

Advanced Data Structure and Algorithms

DCAP605



L OVELY
P ROFESSIONAL
U NIVERSITY



LOVELY
PROFESSIONAL
UNIVERSITY

ADVANCED DATA STRUCTURE AND ALGORITHMS

Copyright © 2011 Anindita Hazra
All rights reserved

Produced & Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Advanced Data Structure and Algorithms

Objectives: To equip the students with the skills to analyze, design, program, and select the appropriate advanced data structures required to solve real world problems.

Sr. No.	Topics
1.	List: Abstract data types, list adts:array implementation, linked list ,common errors, doubly linked list, circularly linked list, cursor implementation of linked list.
2.	Stack: Stack model, implementation of stacks, applications; queues: queue model, array implementation, applications.
3.	Trees: Binary trees, binary search trees, avl trees.
4.	Splay trees, b-trees.
5.	Hashing: Hash functions, open hashing, closed hashing, rehashing.
6.	Heaps: Binary heaps, applications, d-heaps.
7.	Leftist heaps, skew heaps, binomial queues
8.	Sorting: insertion sort, shell sort, heap sort , Merge sort, quick sort, bucket sort, external sort
9.	Graphs: Shortest path algorithms.
10	Network flow problem, minimum spanning tree

CONTENTS

Unit 1:	Introduction to Data Structure and Arrays	1
Unit 2:	Linked Lists	24
Unit 3:	Stacks	58
Unit 4:	Queues	82
Unit 5:	Trees	97
Unit 6:	Binary Search Tree and AVL Trees	116
Unit 7:	Splay Trees	159
Unit 8:	B-trees	170
Unit 9:	Hashing	188
Unit 10:	Heaps	199
Unit 11:	Leftist Heaps and Binomial Queues	219
Unit 12:	Sorting	233
Unit 13:	Graphs	264
Unit 14:	Network Flows	277

Unit 1: Introduction to Data Structure and Arrays

Notes

CONTENTS

Objectives

Introduction

- 1.1 Abstract Data Type (ADT)
- 1.2 Definition of Data Structure
- 1.3 Data Structure Operations
- 1.4 Implementation of Data Structure
- 1.5 List ADT's
 - 1.5.1 ADT of Singly Linked Lists
 - 1.5.2 Implementation
- 1.6 Arrays Implementation
 - 1.6.1 Array
 - 1.6.2 Index or Subscript
 - 1.6.3 Dimensions of an Array
- 1.7 Summary
- 1.8 Keywords
- 1.9 Self Assessment
- 1.10 Review Questions
- 1.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Realise basic concept of data
- Describe Abstract Data Type (ADT)
- Define data structure
- Know data structure types

Introduction

As Static representation of linear ordered list through Array leads to wastage of memory and in some cases overflows. Now we don't want to assign memory to any linear list in advance instead we want to allocate memory to elements as they are inserted in list. This requires Dynamic Allocation of memory.

Semantically data can exist in either of the two forms – atomic or structured. In most of the programming problems data to be read, processed and written are often related to each other. Data items are related in a variety of different ways. Whereas the basic data types such as

Notes

integers, characters etc. can be directly created and manipulated in a programming language, the responsibility of creating the structured type data items remains with the programmers themselves. Accordingly, programming languages provide mechanism to create and manipulate structured data items.

1.1 Abstract Data Type (ADT)

Before we move to abstract data type let us understand what data type is. Most of the languages support basic data types viz. integer, real, character etc. At machine level, the data is stored as strings containing 1's and 0's. Every data type interprets the string of bits in different ways and gives different results. In short, data type is a method of interpreting bit patterns.

Every data type has a fixed type and range of values it can operate on. For example, an integer variable can hold values between the min and max values allowed and carry out operations like addition, subtraction etc. For character data type, the valid values are defined in the character set and the operations performed are like comparison, conversion from one case to another etc. There are fixed operations, which can be carried out on them. We can formally define data types as a formal description of the set of values and operations that a variable of a given type may take. That was about the inbuilt data types. One can also create user defined data types, decide the range of values as well as operations to be performed on them. The first step towards creating a user defined data type or a data structure is to define the logical properties. A tool to specify the logical properties of a data type is Abstract Data Type.

Data abstraction can be defined as separation of the logical properties of the organization of programs' data from its implementation. This means that it states what the data should be like. It does not consider the implementation details. ADT is the logical picture of a data type; in addition, the specifications of the operations required to create and manipulate objects of this data type.

While defining an ADT, we are not concerned with time and space efficiency or any other implementation details of the data structure. ADT is just a useful guideline to use and implement the data type.

An ADT has two parts:

1. Value definition
2. Operation definition.

Value definition is again divided into two parts:

1. Definition clause
2. Condition clause

As the name suggests the definition clause states the contents of the data type and condition clause defines any condition that applies to the data type. Definition clause is mandatory while condition clause is optional.



Example: An integer variable can contain only integer values while a character variable can contain only a valid character value.

To understand the above clauses let us consider the ADT representation of integer data type. In the value definition the definition clause will state that it can contain any number between minimum integer value and the maximum integer value allowed by the computer while the condition clause will impose a restriction saying none of the values will have a decimal point.

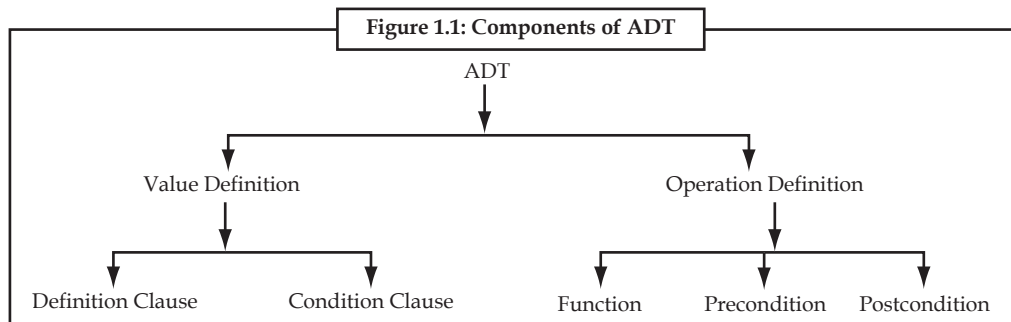
In operation definition, there are three parts:

1. Function
2. Precondition
3. Postcondition

The *function* clause defines the role of the operation. If we consider the addition operation in integers the function clause will state that two integers can be added using this function. In general, precondition specifies any restrictions that must be satisfied before the operation can be applied. This clause is optional. If we consider the division operation on integers then the precondition will state that the divisor should not be zero. So any call for divide operation, which does not satisfy this condition, will not give the desired output.

Precondition specifies any condition that may apply as a pre-requisite for the operation definition. There are certain operations that can be carried out if certain conditions are satisfied. For example, in case of division operation the divisor should never be equal to zero. Only if this condition is satisfied the division operation is carried out. Hence, this becomes a precondition. In that case & (ampersand) should be mentioned in the operation definition.

Postcondition specifies what the operation does. One can say that it specifies the state after the operation is performed. In the addition operation, the post condition will give the addition of the two integers.



As an example, let us consider the representation of integer data type as an ADT. We will consider only two operations addition and division.

Value Definition

1. *Definition clause:* The values must be in between the minimum and maximum values specified for the particular computer.
2. *Condition clause:* Values should not include decimal point.

Operations

1. add (a, b)

Function: add the two integers a and b.

Precondition: no precondition.

Postcondition: output = a + b

Notes

2. Div (a, b)

Function: Divide a by b.

Precondition: $b \neq 0$

Postcondition: $\text{output} = a/b$.

There are two ways of implementing a data structure viz. static and dynamic. In static implementation, the memory is allocated at the compile time. If there are more elements than the specified memory then the program crashes. In dynamic implementation, the memory is allocated as and when required during run time.

Any type of data structure will have certain basic operations to be performed on its data like insert, delete, modify, sort, search etc depending on the requirement. These are the entities that decide the representation of data and distinguish data structures from each other.

Let us see why user defined data structures are essential. Consider a problem where we need to create a list of elements. Any new element added to the list must be added at the end of the list and whenever an element is retrieved, it should be the last element of the list. One can compare this to a pile of plates kept on a table. Whenever one needs a plate, the last one on the pile is taken and if a plate is to be added on the pile, it will be kept on the top. The description wants us to implement a stack. Let us try to solve this problem using arrays.

We will have to keep track of the index of the last element entered in the list. Initially, it will be set to -1. Whenever we insert an element into the list we will increment the index and insert the value into the new index position. To remove an element, the value of current index will be the output and the index will be decremented by one. In the above representation, we have satisfied the insertion and deletion conditions.

Using arrays we could handle our data properly, but arrays do allow access to other values in addition to the top most one. We can insert an element at the end of the list but there is no way to ensure that insertion will be done only at the end. This is because array as a data structure allows access to any of its values. At this point we can think of another representation, a list of elements where one can add at the end, remove from the end and elements other than the top one are not accessible. As already discussed this data structure is called as STACK. The insertion operation is known as push and removal as pop. You can try to write an ADT for stacks.

Another situation where we would like to create a data structure is while working with complex numbers. The operations add, subtract division and multiplication will have to be created as per the rules of complex numbers. The ADT for complex numbers is given below. Only addition and multiplication operations are considered here, you can try to write the remaining operations.

The ADT for complex numbers is as follows:

Value Definition

1. **Definition Clause:** This data type has values $a + ib$ where a and b are integers and i is equal to under root of -1.
2. **Condition Clause:** i term should be present.

Operations

1. **add:**

Function: add, add two complex numbers $a_1 + ib_1$ and $a_2 + ib_2$.

Precondition: no Precondition.

Post condition: $(a_1 + a_2) + i(b_1 + b_2)$.

2, *multiply:*

Function: multiply two complex numbers $a1 + ib1$ and $a2 + ib2$.

Precondition: no Precondition.

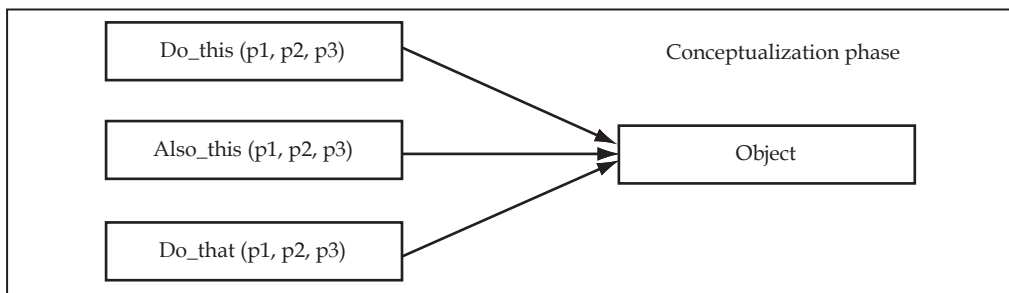
Post condition: $a1a2 + ia1b2 + ib1a2 + i2b1b2$.

i.e. $a1a2 + ia1b2 + ib1a2 - b1b2$.

Abstract data type (ADT) is a mathematical model with a collection of operations defined on that model. Sets of integers, together with the operations of union, intersection and set difference, form a simple example of an ADT. The ADT encapsulates a data type in the sense that the definition of the type and all operations on the type can be localized and are not visible to the users of the ADT. To the users, just the declaration of the ADT and its operations are important.

Abstract Data Type (ADT)

1. A framework for an object interface
2. What kind of stuff it'd be made of (no details)?
3. What kind of messages it would receive and kind of action it'll perform when properly triggered?



From this we figure out

1. Object make-up (in terms of data)
2. Object interface (what sort of messages it would handle?)
3. How and when it should act when triggered from outside (public trigger) and by another object friendly to it?

These concerns lead to an ADT – a definition for the object.

An Abstract Data Type (ADT) is a set of data items and the methods that work on them.


An implementation of an ADT is a translation into statements of a programming language, of the declaration that defines a variable to be of that ADT, plus a procedure in that language for each operation of the ADT. An implementation chooses a data structure to represent the ADT; each data structure is built up from the basic data types of the underlying programming language. Thus, if we wish to change the implementation of an ADT, only the procedures implementing the operations would change. This change would not affect the users of the ADT.

Although the terms 'data type', 'data structure' and 'abstract data type' sound alike, they have different meanings. In a programming language, the data type of a variable is the set of values that the variable may assume. For example, a variable of type boolean can assume either the value true or the value false, but no other value. An abstract data type is a mathematical model, together with various operations defined on the model. As we have indicated, we shall design algorithms in terms of ADTs, but to implement an algorithm in a given programming language

Notes

we must find some way of representing the ADTs in terms of the data types and operators supported by the programming language itself. To represent the mathematical model underlying an ADT, we use data structures, which are a collection of variables, possibly of several data types, connected in various ways.

The cell is the basic building block of data structures. We can picture a cell as a box that is capable of holding a value drawn from some basic or composite data type. Data structures are created by giving names to aggregates of cells and (optionally) interpreting the values of some cells as representing relationships or connections (e.g., pointers) among cells.



Task Value definition is a one part of ADT the second one is

1.2 Definition of Data Structure

A data structure is a set of data values along with the relationship between the data values. Since, the operations that can be performed on the data values depend on what kind of relationships exists amongst the data values, we can specify the relationship amongst the data values by specifying the operations permitted on the data values. Therefore, we can say that a data structure is a set of values along with the set of operations permitted on them. It is also required to specify the semantics of the operations permitted on the data values, and this is done by using a set of axioms, which describes how these operations work, and therefore a data structure is made of:

1. A set of data values.
2. A set of functions specifying the operations permitted on the data values.
3. A set of axioms describing how these operations work.

Hence, we conclude that a data structure is a triple (D,F,A) , where

1. D is a set of data values
2. F is a set of functions
3. A is a set of axioms

A triple (D, F, A) is referred to as an abstract data structure because it does not tell anything about its actual implementation. It does not tell anything about how these values will be physically represented in the computer memory and these functions will be actually implemented. Therefore every abstract data structure is required to be implemented, and the implementation of an abstract data structure requires mapping of the abstract data structure to be implemented into the data structure supported by the computer. For example, if the abstract data structure to be implemented is integer, then it can be implemented by mapping into bits which is a data structure supported by hardware. This requires that every integer data value is to be represented using suitable bit patterns and expressing the operations on integer data values in terms of operations for manipulating bits.

Data Structure mainly two types:

1. Linear type data structure
2. Non-linear type data structure

Linear data structure: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists.

Non-linear data structure: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs.

Basic Concept of Data

The memory (also called storage or core) of a computer is simply a group of bits (switches). At any instant of the computer's operation any particular bit in memory is either 0 or 1 (off or on). The setting or state of a bit is called its value and that is the smallest unit of information. A set of bit values form data.

Some logical properties can be imposed on the data. According to the logical properties data can be segregated into different categories. Each category having unique set of logical properties is known as data type.

Data type are of two types:

1. Simple data type or elementary item like integer, character.
2. Composite data type or group item like array, structure, union.

Data structures are of two types:

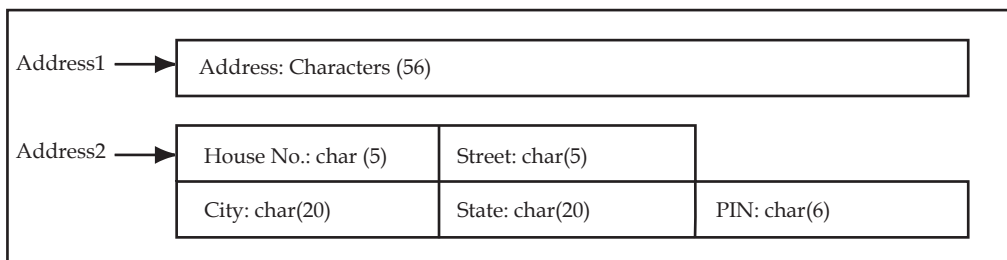
1. **Primitive Data Structures:** Data can be structured at the most primitive level, where they are directly operated upon by machine-level instructions. At this level, data may be character or numeric, and numeric data may consist of integers or real numbers.
2. **Non-primitive Data Structures:** Non-primitive data structures can be classified as arrays, lists, and files.

An array is an ordered set which contains a fixed number of objects. No deletions or insertions are performed on arrays i.e. the size of the array cannot be changed. At best, elements may be changed.

A list, by contrast, is an ordered set consisting of a variable number of elements to which insertions and deletions can be made, and on which other operations can be performed. When a list displays the relationship of adjacency between elements, it is said to be linear; otherwise it is said to be non-linear.

A file is typically a large list that is stored in the external memory of a computer. Additionally, a file may be used as a repository for list items (records) that are accessed infrequently.

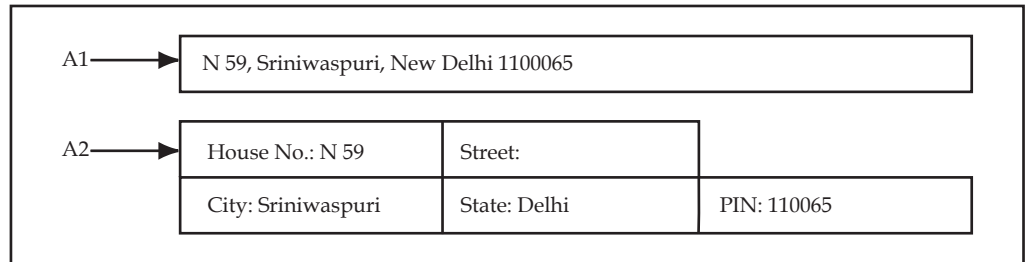
From a real world perspective, very often we have to deal with structured data items which are related to each other. For instance, let us consider the address of an employee. We can take address to be one variable of character type or structured into various fields, as shown below:



As shown above Address1 is unstructured address data. In this form you cannot access individual items from it. You can at best refer to the entire address at one time. While in the second from, i.e., Address2, you can access and manipulate individual fields of the address - House No., Street, PIN etc. Given hereunder are two instances of the address1 and address2 variables.

Notes

Let A1, A2 be two Address1 type variables and B1, B2 be two address2 type variables. The data can now be stored in the following way:



One can access different fields of A2 as shown below:

A 2.HouseNo = N 59

A 2.Street = " "

A 2.City = "Srinivaspuri"

A 2.State = "New Delhi"

A 2.PIN = "110065"

Data structure is a combination of one or more basic data types to form a single addressable data type along with operations defined on it.



Note It is a data type and hence one can create variables of that type in a computer program. Just as each basic data type allows programmers to perform certain operations, data structures also let programmers operate on them.

Data Abstraction: Data abstraction is a tool that allows each data structure to be developed in relative isolation from the rest of the solution. The study of data structure is organized around a collection of abstract data types that includes lists, trees, sets, graphs, and dictionaries.

1.3 Data Structure Operations

The data appearing in our data structure is processed by means of certain operations. The particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following four operations play a major role:

1. **Transversing:** Accessing each record exactly once so that certain items in the record may be processed. (This accessing or processing is sometimes called 'visiting' the records.)
2. **Searching:** Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.
3. **Inserting:** Adding new records to the structure.
4. **Deleting:** Removing a record from the structure.

Sometimes two or more data structure of operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

1.4 Implementation of Data Structure

In this unit, we will discuss the distinguish between the usage and the implementation of a data structure. This difference is crucial in understanding the concept of the separation, definition and use that should be practiced for designing good software. We then present the implementation of a few data structures.

Our main purpose of this course is to describe important data structures and to develop algorithms for the various operations on such data structures.

While presenting a data structure we shall discuss ways to represent it in the computer memory. In many cases, we shall assume array to be the primitive structure to house other data structures. In this context, we use the term array to mean a one-dimensional array having elements of a particular type. Every high level programming language includes facilities to handle arrays and as such implementation of a data structure housed in an array does not offer any problem. Even in assembly or machine languages, a block of contiguous storage locations can be conveniently considered to be an array for the said purpose. Borrowing the notation of C language, we shall denote an array type as follows:

```
<base type> <array name>[<number of elements>;
```

Thus,

```
int A[50];
```

will denote an array of 50 elements where A is the name of the array. An individual element of this array will be denoted by

A[i] where $0 < i < 50$.

In general, an element will be shown as

```
<array name>[<index>]
```

where $0 < \text{index} < \text{number of elements}$

In many cases, the relations among the elements of a data structure need not be explicitly represented because these are implicitly known. When the relations need to be explicitly represented, there are two possible approaches. Either the relations are represented separately (possible as another data structure) or the elements are augmented to include additional fields that represent structural information. Later, we shall see many examples of both these approaches and as such, we do not consider any example at this stage.

It is clear from the above discussion that sometimes an element will also include structural information. In such cases, we shall call it a node to make a minor distinction between the two. Thus the distinction between a node and an element (or a component) is that a node is an element (or a component) plus the structural information (if any) carried into it.



Task

Every high level programming language includes facilities to handle arrays what about machine level and assembly level languages.

1.5 List ADT's

An array is an example of list. Arrays have fixed size, which is declared and fixed at the start of the program, and therefore cannot be changed while it is running.



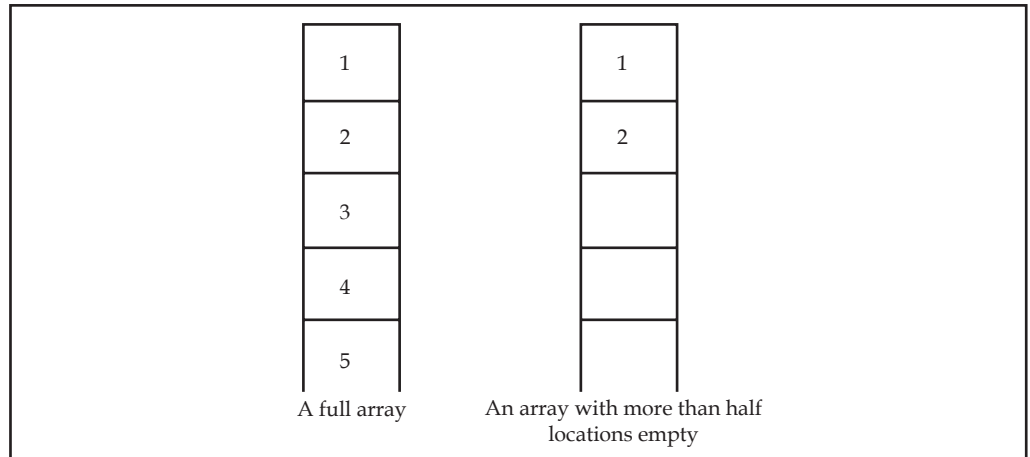
Example: Suppose an array of size 5 has been declared at the start of the program.

Notes

Now, this size cannot be changed while running the program. This we all know is static allocation. When writing the program, we have to decide on the maximum amount of memory that would be needed. If we run the program on a small collection of data, then much of the space will go waste. If program is run on bigger collection of data, then we may exhaust the space and encounter an overflow. Consider the following example:

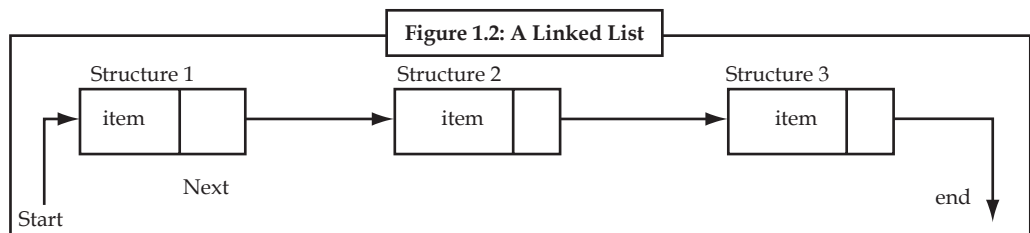


Example: Suppose, we define an array of size 5. If we store 5 elements in it, it is said to be full and no space is left in it. On the contrary, if we store 2 elements in it, then 3 positions are empty and virtually useless, resulting in wastage of memory.



Dynamic data structures can avoid these difficulties. The idea is to use dynamic memory allocation. We allocate memory for individual elements as and when they are required. Each memory location contains a pointer to the location where the successive element is stored. A pointer or a link or a reference is a variable, which stores the memory address of some other variable. If we use pointers to locate the data in which we are interested, then we need not worry about where the data is actually stored, since by using a pointer, we can let the computer system itself locate the data when required.

Linked lists use the concept of dynamic memory allocation. In this respect they are different than arrays. Every node in a linked list contains a 'link' to the next node as shown below. This link is achieved by using pointers.



This type of list is called a linked list because it is a list where order is given by links from one item to the next.

1.5.1 ADT of Singly Linked Lists

There are various operations, which can be performed on lists. The list Abstract Data Type definition given here contains a small collection of basic operations, which can be performed on lists:

List ADT Specification

Notes

Value Definition: The value definition of a linked list contains a data type for storing the value of the node along with the pointer to the next node. The value can be represented using a simple data type or a collection of basic data types. However, it must necessarily contain at least one pointer to the next structure. This can be shown as follows:

```
struct datatype
{
int item;
struct datatype *next;
}
or,
struct datatype
{
int item;
float info;
char str;
struct datatype *next;
}
```

Definition clause: The nodes of the list are all of the same type, and have a key field called key. The list is logically ordered from smallest unique element of key to the largest value i.e. at any position the key of the element is greater than its predecessor and smaller than its successor.

Operations

1. **clist:**

Function: creates a list and initializes it as empty.

Preconditions: none.

Postconditions: list is created and is initialized as empty.

2. **insert:**

Function: inserts new element into the list either at the beginning, in the middle or at the end.

Preconditions: a list already exists.

Postconditions: list is returned with the new element inserted in it.

3. **delete:**

Function: searches a list for the element and removes the element from the list.

Preconditions: the list already exists.

Postconditions: the list is returned with the element removed from it.

4. **print:**

Function: traverses the list and prints each element.

Preconditions: the list already exists.

Notes

Postconditions: list elements are printed in the order they are present in the list. List remains unchanged.

5. **modify:**

Function: searches for an element and replaces it with a new value.

Preconditions: the list already exists.

Postconditions: the element if present is modified by a new value.

These are the basic set of operations that might be needed to create and maintain a list of elements. Other operations, which can be performed on linked lists, are:

1. Counting the elements in a list.
2. Concatenating two lists.

Users can think of more operations like comparing two lists, adding the elements of two lists, etc. depending on a specific problem and try building ADT's of their own.

1.5.2 Implementation

Each element of the list is called a node and consists of two or more members. Some members can contain the information pertaining to that node and the others may be pointers to other nodes. In case of a singly linked list, one member consists of such a pointer. A linked list is therefore a collection of structures ordered not by their physical placement in memory but by logical links that are stored as part of data in the structure itself. The link is in form of a pointer to another structure of the same type.

Such a structure is represented in 'C/C++' as follows:

```
struct node
{
int item;
struct node *next;
};
```

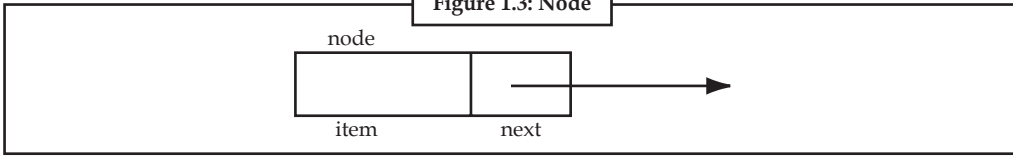
The first member is an integer item and the second a pointer to the next node in the list as shown. The item can be any complex data type. That is it can contain a collection of basic data types. Further, as we will study doubly linked lists later, we can have more than two pointers. One, pointing to the successor node and the other to the predecessor. The pointer type is the type of the node itself. This node can be shown as follows:

```
struct node
{
int item;
float info;
char str;
struct node *next;
};
```

Right now, we will limit our discussion to singly linked lists with only two members, i.e. one containing the data and the other a pointer to the next node.

Notes

Figure 1.3: Node



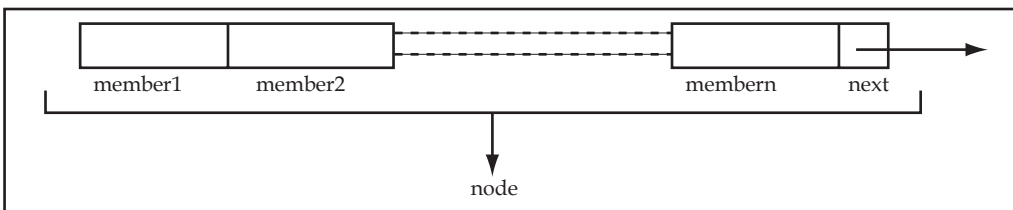
Such structures, which contain a member field that points to the same structure type, are called self-referential structures. A node may be represented in general form as follows:

Figure 1.4: Structure Declaration for the Node

```

struct label-name
{
    type member1;
    type member2;
    type member3;
    . . . .
    . . . .
    struct label-name *next;
};
    
```

The node may contain more than one item with different data types. However, one of the items must be a pointer of the type label-name. The above node with all its members can be depicted as follows:



Consider a simple example to understand the concept of linking. Suppose we define a structure as follows:

```

struct list
{
    int value;
    struct list *next;
};
    
```

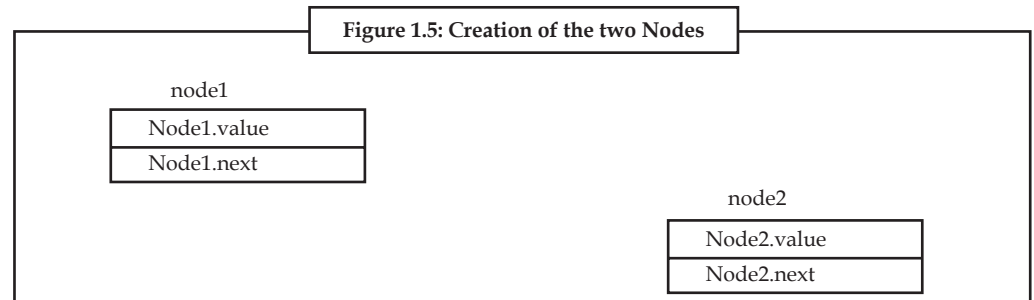
Assume that the list contains two node viz. node1 and node2. They are of type struct list and are defined as follows:

```

struct list node1,node2;
    
```

Notes

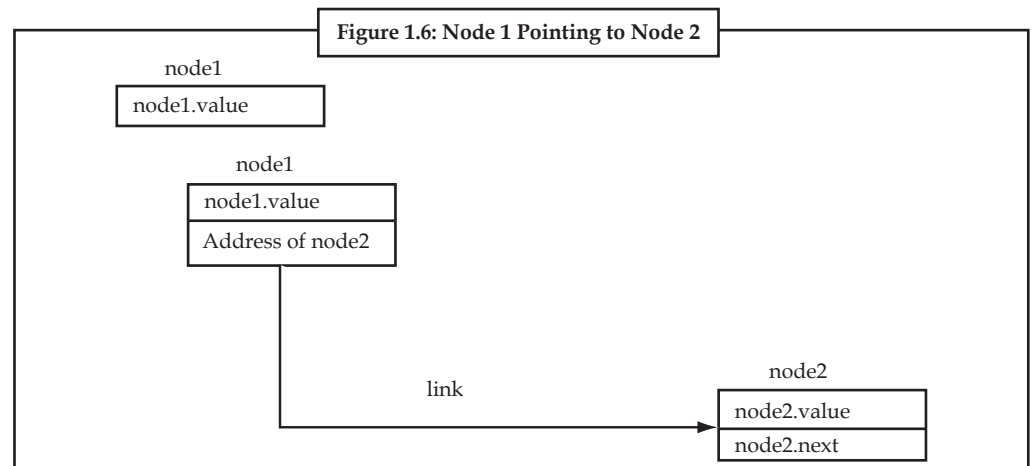
This statement creates space for two nodes each containing two empty fields as shown below:



The next pointer of node1 can be made to point to node2 by the statement

```
node1.next=&node2;
```

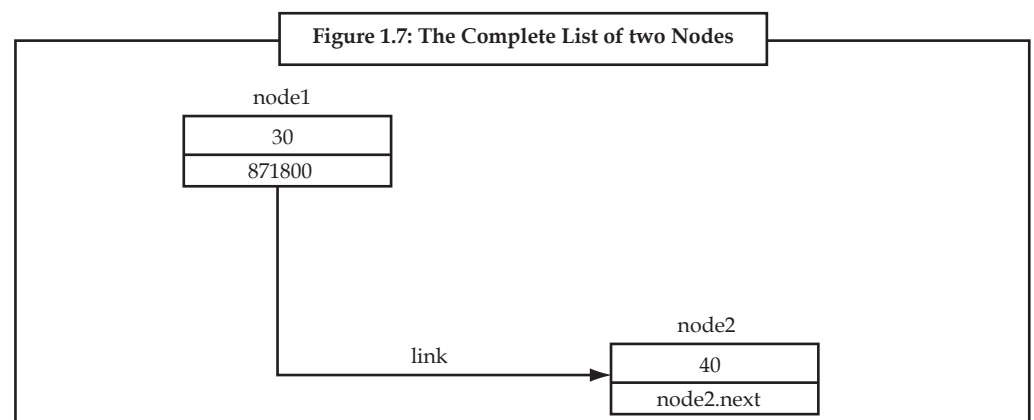
This statement stores the address of node2 into the field node1.next and thus establishes a “link” between node1 and node2 as shown below:



Now we can assign values to the field value.

```
node1.value=30;
node2.value=40;
```

The result is as follows:



Assume that the address of node2 is 871800. As you can see, that address is now stored in the next field of node1.

We may continue this process to create a linked list of any number of values. Each time you need to store a value allocate the node and use it in the list. For example,

```
node2 . next = &node3 ;
```

would add another link.

Also every list must have an end. This is necessary for processing the list. C has a special pointer value called NULL that can be stored in the next field of the last node.

In the above two-node list, the end of the list is marked as follows:

```
node2 . next = NULL ;
```

The value of the value member of node2 can be accessed using the next member of node1 as follows:

```
cout << "\n" << node1 . next -> value ;
```



Task

Write a syntax to represent a node.

1.6 Arrays Implementation

1.6.1 Array

A group of finite number of identical type of elements accessible by a common name. An array has a name which is also the name of each of its elements. For example, let there be an array abc of integers having 6 elements. The same is shown below.

The names of different elements are shown below.

abc[0]	abc[1]	abc[2]	abc[3]	abc[4]	abc[5]
--------	--------	--------	--------	--------	--------

1.6.2 Index or Subscript

The integral value that corresponds to the ordinal position of an array element is called index or subscript.

1.6.3 Dimensions of an Array

On the basis of the number of indices required to access a member element in an array, an array can be classified into the following categories:

1. Single-dimensional Arrays
2. Multi-dimensional Arrays

Each type of Array has different kind of memory representation.

Memory Allocation to Arrays

Member elements of an array are provided contiguous memory locations. When a program requests the operating system that it intends to create an array the operating system computes the size of one element and multiplies the same with the number of elements requested to obtain the total number of memory cells required. The operating system then allocates those many

Notes

memory locations to the program for array. In case the required number of memory cells exceeds the available memory the operating system returns an error code to the program that made the request.

Memory Allocation to One-dimensional Array

A one dimensional array is a list of finite number n of Homogeneous data elements (i.e. data elements of the same type) such that:

1. The elements of the Array are referenced respectively by an index set consisting of n consecutive numbers.
2. The elements of the Array are stored respectively in successive memory locations.

The number n of elements is called the length or size of the Array.


If not explicitly stated, we will assume the index set consists of the integers 1, 2, ...n. In general, the length or the number of data elements of the Array can be obtained from the index set by the formula

$$\text{Length} = \text{UB} - \text{LB} + 1 \quad \dots(1)$$

where,

UB is the largest index, called the upper bound and

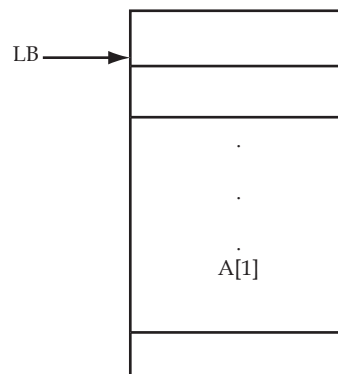
LB is the smallest index, called the lower bound, of the Array.



Note Note that length = UB when LB = 1.

The elements of an Array A may be denoted by the subscript notation A[1], A[2], A[3],... A[n].

In general a linear Array A with a subscript lower bound of “one” can be represented pictorially as in figure given below.



If s words are allocated for each element or node (i.e. size of data types of elements of Array is s), then total memory space allocated to array is given by:

$$\text{Memory Space Request} = (\text{UB} - \text{LB} + 1) * s \quad \dots(2)$$

If we want to calculate the memory space required for first i-1 elements of an Array then slight modification in formula (2) will be needed. i.e.

$$\text{Space Required} = (i - 1 - \text{LB} + 1) * s$$

or $\text{Space Required} = (i - \text{LB}) * s \quad \dots(3)$

Notes

If the Address of A[LB] is $_$ then the Address of i th element of Array will be given by:

$$\text{Address of } A[i] = _ + (i - \text{LB}) * s \quad \dots(4)$$

Now, we can write a Program for addressing the i th element of a single dimensional Array. We can take $_$, i , LB and s as input from user and output the address of $A[i]$.

Consider a One-dimensional Array of figure given below. It has five elements. Each element takes 4 bytes to store.

Suppose Address of $a[0]=1000$, and we want to calculate the address of a_4 then from the formula (4) we have,

$$_ = 1000,$$

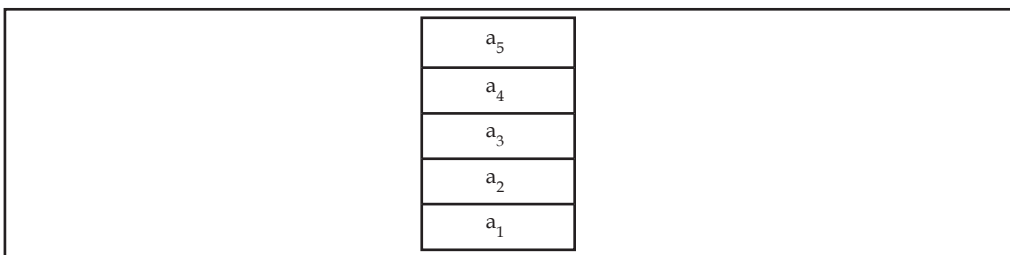
$$i = 4,$$

$$\text{LB} = 1,$$

$$s = 4$$

$$\text{Address of } a_4 = 1000 + (4 - 1) * 4$$

$$\text{Address of } a_4 = 1012.$$



Memory Representation of Two-dimensional Array

Even though Multidimensional Arrays are provided as a standard data object in most of the high level languages, it is interesting to see how they are represented in memory. Memory may be regarded as one dimensional with words numbered from 1 to m . So we are concerned with representing n dimensional Array in a one dimensional memory.

A two dimensional ' $m \times n$ ' Array A is a collection of $m.n$ data elements such that each element is specified by a pair of integers (such as j, k), called subscripts, with property that

$$1 \times j \times m \text{ and } 1 \times k \times n.$$

The element of A with first subscript j and second subscript k will be denoted by A_j, K or $A[j, K]$. Two dimensional arrays are called matrices in mathematics and tables in Business Applications.

Consider the following example of a 3×3 array.

	Columns		
1.	$A [1, 1]$	$A [1, 2]$	$A [1, 3]$
2.	$A [2, 1]$	$A [2, 2]$	$A [2, 3]$
3.	$A [3, 1]$	$A [3, 2]$	$A [3, 3]$

Two-dimensional 3×3 Array A

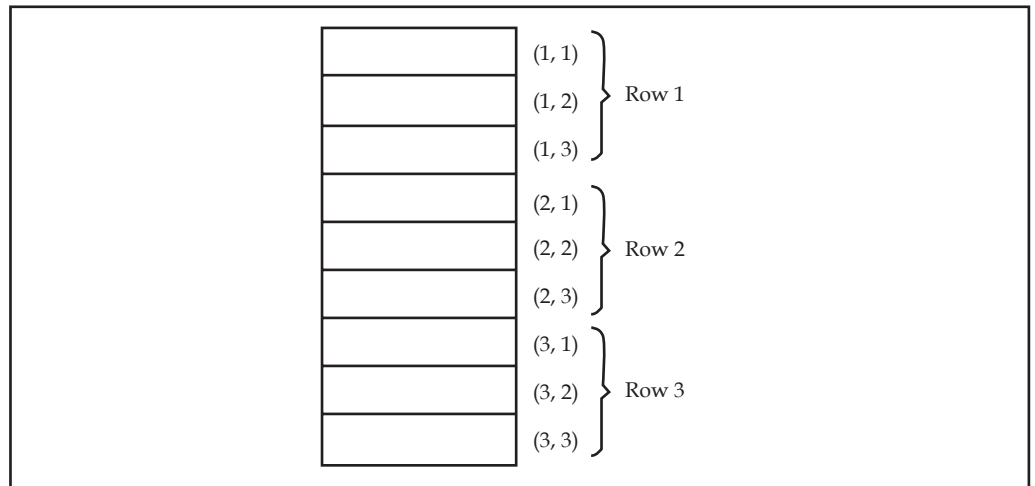
Notes

Programming language stores the array in either of the two way:

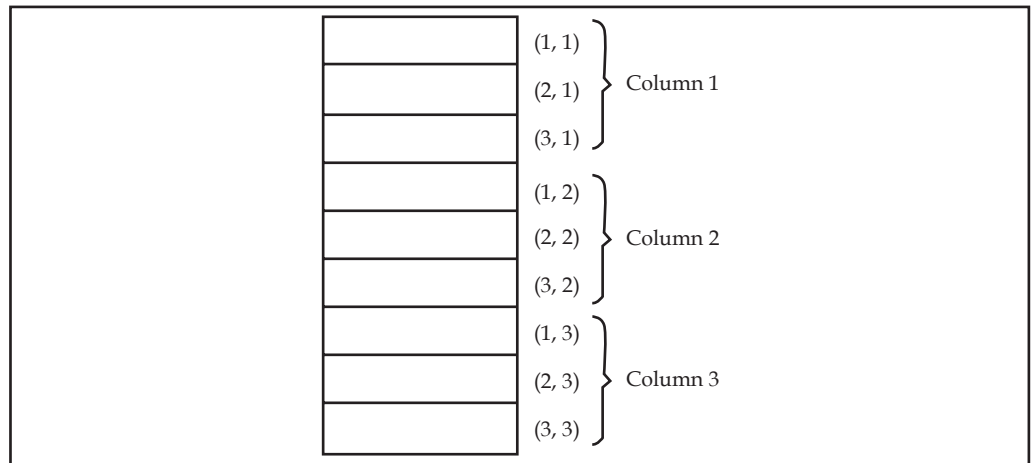
1. Row Major Order
2. Column Major Order

In Row Major Order elements of 1st Row are stored first in linear order and then comes elements of next Row and so on.

In Column Major Order elements of 1st column are stored first linearly and then comes elements of next column. When Above Matrix is stored in memory using Row Major Order form then the representation will be as shown in figure.



Representation with Column Major form will be:



Number of elements in any two-dimensional Array can be given by:

$$\text{No. of elements} = (UB_1 - LB_1 + 1) * (UB_2 - LB_2 + 1) \quad \dots(5)$$

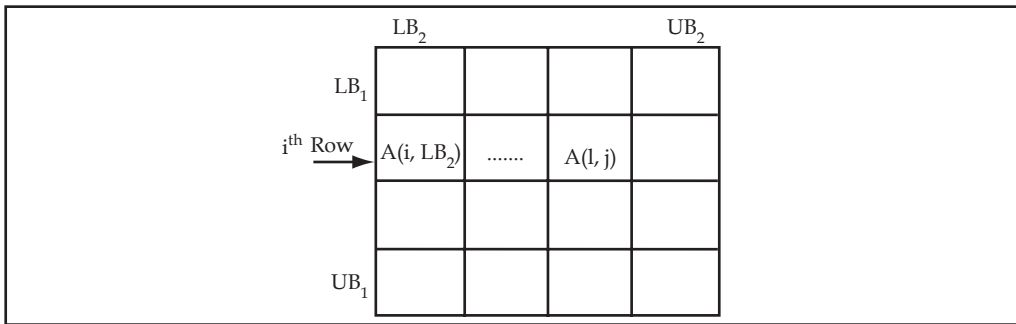
where,

UB_1 is upper bound of 1st dimension

LB_1 is lower bound of 1st dimension UB_2 and LB_2

are upper and lower bounds of 2nd dimensions

Notes



If we want to calculate the number of elements till Ist Row then.

$$\text{No. of elements} = (UB_2 - LB_2 + 1) * (1 - 1 + 1)$$

or $\text{No. of elements} = UB_2 - LB_2 + 1 \dots(6)$

No. of elements in (j - 1) Rows = (j - 1) (UB₂ - LB₂ + 1). If s be the size of data types of Array elements then memory space required for storing i-1 Rows will be.

$$\text{Space Required} = (UB_2 - LB_2 + 1) (i - 1)*s \dots(7)$$

If x be the address of A[LB₁, LB₂] then Address of A(i, LB₂) will be:

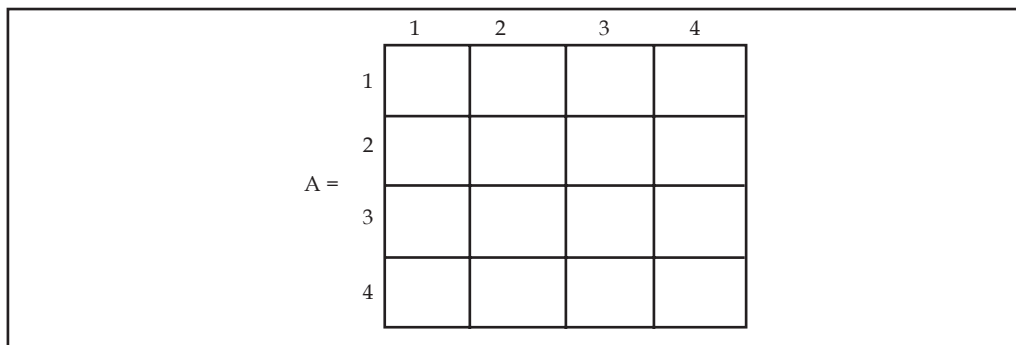
$$\text{Add} = x + (UB_2 - LB_2 + 1) (i-1)*s \dots(8)$$

Address of A[i, j] will be

$$\text{Address of A[i, j]} = x + [(UB_2 - LB_2 + 1) (i - 1) + (j - 1)]*s \dots(9)$$

This is Address Scheme for Row Major form. For Column Major form

$$\text{Address of A[i, j]} = x + [(UB_1 - LB_1 + 1) (j - 1) + (i - 1)]*s \dots(10)$$



Consider a two-dimensional matrix of figure. Suppose address of A11 is 2000 and this two dimensional array contains elements of 4-bytes each we want to calculate the address of A23, then by the formula (9) we have

$$A_{11} = 2000,$$

$$UB_2 = 4, LB_2 = 1$$

$$i = 2, j = 3,$$

and $s = 4$

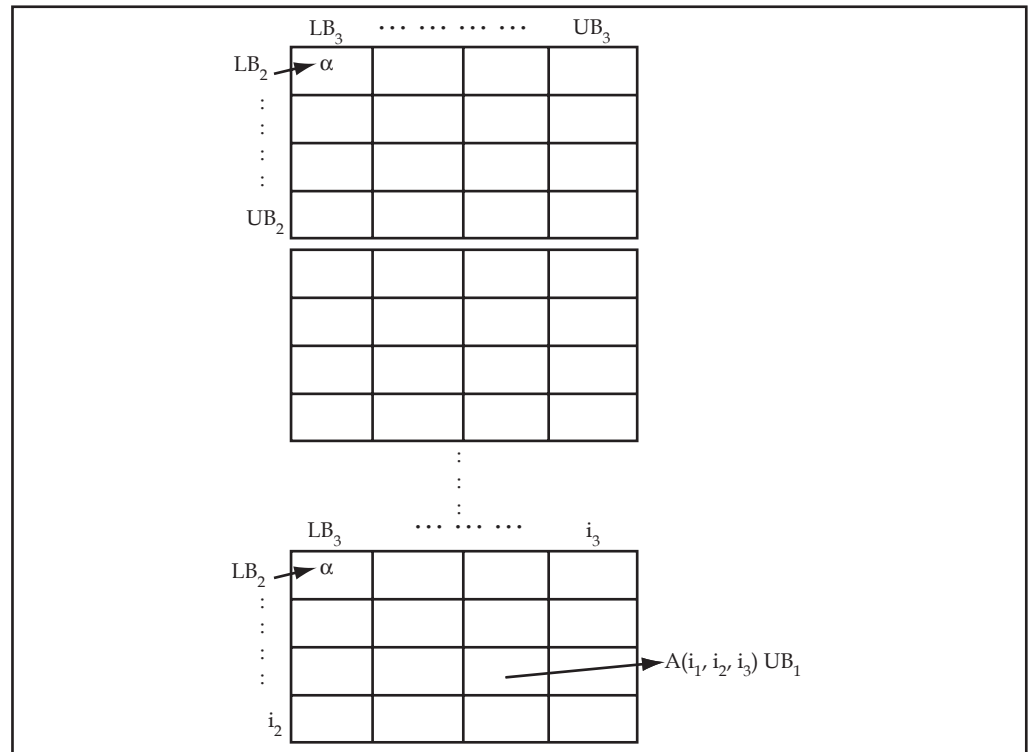
then

$$\text{add. of A23} = 2000 + [(4 - 1 + 1) (2 - 1) + (3 - 1)]*4 = 2024$$

Notes

Memory Allocation to Three-dimensional Array

Suppose we want to Calculate Address of $A(i_1, i_2, i_3)$. i.e. Address of i_1 th window, i_2 th Row and i_3 th Column.



Then, for Row Major Order Form:

Address of $A(i_1, i_2, i_3)$

$$= _ + ((i_3 * (UB_2 - LB_2 + 1) + i_2) * (UB_1 - LB_1 + 1) + i_1) * s \quad \dots(11)$$

Memory Allocation to Multi-dimensional Array

Now if we generalize the formulae (11) for the n dimensions of Array then

For Row Major form

Let, Add. of $A(LB_1, LB_2, LB_3, \dots, LB_n) = x$

Add. of $A [i_1, i_2, i_3, \dots, i_n]$

$$= x [(i_1 - LB_1) (UB_2 - LB_2 + 1)$$

$$\dots (UB_n - LB_n + 1) + (i_3 - LB_3)(LB_4 - LB_4 + 1) \dots (UB_n - LB_n + 1) + \dots$$

$$\dots (i_n - 1 - LB_n - 1)(UB_n - LB_n + 1) + (i_n - LB_n)] * s \quad \dots(12)$$

Now Program for Address Calculation can be made by taking number of dimensions, lower and upper Bounds of each dimensions, address of $A[LB_1, LB_2, \dots, LB_n]$ and size as input from user and give Address of $A[i_1, i_2, \dots, i_n]$ as output.

1.7 Summary

- Data structure is a combination of one or more basic data types to form a single addressable data type.
- An algorithm is a finite set of instructions which, when followed, accomplishes a particular task, the termination of which is guaranteed under all cases, i.e. the termination is guaranteed for every input.
- The instructions must be unambiguous and the algorithm must produce the output within a finite number of executions of its instructions.
- Associated with each data structure, there are some algorithms to perform the basic operations on the elements.
- Abstract data type (ADT) is a mathematical model with a collection of operations defined on that model. Although the terms 'data type', 'data structure' and 'abstract data type' sound alike, they have different meanings.

1.8 Keywords

Linear Data Structure: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached.

Non-linear Data Structure: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.

Searching: Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

Transversing: Accessing each record exactly once so that certain items in the record may be processed.

1.9 Self Assessment

Fill in the blanks:

1. The memory of a computer is simply a group of
2. A is typically a large list that is stored in the external memory of a computer.
3. An is an ordered set which contains a fixed number of objects.
4. is a combination of one or more basic data types to form a single addressable data type along with operations defined on it.
5. Adding new records to the structure is known as

State whether the following statements are true or false:

6. An array is an example of list.
7. Linked lists can not use the concept of dynamic memory allocation.
8. Each element of the list is called a node and consists of two or more members.
9. At machine level, the data is stored as strings containing 1's and 0's.
10. A tool to specify the logical properties of a data type is Abstract Data Type.

Notes

1.10 Review Questions

1. Explain operations of data structure.
2. Distinguish between linear and non-linear type of data structure.
3. Write short note on ADT.
4. "ADT is the logical picture of a data type". Explain
5. "Data abstraction is a tool that allows each data structure to be developed in relative isolation from the rest of the solution". Discuss
6. How will you implement data structure? Explain
7. What do you mean by ADT of singly lists? Discuss
8. "The list is logically ordered from smallest unique element of key to the largest value". Explain
9. "A pointer or a link or a reference is a variable, which stores the memory address of some other variable". Discuss
10. How will you remove a record from the structure? Explain

Answers: Self Assessment

- | | |
|--------------|-------------------|
| 1. bits | 2. file |
| 3. array | 4. Data structure |
| 5. Inserting | 6. True |
| 7. False | 8. True |
| 9. True | 10. True |

1.11 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.

Data Structures and Algorithms; Shi-Kuo Chang; World Scientific.

Data Structures and Efficient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.

Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi

Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing

RG Dromey, How to Solve it by Computer, Cambridge University Press.

Shi-kuo Chang, Data Structures and Algorithms, World Scientific

Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.

Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms. Prentice-Hall of India Pvt. Limited, New Delhi

Timothy A. Budd, Classic Data Structures in C++, Addison Wesley.

Notes



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 2: Linked Lists

CONTENTS

Objectives

Introduction

- 2.1 Concept of Linked Lists
- 2.2 Representation of Linked List
- 2.3 Inserting a Node using Recursive Programs
- 2.4 Deleting the Specified Node in Singly Linked List
- 2.5 Inserting a Node after the Specified Node in a Singly Linked List
- 2.6 Linked List Common Errors
- 2.7 Doubly Linked Lists
- 2.8 Circular Linked List
- 2.9 Sorting and Reversing a Linked List
- 2.10 Merging two Sorted Lists
- 2.11 Merging of two Circular Lists
- 2.12 Application of Linked List
- 2.13 Summary
- 2.14 Keywords
- 2.15 Self Assessment
- 2.16 Review Questions
- 2.17 Further Readings

Objectives

After studying this unit, you will be able to:

- Define linked lists
- Describe linked lists representation
- Know doubly linked lists
- Explain circular linked lists

Introduction

There are many other operations that are also useful to apply to sequences of elements. Thus we can form a wide variety of similar ADTs by utilizing different packages of operations. Any one of these related ADTs could reasonably go by the name of list. However, we fix our attention on one particular list ADT whose operations give a representative sampling of the ideas and problems that arise in working with lists.

The standard template library provides a rather different data structure called a list. The STL (Standard Template Library) list provides only those operations that can be implemented efficiently in a List implementation known as doubly linked, which we shall study shortly. In particular, the STL list does not allow random access to an arbitrary list position, as provided by our List operations for insertion, removal, retrieval, and replacement. Another STL template class, called a vector, does provide some random access to a sequence of data values. An STL vector bears some similarity to our List ADT, in particular, it provides the operations that can be implemented efficiently in the List implementation that we shall call contiguous. In this way, our study of the List ADT provides an introduction to the STL classes list and vector.

2.1 Concept of Linked Lists

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage. It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires that every element be capable of holding the data as well as the address of the next element. Thus every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field.

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

This program uses a strategy of inserting a node in an existing list to get the list created. An insert function is used for this. The insert function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as a second parameter, creates a new node by using the data value, appends it to the end of the list, and returns a pointer to the first node of the list. Initially the list is empty, so the pointer to the starting node is NULL. Therefore, when insert is called first time, the new node created by the insert becomes the start node. Subsequently, the insert traverses the list to get the pointer to the last node of the existing list, and puts the address of the newly created node in the link field of the last node, thereby appending the new node to the existing list. The main function reads the value of the number of nodes in the list. Calls iterate that many times by going in a while loop to create the links with the specified number of nodes.

2.2 Representation of Linked List

Because each node of an element contains two parts, we have to represent each node through a structure.

Notes

While defining linked list we must have recursive definitions:

```
struct node
{
    int data;
    struct node * link;
}
```

Here, link is a pointer of struct node type i.e. it can hold the address of variable of struct node type. Pointers permit the referencing of structures in a uniform way, regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

Initialization:

```
main()
{
    struct node *p, *list, *temp;
    list = p = temp = NULL;
    .
    .
    .
}
```



Lab Exercise **Program:**

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    /* if the existing list is empty then insert a new node as the
    starting node */
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node)); /* creates new
        node data value passes
        as parameter */
        if(p==NULL)
        {
            printf("Error\n");
        }
    }
}
```


Notes

```

        exit(0);
    }
    p-> data = n;
    p-> link = p; /* makes the pointer pointing to itself because
it is a circular list*/
}
else
{
    temp = p;
    /* traverses the existing list to get the pointer to the last node
of it */
    while (temp-> link != p)
        temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
data value passes as
parameter and puts its
address in the link field
of last node of the
existing list*/
    if(temp -> link == NULL)
    {
        printf("Error\n");
        exit(0);
    }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = p;
}
return (p);
}

void printlist ( struct node *p )
{
    struct node *temp;
    temp = p;
    printf("The data values in the list are\n");
    if(p!= NULL)
    {
        do
        {
            printf("%d\t",temp->data);

```

Notes

```
        temp=temp->link;
        } while (temp!= p);
    }
    else
        printf("The list is empty\n");
}
void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf ("%d",&n);
    while ( n -- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
        scanf ("%d",&x);
        start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

2.3 Inserting a Node using Recursive Programs

A linked list is a recursive data structure. A recursive data structure is a data structure that has the same form regardless of the size of the data. You can easily write recursive programs for such data structures.



Lab Exercise Program

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    if(p==NULL)
    {
```

```

    p=(struct node *)malloc(sizeof(struct node));
    if(p==NULL)
    {
printf("Error\n");
        exit(0);
    }
    p-> data = n;
    p-> link = NULL;
}
else
    p->link = insert(p->link,n);/* the while loop replaced by
recursive call */
    return (p);
}
void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
printf("%d\t",p-> data);
        p = p-> link;
    }
}
void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
printf( "Enter the data values to be placed in a node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}

```

Notes

This recursive version also uses a strategy of inserting a node in an existing list to create the list. An insert function is used to create the list. The insert function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as the second parameter. It creates the new node by using the data value, then appends it to the end of the list. It then returns a pointer to the first node of the list. Initially, the list is empty, so the pointer to the starting node is NULL. Therefore, when insert is called the first time, the new node created by the insert function becomes the start node. Subsequently, the insert function traverses the list by recursively calling itself. The recursion terminates when it creates a new node with the supplied data value and appends it to the end of the list.

2.4 Deleting the Specified Node in Singly Linked List

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed. Figures 2.1 and 2.2 show the list before and after deletion, respectively.

*Lab Exercise Program*

```
# include <stdio.h>
# include <stdlib.h>

struct node *delet ( struct node *, int );
int length ( struct node * );

struct node
{
    int data;
    struct node *link;
};

struct node *insert(struct node *p, int n)
{
    struct node *temp;
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node));
        if(p==NULL)
        {
            printf("Error\n");
            exit(0);
        }
        p-> data = n;
        p-> link = NULL;
    }
}
```

```
else
{
    temp = p;
    while (temp-> link != NULL)
        temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node));
    if(temp -> link == NULL)
    {
        printf("Error\n");
        exit(0);
    }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = NULL;
}
return (p);
}

void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
        p = p-> link;
    }
}

void main()
{
    int n;
    int x;
    struct node *start = NULL;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
}
```

Notes

```
printf(" The list before deletion id\n");
printlist ( start );
printf("% \n Enter the node no \n");
scanf ( " %d",&n);
start = delet (start , n );
printf(" The list after deletion is\n");
printlist ( start );
}
/* a function to delete the specified node*/
struct node *delet ( struct node *p, int node_no )
{
    struct node *prev, *curr ;
    int i;
    if (p == NULL )
    {
        printf("There is no node to be deleted \n");
    }
    else
    {
        if ( node_no > length (p))
        {
            printf("Error\n");
        }
        else
        {
            prev = NULL;
            curr = p;
            i = 1 ;
            while ( i < node_no )
            {
                prev = curr;
                curr = curr-> link;
                i = i+1;
            }
            if ( prev == NULL )
            {
                p = curr -> link;
                free ( curr );
            }
            else
```

```

    {
        prev -> link = curr -> link ;
        free ( curr );
    }
}
return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
    int count = 0 ;
    while ( p != NULL )
    {
        count++;
        p = p->link;
    }
    return ( count ) ;
}

```

Figure 2.1: Before Deletion

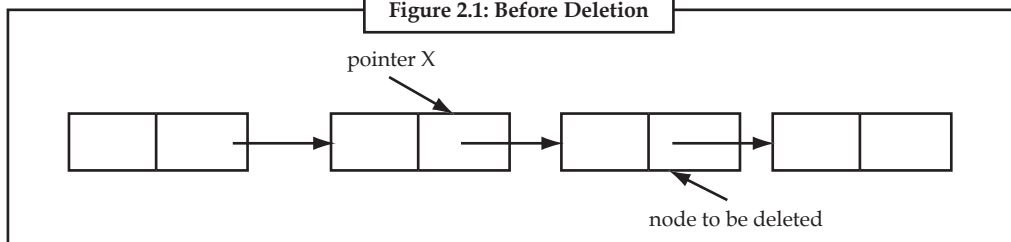
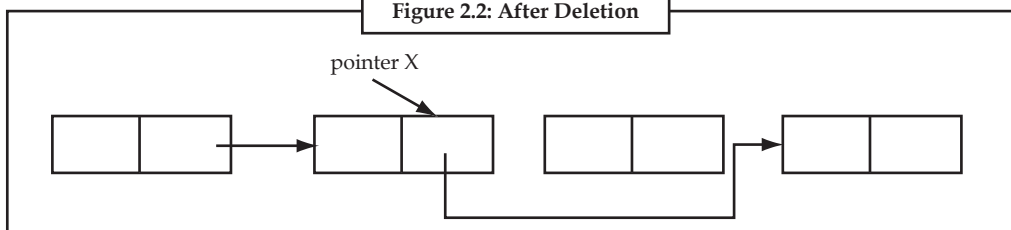


Figure 2.2: After Deletion



2.5 Inserting a Node after the Specified Node in a Singly Linked List

To insert a new node after the specified node, first we get the number of the node in an existing list after which the new node is to be inserted. This is based on the assumption that the nodes of the list are numbered serially from 1 to n. The list is then traversed to get a pointer to the node, whose number is given. If this pointer is x, then the link field of the new node is made to point to the node pointed to by x, and the link field of the node pointed to by x is made to point to the new node. Figures 2.3 and 2.4 show the list before and after the insertion of the node, respectively.

Notes



Lab Exercise Program

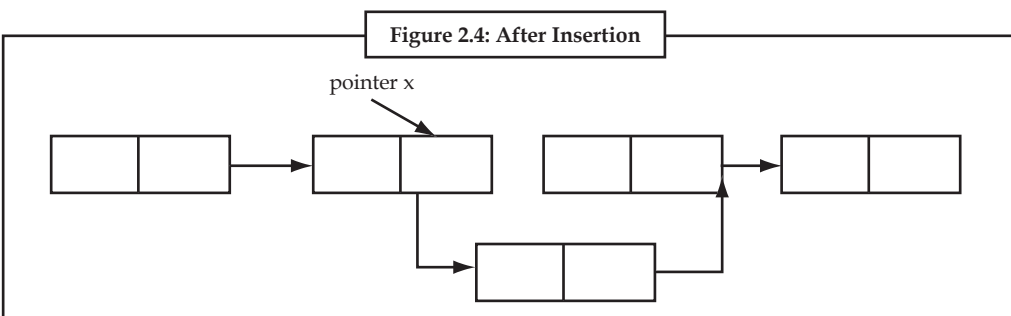
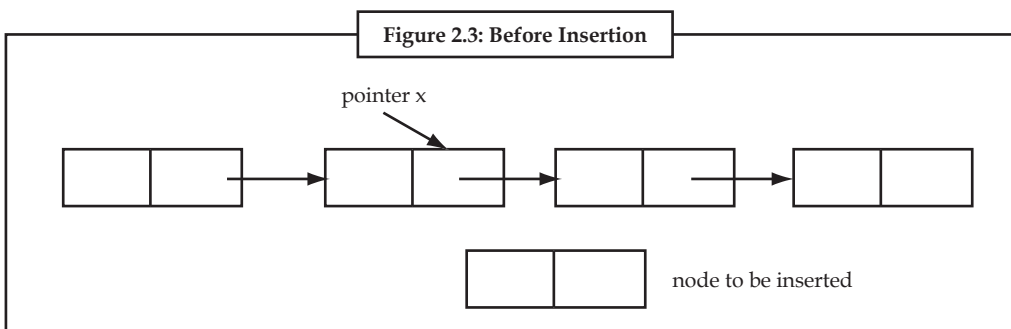
```
# include <stdio.h>
# include <stdlib.h>
int length ( struct node * );
struct node
{
    int data;
    struct node *link;
};
/* a function which appends a new node to an existing list used for
building a list */
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node));
        if(p==NULL)
        {
            printf("Error\n");
            exit(0);
        }
        p-> data = n;
        p-> link = NULL;
    }
    else
    {
        temp = p;
        while (temp-> link != NULL)
            temp = temp-> link;
        temp-> link = (struct node *)malloc(sizeof(struct node));
        if(temp -> link == NULL)
        {
            printf("Error\n");
            exit(0);
        }
        temp = temp-> link;
```



```
        temp-> data = n;
        temp-> link= NULL;
    }
    return (p);
}
/* a function which inserts a newly created node after the specified
node */
struct node * newinsert ( struct node *p, int node_no, int value )
{
    struct node *temp, * temp1;
    int i;
    if ( node_no <= 0 || node_no > length (p))
    {
        printf("Error! the specified node does not exist\n");
        exit(0);
    }
    if ( node_no == 0)
    {
        temp = ( struct node * )malloc ( sizeof ( struct node ));
        if ( temp == NULL )
        {
            printf( " Cannot allocate \n");
            exit (0);
        }
        temp -> data = value;
        temp -> link = p;
        p = temp ;
    }
    else
    {
        temp = p ;
        i = 1;
        while ( i < node_no )
        {
            i = i+1;
            temp = temp-> link ;
        }
        temp1 = ( struct node * )malloc ( sizeof(struct node));
        if ( temp == NULL )
        {
            printf ("Cannot allocate \n");
```

Notes

```
        exit(0)
    }
    temp1 -> data = value ;
    temp1 -> link = temp -> link;
    temp -> link = temp1;
}
return (p);
}
void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
        p = p-> link;
    }
}
void main ()
{
    int n;
    int x;
    struct node *start = NULL;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf(" The list before deletion is\n");
    printlist ( start );
    printf(" \n Enter the node no after which the insertion is to be
done\n");
    scanf ( " %d",&n);
    printf("Enter the value of the node\n");
    scanf("%d",&x);
    start = newinsert(start,n,x);
    printf("The list after insertion is \n");
    printlist(start);
}
```



Task

Write a program to insert a node in singly linked list

2.6 Linked List Common Errors

Here is summary of common errors of linked lists. Read these carefully, and read them again when you have problem that you need to solve.

1. Allocating a new node to step through the linked list; only a pointer variable is needed.
2. Confusing the and the \rightarrow operators.
3. Not setting the pointer from the last node to 0 (null).
4. Not considering special cases of inserting/removing at the beginning or the end of the linked list.
5. Applying the delete operator to a node (calling the operator on a pointer to the node) before it is removed. Delete should be done after all pointer manipulations are completed.
6. Pointer manipulations that are out of order. These can ruin the structure of the linked list.



Task

Draw a diagram to illustrate the configuration of linked nodes that is created by the following statements.

```
Node *p0 = new Node(000);
```

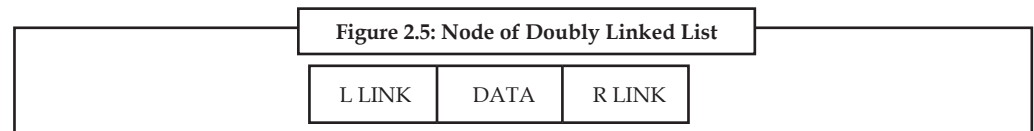
```
Node *p1 = p0->next = new Node(010);
```

```
Node *p2 = p1->next = new Node(020, p1);
```

2.7 Doubly Linked Lists

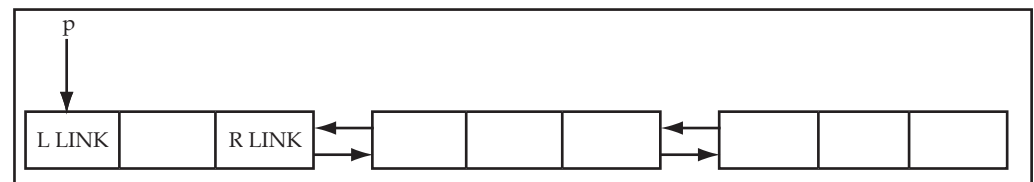
In the single linked list each node provides information about where the next node is in the list. It faces difficulty if we are pointing to a specific node, then we can move only in the direction of the links. It has no idea about where the previous node lies in memory. The only way to find the node which precedes that specific node is to start back at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a single linked list. Since in order to easily delete an arbitrary node one must know the preceding node. This problem can be avoided by using Doubly Linked List, we can store in each node not only the address of next node but also the address of the previous node in the linked list. A node in Doubly Linked List has three fields (Figure 2.5).

1. Data
2. Left Link
3. Right Link



Left link keeps the address of previous node and Right Link keeps the address of next node. Doubly Linked List has following property.

$$p \rightarrow p \rightarrow \text{llink} \rightarrow \text{rlink} = p \rightarrow \text{rlink} \rightarrow \text{llink}.$$



This formula reflects the essential virtue of this structure, namely, that one can go back and forth with equal ease.

Implementation of Doubly Linked List

Structure of a node of Doubly Linked List can be defined as:

```
struct node
{
    int data;
    struct node *llink;
    struct node *rlink;
}
```



Lab Exercise Program

```
# include <stdio.h>
# include <stdlib.h>
struct dnode
```

```
{
int data;
struct node *left, *right;
};
struct dnode *insert(struct dnode *p, struct dnode **q, int n)
{
struct dnode *temp;
/* if the existing list is empty then insert a new node as the
starting node */
if(p==NULL)
{
p=(struct dnode *)malloc(sizeof(struct dnode)); /* creates new
node data value
passed as parameter */
if(p==NULL)
{
printf("Error\n");
exit(0);
}
p-> data = n;
p-> left = p->right =NULL;
*q =p
}
else
{
temp = (struct dnode *)malloc(sizeof(struct dnode)); /* creates
new node using
data value passed as
parameter and puts its
address in the temp
*/
if(temp == NULL)
{
printf("Error\n");
exit(0);
}
temp-> data = n;
temp->left = (*q);
temp->right = NULL;
(*q) = temp;
```

Notes

```
    }
    return (p);
}
void printfor( struct dnode *p )
{
    printf("The data values in the list in the forward order are:\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
        p = p-> right;
    }
}
/* A function to count the number of nodes in a doubly linked list */
int nodecount (struct dnode *p )
{
    int count=0;
    while (p != NULL)
    {
        count ++;
        p = p->right;
    }
    return(count);
}
/* a function which inserts a newly created node after the specified
node in a doubly linked list */
struct node * newinsert ( struct dnode *p, int node_no, int value )
{
    struct dnode *temp, * temp1;
    int i;
    if ( node_no <= 0 || node_no > nodecount (p))
    {
        printf("Error! the specified node does not exist\n");
        exit(0);
    }
    if ( node_no == 0)
    {
        temp = ( struct dnode * )malloc ( sizeof ( struct dnode ));
        if ( temp == NULL )
        {
```

```
printf( " Cannot allocate \n");
exit (0);
}
temp -> data = value;
temp -> right = p;
temp->left = NULL
p = temp ;
}
else
{
temp = p ;
i = 1;
while ( i < node_no )
{
i = i+1;
temp = temp-> right ;
}
temp1 = ( struct dnode * )malloc ( sizeof(struct dnode));
if ( temp == NULL )
{
printf("Cannot allocate \n");
exit(0);
}
temp1 -> data = value ;
temp1 -> right = temp -> right;
temp1 -> left = temp;
temp1->right->left = temp1;
temp1->left->right = temp1
}
return (p);
}
void main()
{
int n;
int x;
struct dnode *start = NULL ;
struct dnode *end = NULL;
printf("Enter the nodes to be created \n");
scanf("%d",&n);
while ( n > 0 )
{
```

Notes

```

printf( "Enter the data values to be placed in a node\n");
scanf( "%d", &x);
start = insert ( start, &end,x );
}
printf("The created list is\n");
printfor ( start );
printf("enter the node number after which the new node is to be
inserted\n");
scanf( "%d", &n);
printf("enter the data value to be placed in the new node\n");
scanf( "%d", &x);
start=newinsert (start,n,x);
printfor(start);
}

```

Explanation

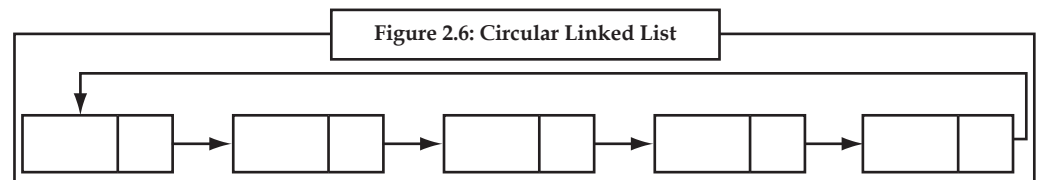
1. To insert a new node in a doubly linked chain, it is required to obtain a pointer to the node in the existing list after which a new node is to be inserted.
2. To obtain this pointer, the node number after which the new node is to be inserted is given as input. The nodes are assumed to be numbered as 1,2,3,..., etc., starting from the first node.
3. The list is then traversed starting from the start node to obtain the pointer to the specified node. Let this pointer be x. A new node is then created with the required data value, and the right link of this node is made to point to the node to the right of the node pointed to by x. And the left link of the newly created node is made to point to the node pointed to by x. The left link of the node which was to the right of the node pointed to by x is made to point to the newly created node. The right link of the node pointed to by x is made to point to the newly created node.

Question

Write a program to delete a specific node from the linked list.

2.8 Circular Linked List

Circular Linked List is another remedy for the drawbacks of the Single Linked List besides Doubly Linked List. A slight change to the structure of a linear list is made to convert it to circular linked list; link field in the last node contains a pointer back to the first node rather than a Null. (See Figure 2.6).



From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point.



Notes

Lab Exercise

Program: Here is a program for building and printing the elements of the circular linked list.

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    /* if the existing list is empty then insert a new node as the
starting node */
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node)); /* creates new
node data value passes
as parameter */
        if(p==NULL)
        {
            printf("Error\n");
            exit(0);
        }
        p-> data = n;
        p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
    }
    else
    {
        temp = p;
        /* traverses the existing list to get the pointer to the last node of
it */
        while (temp-> link != p)
            temp = temp-> link;
        temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
data value passes as
parameter and puts its
address in the link field
of last node of the
existing list*/
```

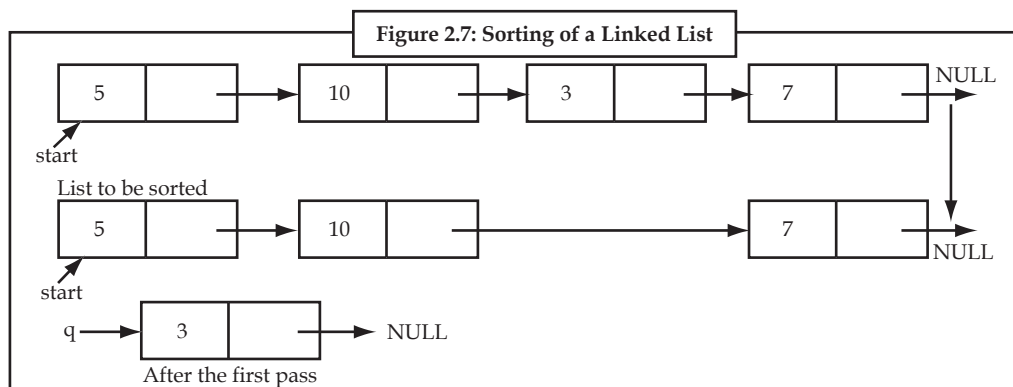
Notes

```
        if(temp -> link == NULL)
        {
            printf("Error\n");
        }
        exit(0);
        temp = temp-> link;
        temp-> data = n;
        temp-> link = p;
    }
    return (p);
}
void printlist ( struct node *p )
{
    struct node *temp;
    temp = p;
    printf("The data values in the list are\n");
    if(p!= NULL)
    {
        do
        {
            printf("%d\t",temp->data);
            temp=temp->link;
        } while (temp!= p)
    }
    else
        printf("The list is empty\n");
}
void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
        printf( "Enter the data values to be placed in a
node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

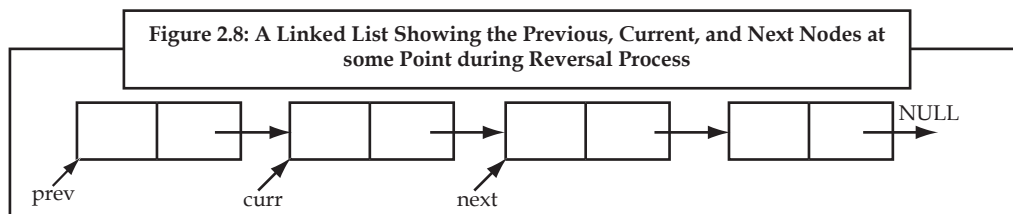
This program appends a new node to the existing list (that is, it inserts a new node in the existing list at the end), and it makes the link field of the newly inserted node point to the start or first node of the list. This ensures that the link field of the last node always points to the starting node of the list.

2.9 Sorting and Reversing a Linked List

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved, as shown in Figure 2.7.



To reverse a list, we maintain a pointer each to the previous and the next node, then we make the link field of the current node point to the previous, make the previous equal to the current, and the current equal to the next, as shown in Figure 2.8.



Therefore, the code needed to reverse the list is:

```
Prev = NULL;
While (curr != NULL)
{
    Next = curr->link;
    Curr -> link = prev;
    Prev = curr;
    Curr = next;
}
```

2.10 Merging two Sorted Lists

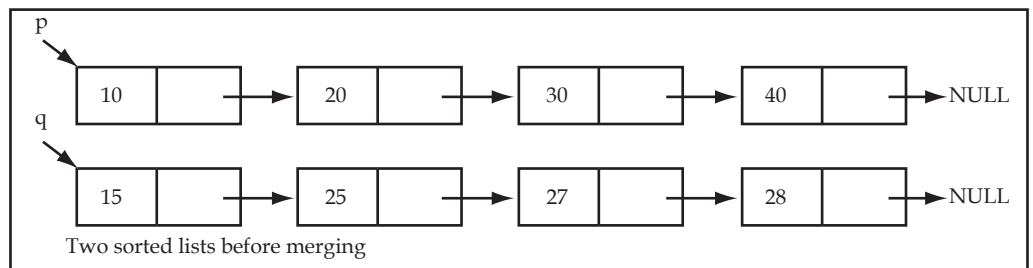
Merging of two sorted lists involves traversing the given lists and comparing the data values stored in the nodes in the process of traversing.

Notes

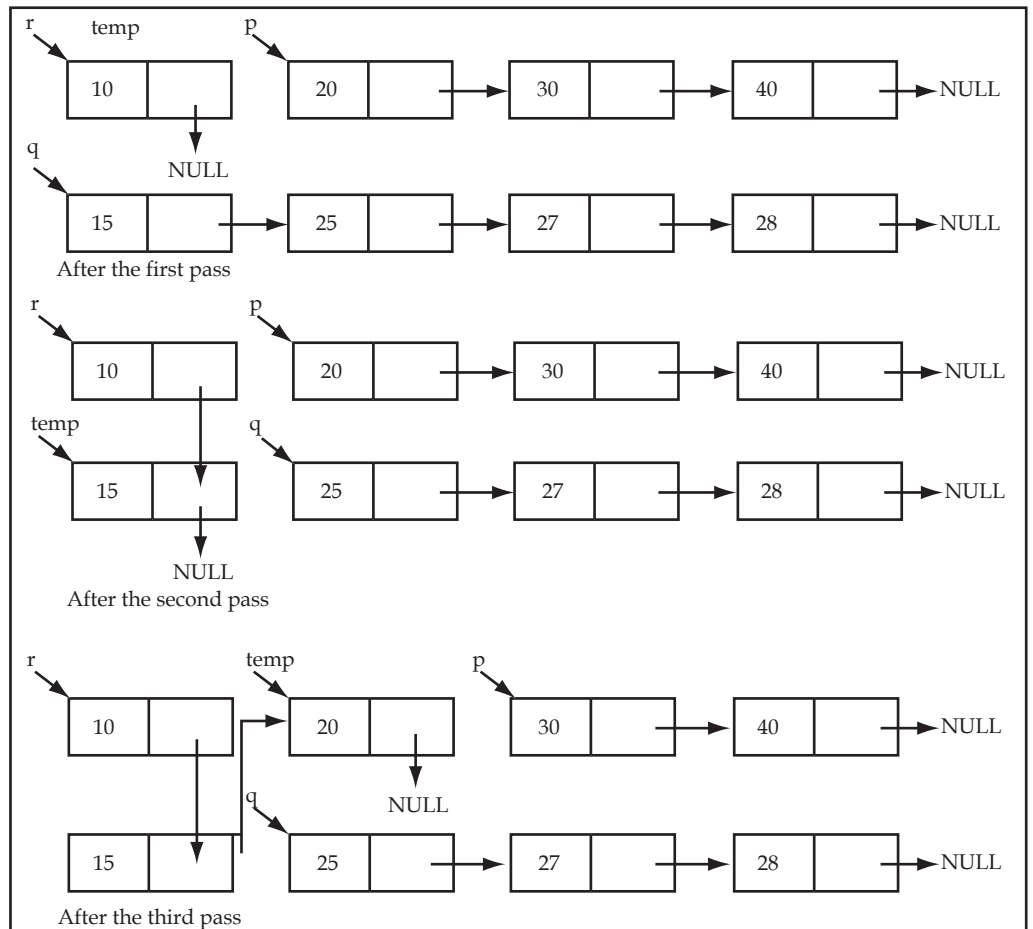
If p and q are the pointers to the sorted lists to be merged, then we compare the data value stored in the first node of the list pointed to by p with the data value stored in the first node of the list pointed to by q. And, if the data value in the first node of the list pointed to by p is less than the data value in the first node of the list pointed to by q, make the first node of the resultant/merged list to be the first node of the list pointed to by p, and advance the pointer p to make it point to the next node in the same list.

If the data value in the first node of the list pointed to by p is greater than the data value in the first node of the list pointed to by q, make the first node of the resultant/merged list to be the first node of the list pointed to by q, and advance the pointer q to make it point to the next node in the same list.

Repeat this procedure until either p or q becomes NULL. When one of the two lists becomes empty, append the remaining nodes in the non-empty list to the resultant list.

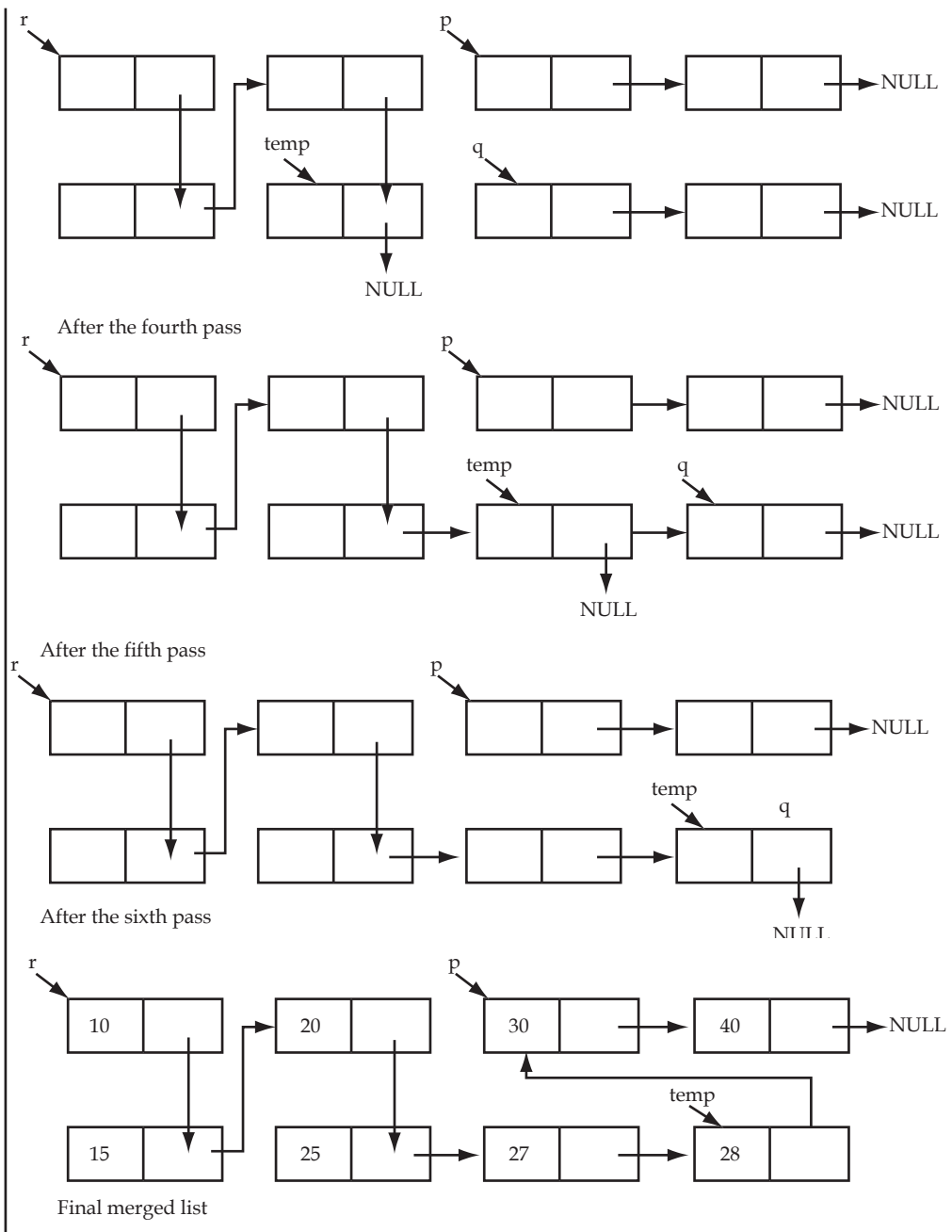


If the above lists given as input, what would be the output of the program after each pass?



Contd...

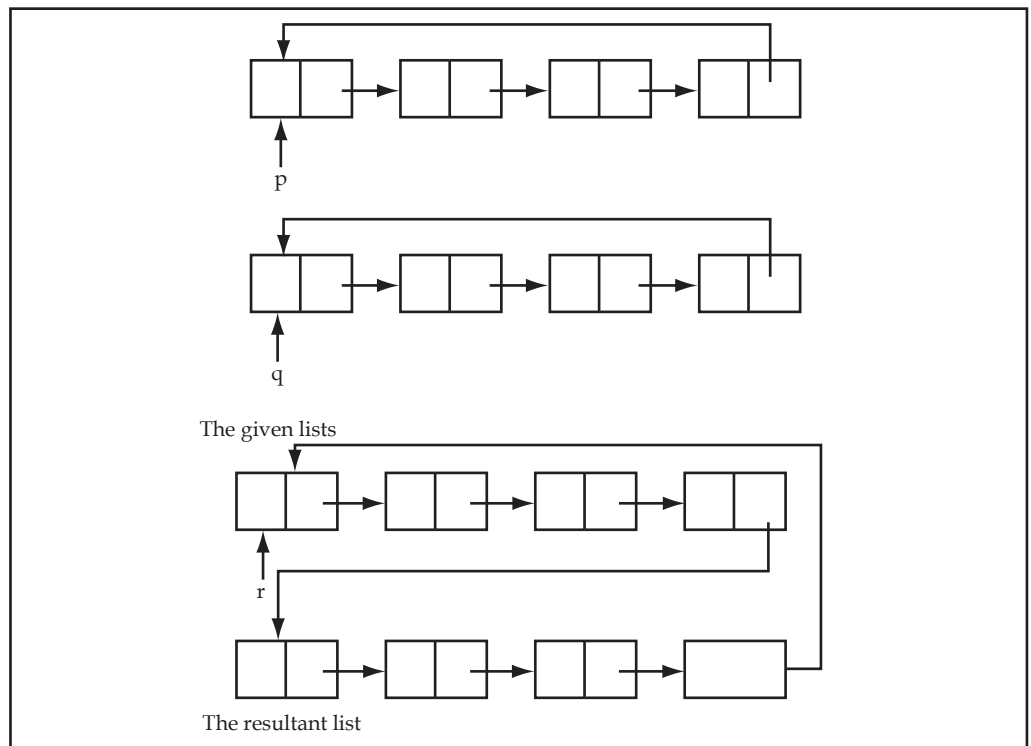
Notes



2.11 Merging of two Circular Lists

In order to merge or concatenate the two non-empty circular lists pointed to by *p* and *q*, it is required to make the start of the resultant list *p*. Then the list pointed to by *p* is required to be traversed until its end, and the link field of the last node must become the pointer *q*. After that, the list pointed to by *q* is required to be traversed until its end, and the link field of the last node is required to be made *p*.

Notes



You can merge two lists into one list. The following program merges two circular lists.



Lab Exercise Program:

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
{
struct node *temp;
/* if the existing list is empty then insert a new node as the
starting node */
if (p==NULL)
{
p=(struct node *)malloc(sizeof(struct node)); /* creates new node
data value passes
as parameter */
if (p==NULL)
```

```

    {
printf("Error\n");
        exit(0);
    }
    p-> data = n;
    p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
    }
else
    {
        temp = p;
/* traverses the existing list to get the pointer to the last node of
it */
while (temp-> link != p)
    temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
    data value passes as
    parameter and puts its
    address in the link field
    of last node of the
    existing list*/
    if(temp -> link == NULL)
    {
printf("Error\n");
        exit(0);
    }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = p;
    }
return (p);
}
void printlist ( struct node *p )
{
struct node *temp;
temp = p;
printf("The data values in the list are\n");
    if(p!= NULL)
    {

```

Notes

```
do
    {
        printf("%d\t",temp->data);
        temp=temp->link;
    } while (temp!= p);
}
else
    printf("The list is empty\n");
}
struct node *merge(struct node *p, struct node *q)
{
    struct node *temp=NULL;
    struct node *r=NULL;
    r = p;
    temp = p;
    while(temp->link != p)
        temp = temp->link;
    temp->link = q;
    temp = q;
    while( temp->link != q)
        temp = temp->link;
    temp->link = r;
    return(r);
}
void main()
{
    int n;
    int x;
    struct node *start1=NULL ;
    struct node *start2=NULL;
    struct node *start3=NULL;
    /* this will create the first circular list nodes*/
    printf("Enter the number of nodes in the first list \n");
    scanf("%d",&n);
    while ( n-- > 0 )
    {
        printf( "Enter the data value to be placed in a node\n");
        scanf("%d",&x);
        start1 = insert ( start1, x );
    }
    printf("The first list is\n");
```



```

printlist ( start1 );
/* this will create the second circular list nodes*/
printf("Enter the number of nodes in the second list \n");
scanf("%d",&n);
while ( n-- > 0 )
{
printf( "Enter the data value to be placed in a node\n");
scanf("%d",&x);
start2 = insert ( start2, x );
}
printf("The second list is:\n");
printlist ( start2 );
start3 = merge(start1,start2);
printf("The resultant list is:\n");
printlist(start3);
}

```



Task

Write a program to reverse a linked list

2.12 Application of Linked List

Polynomials in general are represented by the following equation:

$$f(x) = c_i x_{e_i} + c_{i-1} x_{e_{i-1}} + \dots + c_1 x_{e_1}$$

where C_i are non zero coefficients of variable x and e_i are exponents such that

$$e_i > e_{i-1} > e_{i-2} > \dots > e_1 > 0.$$



Example:

$$F1(x) = 3x^7 - 7x^5 + 3x^2 + 1$$

$$F2(x) = 5x^8 + 6x^5 + 4x^2 + 13$$

These Polynomials can be added to form another Polynomial

Let $f3(x) = f1(x) + f2(x)$

$$f3(x) = 5x^8 + 3x^7 - x^5 + 7x^2 + 14$$

This is achieved by adding coefficients of variables with same exponent value.

These Polynomials can be maintained using a Linked List. To achieve this each term will be represented by a node and each node should consist of three elements namely coefficients, exponents and a link to the next term (represented by another node).

Struct poly

Coef.	Exp.	Link
-------	------	------

{

float coef;

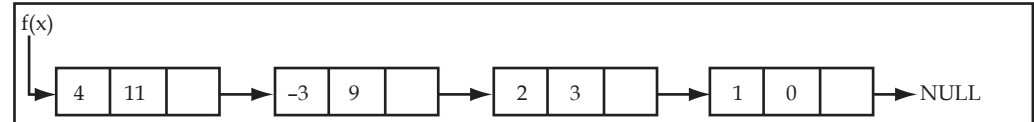
Notes

```

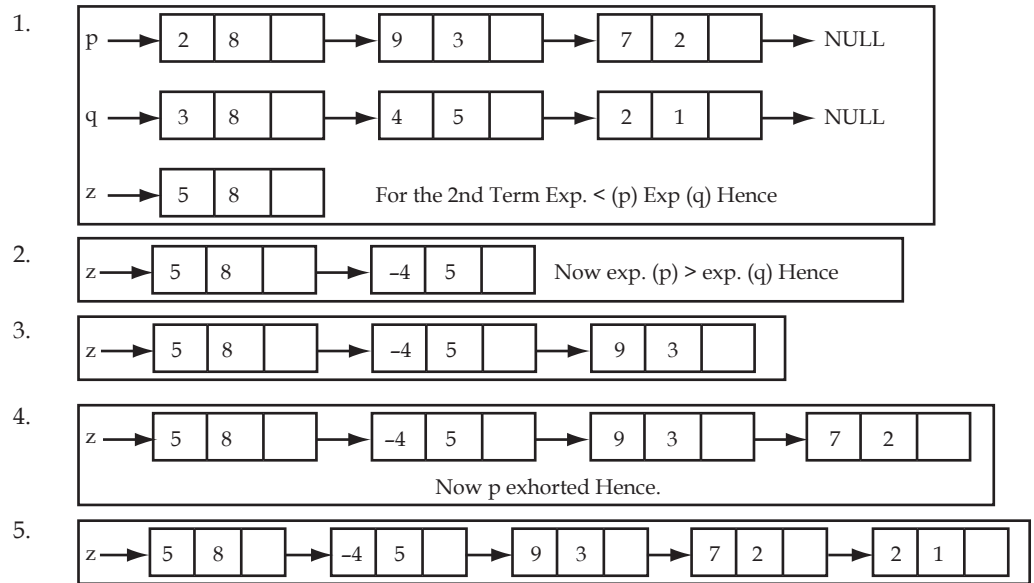
int exp;
struct poly link;
}
    
```

For instance, the polynomial

$f(x) = 4x^{11} - 3x^9 + 2x^3 + 1$ will be stored as:



While maintaining the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. We can use these linked Lists to add two polynomials. To add two polynomials together we examine their terms starting at the nodes pointed to by p and q (two pointers used to move along the terms of two polynomials). If the exponents of two terms are equal, then the coefficients are added and a new term created for the result. If the exponents of the current term in p is less than the exponent of current term of q, then a duplicate of the term q is created and attached to z. The pointers q and z (pointer to result term) are advanced. Similar action is taken if $\text{Exp}(p) > \text{Exp}(q)$.



The function for Adding two polynomials is given below:

```

polyadd (struct poly* p, struct poly*q)
{
    struct poly *z1;
    z= malloc(sizeof(struct poly));
    if (p== NULL && q==NULL) return;
    while (p!=Null && q!=NULL)
    {
        if (p-> exp > q->exp)
        {
    
```

```

        z->exp=p->exp; z1->coef=p->coef;
        p=p-> link;
    }
    if (p->exp<q->exp)
    {
        z->exp=q->exp; z1->coef=q->coef;
    }
    q=q->link;
    if(p-> exp=q->exp)!=0)
    {
        z->exp=p->exp;
        z->coef=p->coef;
        p=p->link;
        q=q->link;
    }
    z->link = malloc(sizeof(struct poly));
    z=z1-> link;
    while (p!=NULL)
    {
        z->exp=p->exp;
        z->coef=p->coef;
        z->link= malloc(sizeof(struct poly));
        z=z->link; p=p->link;
    }
    while (q!=NULL)
    {
        z->exp=q->exp;
        z->coef=q->coef;
        z->link=malloe(sizeof(struct poly));
        z=z->link;
    }
    q=q->link;
}
}

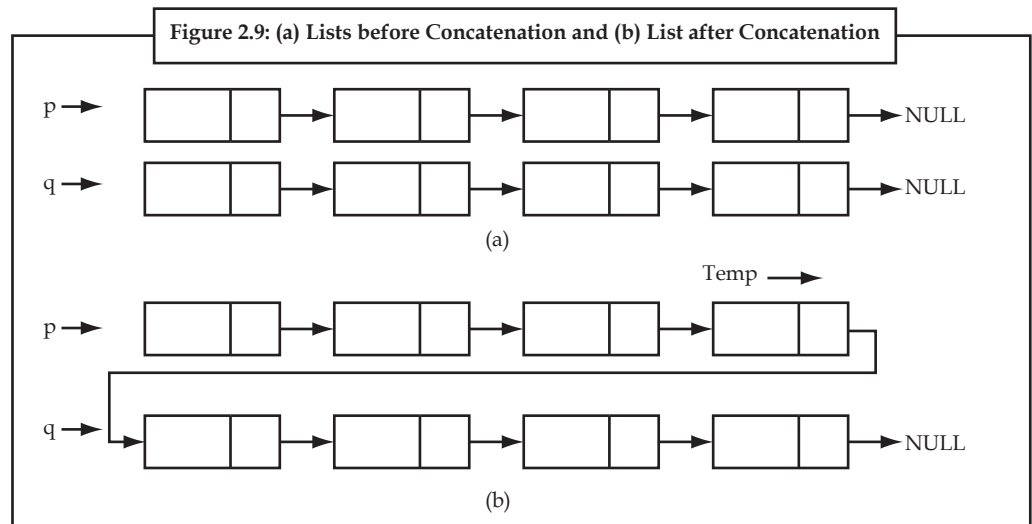
```

Cursor Implementation of Linked Lists

Consider a case where we have two Linked Lists pointed to by two different pointers, say p and q respectively, and we want to concatenate 2nd list at the end of the first list. We can do it by traversing first list till the end and then store the address of first node in the second list, in the link field of last node of first list. Suppose we are traversing first list by pointer temp, then we can concatenate the list by the statement (Figure 2.9)

```
temp -> link = q;
```

Notes



The function to achieve this is given below:

```

Concatenate (struct node *p, struct node *q)
{
    struct node *temp;
    temp = p;
    if (p == NULL) // If first list is NULL then Concatenated
        p = q; // List will be only Second List and will be
    else //pointed by p;
    {
        temp = p;
        while (temp -> link != NULL)
            temp = temp -> link;
        temp -> link = q;
    }
}
    
```

2.13 Summary

- The foremost advantage of linked lists in dynamic storage is flexibility advantages. Overflow is no problem until the computer memory is actually exhausted. Especially when the individual entries are quite large, it may be difficult to determine the overflow amount of contiguous static storage that might be needed for the required arrays while keeping enough free for other needs. With dynamic allocation, there is no need to attempt to make such decisions in advance.
- Changes, especially insertions and deletions, can be made in the middle of a changes linked list more quickly than in the middle of a contiguous list. If the structures are large, then it is much quicker to change the values of a few pointers than to copy the structures themselves from one location to another disadvantages.
- The first drawback of linked lists is that the links themselves take space that might otherwise be needed for additional data. In most systems, a pointer requires the same amount of storage (one word) as does an integer. Thus a list of integers will require double the space in linked storage that it would require in contiguous storage.

2.14 Keywords

Notes

Circular Linked List: A linear linked list in which the last element points to the first element, thus, forming a circle.

Doubly Linked List: A linear linked list in which each element is connected to the two nearest elements through pointers.

Linear List: A one-dimensional list of items.

Linked List: A dynamic list in which the elements are connected by a pointer to another element.

NULL: A constant value that indicates the end of a list.

2.15 Self Assessment

Choose the appropriate answer:

- Pointers permit the referencing of structures in a
 - Normal way
 - Uniform way
 - Common way
 - None
- A recursive data structure is a data structure that has the same form regardless of the
 - Size of the data
 - Shape of the data
 - Origin of the data
 - Form of the data
- The insert function takes a pointer to an existing list as the
 - Second parameter
 - Third parameter
 - First parameter
 - None
- A Circular Linked List has no
 - Beginning and no mid point
 - End and last
 - Beginning and beginning
 - Beginning and no end

Fill in the blanks:

- Memory allocation in Linked Lists is
- Linked list stores two items of information: and
- An function is used to create the list.
- A is a list of elements in which the elements of the list can be placed anywhere in memory.

State whether the following statements are true or false:

- We cannot insert a node after any specific node.
- In the single linked list each node provides information about where the previous node is in the list.
- A linked list is not a recursive data structure.

2.16 Review Questions

1. Write a C program to sort the elements of a linked list.
2. Why are linked list better than arrays? Compare giving examples.
3. Create a linked list and write programs to:
 - (a) Insert a node at the nth position where n is accepted as an input from the keyboard.
 - (b) Delete a node from the nth position where n is accepted as an input from the keyboard.
 - (c) Shift the node at the nth position to the position p where n and p are accepted as inputs from the keyboard.
4. Write a program for building of an element of the circular linked list.
5. Describe the method of inserting a node using recursive program.
6. Write a C program to merge two given lists A and B to form C in the following manner:

The first element of C is the first element of A and the second element of C is the first element of B. The second elements of A and B become the third and fourth elements of C, and so on. If either A or B gets exhausted, the remaining elements of the other are to be copied to C.
7. Write a C program to transform a circular list into a chain.
8. Describe ADT of singly linked lists.
9. Write remove for the (second) implementation of simply linked lists that remembers the last-used position.
10. Prepare a collection of files containing the declarations for a contiguous list and all the functions for list processing.

Answers: Self Assessment

- | | |
|--|----------------|
| 1. (b) | 2. (a) |
| 3. (c) | 4. (d) |
| 5. Dynamic | |
| 6. An element of the list, A link or address of another node | |
| 7. Insert | 8. Linked list |
| 9. False | 10. False |
| 11. False | |

2.17 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Data Structures and Algorithms; Shi-Kuo Chang; World Scientific.

Data Structures and Efficient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.

Notes

Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi

Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Addition. Addison-Wesley publishing

RG Dromey, How to Solve it by Computer, Cambridge University Press.

Shi-kuo Chang, Data Structures and Algorithms, World Scientific

Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.

Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms. Prentice-Hall of India Pvt. Limited, New Delhi

Timothy A. Budd, Classic Data Structures in C++, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 3: Stacks

CONTENTS

Objectives

Introduction

3.1 Stack Model

3.2 Implementation of Stacks

3.2.1 Array-based Implementation

3.2.2 Pointer-based Implementation

3.3 Applications of Stacks

3.3.1 Maze Problem

3.3.2 Simulating Recursive Function using Stack

3.3.3 Simulation of Factorial

3.4 Summary

3.5 Keywords

3.6 Self Assessment

3.7 Review Questions

3.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the stack model
- Explain the implementation and applications of stacks

Introduction

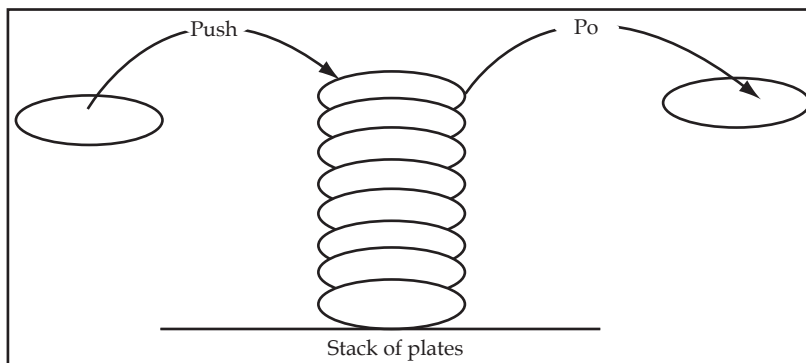
Stack is another linear data structure having a very interesting property. Unlike arrays and link lists, an element can be inserted and deleted not at any arbitrary position but only at one end. Thus, one end of a stack is sealed for insertion and deletion while the other end allows both the operations.

3.1 Stack Model

A stack is a linear data structure in as much as its member elements are ordered as 1st, 2nd,.... and last. However, an element can be inserted in and deleted from only one end. The other end remains sealed. This open end to which elements can be inserted and deleted from is called stack top or top of the stack. Consequently, the elements are removed from a stack in the reverse order of insertion. A stack is said to possess LIFO (Last In First Out) property. A data structure has LIFO property if the element that can be retrieved first is the one that was inserted last.

We come across several such structures in our day-to-day life. Consider a pile of plates as shown below.

Notes

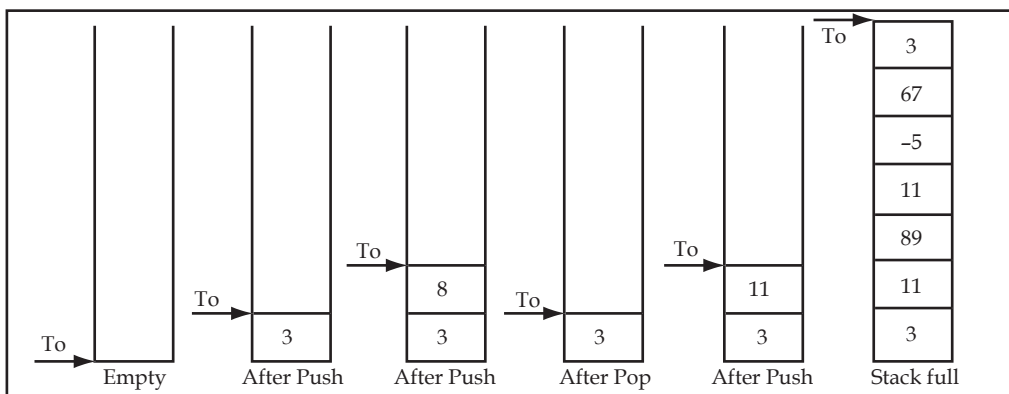


Clearly, an element can be inserted in a stack only on the top and can be retrieved only from the top.

The following are the operations on a stack S:

- Push(S, e) : This inserts an element e into the stack S.
- pop(S) → e : This deletes an element from the stack S and stores it in e.
- Isempty(S) → true/false : This checks if the stack S is empty.
- Istfull(S) → true/false : This checks if the stack S is full or not.

A point to remember is that the above three operations are the only ones permitted with which a stack could be operated on. Consider the following stack of integers to understand the stack operations.



Stack operations in pseudo-code:

STACK-EMPTY(S)

if top[S] = 0

return true

else return false

PUSH(S, x)

top[S] ← top[S] + 1

S[top[S]] ← x

POP(S)

Notes

```

if STACK-EMPTY(S)
then error "underflow"
else top[S] <- top[S] - 1
return S[top[S] + 1]

```

The running time for the operations is $O(1)$ (constant time).

3.2 Implementation of Stacks

There are two basic methods for the implementation of stacks – one where the memory is used statically and the other where the memory is used dynamically.

3.2.1 Array-based Implementation

In this scheme, an array of certain maximum size is allocated memory statically (i.e. once for all). The stack and its operations are implemented using this array.

The following pseudo-code shows the array-based implementation of a stack. In this, the elements of the stack are of type T.

```

struct stk
{ T array[max_size];
/* max_size is the maximum size */
int top = -1;
/* stack top initially given value -1 */
} stack;
void push(T e)
/*inserts an element e into the stack s*/
{
    if (stack.top == max_size)
        printf("Stack is full-insertion not possible");
    else
    {
        stack.top = stack.top + 1;
        stack.array[stack.top] = e;
    }
}
T pop()
/*Returns the top element from the stack */
{
    T x;
    if(stack.top == -1)
        printf("Stack is empty");
    else
    {
        x = stack.array[stack.top];
        stack.top = stack.top - 1;
    }
}

```

```

        return(x);
    }
}
boolean empty()
/* checks if the stack is empty */
{
    boolean empty = false;
    if(stack.top == -1)
        empty = true else empty = false;
return(empty);
}
void initialise()
/* This procedure initializes the stack s */
{
    stack.top = -1;
}

```

The above implementation strategy is easy and fast since it does not have run-time overheads. At the same time it is not flexible since it cannot handle a situation when the number of elements exceeds `max_size`. Also, let us say, if `max_size` is derived statically to 100 and a stack actually has only 10 elements, then memory space for the rest of the 90 elements would be wasted.



Task

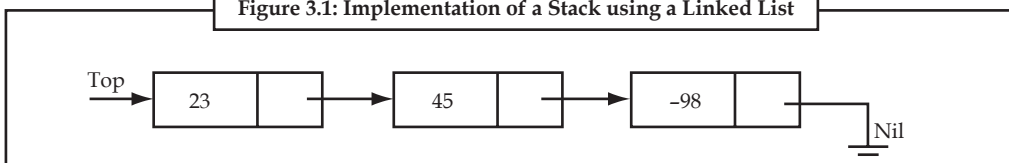
Write a syntax to declare array in a program.

3.2.2 Pointer-based Implementation

Here the memory is used dynamically. For every push operation, the memory space for one element is allocated at run-time and the element is inserted into the stack. For every pop operation, the memory space for the deleted element is de-allocated and returned to the free space pool. Hence the shortcomings of the array-based implementation are overcome. But since, this allocates memory dynamically, the execution is slowed down.

The stack is implemented as a linked list as shown in Figure 3.1.

Figure 3.1: Implementation of a Stack using a Linked List



The following pseudo-code is for the pointer-based implementation of a stack. Each element of the stack is of type T.

```

struct stk
{
    T element;
}

```

Notes

```
        struct stk *next;
    };
    struct stk *stack;
    void push(struct stk *p, T e)
    {
        struct stk *x;
        x = new(stk);
        x.element = e;
        x.next = NULL;
        p = x;
    }
```

Here the stack full condition is checked by the call to new which would give an error if no memory space could be allocated.

```
T pop(struct stk *p)
{
    struct stk *x;
    if (p == NULL)
        printf("Stack is empty");
    else
    {
        x = p;
        x = x.next;
        return(p.element);
    }
}
boolean empty(struct stk *p)
{
    if (p == NULL)
        return(true);
    else
        return(false);
}
void initialize(struct stk *p)
{
    p = NULL;
}
```

3.3 Applications of Stacks

There are numerous applications of the stack data structure in computer algorithms. It is used to store return information in the case of function/procedure/subroutine calls. Hence, one would find a stack in architecture of any Central Processing Unit (CPU). In this section, we would just illustrate a few of them.

3.3.1 Maze Problem

Notes

In a classical experimentation in psychology, a rat is placed through the door of a large box in which walls are set up to restrict movements in most directions. The rat's movement is observed as it finds out, through the obstacles, a way to escape from the box. A two-dimensional version of this problem is easy to formulate where a rectangular maze is conceived as consisting of a number of unit square rooms such that movement from one square to another is restricted.

An m by n maze can be represented by an array maze of size $[1..m, 1..n]$ where each $\text{maze}[i,j]$ represents a unit room. If $\text{maze}[i,j]$ is 0 the room may be entered into but it cannot be entered if $\text{maze}[i,j]$ is 1. The entrance is at $\text{maze}[1,1]$ and the exit is at $\text{maze}[m,n]$.

Evaluation of Expressions

One of the biggest technical hurdles faced when conceiving the idea of higher level Programming Languages, is to generate machine language instructions, which would properly evaluate any arithmetic expression. A complex assignment statement such as:

$$z \rightarrow x / y ** a + b * c - x * a$$

might have several meanings; even if it were uniquely defined, by the full use of parenthesis.

An expression is made up of operands, operators and delimiters. The above expression has five operands x , y , a , b and c . The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. To fix the order of evaluation we assign to each operator a priority. A set of sample priorities are as follows:

Operator	Priority	Associativity
()	8	Left to Right
^ or **, unary -, unary+, ![not]	7	Right to Left
*, /, %	6	Left to Right
+, -	5	Left to Right
<, <=, >, >=	4	Left to Right
=, !=	3	Left to Right
&&	2	Left to Right
	1	Left to Right

But by using parenthesis we can override these rules and such expressions are always evaluated with the inner most parenthesized expression first.

The above notation of any expression is called Infix Notation (in which operators come in between the operands). The notation is a traditional notation, which needs operator's priorities and associativities. But how can a compiler accept such an expression and produce correct Polish Notation or Prefix form (in which operators come before operands) Polish Notation has several advantages over Infix Notation such as: there is no need for considering priorities while evaluating them, there is no need of introducing parenthesis for maintaining order of execution of operators.

Similarly, Reverse Polish Notation or Postfix Form also has same advantages over Infix Notation, in this notation operators come after the operands.

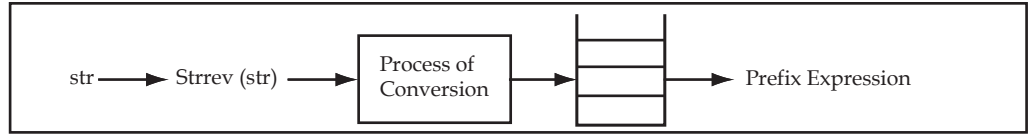
Consider the following examples.

Infix	Prefix	Postfix
$x + y$	$+ x y$	$x y +$
$x + y * z$	$+ x + y z$	$x y z * +$
$x + y - z$	$- + x y z$	$x y z * + -$

Notes

Stacks are frequently used for converting INFIX form into equivalent PREFIX and POSTFIX forms.

The whole logic is same as for Infix to Postfix except, before traversing the string reverse it by `strrev()` function and then process as before, but instead of printing the expression take output in another stack and then print the stack, at last reverses the string.

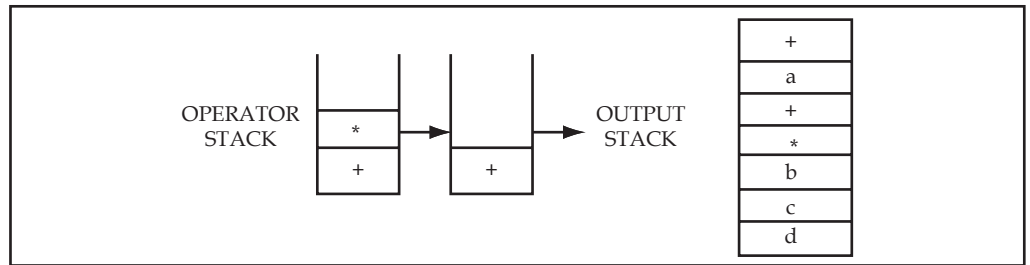


Consider the following example.

Infix expression -> a + b * c + d

After reversing it -> d + c * b + a

Now apply the logic of Infix to Postfix conversion and store the output in another stack.



Now on printing the stack we have, + a + * b c d which is a prefix expression. Here is a C implementation for converting infix into postfix.

```

/* Convert infix expression into corresponding postfix expression */
/* Maximum length of the input expression allowed is 20 characters */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
typedef struct
{
    char data[20]; /* array to hold stack contents */
    int top; /* top of the stack pointer */
}stk;
/* function prototypes */
void initStack(stk *stack);
void get_infix(char infix[]);
void convertToPostfix(char infix[], char postfix[]);
int isOperator(char c);
int precedence(char operator1, char operator2);
int pred_level(char ch);
void push(stk *stack, char value);
  
```

Notes

```

char pop(stk *stack);
char stackTop(stk *stack);
int isEmpty(stk *stack);
int isFull(stk *stack);
void printResult(char infix[], char postfix[]);
/*void print_msg(void);*/
/* program entry point */
int main(void)
{
    char infix[20], postfix[20]="";
    /* convert from infix to postfix main function */
    convertToPostfix(infix, postfix);
    /* display the postfix equivalent */
    infix[strlen(infix)-2] = '\0';
    printResult(infix, postfix);
    return(1);
}

void initStack(stk *stack) /* initialise the stack */
{
    stack->top = -1;          /* stack is initially empty */
}

void get_infix(char infix[]) /* get infix expression from user */
{
    int i;
    printf("Enter infix expression below (max 18 characters excluding spaces) : \n");
    fflush(stdin);
    for ( i=0; i<18; ) /* to read in only 18 characters excluding spaces */
    {
        if ( (infix[i] = getchar()) == '\n' )
        {
            i++; break;
        }
        else if ( !(isspace(infix[i])) )
            i++;
    }
    infix[i] = '\0';
}

void convertToPostfix(char infix[], char postfix[]) /* convert the infix expression
to postfix notation */
{
    int i, length, j=0;

```

Notes

```

char top_ch;
stk stack;
initStack(&stack);          /* initialise stack */
get_infix(infix);          /* get infix expression from user */
length = strlen(infix);
if ( length )              /* if strlen of infix is more than zero */
{
    push(&stack, '(');
    strcat(infix, "(");
    length++;
    for ( i=0; i<length; i++ )
    {
        if ( isdigit(infix[i]) ) /* if current operator in infix is digit */
        {
            postfix[j++] = infix[i];
        }
        else if ( infix[i] == '(' ) /* if current operator in infix is
left parenthesis */
        {
            push(&stack, '(');
        }
        else if ( isOperator(infix[i]) ) /* if current operator is operator */
        {
            while(1)
            {
                top_ch = stackTop(&stack);          /* get tos */
                if ( top_ch == '\0' )                /* no stack left */
                {
                    printf("\nInvalid infix expression\n");
                    exit(1);
                }
                else
                {
                    if ( isOperator(top_ch) )
                    {
                        if ( pred_level(top_ch) >= pred_level(infix[i]) )
                            postfix[j++] = pop(&stack);
                        else
                            break;
                    }
                    else
                }
            }
        }
    }
}

```



```

        break;
    }
}
push(&stack, infix[i]);
}
else if (infix[i] == ')') /* if current operator is right parenthesis */
{
    while (1)
    {
        top_ch = stackTop(&stack); /* get tos */
if ( top_ch == '\0' ) /* no stack left */
        {
            printf("\nInvalid infix expression\n");
/*print_msg();*/
            exit(1);
        }
        else
        {
if ( top_ch != '(' )
            {
                postfix[j++] = top_ch;
                pop(&stack);
            }
            else
            {
                pop(&stack);
                break;
            }
        }
    }
    continue;
}
}
}
postfix[j] = '\0';
}
int isOperator(char c) /* determine if c is an operator */
{
    if ( c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^' )
        return 1;
}

```

Notes

```
        else
            return 0;
    }
int pred_level(char ch)      /* determine precedence level */
{
    if ( ch == '+' || ch == '-' ) return 1;
    else if ( ch == '^' )
        return 3;
    else
        return 2;
}
int precedence(char operator1, char operator2)      /* determine if the
precedence of operator1 is less than, equal to, greater than the precedence of
operator2 */
{
    if ( pred_level(operator1) > pred_level(operator2) )
        return 1;
    else if ( pred_level(operator1) < pred_level(operator2) )
        return -1;
    else
        return 0;
}
void push(stk *stack, char value)      /* push a value on the stack */
{
    if ( !(isFull(stack)) )
    {
        (stack->top)++;
        stack->data[stack->top] = value;
    }
}
char pop(stk *stack)      /* pop a value off the stack */
{
    char ch;
    if ( !(isEmpty(stack)) )
    {
        ch = stack->data[stack->top];
        (stack->top)--;
        return ch;
    }
    else
        return '\0';
}
```

Notes

```

}

char stackTop(stk *stack) /* return the top value of the stack without popping
the stack */
{
    if ( !isEmpty(stack) )
        return stack->data[stack->top];
    else
        return '\0';
}

int isEmpty(stk *stack) /* determine if stack is empty */
{
    if ( stack->top == -1 ) /* empty */
        return 1;
    else /* not empty */
        return 0;
}

int isFull(stk *stack) /* determine if stack is full */
{
    if ( stack->top == 19 ) /* full */
        return 1;
    else /* not full */
        return 0;
}

void printResult(char infix[], char postfix[]) /* display the result postfix
expression */
{
    printf("\n\n"); /*system("cls");*/
    printf("Infix notation : %s\n", infix);
    printf("Postfix notation: %s\n\n", postfix);
}

```

Here are some results of the test run.

Infix	Postfix
(1+2)-(3*4/5)+6	12+34*5/-6+
1-2*3/(4-5)	123*45-/-
(1*2^3)-(4%3-5)^6	123^*43%5-6^-

Evaluating Postfix Expression

An expression may be evaluated by making a left to right scan, stacking operands and evaluating operators using the correct number of operands from the stack and finally placing back the result on to the stack.

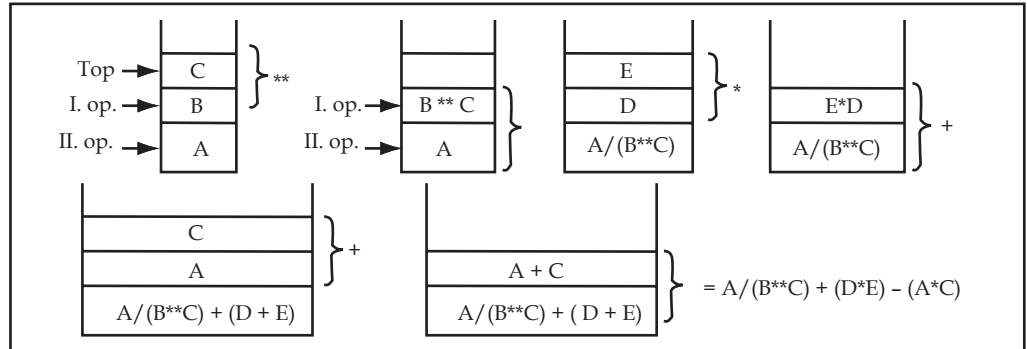
Notes



Example: To evaluate a Postfix Expression:

$ABC^{**}/DE^{*}+AC^{-}$

The following steps will be executed:



Lab Exercise Here is a program that implements the above idea into a program.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXC 80
double eval(charl[]);
double pop(struct stack *);
void push(struct stack *, double);
int empty(struct stack *);
int isdigit(char);
double oper(intn double, double);
void main()
{
char expr[MAXC];
int position = 0;
while((expr[position++] = getchar()) != '\n') ;
expr[--position] = '\0';
printf("%s%s", "the original postfix expression is", expr);
printf("\n%f", eval(expr));
}
struct stack
{
int top;
double data[MAXC];
}
```

```
double eval (char exp[])
{
int c, position;
double opnd1, opnd2, value;
struct stack opndstk;
opndstk.top = -1;
for (position = 0; (c = expr[position]) != '\0'; position++)
if (isdigit(c))
push(&opndstk, (double) (c - '0'));
else {
opnd2 = pop(&opndstk);
opnd1 = pop(&opndstk);
value = oper(c, opnd1, opnd2);
push(&opndstk, value);
}
return(pop(&opndstk));
}

int isdigit(char symb)
{
return (symb >= '0' && symb <= '9');
}

double oper(int symb, double op1, double op2)
{
switch(symb)
{
case '+' : return (op1 + op2);
case '-' : return (op1 - op2);
case '*' : return (op1 * op2);
case '/' : return (op1 / op2);
case '^' : return (op1 ^ op2);
default : printf("%s", "illegal operation");
}
}

double pop(struct stack *s)
{
int x;
if(s.top == -1)
return(0);
else
{
```

Notes

```
x = s.top;
    s.top = x-1;
return (s.data[x]);
}
}
void push(struct stack *s, double e)
{
    if(s.top == MAXC)
        printf("Stack full!");
    else
        {
            s.top = s.top + 1;
            s.data[s.top] = e;
        }
}
int empty(struct stack *p)
{
    if(s.top == -1)
        return 1;
    else
        return 0;
}
```

3.3.2 Simulating Recursive Function using Stack

A recursive solution to a problem is often more expensive than a non-recursive solution, both in terms of time and space. Frequently, this expense is a small price to pay for the logical simplicity and self-documentation of the recursive solution. However, in a production program (such as a compiler, for example) that may be run thousands of times, the recurrent expense is a heavy burden on the system's limited resources.

Thus, a program may be designed to incorporate a recursive solution in order to reduce the expense of design and certification, and then carefully converted to a non-recursive version to be put into actual day-to-day use. As we shall see, in performing such a conversion it is often possible to identify parts of the implementation of recursion that are superfluous in a particular application and thereby significantly reduce the amount of work that the program must perform.

Suppose that we have the statement

route(x); where route is defined as a function by the header

route(a); x is referred to as an argument (of the calling function), and a is referred to as a parameter (of the called function).

What happens when a function is called? The action of calling a function may be divided into three parts:

1. Passing Arguments
2. Allocating and initializing local variables
3. Transferring control to the function.

Let us examine each of these three steps in turn.

1. **Passing arguments:** For a parameter in C, a copy of the argument is made locally within the function, and any changes to the parameter are made to that local copy. The effect to this scheme is that the original input argument cannot be altered. In this method, storage for the argument is allocated within the data area of the function.
2. **Allocating and initializing local variables:** After arguments have been passed, the local variables of the function are allocated. These local variables include all those declared directly in the function and any temporaries that must be created during the course of execution.
3. **Transferring control to the function:** At this point control may still not be passed to the function because provision has not yet been made for saving the return address. If a function is given control, it must eventually restore control to the calling routine by means of a branch. However, it cannot execute that branch unless it knows the location to which it must return. Since this location is within the calling routine and not within the function, the only way that the function can know this address is to have it passed as an argument. This is exactly what happens. Aside from the explicit arguments specified by the programmer, there is also a set of implicit arguments that contain information necessary for the function to execute and return correctly. Chief among these implicit arguments is the return address. The function stores this address within its own data area. When it is ready to return control to the calling program, the function retrieves the return address and branches to that location. Once the arguments and the return address have been passed, control may be transferred to the function, since everything required has been done to ensure that the function can operate on the appropriate data and then return to the calling routine safely.

Return from a Function

When a function returns, three actions are performed. First, the return address is retrieved and stored in a safe location. Second, the function's data area is freed. This data area contains all local variables (including local copies of arguments), temporaries, and the return address. Finally, a branch is taken to the return address, which had been previously saved. This restores control to the calling routine at the point immediately following the instruction that initiated the call. In addition, if the function returns a value, that value is placed in a secure location from which the calling program may retrieve it. Usually this location is a hardware register that is set-aside for this purpose.



Task

Discuss maze problem.

3.3.3 Simulation of Factorial

Let us look at the factorial function as a recursion example. We present the code for that function, including temporary variables explicitly and omitting the test for negative input, as follows:

```
int fact(int n)
{
    int x, y;
    if (n == 0) return(1);
    x = n-1;
    y = fact(x);
```

Notes

```
return (n * y);
}
```

How are we to define the data area for this function? It must contain the parameter *n* and the local variables *x* and *y*. As we shall see, no temporary variables are needed. The data area must also contain a return address. In this case, there are two possible points to which we might want to return: the assignment of *fact(x)* to *y*, and the main program that called *fact*. Suppose that we had two labels and that we let the label *label2* be the label of a section of code.

Label 2: *y* = result;

within the simulating program. Let the label 1 be the label of a statement

Label 1: return(result);

This reflects a convention that the variable to result contains the value to be returned by an invocation of the *fact* function. The return address will be stored as an integer *i* (equal to either 1 or 2). To effect a return from a recursive call the statement

```
switch(i) {
case 1: goto label1;
case 2: goto label2;
}
```

is executed. Thus, if *i* = 1, a return is executed to the main program that called *fact*, and if *i* = 2, a return is simulated to the assignment of the returned value to the variable *y* in the previous execution of *fact*.

Proving Correctness of Parenthesis in an Expression



Lab Exercise This program takes one mathematical expression and checks whether the opening and closing parenthesis match or not.

```
#define LEFTBRACKET '{'
#define RIGHTBRACKET '}'
#include <stdio.h>
main()
{
char t, *p, expr[100];
printf("Enter an expression :");
scanf("%s", expr);
p=expr;
while(p != '\0')
{
if( *p == LEFTBRACKET) push(p);
if(*p == RIGHTBRACKET) t=pop();
if(t < 0) (printf("ERROR:"));
p++;
}
}
```



```

if(Isempty()) print("statement is parenthetically balanced");
else print("statement is parenthetically unbalanced");
}

```

Notes

*Lab Exercise*

1. The program given below implements the stack data structure using an array. In this program the elements are pushed into array `stack[]` through `push()` function. The parameters passed to `push()` are the base address of the array, the position in the stack at which the element is to be placed and the element itself. Care is taken by the `push()` function that the user does not try to place the element beyond the bounds of the stack. This is done by checking the value stored in `pos`. `pop()` function pops out the last element stored in the `stack[]`, because, `pos` holds the position which has the last element in the stack.

```

/* To pop and push items in a stack */
#define MAX 10
void push ( int ) ;
int pop( ) ;
int stack[MAX] ;
int pos ;
void main( )
{
int n ;
clrscr( ) ;
pos = -1 ; /* stack is empty */
push ( 10 ) ;
push ( 20 ) ;
push ( 30 ) ;
push ( 40 ) ;
n = pop( ) ;
printf ( "\nitem popped out is : %d", n ) ;
n = pop( ) ;
printf ( "\nitem popped out is : %d", n ) ;
}
/* pushes item on the stack */
void push ( int data )
{
if ( pos == MAX - 1 )
printf ( "\nStack is full" ) ;
else
{
pos++ ;
stack[ pos ] = data ;
}
}

```

Notes

```

}
/* pops off the items from the stack */
int pop( )
{
int data ;
if ( pos == -1 )
{
printf ( "\nStack is empty" ) ;
return ( -1 ) ;
}
else
{
data = stack[ pos ] ;
pos-- ;
return ( data ) ;
}
}

```

2. In this program stack is implemented and maintained using linked list. This implementation is more sophisticated compared to the one that uses an array, the added advantage being we can push as many elements as we want. A new node is created by push() (using malloc()) every time an element is pushed in the stack. Each node in the linked list contains two members, data holding the data and link holding the address of the next node. The end of the stack is identified by the node holding NULL in its link part. The pop() function pops out the last element inserted in the stack and frees the memory allocated to hold it. stack_display() displays all the elements that stack holds. count() counts and returns the number of elements present in the stack.

```

#include "alloc.h"
struct node
{
int data ;
struct node *link ;
} ;
push ( struct node **, int ) ;
pop ( struct node ** ) ;
main( )
{
struct node *top ; /* top will always point to top of a stack */
int item;
top = NULL ; /* empty stack */
push ( &top, 11 ) ;
push ( &top, 12 ) ;
push ( &top, 13 ) ;
push ( &top, 14 ) ;
push ( &top, 15 ) ;

```

Notes

```
push ( &top, 16 ) ;
push ( &top, 17 ) ;
clrscr ( ) ;
stack_display ( top ) ;
printf ( "No. of items in stack = %d" , count ( top ) ) ;
printf ( "\nItems extracted from stack : " ) ;
item = pop ( &top ) ;
printf ( "%d ", item ) ;
item = pop ( &top ) ;
printf ( "%d ", item ) ;
item = pop ( &top ) ;
printf ( "%d ", item ) ;
stack_display ( top ) ;
printf ( "No. of items in stack = %d" , count ( top ) ) ;
}
/* adds a new element on the top of stack */
push( struct node **s, int item )
{
struct node *q ;
q = malloc ( sizeof ( struct node ) ) ;
q -> data = item ;
q -> link = *s ;
*s = q ;
}
/* removes an element from top of stack */
pop ( struct node **s )
{
int item ;
struct node *q ;
/* if stack is empty */
if ( *s == NULL )
printf ( " stack is empty" ) ;
else
{
q = *s ;
item = q -> data ;
*s = q -> link ;
free ( q ) ;
return ( item ) ;
}
}
/* displays whole of the stack */
stack_display ( struct node *q )
```

Notes

```
{
printf ( "\n" ) ;
/* traverse the entire linked list */
while ( q != NULL )
{
printf ( "%2d ", q -> data ) ;
q = q -> link ;
}
printf ( "\n" ) ;
}
/* counts the number of nodes present in the linked list representing
a stack */
count ( struct node * q )
{
int c = 0 ;
/* traverse the entire linked list */
while ( q != NULL )
{
q = q -> link ;
c++ ;
}
return c ;
}
```

3.4 Summary

- Stack is a linear data structure having a very interesting property that an element can be inserted and deleted not at any arbitrary position but only at one end. One end of a stack remains sealed for insertion and deletion while the other end allows both the operations.
- A stack possesses LIFO (Last In First Out) property. There are two basic methods for the implementation of stacks - one where the memory is used statically and the other where the memory is used dynamically.
- Push is a stack operation in which an element is added to a stack.
- Pop is a stack operation that removes an element from the stack.
- Stack is used to store return information in the case of function/procedure/subroutine calls. Hence, one would find a stack in architecture of any Central Processing Unit (CPU). In infix notation operators come in between the operands. An expression can be evaluated using stack data structure.

3.5 Keywords

Infix: Notation of an arithmetic expression in which operators come in between their operands.

LIFO: (Last In First Out) The property of a list such as stack in which the element which can be retrieved is the last element to enter it.

Pop: Stack operation retrieves a value from the stack.

Postfix: Notation of an arithmetic expression in which operators come after their operands.

Prefix: Notation of an arithmetic expression in which operators come before their operands.

Push: Stack operation which puts a value on the stack.

Stack: A linear data structure where insertion and deletion of elements can take place only at one end.

3.6 Self Assessment

Choose the appropriate answers:

- Basic method of implementation of stack are:
 - Where the memory is used statically
 - Where the memory is used dynamically
 - Both of the above
 - None of the above
- A stack operation that removes an element from the stack
 - Pop
 - Push
 - LIFO
 - FIFO
- The infix of a particular element is $x + y$ what is the prefix of this
 - $x + y +$
 - $+xy$
 - $xy +$
 - None of the above
- The hardware component known as an input device is:
 - RAM
 - Hard-disk
 - Monitor
 - Processor
 - Keyboard
- The hardware component known as an output device is:
 - Keyboard
 - Monitor
 - RAM
 - Hard-disk
 - Processor
- CPU stands for
 - Control processing unit
 - Central processing unit
 - Central programming unit
 - None of these

Fill in the blanks:

- Stack is said to possessproperty.
- is used to implement stacks where the memory is used statically.
- Pointers help in implementation of stacks.
- In the operators come before operands.

3.7 Review Questions

1. Write an algorithm to reverse an input string of characters using a stack.
2. Explain how function calls may be implemented using stacks for return values.
3. What are the advantages of implementing a stack using dynamic memory allocation method?
4. Trace the execution of infix-to-postfix algorithm on the following expression.

$$3 + (23 * 9 / 3 - 67 - (2 * 7) / 9)$$

5. One way to determine if a string of characters is a palindrome is to use one stack and one queue and to apply the following algorithm strategy:
 - (a) Put the input string on the stack and the queue simultaneously.
 - (b) Thus, popping the string from the stack is equivalent to reading it backwards while deleting the string from the queue is reading it forwards.
6. Write a C program to implement a stack of characters.
7. Explain why we cannot use the following implementation for the method push in our linked Stack.

```

Error_code Stack :: push(Stack_entry item)
{
    Node new_top(item, top_node);
    top_node = new_top;
    return success;
}
    
```

8. What is wrong with the following attempt to use the copy constructor to implement the overloaded assignment operator for a linked Stack?

```

void Stack :: operator = (const Stack &original)
{
    Stack new_copy(original);
    top_node = new_copy.top_node;
}
    
```

How can we modify this code to give a correct implementation?

Answers: Self Assessment

- | | |
|-----------------------------|---------------------|
| 1. (c) | 2. (a) |
| 3. (b) | 4. (e) |
| 5. (b) | 6. (b) |
| 7. LIFO (Last In First Out) | 8. Array |
| 9. Dynamic | 10. Polish Notation |

3.8 Further Readings

Notes



Books

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.

Data Structures and Algorithms; Shi-Kuo Chang; World Scientific.

Data Structures and Efficient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.

Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi

Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Edition. Addison-Wesley publishing

RG Dromey, How to Solve it by Computer, Cambridge University Press.

Shi-kuo Chang, Data Structures and Algorithms, World Scientific

Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.

Thomas H. Cormen, Charles E. Leiserson & Ronald L. Rivest: Introduction to Algorithms. Prentice-Hall of India Pvt. Limited, New Delhi

Timothy A. Budd, Classic Data Structures in C++, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 4: Queues

CONTENTS

Objectives

Introduction

4.1 Queue Model

4.2 Array Implementation

4.2.1 Array-based Implementation

4.2.2 Pointer-based Implementation

4.3 Applications of Queues

4.4 Summary

4.5 Keywords

4.6 Self Assessment

4.7 Review Questions

4.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe queue model
- Know array implementation
- Describe various applications of queues

Introduction

The queue abstract data type is also a widely used one with applications very common in real life. An example comes from the operating system software where the scheduler picks up the next process to be executed on the system from a queue data structure. In this unit, we would study the various properties of queues, their operations and implementation strategies.

4.1 Queue Model

A queue is a list of elements in which insertions are made at one end-called the rear and deletions are made from the other end-called the front. This means, in particular, that elements are removed from a queue in the same order as that in which they were inserted into the queue. Thus, a queue exhibits the FIFO (First In First Out) property.

The following are the allowed operations on a queue Q:

1. *insert(Q, e)*: This inserts an element e into the queue Q.
2. *delete(Q) → e*: This deletes an element from the queue Q and stores it into e.
3. *empty(Q) → true/false*: This checks whether a queue is empty or not.

Queue operations in pseudo-code:

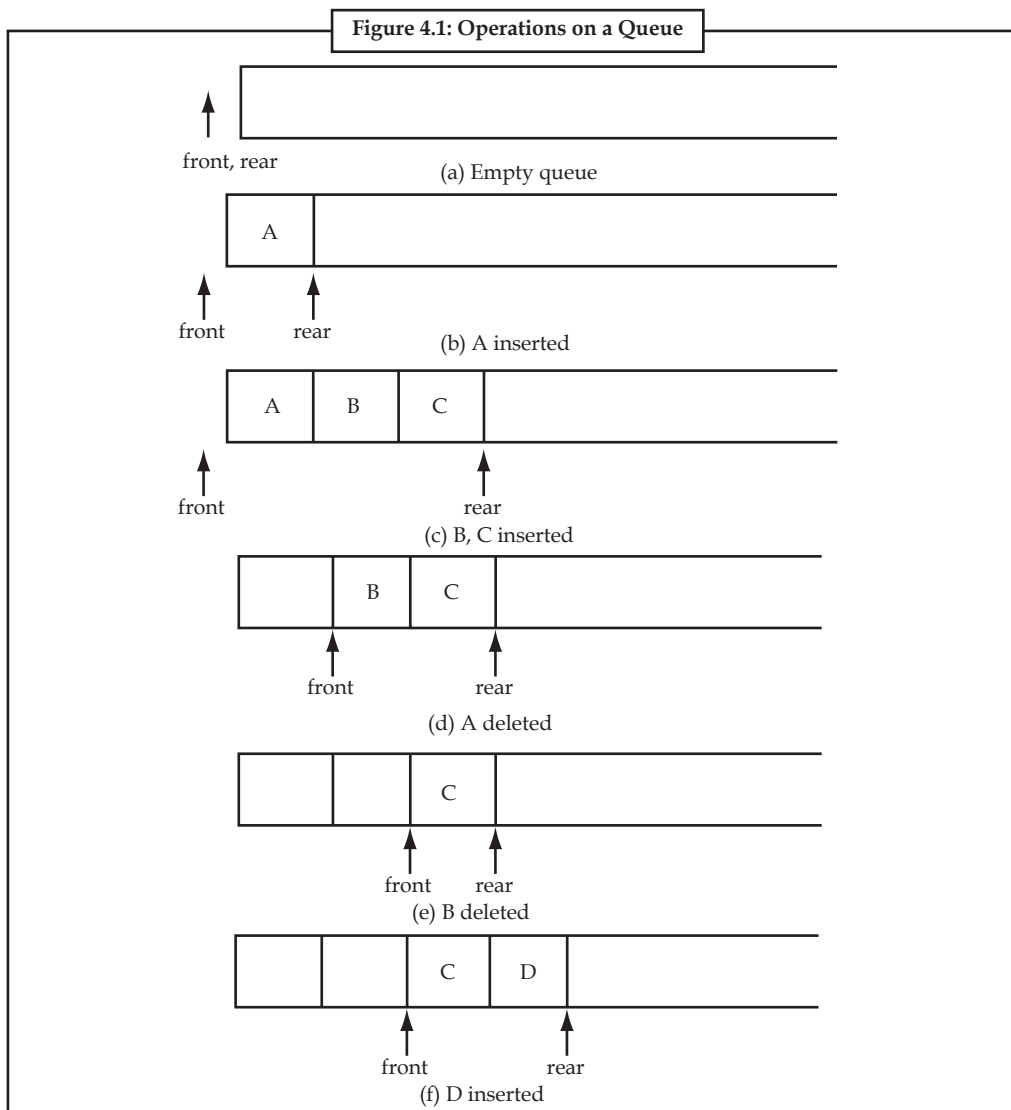
```

ENQUEUE(Q, x)
Q[tail[Q]] <- x
if tail[Q] = length[Q]
then tail[Q] <- 1
else tail[Q] <- tail[Q] + 1
DEQUEUE(Q)
x <- Q[head[Q]]
if head[Q] = length[Q]
then head[Q] <- 1
else head[Q] <- head[Q] + 1
return x

```

These are also $O(1)$ time operations.

Figure 4.1 shows the operations on a queue.



4.2 Array Implementation

Here too, there are two possible implementation strategies – one where the memory is used statically and the other where memory is used dynamically.

4.2.1 Array-based Implementation

To represent a queue we require a one-dimensional array of some maximum size say n to hold the data items and two other variables `front` and `rear` to point to the beginning and the end of the queue. Hence a queue data type may be defined in C as follows:

```
struct queue
{
    T data[n];
    int front, rear;
}
```

During the initialization of a queue q , its `front` and `rear` are made to 0. At each insertion to the queue, which takes place at the rear end, '`rear`' is incremented by 1. At each deletion, '`front`' is incremented by 1. The following procedures show the implementation of the queue operations:

```
void insert(queue q[], T x)
/* This function inserts an element x into the queue q */
{
    if(q.rear == n) printf("Queue is full!");
    else
    {
        q.rear = q.rear + 1;
        q.data[q.rear] = x;
    }
}

void delete(queue q[], T x)
/*deletes an element from the queue and stores in x*/
{
    if(q.front == q.rear)
    {
        printf("queue is empty!")
        q.front = 0;
        q.rear = 0;
    }
    else
    {
        q.front = q.front + 1;
        x = q.data[q.front];
    }
}

boolean empty(queue q[])
```

```

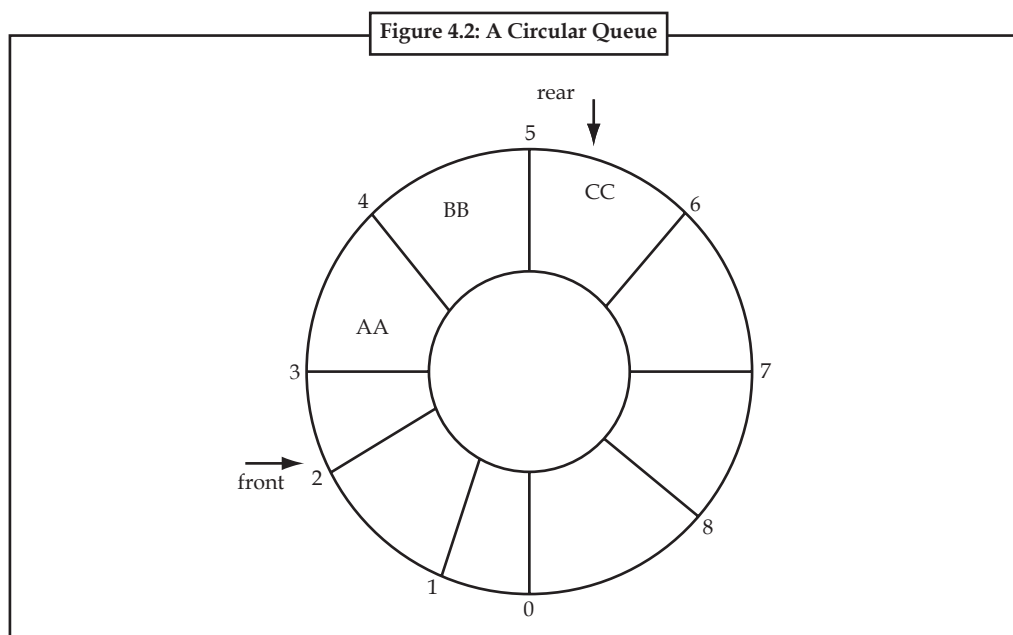
/* This function checks if the queue q is empty or not */
{
    if(q.front == q.rear)
        return(true)
    else
        return(false);
}
void initialize(queue q[])
/* This procedure initializes a queue q */
{
    q.front = 0;
    q.rear = 0;
}

```

An important point to be noticed in the above implementation is that the queue has a tendency to move to the right unless occasionally 'front' catches up with 'rear' and both are reset to 0 again (in the delete procedure). As a result it is possible that a considerable portion of the elements array is free but the queue full signal is on. To get rid of this problem, we may shift the elements array towards left by 1 whenever a deletion is made. Such a shifting is time consuming since it has to be done for each deletion operation.

A more efficient queue representation is obtained by regarding the array of elements of a queue to be circular. For easier algebraic manipulation we consider the array subscript ranging from 0.. n-1; As before, rear will point to the element which is at the end of the queue. The front points to one position anticlockwise to the beginning of the queue. Initially, front = rear = 1 and front = rear only if the queue is empty. Every insertion implies that the rear of the queue increases by 1 except when rear = n - 1. In that case, the rear is made zero if the queue is not full yet.

Figure 4.2 shows a circular queue with 3 elements where n = 9 front=2 and rear=5.



Notes

An implementation of the circular queue is shown below:

```
struct Cqueue
{
    T data[n];
    int front, rear;
}

void insert(queue q[], T x)
/* This procedure inserts an element x into the queue q */
{
    q.rear = (q.rear + 1) % n;
    if(q.rear == q.front)
        printf("Queue full");
    else
        q.data[q.rear] = x;
}

void delete(queue q, T x)
/* deletes an element & stores it in x */
{
    if(q.front == q.rear)
        printf("queue empty");
    else
    {
        x = q.data[q.front];
        q.front = (q.front + 1) % n;
    }
}

boolean empty(queue q[])
/* This function checks if the queue q is empty or not */
{
    if(q.front == q.rear) return(true);
    else return(false);
}

void initialize(queue q[])
/* This procedure initializes a queue q */
{
    q.front = 0;
    q.rear = 0;
}
```

In this implementation, we are using only n-1 entries of the 'elements' array. To use all the elements, a special flag has to be maintained to distinguish between the queue_full and queue_empty situations.

4.2.2 Pointer-based Implementation

Notes

As in stacks, a queue could be implemented using a linked list.

A queue can be implemented by the following declarations and algorithms:

```

struct queueNode
{
    T item;
    queueNode *next;
};
struct queueNode, *front, *rear;
void insert(queueNode *front, queueNode *rear, T e)
{
    struct queueNode *x;
    x = new(queueNode);
    x->item = e;
    if(front == NULL) front = x;
    else rear->next = x;
    rear = x;
}
void delete(queueNode *front, T e, queueNode *front)
/*This procedure deletes the element from the front of the queue & stores it in
e */
{
    if(front == NULL) printf("Queue is empty");
    else
    {
        e = front->item;
        front = front->next;
        if(front == NULL) rear = NULL;
    }
}
boolean empty(queueNode *q) /* check if queue is empty*/
{
    if(q == NULL) return(true) else return(false);
}
void initialize(queueNode *front, queueNode *rear)
{
    front = NULL; rear = NULL;
}

```



Task

Write a program to implement a circular queue.

4.3 Applications of Queues

One major application of the queue data structure is in the computer simulation of a real-world situation. Queues are also used in many ways by the operating system, the program that schedules and allocates the resources of a computer system. One of these resources is the CPU (Central Processing Unit) itself. If you are working on a multi-user system and you tell the computer to run a particular program, the operating system adds your request to its "job queue". When your request gets to the front of the queue, the program you requested is executed. Similarly, the various users for the system must share the I/O devices (printers, disks etc.). Each device has its own queue of requests to print, read or write to these devices. The following subsection discusses one application of the queues - the priority queue. It is used in time-sharing multi-user systems where programs of high priority are processed first and programs with the same priority form a standard queue.

Priority Queues

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed is defined by the following rules:

An element of higher priority is processed before any element of lower priority. Two elements with the same priority are processed according to the order in which they were inserted into the queue.

We would use a singly linked list to implement the priority queue. Each node of the linked list would have a type definition as follows:

```
struct qElement
{
    T item;
    int priority;
    qElement *next;
} *Pqueue, *front, *rear;
```

The algorithm for the insertion would change now. Insertion would insert the new element at the correct position according to the priority of the element. The elements of the priority queue would be sorted in a non-descending order of the priority with the front of the queue having the element with the highest priority. The deletion procedure need not change since the element at the front is the one with the highest priority and that is the one that should be deleted.

```
void insert(Pqueue *front, Pqueue *rear, T e, int p)
/* this inserts an element having data e and priority p into the priority queue */
/*the insertion maintains the sorted order of the priority queue */
{
    Pqueue *f, *r;
    Pqueue *x;
    int pr;
    {
        x = new(Pqueue);
        x->item = e; x->priority = p;
```

```

    if(front == NULL)
    {
        front = x;
        x->next = NULL;
        rear = x;
    }
    /* x is the first node being added to the priority queue*/
    elseif(front->priority < p)
    {
        x->next = front;
        front = x;
    }
    /* x has the highest priority hence should be at the front*/
    elseif(rear->priority > p)
    {
        x->next = NULL;
        rear->next = x;
        rear = x;
    }
    /* x has the least priority hence should be at the rear*/
    else
    {
        /* x has to be inserted in between according to its priority*/
        f = front;
        pr = f->pri; r = NULL;
        while(pr > p) /* Advance through the queue till the proper position is
reached */
        {
            f = f->next; r = f; pr = f->priority;
        }
        /* f now points to the node before which x has to be inserted and r points
to the node which should be before x*/
        r->next = x; x->next = f;
    }
}

```



Task Write an implementation of the Extended_queue method full. In light of the simplicity of this method in the linked implementation, why is it still important to include it in the linked class Extended_queue?

Notes



Case Study Convert an infix expression to prefix form.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define N 80
typedef enum {FALSE, TRUE} bool;
#include "stack.h"
#include "queue.h"
#define NOPS 7
char operators [] = "()^/*+-";
int priorities[] = {4,4,3,2,2,1,1};
char associates[] = " RLLLL";
char t[N]; char *tptr = t; // this is where prefix will be saved.
int getIndex( char op ) {
    /*
     * returns index of op in operators.
     */
    int i;
    for( i=0; i<NOPS; ++i )
        if( operators[i] == op )
            return i;
    return -1;
}
int getPriority( char op ) {
    /*
     * returns priority of op.
     */
    return priorities[ getIndex(op) ];
}
char getAssociativity( char op ) {
    /*
     * returns associativity of op.
     */
    return associates[ getIndex(op) ];
}
void processOp( char op, queue *q, stack *s ) {
    /*
```

Contd...

Notes

```

* performs processing of op.
*/
switch(op) {
    case ')':
        printf( "\t S pushing )...\n" );
        sPush( s, op );
        break;
    case '(':
        while( !qEmpty(q) ) {
            *tptr++ = qPop(q);
            printf( "\tQ popping %c...\n", *(tptr-1) );
        }
        while( !sEmpty(s) ) {
            char popop = sPop(s);
            printf( "\tS popping %c...\n", popop );
            if( popop == ')' )
                break;
            *tptr++ = popop;
        }
        break;
    default: {
        int priop;    // priority of op.
        char topop;  // operator on stack top.
        int pritop;  // priority of topop.
        char asstop; // associativity of topop.
        while( !sEmpty(s) ) {
            priop = getPriority(op);
            topop = sTop(s);
            pritop = getPriority(topop);
            asstop = getAssociativity(topop);
            if( pritop < priop || (pritop == priop && asstop == 'L')
                || topop == ')' ) // IMP.
                break;
            while( !qEmpty(q) ) {
                *tptr++ = qPop(q);
                printf( "\tQ popping %c...\n", *(tptr-1) );
            }
            *tptr++ = sPop(s);
            printf( "\tS popping %c...\n", *(tptr-1) );
        }
    }
}

```

Contd...

Notes

```

    }
    printf( "\tS pushing %c...\n", op );
    sPush( s, op );
    break;
}
}
}
bool isop( char op ) {
    /*
     * is op an operator?
     */
    return (getIndex(op) != -1);
}
char *in2pre( char *str ) { /*
    * returns valid infix expr in str to prefix.
    */
    char *sptr;
    queue q = {NULL};
    stack s = NULL;
    char *res = (char *)malloc( N*sizeof(char) );
    char *resptr = res;
    tptr = t;
    for( sptr=str+strlen(str)-1; sptr!=str-1; -sptr ) {
        printf( "processing %c tptr-t=%d...\n", *sptr, tptr-t );
        if( isalpha(*sptr) ) // if operand.
            qPush( &q, *sptr );
        else if( isop(*sptr) ) // if valid operator.
            processOp( *sptr, &q, &s );
        else if( isspace(*sptr) ) // if whitespace.
            ;
        else {
            fprintf( stderr, "ERROR:invalid char %c.\n", *sptr );
            return "";
        }
    }
    while( !qEmpty(&q) ) {
        *tptr++ = qPop(&q);
        printf( "\tQ popping %c...\n", *(tptr-1) );
    }
    while( !sEmpty(&s) ) {
        *tptr++ = sPop(&s);
        printf( "\tS popping %c...\n", *(tptr-1) );
    }
}

```

Contd...

Notes

```

*tptr = 0;
printf( "t=%s.\n", t );
for( -tptr; tptr!=t-1; -tptr ) {
    *respstr++ = *tptr;
}
*respstr = 0;
return res;
}
int main() {
    char s[N];

    puts( "enter infix freespaces max 80." );
    gets(s);
    while(*s) {
        puts( in2pre(s) );
        gets(s);
    }
    return 0;
}

```

Explanation

1. In an infix expression, a binary operator separates its operands (a unary operator precedes its operand). In a postfix expression, the operands of an operator precede the operator. In a prefix expression, the operator precedes its operands. Like postfix, a prefix expression is parenthesis-free, that is, any infix expression can be unambiguously written in its prefix equivalent without the need for parentheses.
2. To convert an infix expression to reverse-prefix, it is scanned from right to left. A queue of operands is maintained noting that the order of operands in infix and prefix remains the same. Thus, while scanning the infix expression, whenever an operand is encountered, it is pushed in a queue. If the scanned element is a right parenthesis (')', it is pushed in a stack of operators. If the scanned element is a left parenthesis ('('), the queue of operands is emptied to the prefix output, followed by the popping of all the operators up to, but excluding, a right parenthesis in the operator stack.
3. If the scanned element is an arbitrary operator o, then the stack of operators is checked for operators with a greater priority than o. Such operators are popped and written to the prefix output after emptying the operand queue. The operator o is finally pushed to the stack.
4. When the scanning of the infix expression is complete, first the operand queue, and then the operator stack, are emptied to the prefix output. Any whitespace in the infix input is ignored. Thus the prefix output can be reversed to get the required prefix expression of the infix input.

Question

Write a C program to implement a queue by using an array.

Notes

4.4 Summary

- A queue is a list of elements in which insertions are made at one end – called the rear and deletions are made from the other end - called the front.
- A queue exhibits the FIFO (First In First Out) property. To represent a queue we require a one-dimensional array of some maximum size say n to hold the data items and two other variables front and rear to point to the beginning and the end of the queue.
- As in stacks, a queue could be implemented using a linked list. One major application of the queue data structure is in the computer simulation of a real-world situation.
- Queues are also used in many ways by the operating system, the program that schedules and allocates the resources of a computer system. Each device has its own queue of requests to print, read or write to these devices.
- A priority queue is a collection of elements such that each element has been assigned a priority. An element of higher priority is processed before any element of lower priority.
- Two elements with the same priority are processed according to the order in which they were inserted into the queue.

4.5 Keywords

FIFO: (First In First Out) The property of a linear data structure which ensures that the element retrieved from it is the first element that went into it.

Front: The end of a queue from where elements are retrieved.

Queue: A linear data structure in which the element is inserted at one end while retrieved from another end.

Rear: The end of a queue where new elements are inserted.

4.6 Self Assessment

Choose the appropriate answer:

1. Priority queue is a
 - (a) Collection of elements such that each element has been assigned a priority.
 - (b) Collection of I/O devices
 - (c) Collection of FIFO
 - (d) None of the above
2. Every insertion implies that the rear of the queue increases by 1 except when rear = .
 - (a) $n-2$
 - (b) $n-1$
 - (c) $n-0$
 - (d) $n-3$

Fill in the blanks:

3. During the initialization of a queue q , its front and rear are made to

4. A more efficient queue representation is obtained by regarding the array of elements of a queue to be
5. A queue could be implemented using a
6. Reverse Polish Notation is also called
7. is used in time-sharing multi-user systems where programs of high priority are processed first amid programs with the same priority form a standard queue.
8. A queue exhibits the property.
9. A queue could be implemented using and
10. A is a collection of elements such that each element has been assigned a priority.

4.7 Review Questions

1. Write a segment of code to create a copy of a given queue. Let q1 be the given queue and q2 be the copy. All the elements of q2 must be equal to q1. Do not assume any implementation of a queue.
2. Describe the application of queue.
3. How will you insert and delete an element in queue?
4. Write a C program to implement a double-ended queue, which is a queue in which insertions and deletions may be performed at either end. Use a linked representation.
5. Write the following methods for linked queues:
 - (a) The method empty,
 - (b) The method retrieve,
 - (c) The destructor,
 - (d) The copy constructor,
 - (e) The overloaded assignment operator.
6. Assemble specification and method files, called queue.h and queue.c, for linked queues, suitable for use by an application program.
7. For a linked Extended_queue, the function size requires a loop that moves through the entire queue to count the entries, since the number of entries in the queue is not kept as a separate member in the class. Consider modifying the declaration of a linked Extended_queue to add a count data member to the class. What changes will need to be made to all the other methods of the class? Discuss the advantages and disadvantages of this modification compared to the original implementation.
8. Write an implementation of the Extended_queue method full. In light of the simplicity of this method in the linked implementation, why is it still important to include it in the linked class Extended_queue?
9. Give a formal definition of the term deque, using the definitions given for stack and queue as models. Recall that entries may be added to or deleted from either end of a deque, but nowhere except at its ends.
10. Give two reasons why dynamic memory allocation is valuable.

Notes

Answers: Self Assessment

- | | |
|-----------------------|------------------------------|
| 1. (a) | 2. (b) |
| 3. 0 | 4. circular |
| 5. linked list | 6. Postfix Notation |
| 7. Priority Queue | 8. FIFO (First In First Out) |
| 9. Array, Linked List | 10. Priority queue |

4.8 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.

Data Structures and Algorithms; Shi-Kuo Chang; World Scientific.

Data Structures and Efficient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.

Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi

Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing

RG Dromey, How to Solve it by Computer, Cambridge University Press.

Shi-kuo Chang, Data Structures and Algorithms, World Scientific

Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.

Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms. Prentice-Hall of India Pvt. Limited, New Delhi

Timothy A. Budd, Classic Data Structures in C++, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 5: Trees

Notes

CONTENTS

Objectives

Introduction

5.1 Concept of Tree

5.2 Binary Tree

5.2.1 Types of Binary Tree

5.2.2 Properties of a Binary Tree

5.3 Binary Representation

5.4 Implementation of a Binary Tree

5.5 Binary Tree Traversal

5.5.1 Order of Traversal of Binary Tree

5.5.2 Procedure for Inorder Traversal

5.6 Summary

5.7 Keywords

5.8 Self Assessment

5.9 Review Questions

5.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the concept of tree
- Explain concept of binary tree
- Know binary tree traversal

Introduction

While dealing with many problems in computer science, engineering and many other disciplines, it is needed to impose a hierarchical structure on a collection of data items. For example, we need to impose a hierarchical structure on a collection of data items while preparing organizational charts and genealogies, to represent the syntactic structure of source programs in compilers. A tree is a data structure that is used to model such a hierarchical structure on data items, hence the study of tree as one of the data structures is important. This module discusses tree as a data structure.

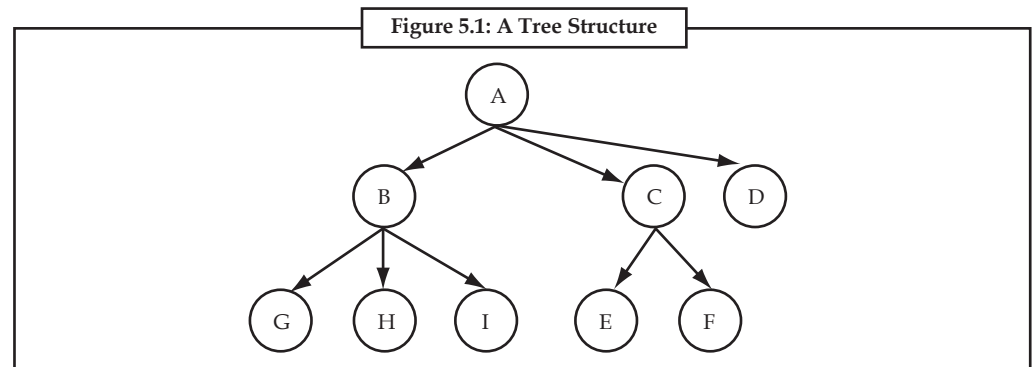
Notes

5.1 Concept of Tree

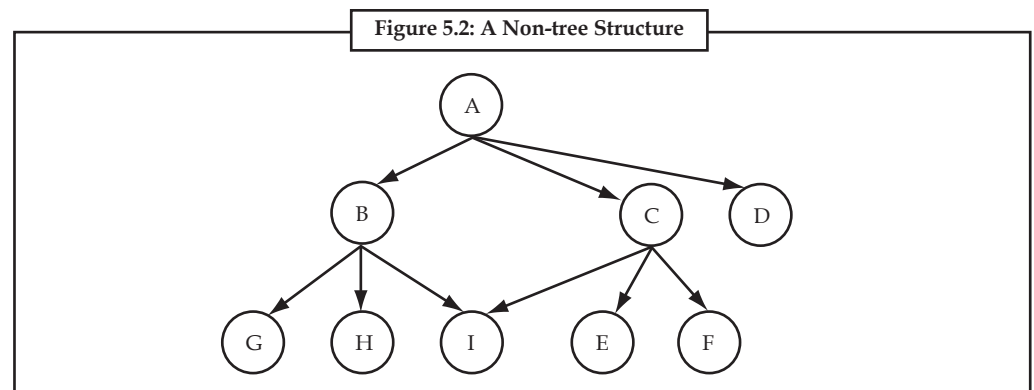
A tree is a set of one or more nodes T such that:

1. There is a specially designated node called root, and
2. Remaining nodes are partitioned into $n \geq 0$ disjoint set of nodes T_1, T_2, \dots, T_n each of which is a tree.

Shown below in Figure 5.1 is a structure, which is tree.



This is a tree because it is a set of nodes {A, B, C, D, E, F, G, H, I}, with node A as a root node, and the remaining nodes are partitioned into three disjoint sets: {B, G, H, I}, {C, E, F} AND {D} respectively. Each of these sets is a tree individually because each of these sets satisfies the above properties. Shown below in Figure 5.2 is a structure, which is not a tree:



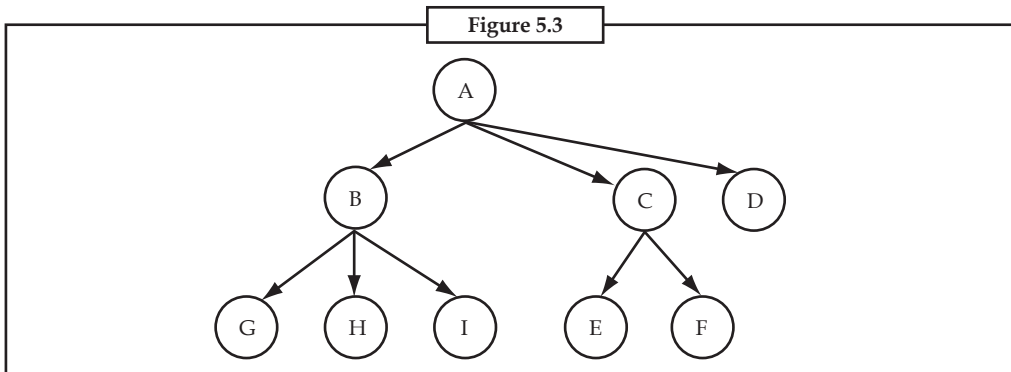
This is not a tree because it is a set of nodes {A, B, C, D, E, F, G, H, I}, with node A as a root node, but the remaining nodes cannot be partitioned into disjoint sets, because the node I is shared.

Given below are some of the important definitions, which are used in connection with trees.

1. **Degree of Node of a Tree:** The degree of a node of a tree is the number of sub-trees having this node as a root, or it is a number of decedents of a node. If degree is zero then it is called terminal node or leaf node of a tree.
2. **Degree of a Tree:** It is defined as the maximum of degree of the nodes of the tree, i.e. degree of tree = max (degree (node i) for $i = 1$ to n).
3. **Level of a Node:** We define the level of the node by taking the level of the root node to be 1, and incrementing it by 1 as we move from the root towards the sub-trees i.e. the level of all the decedents of the root nodes will be 2. The level of their decedents will be 3 and so on. We then define depth of the tree to be the maximum value of level for node of a tree.

Consider the tree given in Figure 5.3:

Notes



The degree of each node of the tree:

Node	Degree
A	3
B	3
C	2
D	0
E	0
F	0
G	0
H	0
I	0

The degree of the tree: Maximum (Degree of all the nodes) = 3

The level of nodes of the tree:

Node	Level
A	1
B	2
C	2
D	2
E	3
F	3
G	3
H	3
I	3



Task

In figure 5.1 calculate the degree of a tree.

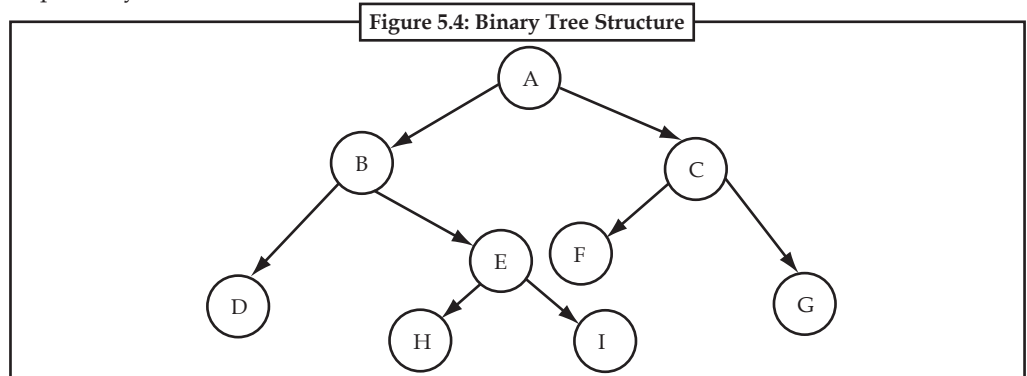
5.2 Binary Tree

A binary tree is a special tree where each non-leaf node can have at most two child nodes. Most important types of trees which are used to model yes/no, on/off, higher/lower, i.e., binary decisions are binary trees.

Recursive Definition: "A binary tree is either empty or a node that has left and right sub-trees that are binary trees. Empty trees are represented as boxes (but we will almost always omit the boxes)".

Notes

In a formal way, we can define a binary tree as a finite set of nodes which is either empty or partitioned into sets of T_0, T_l, T_r , where T_0 is the root and T_l and T_r are left and right binary trees, respectively.



So, for a binary tree we find that:

1. The maximum number of nodes at level i will be 2^{i-1}
2. If k is the depth of the tree then the maximum number of nodes that the tree can have is

$$2^k - 1 = 2^{k-1} + 2^{k-2} + \dots + 2^0$$

5.2.1 Types of Binary Tree

There are two main binary tree and these are:

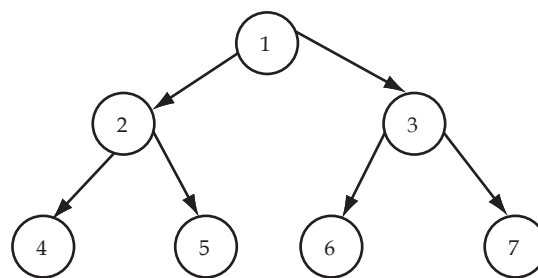
1. Full binary tree
2. Complete binary tree

Full Binary Tree

A full binary tree is a binary of depth k having $2^k - 1$ nodes. If it has $< 2^k - 1$, it is not a full binary tree.



Example: For $k = 3$, the number of nodes = $2^k - 1 = 2^3 - 1 = 8 - 1 = 7$. A full binary tree with depth $k = 3$ is shown in figure.



A Full Binary Tree

We use numbers from 1 to $2^k - 1$ as labels of the nodes of the tree.

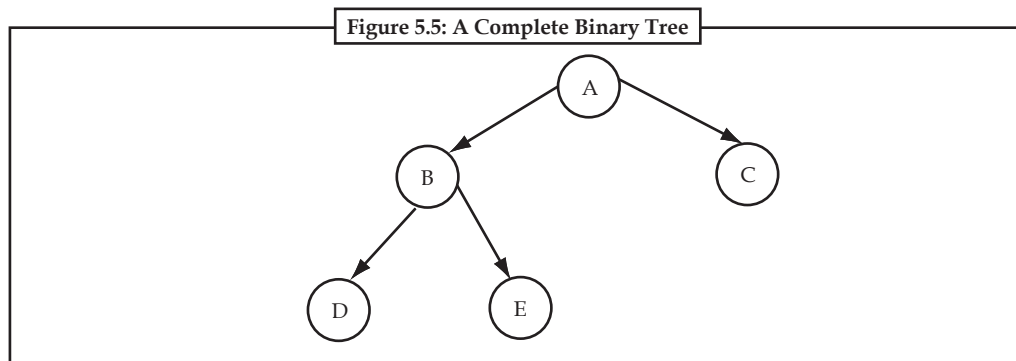
If a binary tree is full, then we can number its nodes sequentially from 1 to $2^k - 1$, starting from the root node, and at every level numbering the nodes from left to right.

Complete Binary Tree

Notes

A complete binary tree of depth k is a tree with n nodes in which these n nodes can be numbered sequentially from 1 to n , as if it would have been the first n nodes in a full binary tree of depth k .

A complete binary tree with depth $k = 3$ is shown in Figure 5.5.



5.2.2 Properties of a Binary Tree

Main properties of binary tree are:

1. If a binary tree contains n nodes, then it contains exactly $n - 1$ edges;
2. A Binary tree of height h has $2^h - 1$ nodes or less.
3. If we have a binary tree containing n nodes, then the height of the tree is at most n and at least $\lceil \log_2(n + 1) \rceil$.
4. If a binary tree has n nodes at a level l then, it has at most $2n$ nodes at a level $l + 1$
5. The total number of nodes in a binary tree with depth k (root has depth zero) is $N = 2^0 + 2^1 + 2^2 + \dots + 2^k = 2^{k+1} - 1$.



Task

In figure 5.4 total no. of node present.

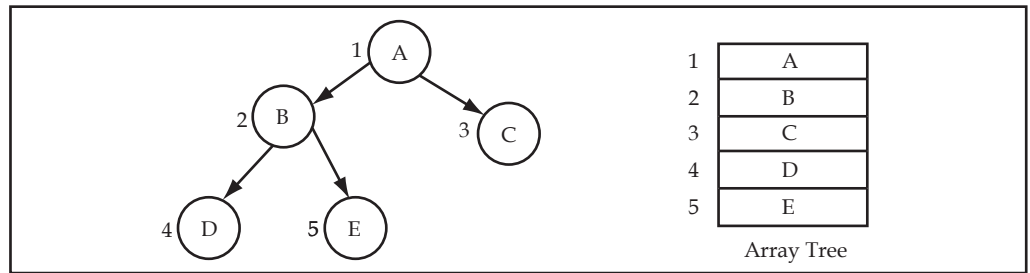
5.3 Binary Representation

If a binary tree is a *complete binary tree*, it can be represented using an array capable of holding n elements where n is the number of nodes in a complete binary tree. If the tree is an array of n elements, we can store the data values of the i^{th} node of a complete binary tree with n nodes at an index i in an array tree. That means we can map node i to the i^{th} index in the array, and the parent of node i will get mapped at an index $i/2$, whereas the left child of node i gets mapped at an index $2i$ and the right child gets mapped at an index $2i + 1$.



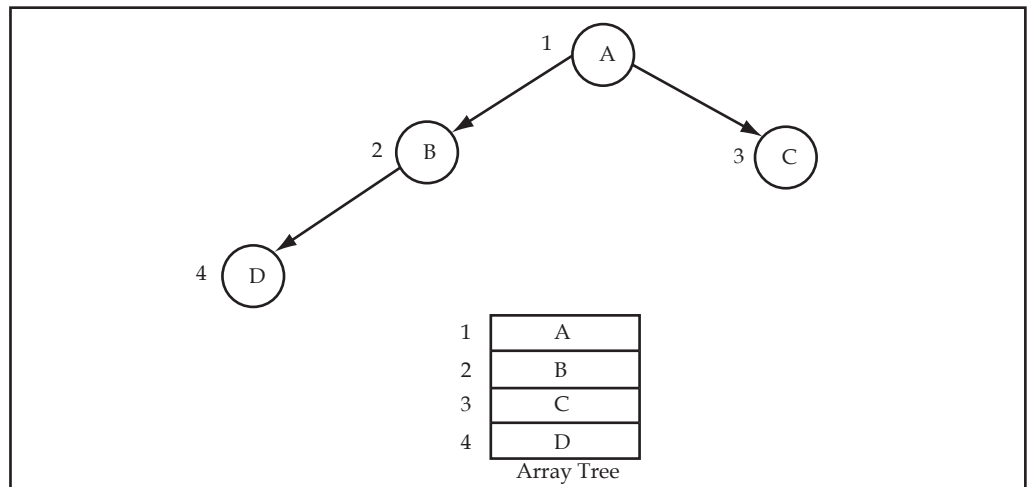
Example: A complete binary tree with depth $k = 3$, having the number of nodes $n = 5$, can be represented using an array of 5 as shown in figure.

Notes



An Array representation of a Complete Binary Tree having 5 Nodes and Depth 3

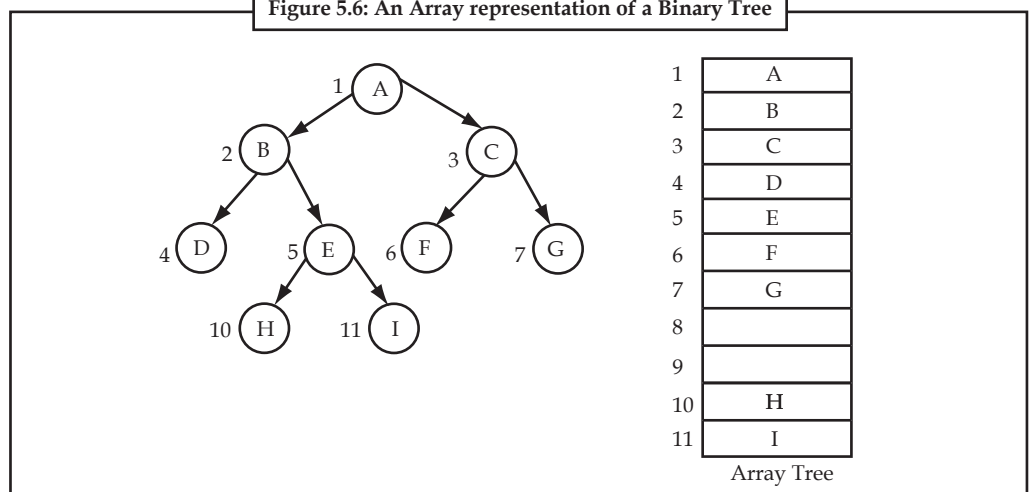
Shown in figure is another example of an array representation of a complete binary tree with depth $k = 3$, with the number of nodes $n = 4$.



An Array representation of a Complete Binary Tree with 4 Nodes and Depth 3

In general, any binary tree can be represented using an array. We see that an array representation of a complete binary tree does not lead to the waste of any storage. But if you want to represent a binary tree that is not a complete binary tree using an array representation, then it leads to the waste of storage as shown in Figure 5.6.

Figure 5.6: An Array representation of a Binary Tree



An array representation of a binary tree is not suitable for frequent insertions and deletions, even though no storage is wasted if the binary tree is a complete binary tree. It makes insertion

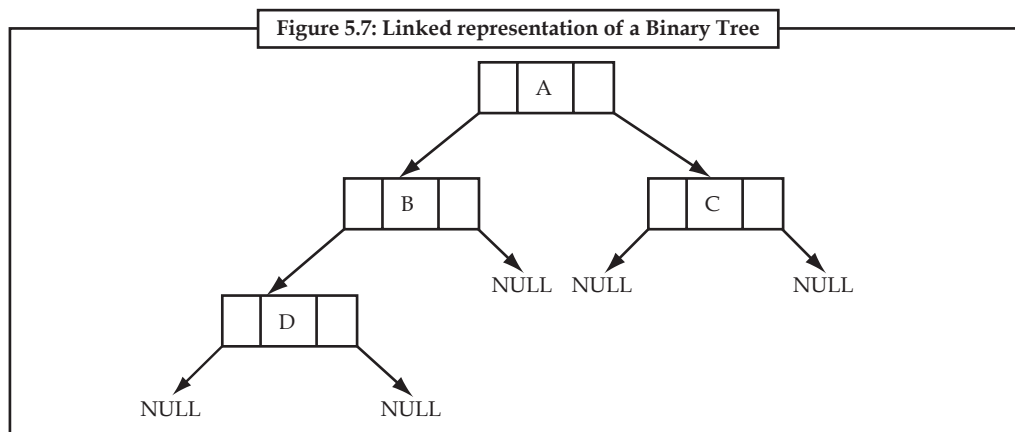
and deletion in a tree costly. Therefore, instead of using an array representation, we can use a linked representation, in which every node is represented as a structure with three fields: one for holding data, one for linking it with the left subtree, and the third for linking it with right subtree as shown here:



We can create such a structure using the following C declaration:

```
struct tnode
{
    int data
    struct tnode *lchild,*rchild;
};
```

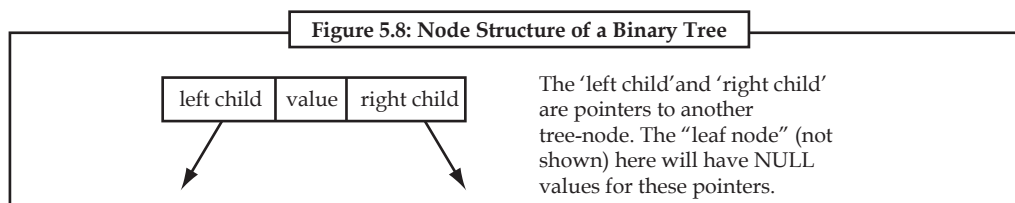
A tree representation that uses this node structure is shown in Figure 5.7.



5.4 Implementation of a Binary Tree

Like general tree, binary trees are implemented through linked lists. A typical node in a Binary tree has a structure as follows (refer to Figure 5.8):

```
struct NODE
{
    struct NODE *leftchild;
    int nodevalue; /* this can be of any data type */
    struct NODE *rightchild;
};
```




The binary tree creation follows a very simple principle. For the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree, then move towards the left side of that element or else to its right. If there is no sub tree on

Notes

the left, then make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly, the same has to be done for the case when your new element is greater than the current element in the tree but this time with the right child. Though this logic is followed for the creation of a Binary tree, this logic is often suitable to search for a key value in the binary tree.

Algorithm for the Implementation of a Binary Tree

1. **Step-1:** If value of new element < current element, then go to step-2 or else step-3.
2. **Step-2:** If the current element does not have a left sub-tree, then make your new element the left child of the current element; else make the existing left child as your current element and go to step-1.
3. **Step-3:** If the current element does not have a right sub-tree, then make your new element the right child of the current element; else make the existing right child as your current element and go to step-1.



Task "If a binary tree is a complete binary tree, it can be represented using an array capable of holding n elements where n is the number of nodes in a complete binary tree." Discuss.



Lab Exercise Program: Depicts the segment of code for the creation of a binary tree.

```

struct NODE
{
    struct NODE *left;
    int value;
    struct NODE *right;
};

create_tree(struct NODE *curr, struct NODE *new )
{
    if(new->value <= curr->value)
    {
        if(curr->left != NULL)
            create_tree(curr->left, new);
        else
            curr->left = new;
    }
    else
    {
        if(curr->right != NULL)
            create_tree(curr->right, new);
        else
    
```

```
curr->right = new;
}
}
```

Notes

Program: Binary tree creation

Array-based representation of a Binary Tree

Consider a complete binary tree T having n nodes where each node contains an item (value). Label the nodes of the complete binary tree T from top to bottom and from left to right $0, 1, \dots, n-1$. Associate with T the array A where the i entry of A is the item in the node labelled i of T , $i = 0, 1, \dots, n-1$. Figure 5.8 depicts the array representation of a Binary tree of figure.

Given the index i of a node, we can easily and efficiently compute the index of its parent and left and right children:

Index of Parent: $(i - 1)/2$, **Index of Left Child:** $2i + 1$, **Index of Right Child:** $2i + 2$.

Table 5.1: Array representation of a Binary Tree

Node	Item	Left child	Right child
0	A	1	2
1	B	3	4
2	C	-1	-1
3	D	5	6
4	E	7	8
5	G	-1	-1
6	H	-1	-1
7	I	-1	-1
8	J	-1	-1
9	?	?	?

First column represents index of node, second column consist of the item stored in the node and third and fourth columns indicate the positions of left and right children (-1 indicates that there is no child to that particular node.)

5.5 Binary Tree Traversal

This section discusses different orders in which a binary tree can be traversed. The algorithms for some commonly used orders of traversal are also presented. It also discusses the issue of construction of a unique binary tree given the orders of traversal.

5.5.1 Order of Traversal of Binary Tree

The following are the possible orders in which a binary tree can be traversed:

1. LDR
2. LRD
3. DLR
4. RDL
5. RLD
6. DRL

Notes

where,

L stands for traversing the left sub-tree,

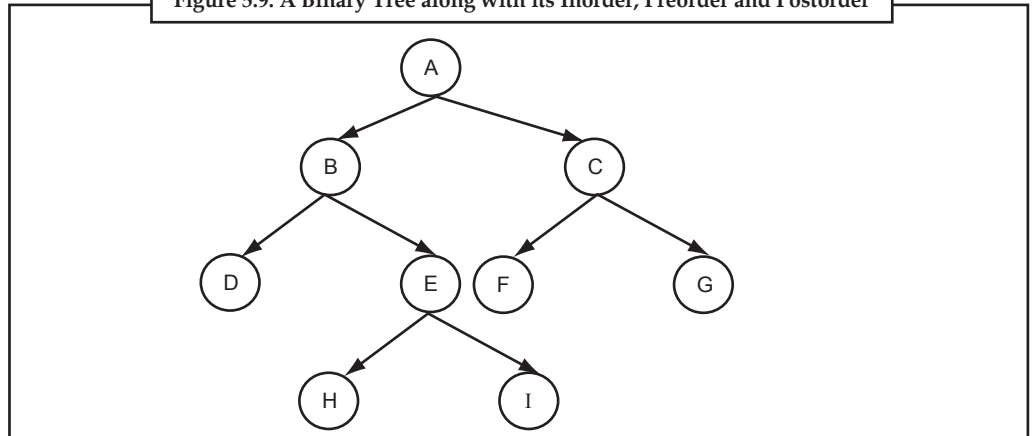
R stands for traversing the right sub-tree, and

D stands for processing the data of the node.

Therefore, the order LDR is the order of traversal in which we start with the root node, visit the left sub-tree first, then process the data of the root node, and then go for visiting the right sub-tree. Since the left, as well as right sub-trees are also the binary trees, the same procedure is used recursively while visiting the left and right sub-trees.

The order LDR is called inorder, the order LRD is called postorder, and the order DLR is called preorder. The remaining three orders are not used. If the processing that we do with the data in the node of tree during the traversal is simply printing the data value, then the output generated for a tree given below in Figure 5.9, using the inorder, preorder and postorder is the one shown below in Figure 5.9 itself.

Figure 5.9: A Binary Tree along with its Inorder, Preorder and Postorder



Different order path in Figure 5.9 are:

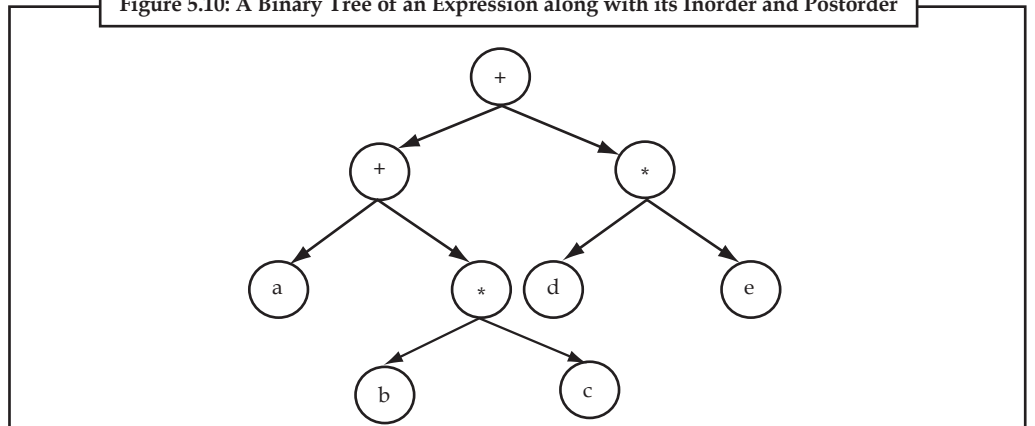
Inorder: DBHEIAFCG

preorder: ABDEHICFG

postorder: DHIEBFGCA

If an expression is represented as a binary tree then the inorder traversal of the tree gives us an infix expression, whereas the postorder traversal gives us postfix expression as shown below in Figure 5.10.

Figure 5.10: A Binary Tree of an Expression along with its Inorder and Postorder



Different order path in Figure 5.10 are:

Inorder : $a + b * c + d * e$

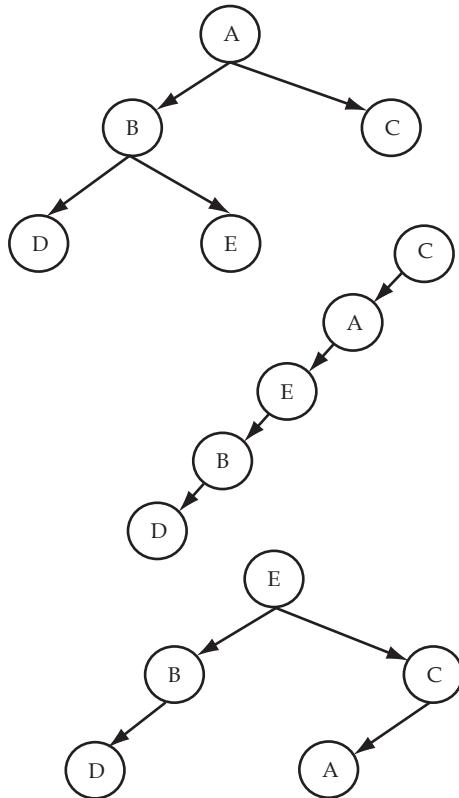
postorder: $abc*+de**$

Given an order of traversal of a tree it is possible to construct a tree.



Example: Consider the following order: Inorder = DBEAC

We can construct the binary trees shown below in figure using this order of traversal:



Binary Trees Constructed using given Inorder

Therefore we conclude that given only one order of traversal of a tree it is possible to construct a number of binary trees, a unique binary tree is not possible to be constructed. For construction of a unique binary tree we require two orders in which one has to be inorder, the other can be preorder or postorder.



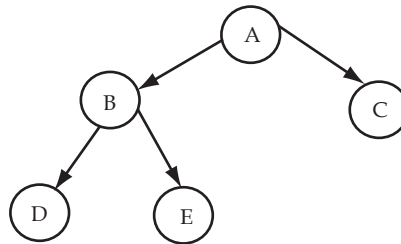
Example: Consider the following orders:

Inorder = DBEAC

Postorder = DEBCA

We can construct a unique binary tree shown in Figure using these orders of traversal.

Notes



A Unique Binary Constructed using the Inorder and Postorder

5.5.2 Procedure for Inorder Traversal

```
void inorder(tnode *p)
{
    if(p != NULL)
    {
        inorder(p->lchild);
        printf(p->data);
        inorder(p->rchild);
    }
}
```

A non-recursive/iterative procedure for traversing a binary tree in inorder is given below for the purpose of doing the analysis.

```
void inorder(tnode *p)
{
    tnode *stack[100];
    int top;
    {
        top = 0;
        if(p != NULL)
        {
            top = top + 1;
            stack[top] = p;
            p = p->lchild;
            while(top > 0)
            {
                while(p != NULL)
                    /*push the left child onto the stack*/
                {
                    top = top + 1;
                    stack[top] = p;
                    p = p->lchild;
                }
            }
        }
    }
}
```

```

    }
    p = stack[top];
    top = top-1;
    printf(p->data);
    p = p->rchild;
    if(p != NULL)
        /*push right child*/
        {
            top = top+1;
            stack[top] = p;
            p = p->lchild;
        }
    }
}

```

Analysis

Consider the iterative version of the inorder given above. If the binary tree to be traversed is having n nodes, then the number of nil links is $n + 1$. Since every node is placed on the stack once, the statements $\text{stack}[\text{top}] := p$ and $p := \text{stack}[\text{top}]$ are executed n times. The test for nil links will be done exactly $n+1$ times. So every step will be executed no more than some small constant times n , hence the order of algorithm $O(n)$. Similar analysis can be done to obtain the estimate of the computation time for preorder and post order.

Preorder Traversal

```

void preorder(tnode *p)
{
    if(p != NULL)
    {
        printf(p->data);
        preorder(p->lchild);
        preorder(p->rchild);
    }
}

```

Postorder Traversal

```

void postorder(tnode *p)
{
    if(!p)
    {
        postorder(p->lchild);
        postorder(p->rchild);
        printf(p->data);
    }
}

```

Notes

```

    }
}

```

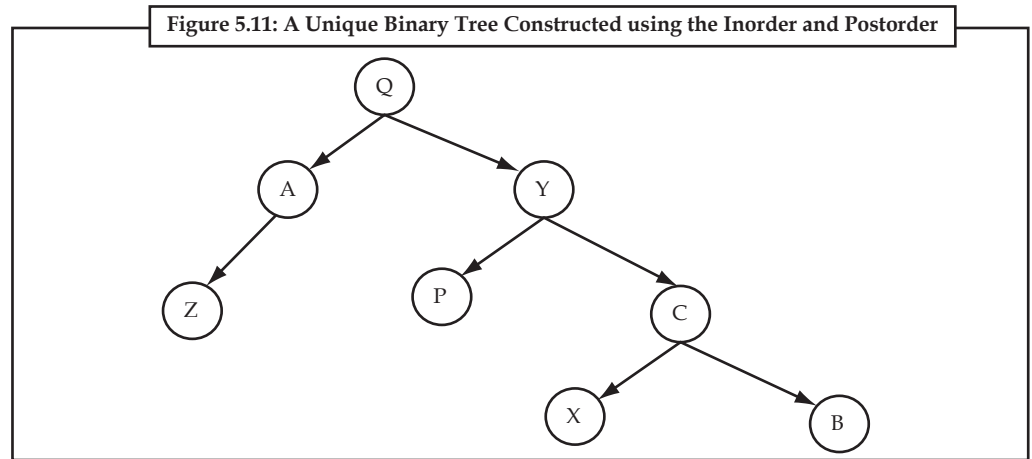
Consider the following example.

Given the preorder and inorder traversal of a binary tree. Draw the tree and write down its postorder traversal.

Inorder: Z, A, Q, P, Y, X, C, B

Preorder: Q, A, Z, Y, P, C, X, B

To obtain the binary tree take the first node in preorder, it is a root node, we then search for this node in the inorder traversal, all the nodes to the left of this node in the inorder traversal will be the part of the left sub-tree, and all the nodes to the right of this node in the inorder traversal will be the part of the right sub-tree. We then consider the next node in the preorder, if it is a part of the left sub-tree, then we make it as left child of the root, otherwise if it is part of the right sub-tree then we make it as part of right sub-tree. This procedure is repeated recursively to get the tree shown below in Figure 5.11:



The post order for this tree is:

Z, A, P, X, B, C, Y, Q

The following function counts the number of leaf node in a binary tree.

```

int count (tnode *p)
{
    if (p == NULL)
        count = 0;
    else
        if ((p->lchild == NULL) && (p->rchild == NULL))
            count = 1;
        else
            count = count (p->lchild) + count (p->rchild);
}

```

The following procedure swaps the left and the right child of every node of a given binary tree.

```

void swaptree (tnode *p)
{

```

```

tnode *temp;
if (p != NULL)
{
    swaptree(p->lchild);
    swaptree(p->rchild);
    temp = p->lchild;
    p->lchild = p->rchild;
    p->rchild = temp;
}
}

```

The following function checks whether the two binary trees are equal or not.

```

boolean equal(tnode *p1, tnode *p2)
{
    boolean ans;
    if ((p1 == NULL) && (p2 == NULL))
        ans = true;
    else
    if (((p1==NULL) && (p2!=NULL)) || ((p1!=NULL) && (p2==NULL))) ans = false;
    else
        while((p1 != NULL) && (p2 != NULL))
        {
            if (p1 != NULL) && (p2 != NULL)
                if ((equal(p1->lchild, p2->lchild))
                    ans = equal(p1->rchild, p2->rchild);
                else
                    ans = false;
            else
                ans = false;
        }
    return (ans);
}

```

The following function creates exact copy of a given binary trees.

```

Tnode *copytree(tnode *p)
{
    tnode *q;
    {
    if (p == NULL)
        return (NULL);
    else
        {

```

Notes

```
q = new(tnode);
q->data = p->data;
q->lchild = copytree(p->lchild);
q->rchild = copytree(p->rchild);
return(q);
}
}
d
```

5.6 Summary

- A tree is a set of one or more nodes T such that there is a specially designated node called root, and remaining nodes are partitioned into disjoint set of nodes.
- The degree of a node of a tree is the number of sub-trees having this node as a root, or it is a number of decedents of a node. If degree is zero then it is called terminal node or leaf node of a tree.
- Degree of a tree is defined as the maximum of degree of the nodes of the tree.
- A tree non of whose nodes has more than N children is known as N-ary tree. In other words, an N-ary tree is a tree whose degree is at the most N. Binary tree is a special type of tree having degree 2.
- A binary tree of depth k can have maximum $2^k - 1$ number of nodes. If a binary tree has fewer than $2^k - 1$ nodes, it is not a full binary tree.

5.7 Keywords

Degree of a tree: The highest degree of a node appearing in the tree.

Inorder: A tree traversing method in which the tree is traversed in the order of left-tree, node and then right-tree.

Level of a node: The number of nodes that must be traversed to reach the node from the root.

Node: A data structure that holds information and links to other nodes.

Postorder: A tree traversing method in which the tree is traversed in the order of left-tree, right-tree and then node.

Preorder: A tree traversing method in which the tree is traversed in the order of node, left-tree and then right-tree.

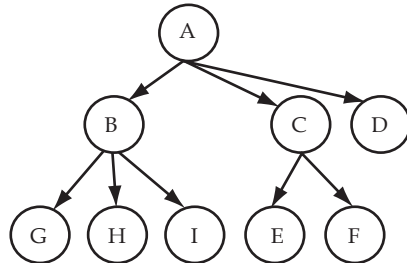
Root node: The node in a tree which does not have a parent node.

Tree: A two-dimensional data structure comprising of nodes where one node is the root and rest of the nodes form two disjoint sets each of which is a tree.

5.8 Self Assessment

Notes

In the context of the figure given below give the answers of self assessment.



1. What is the degree of node E

(a) 3	(b) 2
(c) 1	(d) 0
2. What is the degree of node C

(a) 4	(b) 2
(c) 1	(d) 3
3. What is the level of node A

(a) 1	(b) 0
(c) 3	(d) 4
4. What is the level of node H

(a) 2	(b) 3
(c) 4	(d) 0

Fill in the blanks:

5. To delete a node from a binary search tree the method to be used depends on
6. The order LDR is called, the order LRD is called and the order DLR is called
7. Binary tree is a special type of tree having degree
8. A degree three tree is called a tree.
9. A node with degree zero is called a node.

State the following statements are true or false:

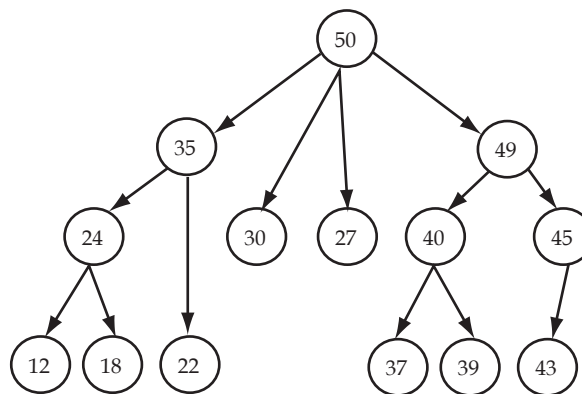
10. A binary tree of depth k can have maximum 2^{k-1} number of nodes.
11. For a binary tree, maximum number of nodes at level i is 2^{i-1} .
12. A full binary tree is a binary of depth k having $2^k - 1$ nodes. If it has $< 2^k - 1$, it is not a full binary tree.

5.9 Review Questions

1. Give the array representation of a complete binary tree with depth $k = 3$, having the number of nodes $n = 7$.

Notes

2. How many binary trees are possible with three nodes?
3. Construct a binary tree whose in-order and pre-order traversal is given below.
 In-order: 5,1,3,11,6,8,4,2,7
 Pre-order: 6,1,5,11,3,4,8,7,2
4. What do you mean by binary tree? Also explain the various properties of binary tree.
5. Consider the tree given below:
 - (a) Find the degree of each node of the tree.
 - (b) The degree of the tree.
 - (c) The level of each node of the tree.



6. Determine the order in which the vertices of the binary tree given in question No. 1 will be visited under:
 - (a) Inorder
 - (b) Preorder
 - (c) Postorder
7. Write a C program to delete all the leaf nodes of a binary tree.
8. Write a method and the corresponding recursive function to count the leaves (i.e., the nodes with both subtrees empty) of a linked binary tree.
9. Write a method and the corresponding recursive function to insert an Entry, passed as a parameter, into a linked binary tree. If the root is empty, the new entry should be inserted into the root, otherwise it should be inserted into the shorter of the two subtrees of the root (or into the left subtree if both subtrees have the same height).
10. Write a function to perform a double-order traversal of a binary tree, meaning that at each node of the tree, the function first visits the node, then traverses its left subtree (in double order), then visits the node again, then traverses its right subtree (in double order).

Answers: Self Assessment

1. (d)
2. (b)
3. (a)
4. (b)
5. number of children for the node to be deleted

6. inorder, postorder, preorder	7. 2	Notes
8. ternary	9. terminal/leaf	
10. True	11. True	
12. True		

5.10 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.

Data Structures and Algorithms; Shi-Kuo Chang; World Scientific.

Data Structures and Efficient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.

Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi

Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing

RG Dromey, How to Solve it by Computer, Cambridge University Press.

Shi-kuo Chang, Data Structures and Algorithms, World Scientific

Sorenson and Tremblay: An Introduction to Data Structure with Algorithms

Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms. Prentice-Hall of India Pvt. Limited, New Delhi

Timothy A. Budd, Classic Data Structures in C++, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 6: Binary Search Tree and AVL Trees

CONTENTS

Objectives

Introduction

6.1 Binary Search Tree

6.2 Counting the Number of Nodes in a Binary Search Tree

6.3 Searching for a Target Key in a Binary Search Tree

6.4 Deletion of a Node from a Binary Search Tree

6.4.1 Deletion of a Node with Two Children

6.4.2 Deletion of a Node with one Child

6.4.3 Deletion of a Node with no Child

6.5 Application of a Binary Search Tree

6.6 AVL Tree

6.7 Summary

6.8 Keywords

6.9 Self Assessment

6.10 Review Questions

6.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe binary search tree
- Explain deletion of node with two child and with single child
- Describe applications of BST

Introduction

Consider the problem of searching a linked list for some target key. There is no way to move through the list other than one node at a time, and hence searching through the list must always reduce to a sequential search. As you know, sequential search is usually very slow in comparison with binary search. Hence, assuming we can keep the keys in order, searching becomes much faster if we use a contiguous list and binary search. Suppose we also frequently need to make changes in the list, inserting new entries or deleting old entries.

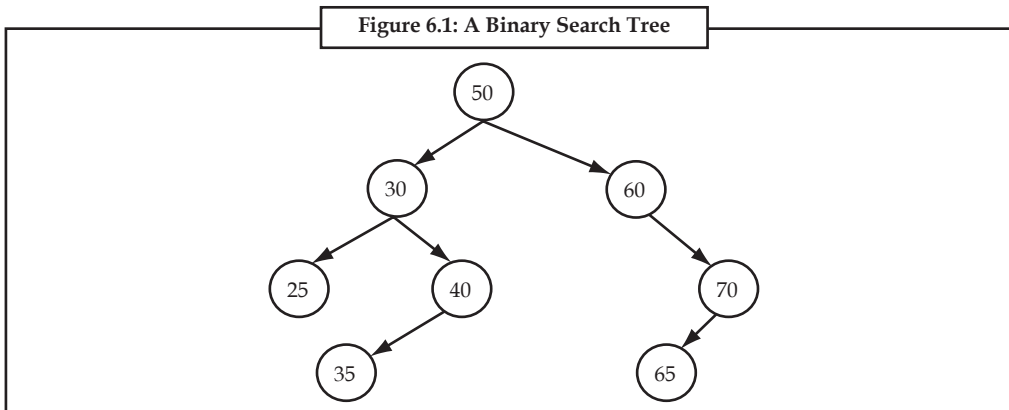
6.1 Binary Search Tree

A binary search tree is a binary tree which may be empty, and every node contains an identifier and

1. Identifier of any node in the left sub-tree is less than the identifier of the root
2. Identifier of any node in the right sub-tree is greater than the identifier of the root and the left sub-tree as well as right sub-tree both are binary search trees.

A tree shown below in Figure 6.1 is a binary search tree:

Notes



A binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder, and postorder. If we traverse a binary search tree in inorder and print the identifiers contained in the nodes of the tree, we get a sorted list of identifiers in the ascending order.

A binary search tree is an important search structure. For example, consider the problem of searching a list. If a list is ordered then searching becomes faster, if we use a contiguous list and binary search, but if we need to make changes in the list like inserting new entries and deleting old entries. Then it is much slower to use a contiguous list because insertion and deletion in a contiguous list requires moving many of the entries every time. So we may think of using a linked list because it permits insertions and deletions to be carried out by adjusting only few pointers, but in a linked list there is no way to move through the list other than one node at a time hence permitting only sequential access. Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary search tree, we find that we can search for a key in $O(n \log n)$ steps.

Creating a Binary Search Tree

We assume that every node in a binary search tree is capable of holding an integer data item and the links which can be made pointing to the root of the left and the right sub-tree respectively. Therefore the structure of the node can be defined using the following declaration:

```

struct tnode
{
    int data;
    tnode *lchild;
    tnode *rchild;
}
  
```

To create a binary search tree we use a procedure named insert which creates a new node with the data value supplied as a parameter to it, and inserts into an already existing tree whose root pointer is also passed as a parameter. The procedure accomplishes this by checking whether the tree whose root pointer is passed as a parameter is empty. If it is empty then the newly created node is inserted as a root node. If it is not empty then it copies the root pointer into a variable temp1, it then stores value of temp1 in another variable temp2, compares the data value of the node pointed to by temp1 with the data value supplied as a parameter, if the data value supplied as a parameter is smaller than the data value of the node pointed to by temp1 then it copies the left link of the node pointed to by temp1 into temp1 (goes to the left), otherwise it copies the right link of the node pointed to by temp1 into temp1 (goes to the right). It repeats this process till temp1 becomes nil.

Notes

When temp1 becomes nil, the new node is inserted as a left child of the node pointed to by temp2 if data value of the node pointed to by temp2 is greater than data value supplied as parameter. Otherwise the new node is inserted as a right child of node pointed to by temp2. Therefore the insert procedure is

```
void insert(tnode *p, int val)
{
    tnode *temp1, *temp2;
    if (p == NULL)
    {
        p = new(tnode);
        p->data = val;
        p->lchild = NULL;
        p->rchild = NULL;
    }
    else
    {
        temp1 = p;
        while(temp1 != NULL)
        {
            temp2 = temp1;
            if(temp1->data > val)
                temp1 = temp1->left;
            else
                temp1 = temp1->right;
        }
        if(temp2->data > val)
        {
            temp2->left = new(tnode);
            temp2 = temp2->left;
            temp2->data = val;
            temp2->left = NULL;
            temp2->right = NULL;
        }
        else
        {
            temp2->right = new(tnode);
            temp2 = temp2->right;
            temp2->data = val;
            temp2->left = NULL;
            temp2->right = NULL;
        }
    }
}
```

6.2 Counting the Number of Nodes in a Binary Search Tree

Notes

Counting the number of nodes in a given binary tree, the tree is required to be traversed recursively until a leaf node is encountered. When a leaf node is encountered, a count of 1 is returned to its previous activation (which is an activation for its parent), which takes the count returned from both the children's activation, adds 1 to it, and returns this value to the activation of its parent. This way, when the activation for the root of the tree returns, it returns the count of the total number of the nodes in the tree.



Lab Exercise Program: A complete C program to count the number of nodes is as follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
    int data;
    struct tnode *lchild, *rchild;
};
int count(struct tnode *p)
{
    if( p == NULL)
        return(0);
    else
        if( p->lchild == NULL && p->rchild == NULL)
            return(1);
        else
            return(1 + (count(p->lchild) + count(p->rchild)));
}
struct tnode *insert(struct tnode *p,int val)
{
    struct tnode *temp1,*temp2;
    if(p == NULL)
    {
        p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the
new node as root node*/
        if(p == NULL)
        {
            printf("Cannot allocate\n");
            exit(0);
        }
        p->data = val;
        p->lchild=p->rchild=NULL;
```

Notes

```

    }
else
{
    temp1 = p;
    /* traverse the tree to get a pointer to that node whose child will
be the newly created node*/
    while(temp1 != NULL)
    {
        temp2 = temp1;
        if( temp1 ->data > val)
            temp1 = temp1->lchild;
        else
            temp1 = temp1->rchild;
    }
    if( temp2->data > val)
    {
        temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode)); /
*inserts the newly created node
        as left child*/
        temp2 = temp2->lchild;
        if(temp2 == NULL)
        {
            printf("Cannot allocate\n");
            exit(0);
        }
        temp2->data = val;
        temp2->lchild=temp2->rchild = NULL;
    }
else
{
    temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts
the newly created node
    as left child*/
    temp2 = temp2->rchild;
    if(temp2 == NULL)
    {
        printf("Cannot allocate\n");
        exit(0);
    }
    temp2->data = val;
    temp2->lchild=temp2->rchild = NULL;
}

```

```

}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
    if(p != NULL)
    {
        inorder(p->lchild);
        printf("%d\t",p->data);
        inorder(p->rchild);
    }
}
void main()
{
    struct tnode *root = NULL;
    int n,x;
    printf("Enter the number of nodes\n");
    scanf("%d",&n);
    while( n --- > 0)
    {
        printf("Enter the data value\n");
        scanf("%d",&x);
        root = insert(root,x);
    }
    inorder(root);
    printf("\nThe number of nodes in tree are :%d\n",count(root));
}

```

Input

1. The number of nodes the created tree should have = 5
2. The data values of the nodes in the tree to be created are: 10, 20, 5, 9, 8

Output

1. 5 8 9 10 20
2. The number of nodes in the tree is 5

6.3 Searching for a Target Key in a Binary Search Tree

Searching for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer. If the key is less than the data value of the root node, repeat the process by using the left subtree. Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.

Notes

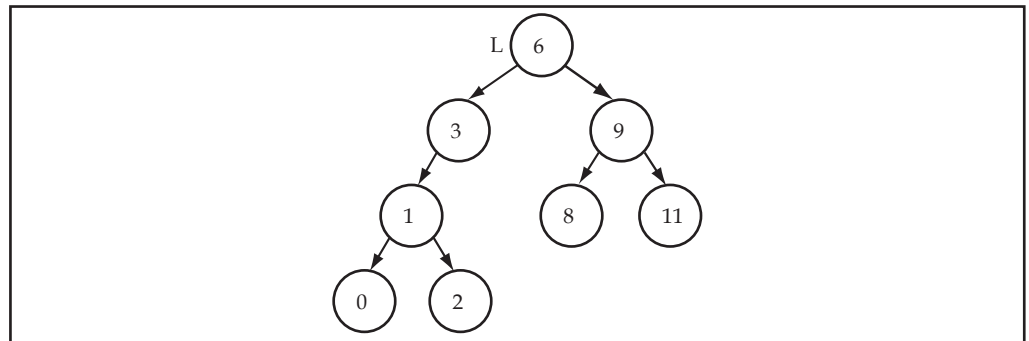
```

boolean search(tnode *p, int val)
{
    boolean ans;
    tnode *temp;
    temp = p;
    ans = false;
    while ((temp != NULL) && (!ans))
    {
        if(temp->data == val)
            ans = true;
        else
            if(temp->data > val)
                temp = temp->left;
            else
                temp = temp->right;
    }
}

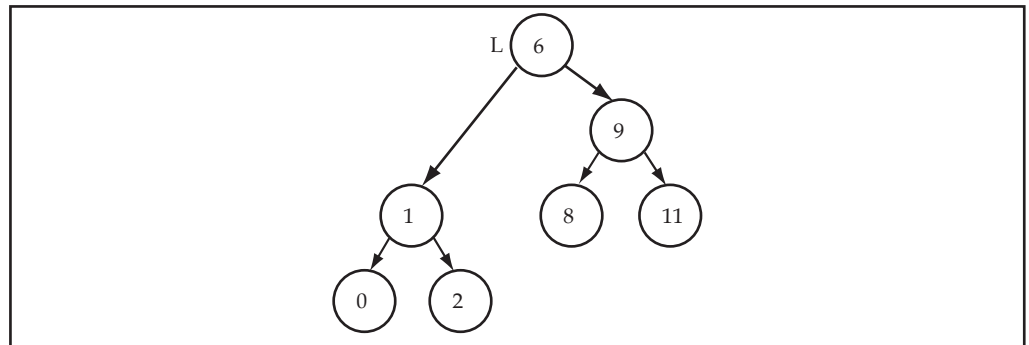
```

6.4 Deletion of a Node from a Binary Search Tree

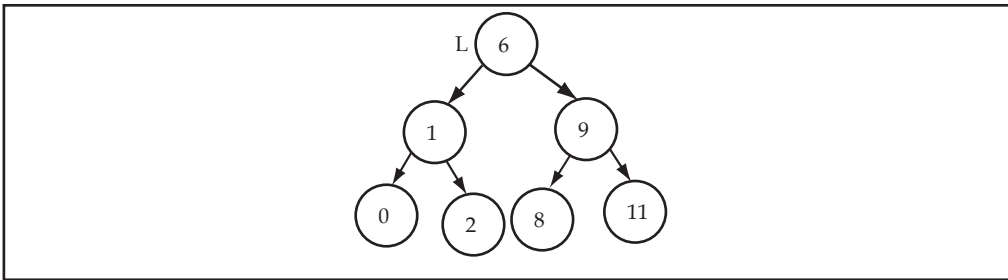
There is another simple situation: suppose the node we're deleting has only one subtree. In the following example, '3' has only 1 subtree.



To delete a node with 1 subtree, we just 'link past' the node, i.e. connect the parent of the node directly to the node's only subtree. This always works, whether the one subtree is on the left or on the right. Deleting '3' gives us:



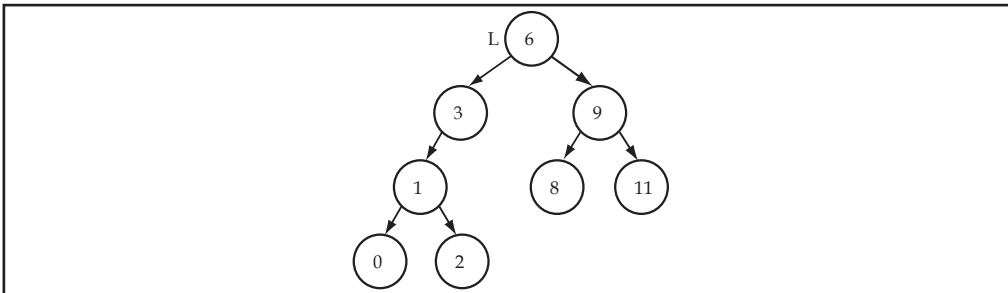
which we normally draw:



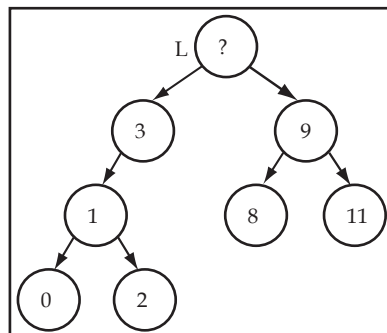
Finally, let us consider the only remaining case: how to delete a node having two subtrees. For example, how to delete '6'? We'd like to do this with minimum amount of work and disruption to the structure of the tree.

The standard solution is based on this idea: we leave the node containing '6' exactly where it is, but we get rid of the value 6 and find another value to store in the '6' node. This value is taken from a node below the '6's node, and it is that node that is actually removed from the tree.

So, here is the plan. Starting with:



Erase 6, but keep its node:



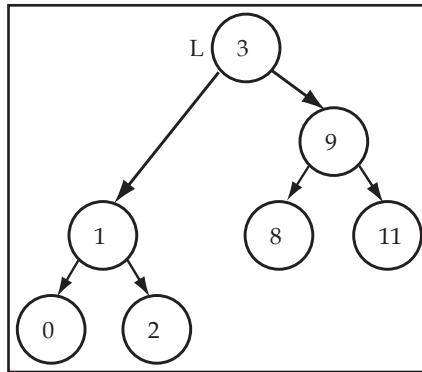
Now, what value can we move into the vacated node and have a binary search tree? Well, here's how to figure it out. If we choose value X, then:

1. Everything in the left subtree must be smaller than X.
2. Everything in the right subtree must be bigger than X.

Let's suppose we're going to get X from the left subtree. (2) is guaranteed because everything in the left subtree is smaller than everything in the right subtree. What about (1)? If X is coming from the left subtree, (1) says that there is a unique choice for X - we must choose X to be the largest value in the left subtree. In our example, 3 is the largest value in the left subtree. So if we put 3 in the vacated node and delete it from its current position we will have a BST with 6 deleted.

Notes

Here it is:



So our general algorithm is: to delete N, if it has two subtrees, replace the value in N with the largest value in its left subtree and then delete the node with the largest value from its left subtree.



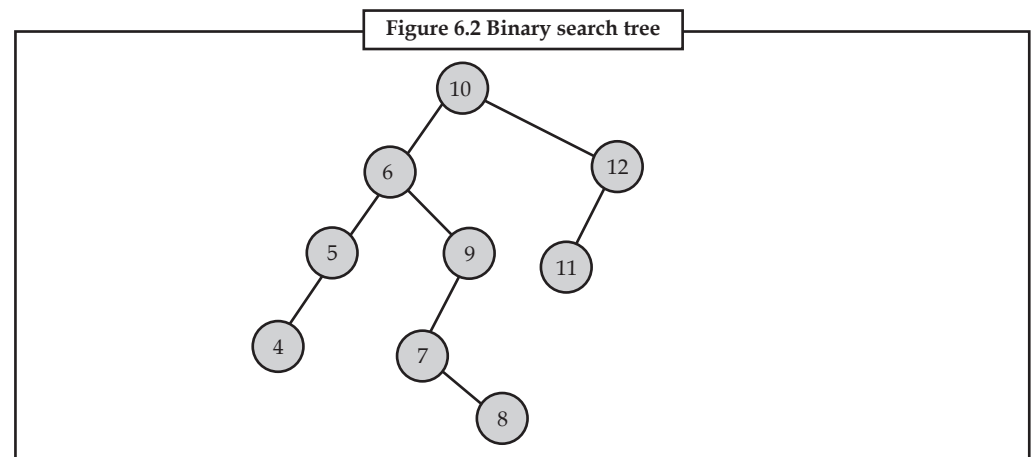
Note The largest value in the left subtree will never have two subtrees. Why? Because if it's the largest value it cannot have a right subtree.

Finally, there is nothing special about the left subtree. We could do the same thing with the right subtree: just use the smallest value in the right subtree.

To delete a node from a binary search tree the method to be used depends on whether a node to be deleted has one child, two children, or has no child.

6.4.1 Deletion of a Node with Two Children

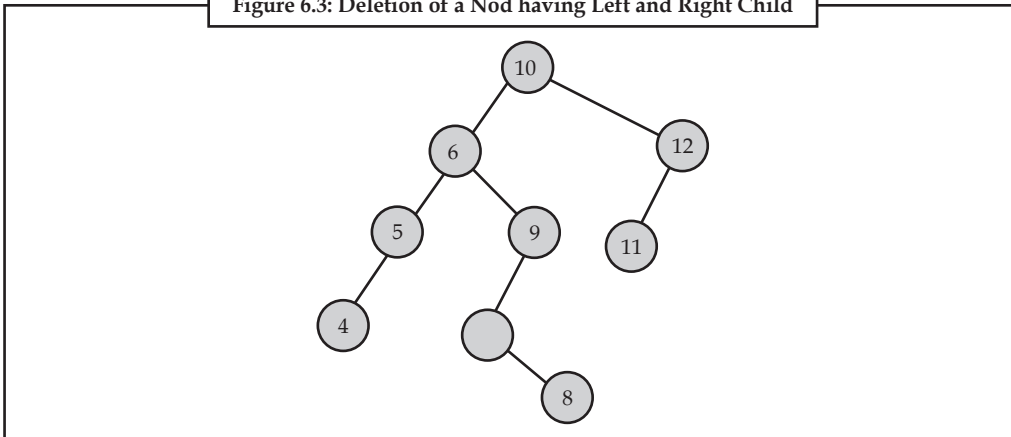
If the node to be deleted has two children; then the value is replaced by the smallest value in the right sub tree or the largest key value in the left sub tree; subsequently the empty node is recursively deleted. Consider the BST in Figure 6.2.



If the node 6 is to be deleted then first its value is replaced by smallest value in its right subtree i.e. by 7. So we will have Figure 6.3.

Notes

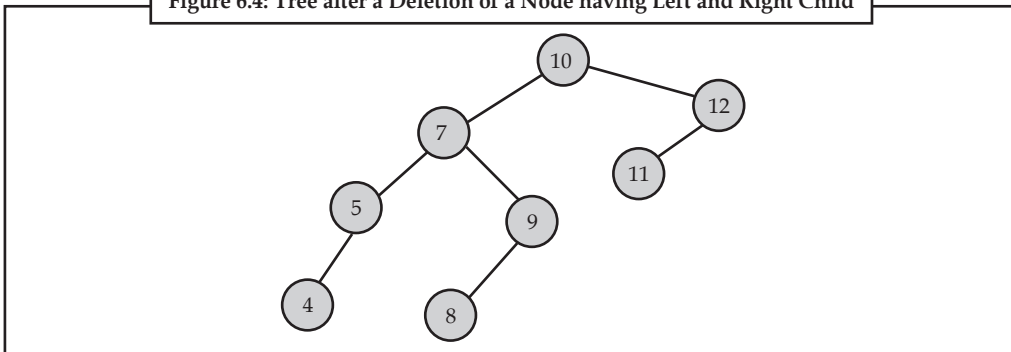
Figure 6.3: Deletion of a Node having Left and Right Child



Now we need to, delete this empty node shown in Figure 6.3.

Therefore, the final structure would be Figure 6.4.

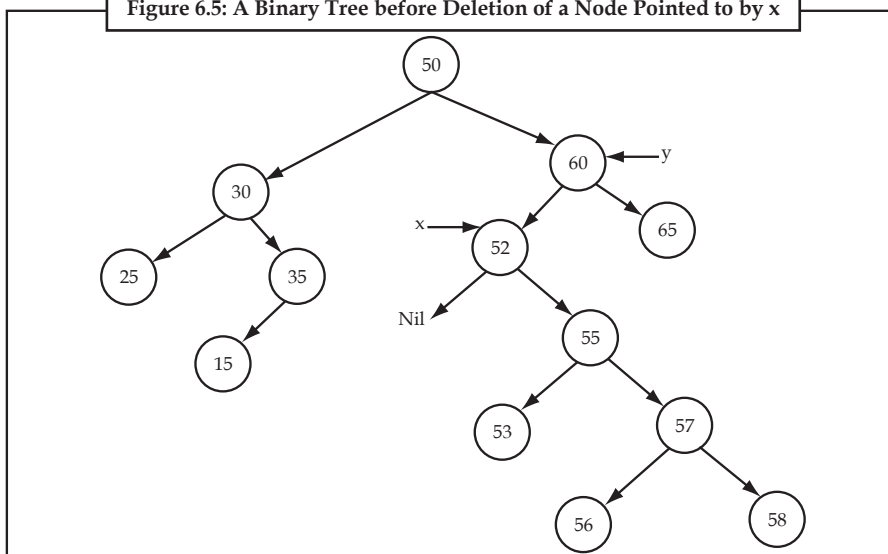
Figure 6.4: Tree after a Deletion of a Node having Left and Right Child



6.4.2 Deletion of a Node with one Child

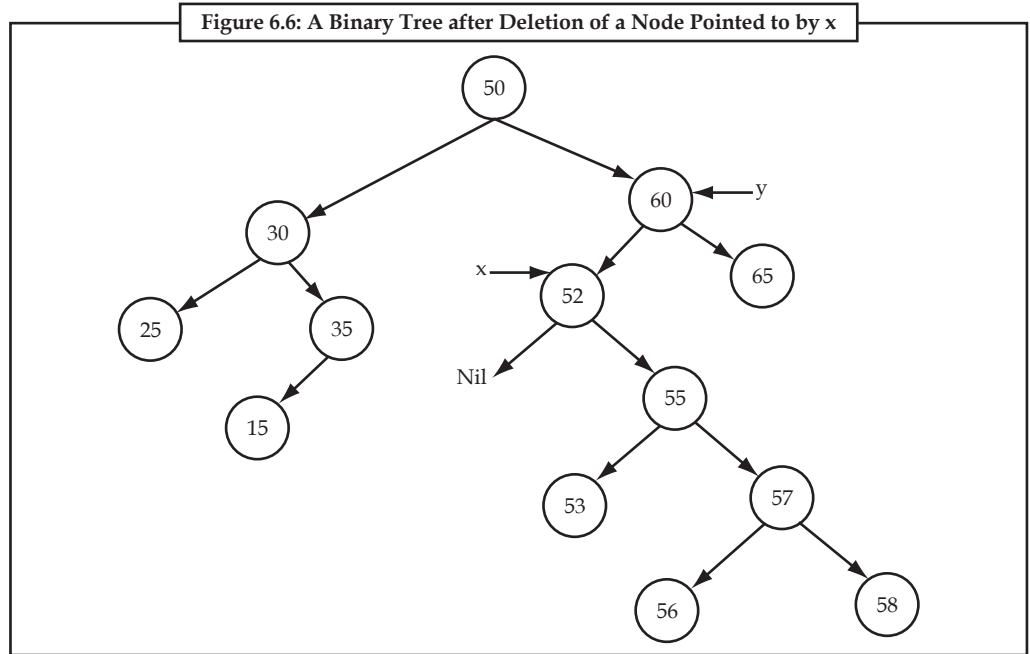
Consider the binary search tree shown below in Figure 6.5.

Figure 6.5: A Binary Tree before Deletion of a Node Pointed to by x



Notes

If we want to delete a node pointed to by x, then we can do it as follows: Let y be a pointer to the node which is the root of the node pointed to by x. Make the left child of the node pointed by y to be the right child of the node pointed by x, and dispose the node pointed by x as shown below in Figure 6.6:

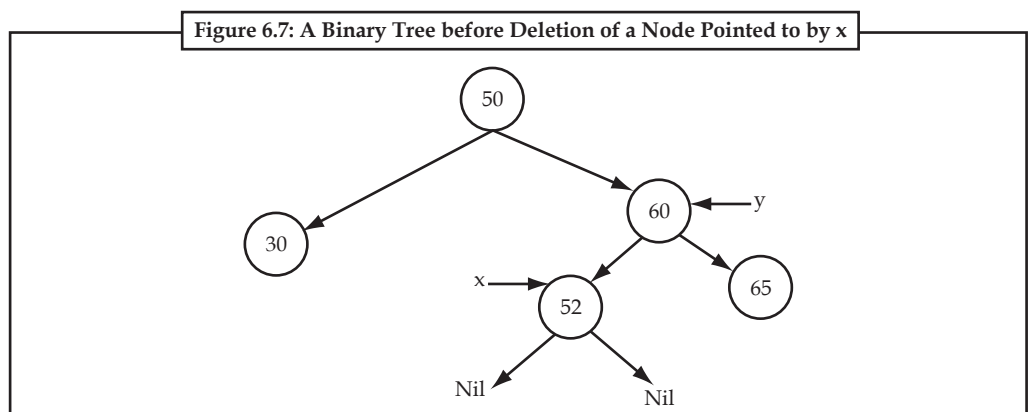


```

y->lchild = x->rchild;
x->rchild = NULL;
delete(x);
    
```

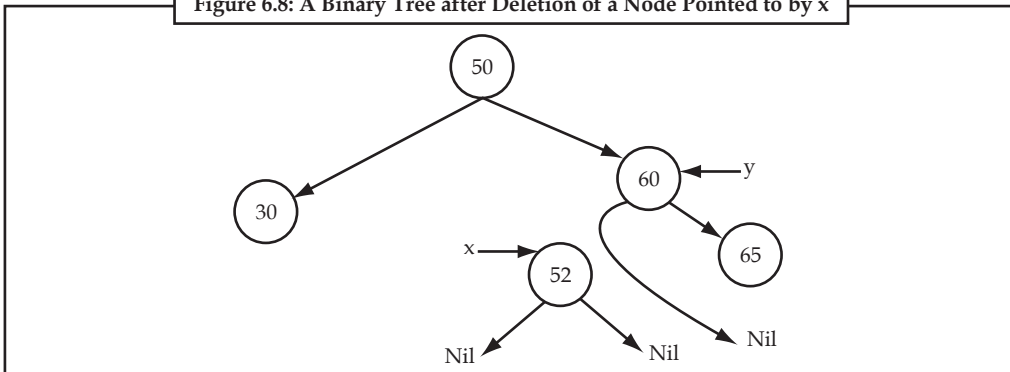
6.4.3 Deletion of a Node with no Child

Consider the binary search tree shown below in Figure 6.7:



Let the left child of the node pointed by y be nil, and dispose node pointed by x as shown in Figure 6.8.

Figure 6.8: A Binary Tree after Deletion of a Node Pointed to by x

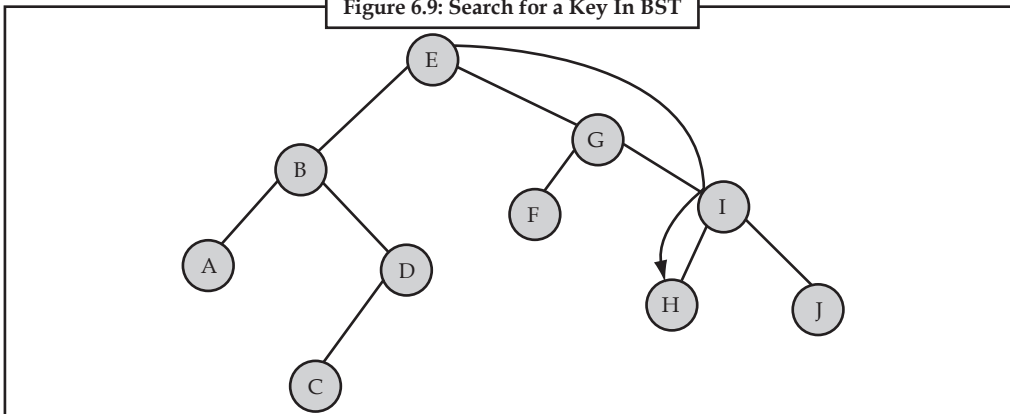


Notes

Search for a key in a BST

To search the binary tree for a particular node, we use procedures similar to those we used when adding to it. Beginning at the root node, the current node and the entered key are compared. If the values are equal success is output. If the entered value is less than the value in the node, then it must be in the left-child sub tree. If there is no left-child sub tree, the value is not in the tree i.e. a failure is reported. If there is a left-child subtree, then it is examined the same way. Similarly, if the entered value is greater than the value in the current node, the right child is searched. Figure 18 shows the path through the tree followed in the search for the key H.

Figure 6.9: Search for a Key In BST



```

find-key (key value, node)
{
  if (two values are same)
  {
    print value stored in node;
    return (SUCCESS);
  }
  else if (key value value stored in current node)
  {
    if (left child exists)
    {
      find-key (key-value, left hand);
    }
  }
}

```

Notes

```

else
{
there is no left subtree.,
return (string not found) }
}
else if (key-value value stored in current node)
{
if (right child exists)
{
find-key (key-value, rot child);
}
else
{
there is no right subtree;
return (string not found)
}
}
}
}

```



Lab Exercise Program: C program to delete a node, where the data value of the node to be deleted is known, is as follows:

```

#include <stdio.h>
#include <stdlib.h>
struct tnode
{
    int data;
    struct tnode *lchild, *rchild;
};
/* A function to get a pointer to the node whose data value is given
as well as the pointer to its root */
struct tnode *getptr(struct tnode *p, int key, struct tnode **y)
{
    struct tnode *temp;
    if( p == NULL)
        return(NULL);
    temp = p;
    *y = NULL;
    while( temp != NULL)
    {
        if(temp->data == key)
            return(temp);
    }
}

```

```

        else
    {
        *y = temp; /*store this pointer as root */
    if(temp->data > key)
        temp = temp->lchild;
    else
        temp = temp->rchild;
    }
}
return(NULL);
}

/* A function to delete the node whose data value is given */
struct tnode *delete(struct tnode *p,int val)
{
    struct tnode *x, *y, *temp;
    x = getptr(p,val,&y);
    if( x == NULL)
    {
        printf("The node does not exists\n");
        return(p);
    }
    else
    {
        /* this code is for deleting root node*/
        if( x == p)
        {
            temp = x->lchild;
            y = x->rchild;
            p = temp;
            while(temp->rchild != NULL)
                temp = temp->rchild;
            temp->rchild=y;
            free(x);
            return(p);
        }
        /* this code is for deleting node having both children */
        if( x->lchild != NULL && x->rchild != NULL)
        {
            if(y->lchild == x)
            {

```

Notes

```
temp = x->lchild;
y->lchild = x->lchild;
while(temp->rchild != NULL)
    temp = temp->rchild;
temp->rchild=x->rchild;
x->lchild=NULL;
x->rchild=NULL;
    }
    else
    {
        temp = x->rchild;
        y->rchild = x->rchild;
        while(temp->lchild != NULL)
            temp = temp->lchild;
        temp->lchild=x->lchild;
x->lchild=NULL;
x->rchild=NULL;
    }
free(x);
return(p);
}
/* this code is for deleting a node with on child*/
if(x->lchild == NULL && x->rchild != NULL)
{
    if(y->lchild == x)
y->lchild = x->rchild;
    else
        y->rchild = x->rchild;
        x->rchild; = NULL;
        free(x);
        return(p);
}
if( x->lchild != NULL && x->rchild == NULL)
{
    if(y->lchild == x)
        y->lchild = x->lchild ;
    else
        y->rchild = x->lchild;
        x->lchild = NULL;
        free(x);
```



```

        return(p);
    }
    /* this code is for deleting a node with no child*/
    if(x->lchild == NULL && x->rchild == NULL)
    {
        if(y->lchild == x)
            y->lchild = NULL ;
        else
            y->rchild = NULL;
        free(x);
        return(p);
    }
}

/*an iterative function to print the binary tree in inorder*/
void inorder1(struct tnode *p)
{
    struct tnode *stack[100];
    int top;
    top = -1;
    if(p != NULL)
    {
        top++;
        stack[top] = p;
        p = p->lchild;
        while(top >= 0)
        {
            while ( p!= NULL)/* push the left child onto stack*/
            {
                top++;
                stack[top] =p;
                p = p->lchild;
            }
            p = stack[top];
            top--;
            printf("%d\t",p->data);
            p = p->rchild;
            if ( p != NULL) /* push right child*/
            {
                top++;
                stack[top] = p;
                p = p->lchild;
            }
        }
    }
}

```

Notes

```
    }
  }
}

/* A function to insert a new node in binary search tree to get a tree
created*/
struct tnode *insert(struct tnode *p,int val)
{
  struct tnode *temp1,*temp2;
  if(p == NULL)
  {
    p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new
node as root node*/
    if(p == NULL)
    {
      printf("Cannot allocate\n");
      exit(0);
    }
    p->data = val;
    p->lchild=p->rchild=NULL;
  }
  else
  {
    temp1 = p;
    /* traverse the tree to get a pointer to that node whose child will be
the newly created node*/
    while(temp1 != NULL)
    {
      temp2 = temp1;
      if( temp1 ->data > val)
        temp1 = temp1->lchild;
      else
        temp1 = temp1->rchild;
    }
    if( temp2->data > val)
    {
      temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/* inserts
the newly created node
as left child*/
      temp2 = temp2->lchild;
      if(temp2 == NULL)
      {
        printf("Cannot allocate\n");

```

```

    exit(0);
    }
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
    temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts
the newly created node
as left child*/
    temp2 = temp2->rchild;
    if(temp2 == NULL)
    {
        printf("Cannot allocate\n");
        exit(0);
    }
    temp2->data = val;
    temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
void main()
{
    struct tnode *root = NULL;
    int n,x;
    printf("Enter the number of nodes in the tree\n");
    scanf("%d",&n);
    while( n > 0)
    {
        printf("Enter the data value\n");
        scanf("%d",&x);
        root = insert(root,x);
    }
    printf("The created tree is :\n");
    inorder1(root);
    printf("\n Enter the value of the node to be deleted\n");
    scanf("%d",&n);
    root=delete(root,n);
    printf("The tree after deletion is \n");
    inorder1(root);
}

```

Notes

Explanation

This program first creates a binary tree with a specified number of nodes with their respective data values. It then takes the data value of the node to be deleted, obtains a pointer to the node containing that data value, and obtains another pointer to the root of the node to be deleted. Depending on whether the node to be deleted is a root node, a node with two children a node with only one child, or a node with no children, it carries out the manipulations as discussed in the section on deleting a node. After deleting the specified node, it returns the pointer to the root of the tree.

Input:

1. The number of nodes that the tree to be created should have
2. The data values of each node in the tree to be created
3. The data value in the node to be deleted

Output:

1. The data values of the nodes in the tree in inorder before deletion
2. The data values of the nodes in the tree in inorder after deletion

Question

Write a C program to count the number of non-leaf nodes of a binary tree.

6.5 Application of a Binary Search Tree

1. A prominent data structure used in many systems programming applications for representing and managing dynamic sets.
2. Average case complexity of Search, Insert, and Delete Operations is $O(\log n)$, where n is the number of nodes in the tree.

One of the applications of a binary search tree is the implementation of a dynamic dictionary. A dictionary is an ordered list which is required to be searched frequently, and is also required to be updated (insertions and deletions) frequently. Hence can be very well implemented using a binary search tree, by making the entries of dictionary into the nodes of binary search tree. A more efficient implementation of a dynamic dictionary involves considering a key to be a sequence of characters, and instead of searching by comparison of entire keys, we use these characters to determine a multi-way branch at each step, this will allow us to make a 26-way branching according the first letter, followed by another branch according to the second letter and so on.

A program to create a binary search tree, given a list of identifiers is given below:

```
char key[MAXLEN];
struct tnode
{
    key name;
    tnode *lchild;
    tnode *rchild;
}
void btree()
{
```

```

tnode *root;
key item;
int n;
root = NULL;
printf("Number of data values:");
scanf("%d", &n);
while( n > 0)
{
    printf("Enter the data value");
    scanf("%s", item);
    insert(root, item);
    n = n-1;
}
printtree(root);
}

```



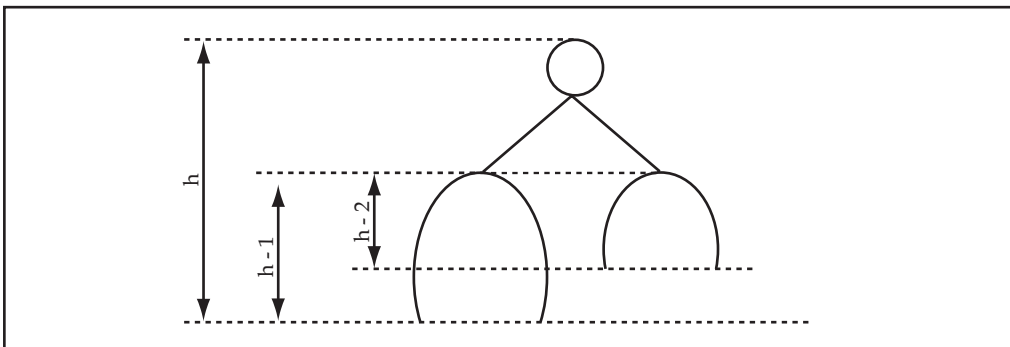
Task

Discuss how will you delete a node with one child.

6.6 AVL Tree

An AVL tree is another balanced binary search tree. It takes its name from the initials of its inventors – Adelson, Velskii and Landis. An AVL tree has the following properties:

1. The sub-trees of every node differ in height by at most one level.
2. Every sub-tree is an AVL tree.



Here, the height of the tree is h . Height of one subtree is $h-1$ while that of another subtree of the same node is $h-2$, differing from each other by just 1. Therefore, it is an AVL tree.

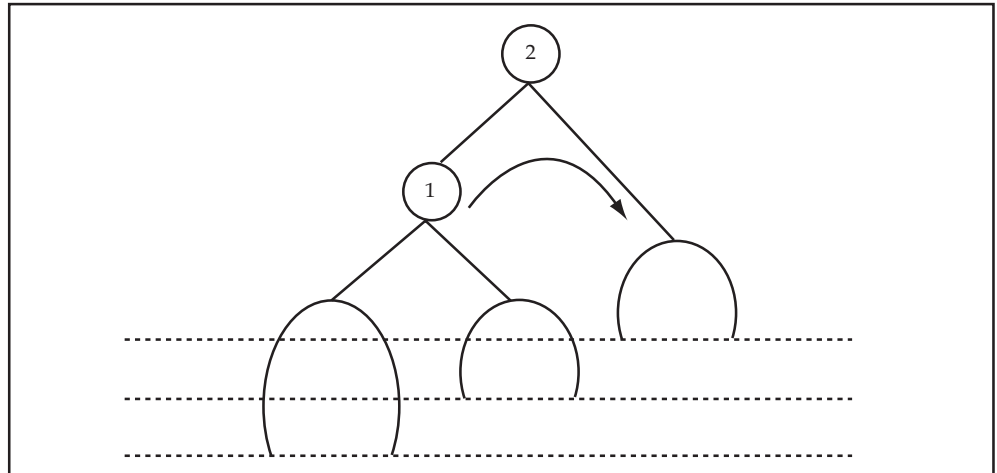
Inserting a Node into AVL Tree

Inserting a node is somewhat complex and involves a number of cases. Implementations of AVL tree insertion rely on adding an extra attribute – the balance factor – to each node. This factor indicates whether the tree is left-heavy (the height of the left sub-tree is 1 greater than the right

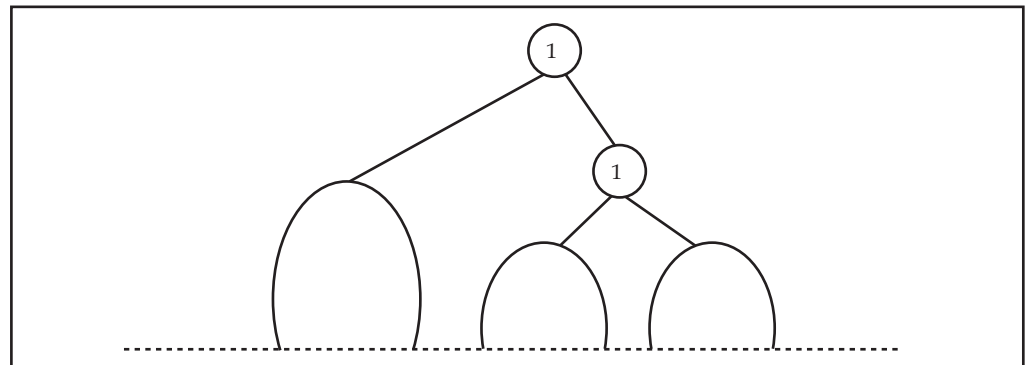
Notes

sub-tree), balanced (both sub-trees are the same height) or right-heavy (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

Let us consider the following AVL tree in which a node has been inserted in the left subtree of node 1.



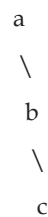
This insertion causes its height to become 2 greater than node-2's right sub-tree. A right-rotation is performed to correct the imbalance, as shown below.



AVL Rotation: A tree rotation can be an intimidating concept at first. You end up in a situation where you're juggling nodes, and these nodes have trees attached to them, and it can all become confusing very fast. I find it helps to block out what's going on with any of the subtrees which are attached to the nodes you're fumbling with, but that can be hard.

Left Rotation (LL)

Imagine we have this situation:



To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:

Notes

b becomes the new root.

a takes ownership of b's left child as its right child, or in this case, null.

b takes ownership of a as its left child.

The tree now looks like this:

```

b
 / \
a  c

```

Right Rotation (RR)

A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

```

c
 /
b
 /
a

```

To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:

b becomes the new root.

c takes ownership of b's right child, as its left child. In this case, that value is null.

b takes ownership of a, as it's right child.

The resulting tree:

```

b
 / \
a  c

```

Left-Right Rotation (LR) or "Double left"

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```

a
 \
c

```

Perfect. It's balanced. Let's insert 'b'.

```

a
 \
c

```

Notes

/

b

Our initial reaction here is to do a single left rotation. Let's try that.

c

/

a

\

b

Our left rotation has completed, and we're stuck in the same situation. If we were to do a single right rotation in this situation, we would be right back where we started. What's causing this? The answer is that this is a result of the right subtree having a negative balance. In other words, because the right subtree was left heavy, our rotation was not sufficient. What can we do? The answer is to perform a right rotation on the right subtree. Read that again. We will perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child. Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

Before:

c

/

b

After:

b

\

c

After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

a

\

b

\

c

Looks like we're ready for a left rotation. Let's do that:

b

/ \

a c

Voila. Problem solved.

Right-Left Rotation (RL) or "Double right"

Notes

A double right rotation, or right-left rotation, or simply RL, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy. This is a mirror operation of what was illustrated in the section on Left-Right Rotations, or double left rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

```

c
/
a
 \
  b

```

In this situation, we have a tree that is unbalanced. The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. What do we do? Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem. Let's try it:

```

a
 \
  c
/
b

```

Looks like that didn't work. Now we have a tree that has a balance of 2. It would appear that we did not accomplish much. That is true. What do we do? Well, let's go back to the original tree, before we did our pointless right rotation:

```

c
/
a
 \
  b

```

The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem we just witnessed. What do we do? The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:

```

c
/
b
/
a

```

Notes

This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

```

b
 / \
a   c
    
```

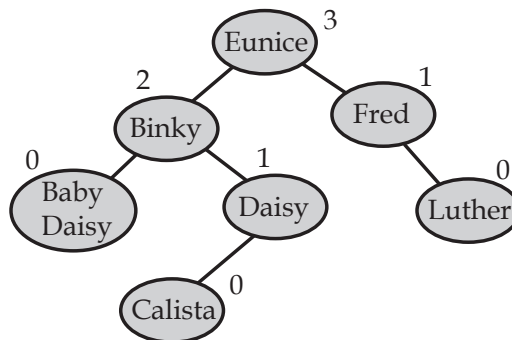
Balance at last.

AVL Operation

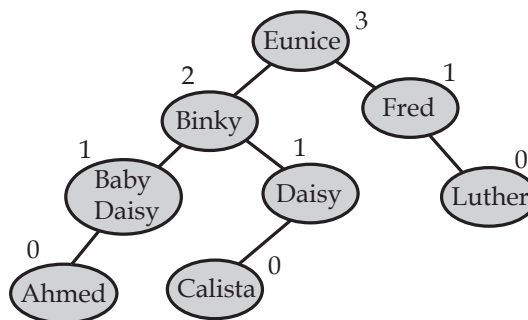
Insertion into AVL Trees

To implement AVL trees, you need to maintain the height of each node. You insert into an AVL tree by performing a standard binary tree insertion. When you're done, you check each node on the path from the new node to the root. If that node's height hasn't changed because of the insertion, then you are done. If the node's height has changed, but it does not violate the balance property, then you continue checking the next node in the path. If the node's height has changed and it now violates the balance property, then you need to perform one or two rotations to fix the problem, and then you are done.

Let's try some examples. Suppose I have the following AVL tree -- I now annotate the nodes with their heights:



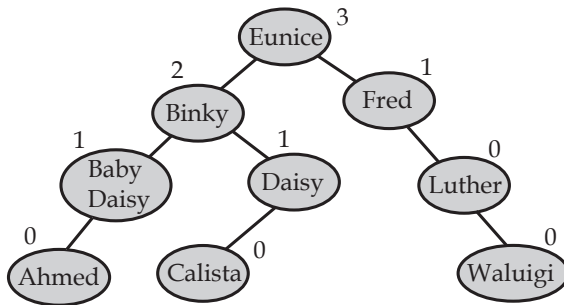
If I insert Ahmad, take a look at the resulting tree:



The new node Ahmad has a height of zero, and when I travel the path up to the root, I change Baby Daisy's height to one. However, her node is not imbalanced, since the height of her subtrees are 0 and -1. Moving on, Binky's height is unchanged, so we can stop – the resulting tree is indeed an AVL tree.

However, suppose I now try to insert Waluigi. I get the following tree:

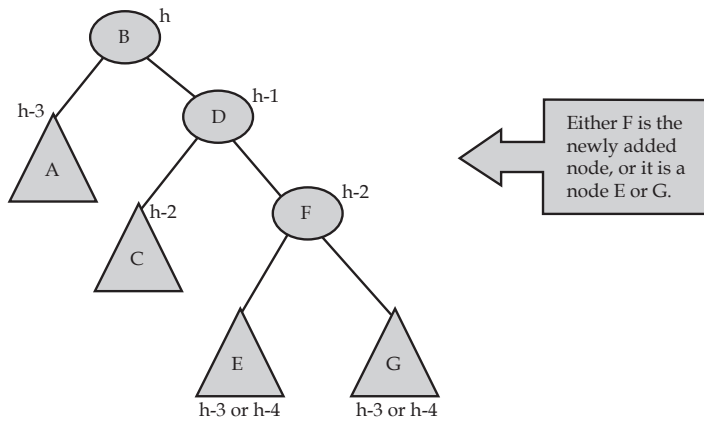
Notes



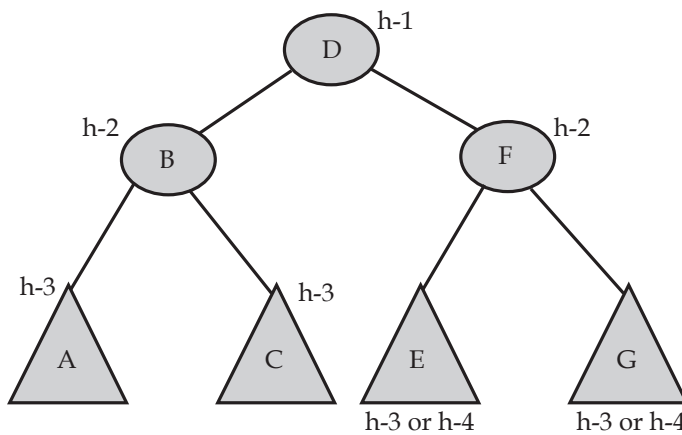
Traveling from the new node to the root, I see that Fred violates the balance condition. It's left child has a height of -1 and its right child has a height of 1. I have to rebalance the tree.

Rebalancing

When I rebalance, I have a node whose height is h , that has two children of differing heights. One child has a height of $h-1$ and the other has a height of $h-3$. The one whose height is $h-1$ contains the new node. We have to consider two cases, which I'll name zig-zig and zig-zag. You'll see the reasons for the names pretty clearly. Here is the Zig-Zig case:



In this case, the path from the imbalanced node (B) to the newly added node starts with two right children. To rebalance the tree, you perform a rotation about node D:

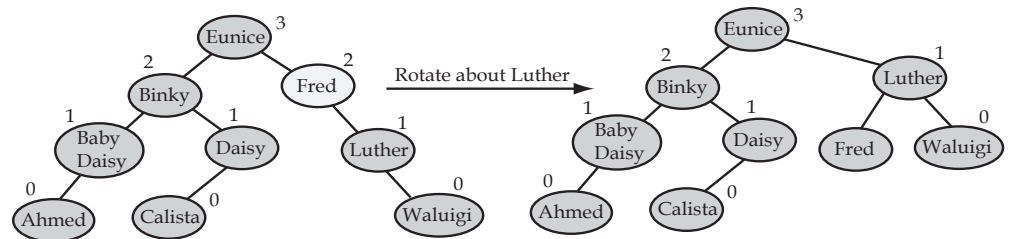


Notes

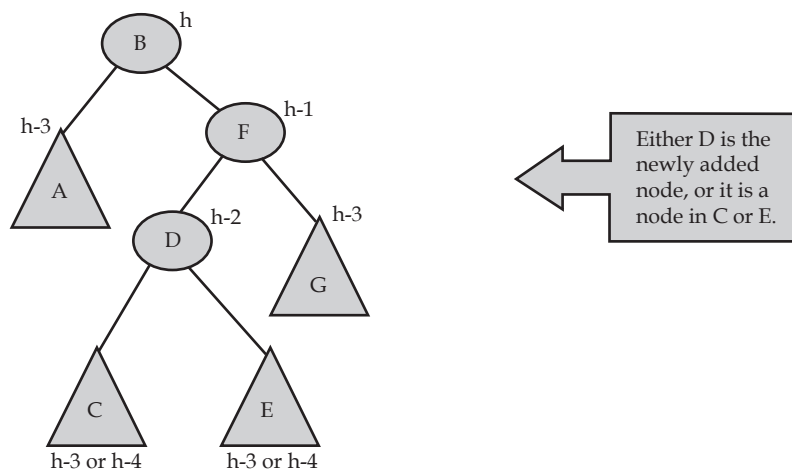
Node D is the new root of the subtree. Before the insertion, node B's height was $h-1$, so the height of the subtree has not changed because of the rotation. Thus, insertion is over, and you are left with a valid AVL tree.

If the path from the imbalanced node to the newly added node starts with two left children, then you have another Zig-Zig case (the mirror image). You treat it in the same way: rotate about the imbalanced node's left child.

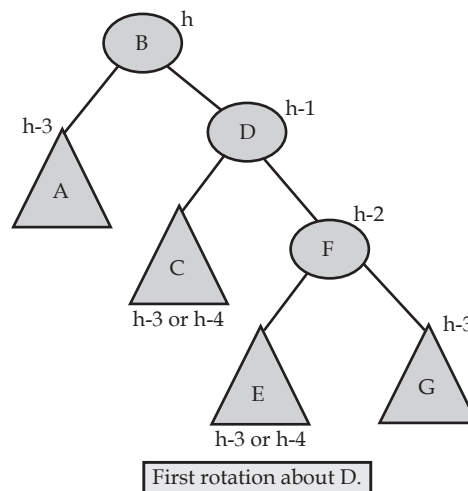
Before going on, take a look at our example above where Fred was imbalanced. That is a Zig-Zig case, so we can fix it by rotating about Luther:

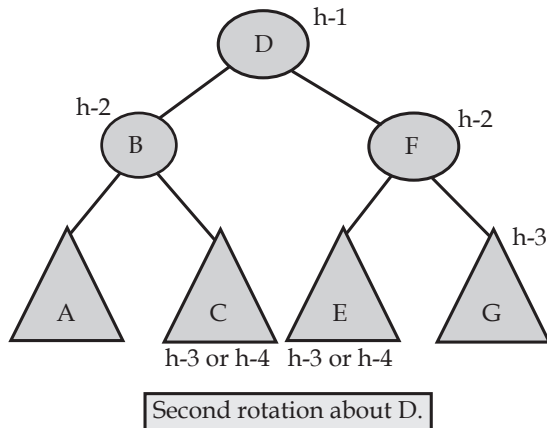


The other rebalancing case is the Zig-Zag case, pictured below:



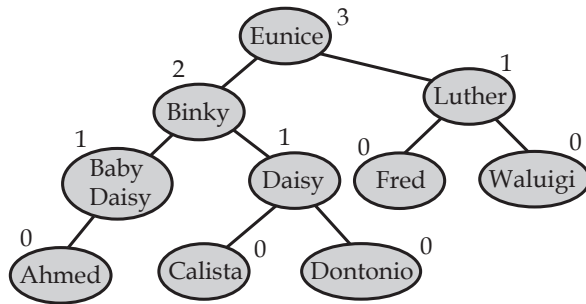
To fix this, you perform two rotations. You rotate about node D, and then you rotate about node D again. This is called a double rotation about node D. Here are the two rotations:



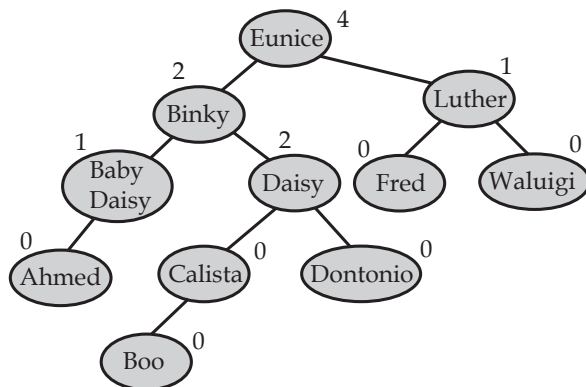


Once again, the height of the subtree before deletion was $h-1$, so when you're done with the double rotation, you are done - your tree is balanced. Again, the mirror image case is treated in the exact same manner.

Here's an example. Suppose our tree is the following, rather large tree:

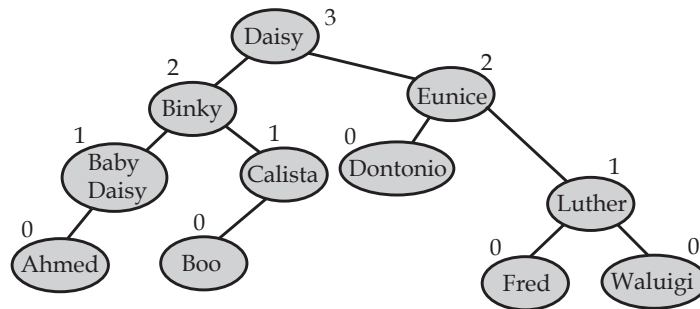


And suppose we insert Boo into the tree:



Checking for balancing, we have to increment every height up to the root, and the root node Eunice is imbalanced. Since the path to the new node starts with a left child and a right child, this is a Zig-Zag case, and we need to perform a double rotation about the grandchild of the imbalanced node -- Daisy. Below is the result. We have a nicely balanced AVL tree!

Notes



Deletion

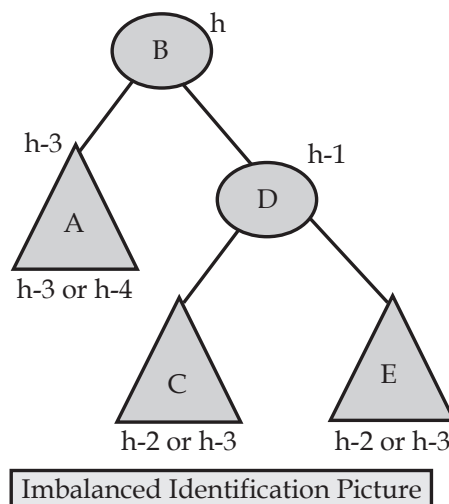
When you delete a node, there are three things that can happen to the parent:

1. Its height is decremented by one. An example of this is if Boo is deleted from the above tree - Baby Daisy's height is decremented by one.
2. Its height doesn't change and it stays balanced. An example of this is if Fred is deleted -- Luther's height is unchanged and it is balanced.
3. Its height doesn't change, but it becomes imbalanced. An example of this is if Dontonio is deleted - Eunice's height is unchanged, but she is imbalanced.

You handle these three cases in different ways:

1. The parent's height is decremented by one. When this happens, you check the parent's parent: you keep doing this until you return or you reach the root of the tree.
2. The parent's height doesn't change and it stays balanced. When this happens you may return - deletion is over.
3. The parent's height doesn't change, but it becomes imbalanced. When this happens, you have to rebalance the subtree rooted at the parent. After rebalancing, the subtree's height may be one smaller than it was originally. If so, you must continue checking the parent's parent.

To rebalance, you need to identify whether you are in a zig-zig situation or a zig-zag situation and rebalance accordingly. How do you do this? Look at the following picture:

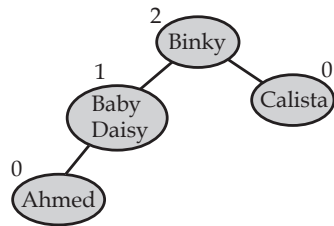


Notes

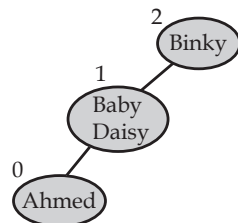
The imbalanced node is B. If the height of subtree C is $h-3$, then the height of E will be $h-2$ and the tree is a Zig-Zig – you can rebalance by rotating about node D. If the height of subtree E is $h-3$, then the height of C is $h-2$ and the tree is a Zig-Zag – you rebalance by doing a double rotation about the root of C. If both C and E have heights of $h-2$, then you treat it as either a Zig-Zig or a Zig-Zag. Both work. For the purposes of your lab, treat this case like a Zig-Zig.

The mirror image works the same way.

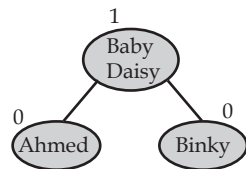
Let's look at some examples. First, suppose we delete Calista from the following tree:



You're left with:

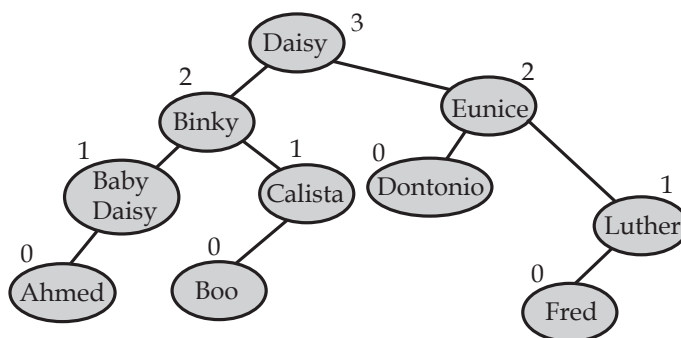


You check Calista's old parent – Binky and although Binky's height hasn't changed, the node is imbalanced. It's clearly a Zig-Zig tree, so you rotate about Baby Daisy to yield the following tree:



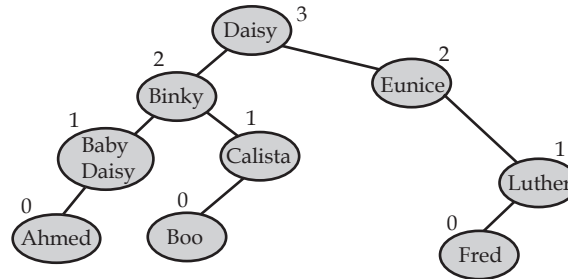
Since *Baby Daisy* is the root, we're done.

Let's try a more complex example -- deleting Eunice from the following tree:

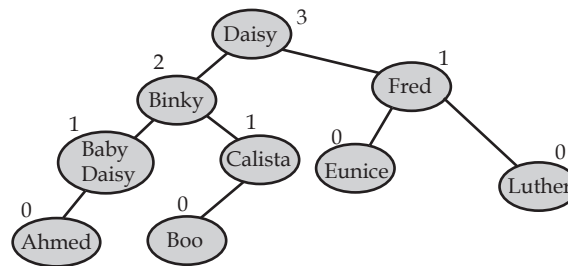


Notes

First, Eunice has two children. So, we find the child with the greatest key in Eunice's left subtree, which is Dontonio, delete it, and replace Eunice with Dontonio. We start by deleting Dontonio:



And we start checking at Eunice. It is imbalanced. Looking at it, we see that it's a Zig-Zag, so we have to double-rotate about Fred:



Now, the subtree rooted by Fred is balanced, but the subtree's height is one less than it used to be, so we need to move to its parent and check it. Its height is unchanged, and it is balanced, so we're done - as a last step, we replace Eunice with Dontonio:

AVL Tree Algorithms

```
#ifndef AVL_TREE_H
#define AVL_TREE_H
#include "dsexceptions.h"
#include <iostream> // For NULL
using namespace std;
// AvlTree class
//
// CONSTRUCTION: with ITEM_NOT_FOUND object used to signal failed finds
//
// *****PUBLIC OPERATIONS*****
// void insert( x ) --> Insert x
// void remove( x ) --> Remove x (unimplemented)
// bool contains( x ) --> Return true if x is present
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( ) --> Return largest item
// boolean isEmpty( ) --> Return true if empty; else false
// void makeEmpty( ) --> Remove all items
```



```
// void printTree( )      --> Print tree in sorted order
// *****ERRORS*****
// Throws UnderflowException as warranted
template <typename Comparable>
class AvlTree
{
public:
    AvlTree( ) : root( NULL )
        { }
    AvlTree( const AvlTree & rhs ) : root( NULL )
        {
            *this = rhs;
        }

    ~AvlTree( )
    {
        makeEmpty( );
    }

    /**
     * Find the smallest item in the tree.
     * Throw UnderflowException if empty.
     */
    const Comparable & findMin( ) const
    {
        if( isEmpty( ) )
            throw UnderflowException( );
        return findMin( root )->element;
    }

    /**
     * Find the largest item in the tree.
     * Throw UnderflowException if empty.
     */
    const Comparable & findMax( ) const
    {
        if( isEmpty( ) )
            throw UnderflowException( );
        return findMax( root )->element;
    }
}
```

Notes

```
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x, int &comps ) const
{
return contains( x, root, comps);
}

/**
 * Test if the tree is logically empty.
 * Return true if empty, false otherwise.
 */
bool isEmpty( ) const
{
    return root == NULL;
}

/**
 * Print the tree contents in sorted order.
 */
void printTree( ) const
{
    if( isEmpty( ) )
        cout << "Empty tree" << endl;
    else
        printTree( root );
}

/**
 * Make the tree logically empty.
 */
void makeEmpty( )
{
    makeEmpty( root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
```

Notes

```

void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    cout << "Sorry, remove unimplemented; " << x <<
        " still present" << endl;
}

/**
 * Deep copy.
 */
const AvlTree & operator=( const AvlTree & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        root = clone( rhs.root );
    }
    return *this;
}

private:
struct AvlNode
{
    Comparable element;
    AvlNode *left;
    AvlNode *right;
    int height;

    AvlNode( const Comparable & theElement, AvlNode *lt,
              AvlNode *rt, int h = 0 )
        : element( theElement ), left( lt ), right( rt ), height( h ) { }
};

```

Notes

```
AvlNode *root;

/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );
    }
    else
        ; // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}

/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
AvlNode * findMin( AvlNode *t ) const
```

Notes

```

{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}

/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
AvlNode * findMax( AvlNode *t ) const
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t;
}

/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the tree.
 */
bool contains( const Comparable & x, AvlNode *t, int &comps ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left, ++comps);
    else if( t->element < x )
        return contains( x, t->right, ++comps);
    else
        return true;    // Match
}

/***** NONRECURSIVE VERSION*****/
bool contains( const Comparable & x, AvlNode *t ) const
{
    while( t != NULL )

```

Notes

```
        if( x < t->element )
            t = t->left;
        else if( t->element < x )
            t = t->right;
        else
            return true;    // Match

    return false;    // No match
}

    *****/

/**
 * Internal method to make subtree empty.
 */
void makeEmpty( AvlNode * &t )
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}

/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
void printTree( AvlNode *t ) const
{
    if( t != NULL )
    {
        printTree( t->left );
        cout << t->element << endl;
        printTree( t->right );
    }
}

/**
 * Internal method to clone subtree.
```

```

*/
AvlNode * clone( AvlNode *t ) const
{
    if( t == NULL )
        return NULL;
    else
        return new AvlNode( t->element, clone( t->left ), clone( t->right ),
t->height );
}

// Avl manipulations
/**
 * Return the height of node t or -1 if NULL.
 */
int height( AvlNode *t ) const
{
    return t == NULL ? -1 : t->height;
}

int max( int lhs, int rhs ) const
{
    return lhs > rhs ? lhs : rhs;
}

/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * &k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}

/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.

```

Notes

```
* Update heights, then set new root.
*/
void rotateWithRightChild( AvlNode * & k1 )
{
    AvlNode *k2 = k1->right;
    k1->right = k2->left;
    k2->left = k1;
    k1->height = max( height( k1->left ), height( k1->right ) ) + 1;
    k2->height = max( height( k2->right ), k1->height ) + 1;
    k1 = k2;
}

/**
 * Double rotate binary tree node: first left child.
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}

/**
 * Double rotate binary tree node: first right child.
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then set new root.
 */
void doubleWithRightChild( AvlNode * & k1 )
{
    rotateWithLeftChild( k1->right );
    rotateWithRightChild( k1 );
}
};

#endif
```


Applications of AVL Trees

Notes

AVL trees are applied in the following situations:

1. There are few insertion and deletion operations
2. Short search time is needed
3. Input data is sorted or nearly sorted

AVL tree structures can be used in situations which require fast searching. But, the large cost of rebalancing may limit the usefulness.

Consider the following:

1. A classic problem in computer science is how to store information dynamically so as to allow for quick look up. This searching problem arises often in dictionaries, telephone directory, symbol tables for compilers and while storing business records etc. The records are stored in a balanced binary tree, based on the keys (alphabetical or numerical) order. The balanced nature of the tree limits its height to $O(\log n)$, where n is the number of inserted records.
2. AVL trees are very fast on searches and replacements. But, have a moderately high cost for addition and deletion. If application does a lot more searches and replacements than it does addition and deletions, the balanced (AVL) binary tree is a good choice for a data structure.
3. AVL tree also has applications in file systems.

6.7 Summary

- Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary tree, we shall find that we can search for a target key in $O(\log n)$ steps, just as with binary search, and we shall obtain algorithms for inserting and deleting entries also in time $O(\log n)$
- When we studied binary search, we drew comparison trees showing the progress of binary search by moving either left (if the target key is smaller than the one in the current node of the tree) or right (if the target key is larger).
- We can regard binary search trees as a new abstract data type with its own definition and its own methods;
- Since binary search trees are special kinds of binary trees, we may consider their methods as special kinds of binary tree methods;
- Since the entries in binary search trees contain keys, and since they are applied for information retrieval in the same way as ordered lists, we may study binary search trees as a new implementation of the abstract data type ordered list.

6.8 Keywords

Binary Search Tree: A binary search tree is a binary tree which may be empty, and every node contains an identifier.

Searching: Searching for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node.

Notes

6.9 Self Assessment

State whether the following statements are true or false:

1. A binary search tree is not a binary tree which may be full, and every node contains an identifier.
2. A dictionary is an ordered list which is required to be searched frequently, and is also required to be updated frequently.
3. AVL Tree invented by Adelson, Velskii and Landis.
4. Implementations of AVL tree insertion rely on deleting an extra attribute - the balance factor - to each node.
5. Identifier of any node in the right sub-tree is greater than the identifier of the root and the left sub-tree as well as right sub-tree both are binary search trees.

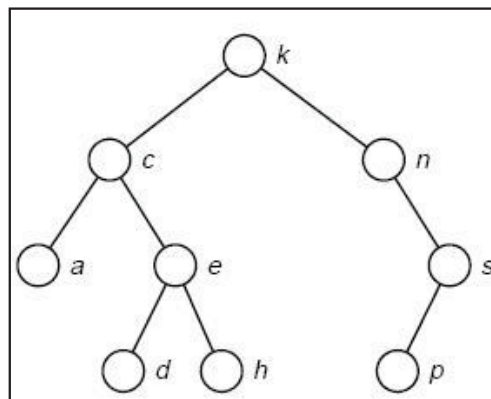
Fill in the blanks:

6. A binary search tree is basically a binary tree, and therefore it can be traversed is inorder, preorder, and
7. To create a we use a procedure named insert which creates a new node with the data value supplied as a parameter to it, and inserts into an already existing tree whose root pointer is also passed as a parameter.
8. To from a binary search tree the method to be used depends on whether a node to be deleted has one child, two children, or has no child.
9. To search the binary tree for a particular node, we use procedures similar to those we used when to it.
10. A more efficient implementation of a dynamic dictionary involves considering a key to be a sequence of

6.10 Review Questions

1. Describe binary search tree. Also explain how will you create binary search tree.
2. How will counting the number of nodes in a binary search tree? Explain
3. Describe deletion of a node with one child.

Question 4-6 are based on the following binary search tree. Answer each question independently, using the original tree as the basis for each part.



4. Show the keys with which each of the following targets will be compared in a search of the preceding binary search tree.
- | | | |
|------|------|------|
| a. c | b. s | c. k |
| d. a | e. d | f. m |
| g. f | h. b | i. t |
5. Insert each of the following keys into the preceding binary search tree. Show the comparisons of keys that will be made in each case. Do each part independently, inserting the key into the original tree.
- | | | |
|------|------|------|
| a. m | b. f | c. b |
| d. t | e. c | f. s |
6. Delete each of the following keys from the preceding binary search tree, using the algorithm developed in this section. Do each part independently, deleting the key from the original tree.
- | | | |
|------|------|------|
| a. a | b. p | c. n |
| d. s | e. e | f. k |
7. Draw the binary search trees that function insert will construct for the list of 14 names presented in each of the following orders and inserted into a previously empty binary search tree.
- Jan Guy Jon Ann Jim Eva Amy Tim Ron Kim Tom Roy Kay Dot
 - Amy Tom Tim Ann Roy Dot Eva Ron Kim Kay Guy Jon Jan Jim
 - Jan Jon Tim Ron Guy Ann Jim Tom Amy Eva Roy Kim Dot Kay
 - Jon Roy Tom Eva Tim Kim Ann Ron Jan Amy Dot Guy Jim Kay
8. Consider building two binary search trees containing the integer keys 1 to 63, inclusive, received in the orders
- All the odd integers in order (1, 3, 5, ..., 63), then 32, 16, 48, then the remaining even integers in order (2, 4, 6, ...).
 - 32, 16, 48, then all the odd integers in order (1, 3, 5, ..., 63), then the remaining even integers in order (2, 4, 6, ...).
- Which of these trees will be quicker to build? Explain why. [Try to answer this question without actually drawing the trees.]
9. Prepare a package containing the declarations for a binary search tree and the functions developed in this section. The package should be suitable for inclusion in any application program.
10. In each of the following, insert the keys, in the order shown, to build them into an AVL tree.
- A, Z, B, Y, C, X.
 - A, B, C, D, E, F.
 - M, T, E, A, Z, G, P.
 - A, Z, B, Y, C, X, D, W, E, V, F.
 - A, B, C, D, E, F, G, H, I, J, K, L.
 - A, V, L, T, R, E, I, S, O, K.

Notes

Answers: Self Assessment

- | | |
|-----------------------|------------------|
| 1. False | 2. True |
| 3. True | 4. False |
| 5. True | 6. postorder |
| 7. binary search tree | 8. delete a node |
| 9. adding | 10. characters |

6.11 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*, Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 7: Splay Trees

Notes

CONTENTS

Objectives

Introduction

7.1 Splay Trees

7.2 Operations

7.3 Splay Operation

7.4 Rotations

7.5 Splay Tree Application

7.6 Summary

7.7 Keywords

7.8 Self Assessment

7.9 Review Questions

7.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe splay trees
- Explain splay tree operation

Introduction

Splay trees are binary search trees that achieve our goals by being self-adjusting in a quite remarkable way: Every time we access a node of the tree, whether for insertion or retrieval, we perform radical surgery on the tree, lifting the newly accessed node all the way up, so that it becomes the root of the modified tree. Other nodes are pushed out of the way as necessary to make room for this new root. Nodes that are frequently accessed will frequently be lifted up to become the root, and they will never drift too far from the top position. Inactive nodes, on the other hand, will slowly be pushed farther and farther from the root.

It is possible that splay trees can become highly unbalanced, so that a single access to a node of the tree can be quite expensive. Later in this section, however, we shall prove that, over a long sequence of accesses, splay trees are not at all expensive and are guaranteed to require not many more operations even than AVL trees. The analytical tool used is called amortized algorithm analysis, since, like insurance calculations, the few expensive cases are averaged in with many less expensive cases to obtain excellent performance over a long sequence of operations.

7.1 Splay Trees

Splay Trees were invented by Sleator and Tarjan. This data structure is essentially a binary tree with special update and access rules. It has the property to adapt optimally to a sequence of tree operations. More precisely, a sequence of m operations on a tree with initially n nodes takes time $O(n \ln(n) + m \ln(n))$.

7.2 Operations

Splay trees support the following operations. We write S for sets, x for elements and k for key values.

$\text{splay}(S, k)$ returns an access to an element x with key k in the set S . In case no such element exists, we return an access to the next smaller or larger element.

$\text{split}(S, k)$ returns (S_1, S_2) , where for each x in S_1 holds: $\text{key}[x] \leq k$, and for each y in S_2 holds: $k < \text{key}[y]$.

$\text{join}(S_1, S_2)$ returns the union $S = S_1 + S_2$. Condition: for each x in S_1 and each y in S_2 : $x \leq y$.

$\text{insert}(S, x)$ augments S by x .

$\text{delete}(S, x)$ removes x from S .

Each split, join, delete and insert operation can be reduced to splay operations and modifications of the tree at the root which take only constant time. Thus, the run time for each operation is essentially the same as for a splay operation.

7.3 Splay Operation

The most important tree operation is $\text{splay}(x)$, which moves an element x to the root of the tree. In case x is not present in the tree, the last element on the search path for x is moved instead.

The run time for a $\text{splay}(x)$ operation is proportional to the length of the search path for x . While searching for x we traverse the search path top-down. Let y be the last node on that path. In a second step, we move y along that path by applying rotations as described later.

The time complexity of maintaining a splay tree is analyzed using an Amortized Analysis. Consider a sequence of operations op_1, op_2, \dots, op_m . Assume that our data structure has a potential. One can think of the potential as a bank account. Each tree operation op_i has actual costs proportional to its running time. We're paying for the costs c_i of op_i with its amortized costs a_i . The difference between concrete and amortized costs is charged against the potential of the data structure. This means that we're investing in the potential if the amortized costs are higher than the actual costs, otherwise we're decreasing the potential.

Thus, we're paying for the sequence op_1, op_2, \dots, op_m no more than the initial potential plus the sum of the amortized costs $a_1 + a_2 + \dots + a_m$.

The trick of the analysis is to define a potential function and to show that each splay operation has amortized costs $O(\ln n)$. It follows that the sequence has costs $O(m \ln n) + n \ln n$.

7.4 Rotations

The splay operation moves the accessed element x to the root of the tree T . This is done using rotations on x and parent y and grandparent z .

There are two kinds of double rotations and one single rotation. Due to symmetry, we need mirror-image versions of each rotation.

Type 1: $x < y < z$, or $z < y < x$ respectively

Type 2: $y < x < z$, or $z < x < y$ respectively.

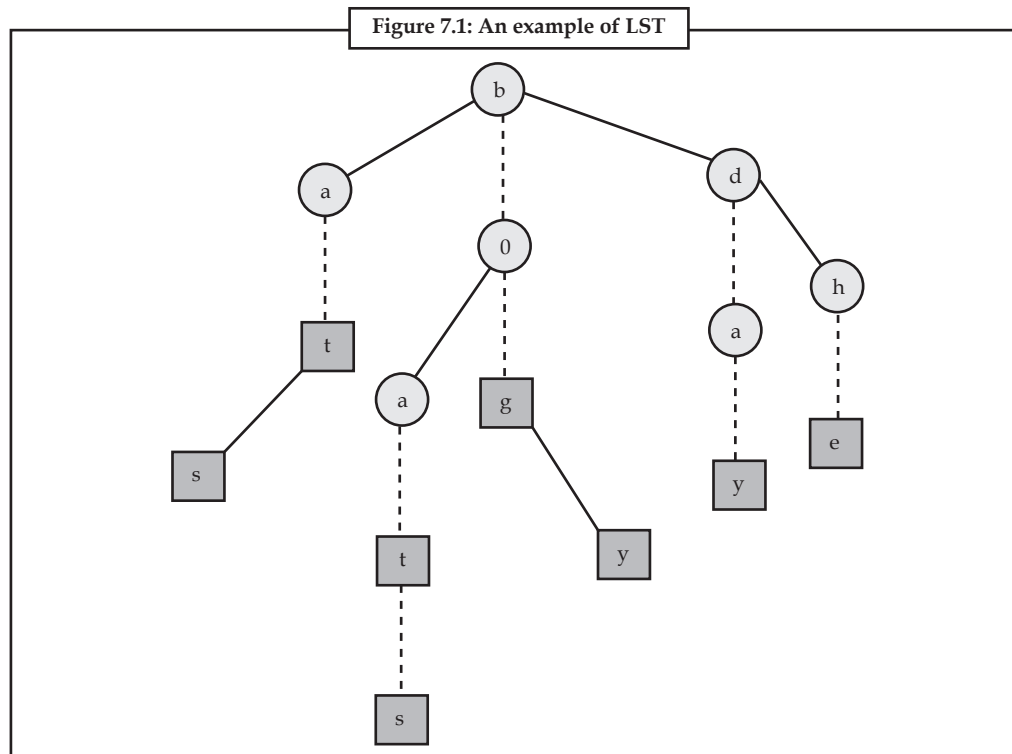
Type 3: The last case deals with the situation that the splay node x is a child of the root. Thus, we need a single rotation.

$x < y$, or $y < x$ respectively.

7.5 Splay Tree Application

Notes

The application is Lexicographic Search Tree (LST).



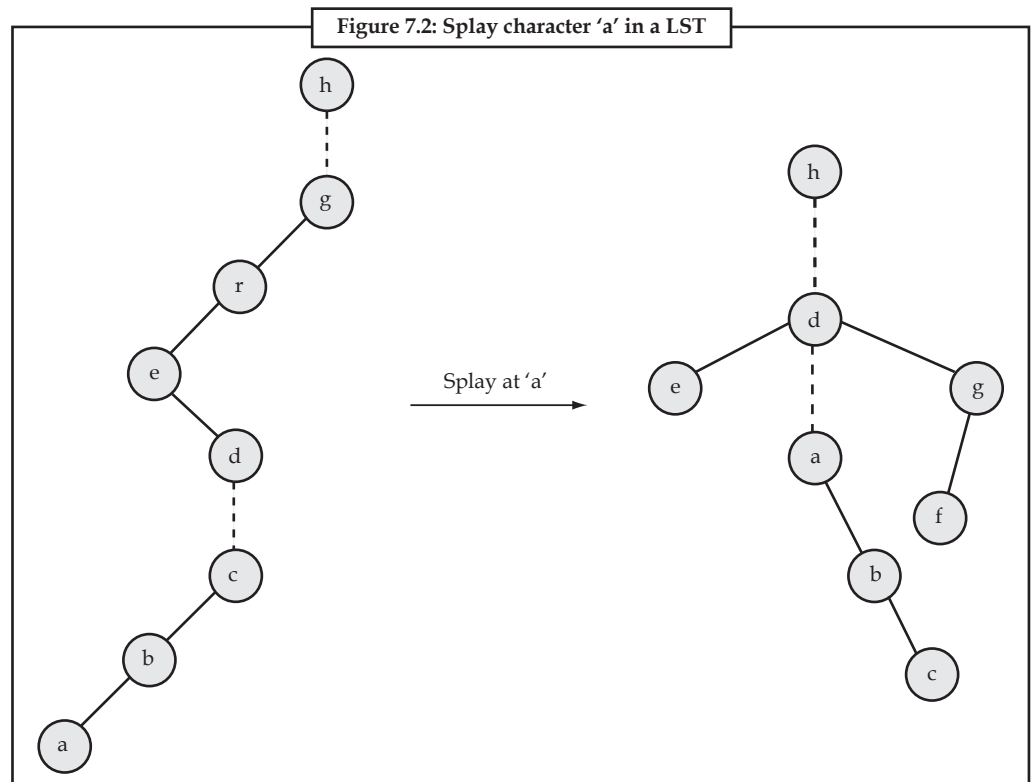
In a LST all the circles and squares are the nodes of the tree. Each node will contain a character. And the square nodes are terminal nodes of the strings. There are two kinds of edges, solid line edges and dashed line edges in LST. Solid line edges forms the splay tree and the nodes connected by a solid line represent different strings. The dashed line edges are used to represent a single string therefore the order of the nodes connected by a dashed line cannot be changed.

Figure 7.1 shows an example of a LST. From the Figure 7.1, it stores different words: "at", "as", "bat", "bats", "bog", "boy", "day" and "he".

When a string is requested you just splay the character in the string one by one. Figure 7.2 shows how to splay a character 'a' in a LST.

As you can see, the character 'a' will not be splayed to the root because 'h' and 'd' are the prefix of 'a' but it will move to the position most near the root. In LST, after the string is accessed, the first character of the string will become the root as a result the most frequent accessed string will be near the root. And LST store the prefix for different strings this reduce the redundancy for storing the strings. Besides LST, the splay tree can also be used for data compression, e.g. dynamic Huffman coding.

Notes



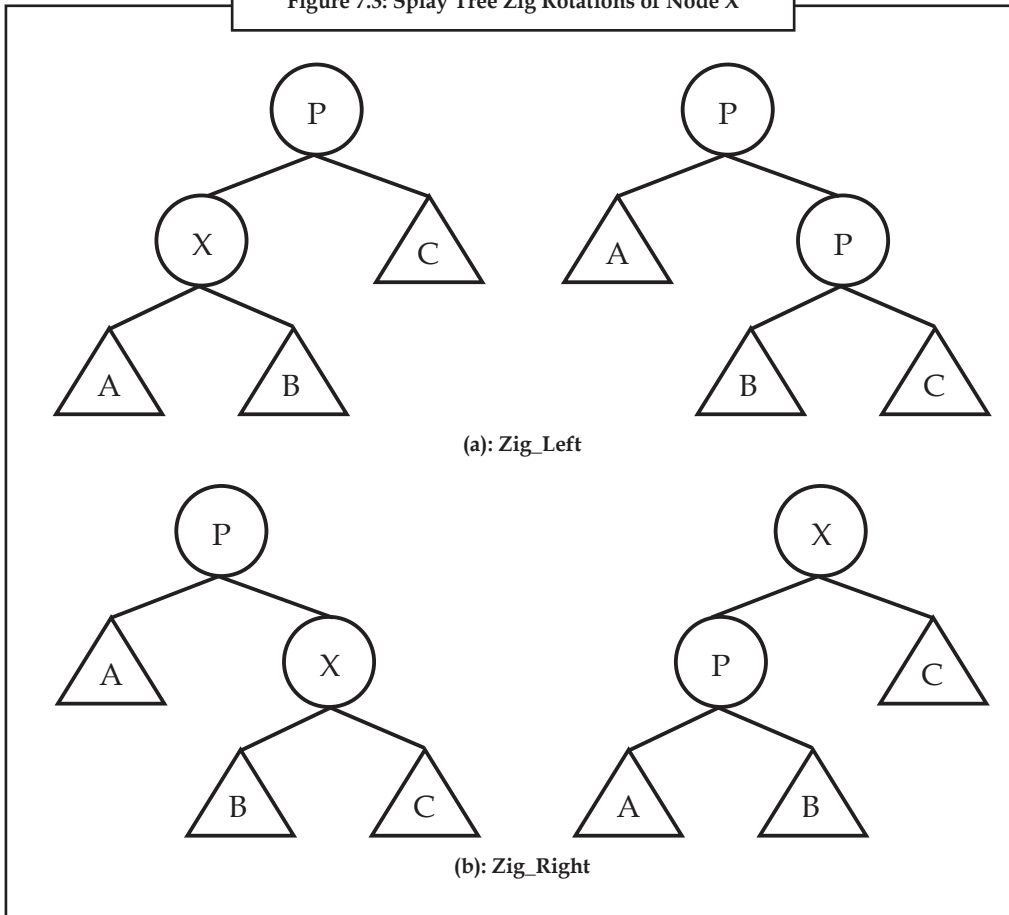
Rules for Splaying

Here are the rules for splaying. Denote the node being accessed as X, the parent of X as P and the grandparent of X as G:

Zig: In this case, P is the root of the tree.

1. Zig_Left(X) if X is a left child.
2. Zig_Right(X) if X is a right child.

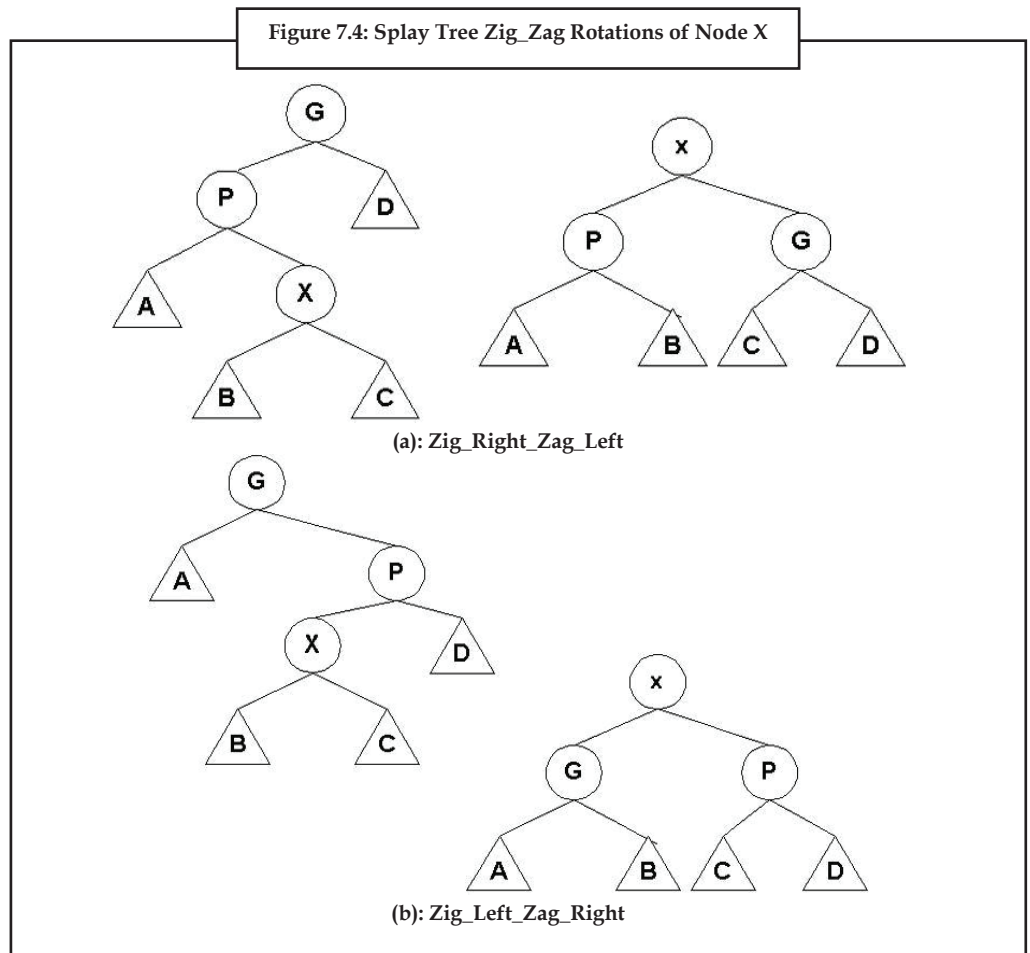
Figure 7.3: Splay Tree Zig Rotations of Node X



ZigZag: P is not the root of the tree. X and P are opposite types of children (left or right). The path from X to G is “crooked”.

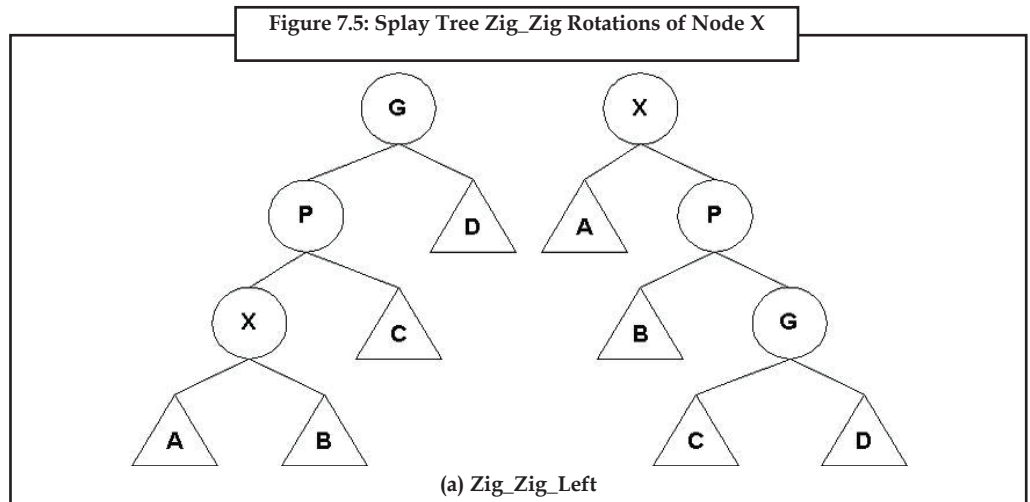
1. Zig_Right_Zag_Left if X is a right child and P is a left child.
2. Zig_Left_Zag_Right if X is a left child and P is a right child.

Notes

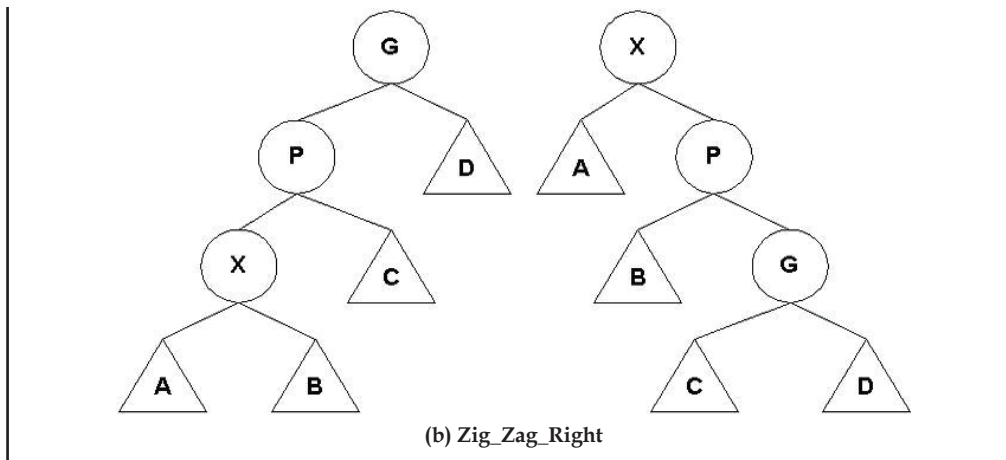


ZigZig: P is not the root of the tree. X and P are the same type of children (left or right). X, P and G are in a straight line.

1. Zig_Zig_Left if X and P are both right children.
2. Zig_Zig_Right if X and P are both left children.



Contd....



Splay Tree Algorithms

To show that splay trees deliver the promised amortized performance, we define a potential function $\Phi(T)$ to keep track of the extra time that can be consumed by later operations on the tree T . As before, we define the amortized time taken by a single tree operation that changes the tree from T to T' as the actual time t , plus the change in potential $\Phi(T') - \Phi(T)$. Now consider a sequence of M operations on a tree, taking actual times $t_1, t_2, t_3, \dots, t_M$ and producing trees T_1, T_2, \dots, T_M . The amortized time taken by the operations is the sum of the actual times for each operation plus the sum of the changes in potential: $t_1 + t_2 + \dots + t_M + (\Phi(T_2) - \Phi(T_1)) + (\Phi(T_3) - \Phi(T_2)) + \dots + (\Phi(T_M) - \Phi(T_{M-1})) = t_1 + t_2 + \dots + t_M + \Phi(T_M) - \Phi(T_1)$. Therefore the amortized time for a sequence of operations underestimates the actual time by at most the maximum drop in potential $\Phi(T_M) - \Phi(T_1)$ seen over the whole sequence of operations.

The key to amortized analysis is to define the right potential function. Given a node x in a binary tree, let $\text{size}(x)$ be the number of nodes below x (including x). Let $\text{rank}(x)$ be the log base 2 of $\text{size}(x)$. Then the potential $\Phi(T)$ of a tree T is the sum of the ranks of all of the nodes in the tree. Note that if a tree has n nodes in it, the maximum rank of any node is $\lg n$, and therefore the maximum potential of a tree is $n \lg n$. This means that over a sequence of operations on the tree, its potential can decrease by at most $n \lg n$. So the correction factor to amortized time is at most $n \lg n$, which is good.

Now, let us consider the amortized time of an operation. The basic operation of splay trees is splaying; it turns out that for a tree t , any splaying operation on a node x takes at most amortized time $3 \cdot \text{rank}(t) + 1$. Since the rank of the tree is at most $\lg(n)$, the splaying operation takes $O(\lg n)$ amortized time. Therefore, the actual time taken by a sequence of n operations on a tree of size n is at most $O(\lg n)$ per operation.

To obtain the amortized time bound for splaying, we consider each of the possible rotation operations, which take a node x and move it to a new location. We consider that the rotation operation itself takes time $t = 1$. Let $r(x)$ be the rank of x before the rotation, and $r'(x)$ the rank of node x after the rotation. We will show that simple rotation takes amortized time at most $3(r'(x) - r(x)) + 1$, and that the other two rotations take amortized time $3(r'(x) - r(x))$. There can be only one simple rotation (at the top of the tree), so when the amortized time of all the rotations performed during one splaying is added, all the intermediate terms $r(x)$ and $r'(x)$ cancel out and we are left with $3(r(t) - r(x)) + 1$. In the worst case where x is a leaf and has rank 0, this is equal to $3 \cdot r(t) + 1$.

Notes

Simple Rotation

The only two nodes that change rank are x and y . So the cost is $1 + r'(x) - r(x) + r'(y) - r(y)$. Since y decreases in rank, this is at most $1 + r'(x) - r(x)$. Since x increases in rank, $r'(x) - r(x)$ is positive and this is bounded by $1 + 3(r'(x) - r(x))$.

Zig-Zig Rotation

Only the nodes x , y , and z change in rank. Since this is a double rotation, we assume it has actual cost 2 and the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Since the new rank of x is the same as the old rank of z , this is equal to

$$2 - r(x) + r'(y) - r(y) + r'(z)$$

The new rank of x is greater than the new rank of y , and the old rank of x is less than the old rank y , so this is at most

$$2 - r(x) + r'(x) - r(x) + r'(z) = 2 + r'(x) - 2r(x) + r'(z)$$

Now, let $s(x)$ be the old size of x and let $s'(x)$ be the new size of x . Consider the term $2r'(x) - r(x) - r'(z)$. This must be at least 2 because it is equal to $\lg(s'(x)/s(x)) + \lg(s'(x)/s'(z))$. Notice that this is the sum of two ratios where $s'(x)$ is on top. Now, because $s'(x) \geq s(x) + s'(z)$, the way to make the sum of the two logarithms as small as possible is to choose $s(x) = s'(z) = s'(x)/2$. But in this case the sum of the logs is $1 + 1 = 2$. Therefore the term $2r'(x) - r(x) - r'(z)$ must be at least 2. Substituting it for the red 2 above, we see that the amortized time is at most

$$(2r'(x) - r(x) - r'(z)) + r'(x) - 2r(x) + r'(z) = 3(r'(x) - r(x)) \text{ as required.}$$

Zig-Zag Rotation

Again, the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Because the new rank of x is the same as the old rank of z , and the old rank of x is less than the old rank of y , this is

$$\begin{aligned} & 2 - r(x) + r'(y) - r(y) + r'(z) \\ & \leq 2 - 2r(x) + r'(y) + r'(z) \end{aligned}$$

Now consider the term $2r'(x) - r'(y) - r'(z)$. By the same argument as before, this must be at least 2, so we can replace the constant 2 above while maintaining a bound on amortized time:

$$\leq (2r'(x) - r'(y) - r'(z)) - 2r(x) + r'(y) + r'(z) = 2(r'(x) - r(x))$$

Therefore amortized run time in this case too is bounded by $3(r'(x) - r(x))$, and this completes the proof of the amortized complexity of splay tree operations.

7.6 Summary

- Splay trees are self-adjusting binary search trees in which every access for insertion or retrieval of a node, lifts that node all the way up to become the root, pushing the other nodes out of the way to make room for this new root of the modified tree. Hence, the frequently accessed nodes will frequently be lifted up and remain around the root position; while the most infrequently accessed nodes would move farther and farther away from the root.

7.7 Keywords

Notes

Access Time: The time required to access and retrieve a word from high-speed memory is a few microseconds at most.

Splay Trees: Splay trees are binary search trees which are self adjusting.

7.8 Self Assessment

Choose the appropriate answer:

1. Splay Trees were invented by
 - (a) Sleator
 - (b) Tarjan
 - (c) Newton
 - (d) Both (a) and (b)
2. The run time for each operation is essentially the same as for a
 - (a) Splay operation.
 - (b) Zero operation
 - (c) Play operation
 - (d) None of the above

State whether the following statements are True or False:

3. The time complexity of maintaining a splay tree is analyzed using an Amortized Analysis.
4. Each node does not contain a character.

Fill in the blanks:

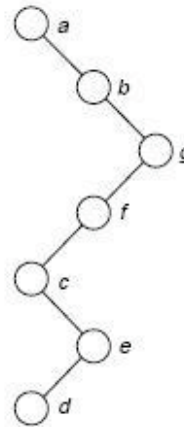
5. The of maintaining a splay tree is analyzed using an Amortized Analysis.
6. The trick of the analysis is to define a potential function and to show that each splay operation has
7. The key to is to define the right potential function.
8. There are two kinds of rotations and one single rotation.

7.9 Review Questions

1. Explain splay operation in splay trees.
2. "The time complexity of maintaining a splay tree is analyzed using an Amortized Analysis." Explain
3. "A splay tree does not keep track of heights and does not use any balance factors like an AVL tree". Explain

Notes

4. Consider the following binary search tree:



Splay this tree at each of the following keys in turn:

d b g f a d b d

Each part builds on the previous; that is, use the final tree of each solution as the starting tree for the next part.

5. “Splay trees are binary search trees that achieve our goals by being self-adjusting in a quite remarkable way”. Discuss
6. “It is possible that splay trees can become highly unbalanced, so that a single access to a node of the tree can be quite expensive”. Explain

Answers: Self Assessment

- | | |
|-----------------------|--------------------------------|
| 1. (d) | 2. (a) |
| 3. True | 4. True |
| 5. time complexity | 6. amortized costs $O(\ln(n))$ |
| 7. amortized analysis | 8. double |

7.10 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E. Leiserson & Ronald L. Rivest: *Introduction to Algorithms*, Prentice-Hall of India Pvt. Limited, New Delhi.

Notes

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 8: B-trees

CONTENTS

Objectives

Introduction

8.1 B-trees

8.2 Structure of B-trees

8.3 Height of B-trees

8.4 Operations on B-trees

8.5 Inserting a New Item

8.6 B-tree-Deleting an Item

8.7 B-tree Algorithms

8.8 Applications

8.9 Summary

8.10 Keywords

8.11 Self Assessment

8.12 Review Questions

8.13 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the concept of B-trees
- Describe B-trees algorithms

Introduction

The key factors which have been discussed in this unit about the b-trees in data structures. While dealing with many problems in computer science, engineering and many other disciplines, it is needed to impose a hierarchical structure on a collection of data items. For example, we need to impose a hierarchical structure on a collection of data items while preparing organizational charts and genealogies, to represent the syntactic structure of source programs in compilers.

A B-tree is a tree data structure that keeps data sorted and allows insertions and deletions that is logarithmically proportional to file size. It is commonly used in databases and file systems.

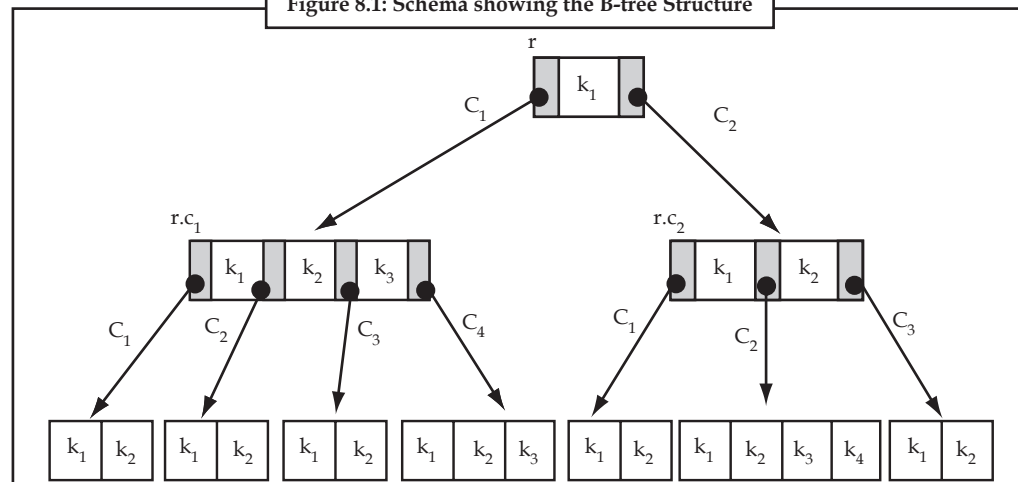
8.1 B-trees

In B-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self balancing

search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation.

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

Figure 8.1: Schema showing the B-tree Structure



B-trees are balanced trees that are optimized for situations when part or the entire tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses.



Example: A b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node.

8.2 Structure of B-trees

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minimum number of allowable children for each node known as the minimization factor. If t is this minimization factor, every node must have at least $t - 1$ keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than $t - 1$ keys. Every node may have at most $2t - 1$ keys or, equivalently, $2t$ children.

Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimization factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

8.3 Height of B-trees

For n greater than or equal to one, the height of an n -key b -tree T of height h with a minimum degree t greater than or equal to 2,

$$h \leq \log_t \frac{n+1}{2}$$

The worst case height is $O(\log n)$. Since the “branchiness” of a b -tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b -trees tend to have smaller heights than other trees with the same asymptotic height.

8.4 Operations on B-trees

The algorithms for the search, create, and insert operations are shown below. Note that these algorithms are single pass; in other words, they do not traverse back up the tree. Since b -trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses. Simpler double-pass approaches that move back up the tree to fix violations are possible.

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be preceded by a read operation denoted by Disk-Read. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by Disk-Write. The algorithms below assume that all nodes referenced in parameters have already had a corresponding Disk-Read operation. New nodes are created and assigned storage with the Allocate-Node call. The implementation details of the Disk-Read, Disk-Write, and Allocate-Node functions are operating system and implementation dependent.

```
B-Tree-Search(x, k)
i ← 1
while i ≤ n[x] and k > keyi[x]
    do i ← i + 1
if i ≤ n[x] and k = keyi[x]
    then return (x, i)
if leaf[x]
    then return NIL
else Disk-Read(ci[x])
return B-Tree-Search(ci[x], k)
```

The search operation on a b -tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b -tree search must make an n -way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, B-Tree-Search is $O(\log t n)$.

```
B-Tree-Create(T)
x ← Allocate-Node()
```

Notes

```
leaf[x] <- TRUE
n[x] <- 0
Disk-Write(x)
root[T] <- x
```

The B-Tree-Create operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The B-Tree-Create operation runs in time $O(1)$.

```
B-Tree-Split-Child(x, i, y)
z <- Allocate-Node()
leaf[z] <- leaf[y]
n[z] <- t - 1
for j <- 1 to t - 1
  do keyj[z] <- keyj+t[y]
if not leaf[y]
  then for j <- 1 to t
    do cj[z] <- cj+t[y]
n[y] <- t - 1
for j <- n[x] + 1 downto i + 1
  do cj+1[x] <- cj[x]
ci+1 <- z
for j <- n[x] downto i
  do keyj+1[x] <- keyj[x]
keyi[x] <- keyt[y]
n[x] <- n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

If a node becomes “too full,” it is necessary to perform a split operation. The split operation moves the median key of node x into its parent y where x is the i th child of y . A new node, z , is allocated, and all keys in x right of the median key are moved to z . The keys left of the median key remain in the original node x . The new node, z , becomes the child immediately to the right of the median key that was moved to the parent y , and the original node, x , becomes the child immediately to the left of the median key that was moved into the parent y .

The split operation transforms a full node with $2t - 1$ keys into two nodes with $t - 1$ keys each.



Note One key is moved into the parent node. The B-Tree-Split-Child algorithm will run in time $O(t)$ where t is constant.

```
B-Tree-Insert(T, k)
r <- root[T]
if n[r] = 2t - 1
```

Notes


```

then s <- Allocate-Node()
    root[T] <- s
    leaf[s] <- FALSE
    n[s] <- 0
    c1 <- r
    B-Tree-Split-Child(s, 1, r)
    B-Tree-Insert-Nonfull(s, k)
else B-Tree-Insert-Nonfull(r, k)
B-Tree-Insert-Nonfull(x, k)
i <- n[x]
if leaf[x]
    then while i >= 1 and k < keyi[x]
        do keyi+1[x] <- keyi[x]
        i <- i - 1
        keyi+1[x] <- k
    n[x] <- n[x] + 1
    Disk-Write(x)
else while i >= and k < keyi[x]
    do i <- i - 1
    i <- i + 1
    Disk-Read(ci[x])
    if n[ci[x]] = 2t - 1
        then B-Tree-Split-Child(x, i, ci[x])
        if k > keyi[x]
            then i <- i + 1
    B-Tree-Insert-Nonfull(ci[x], k)

```

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similar to B-Tree-Search. Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unnecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree.



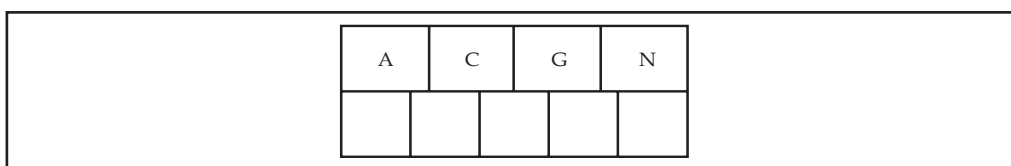
Task In twenty words or less, explain how treesort works.

8.5 Inserting a New Item

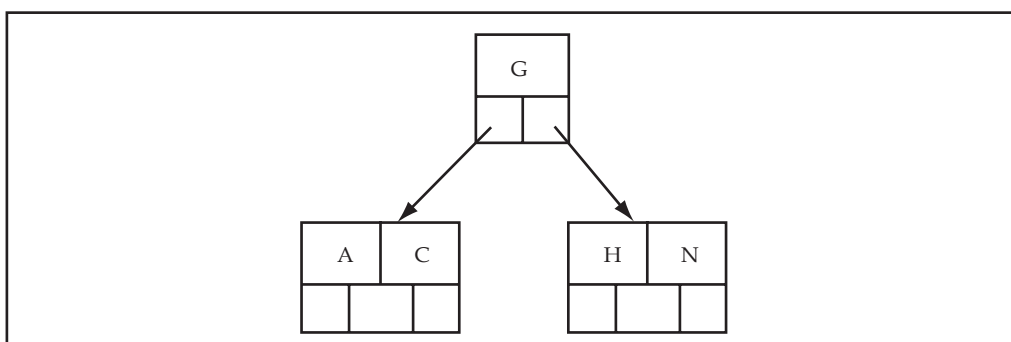
Notes

When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.) Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

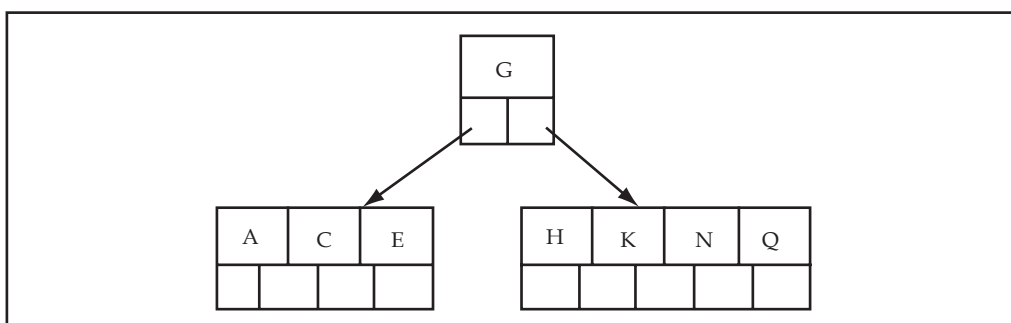
Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.

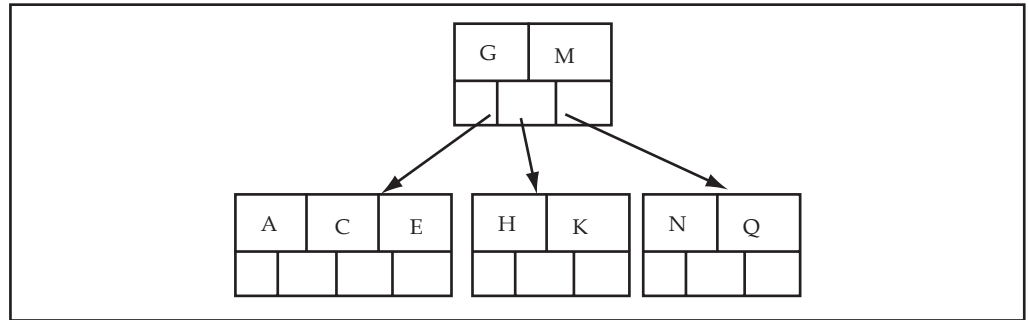


Inserting E, K, and Q proceeds without requiring any splits:

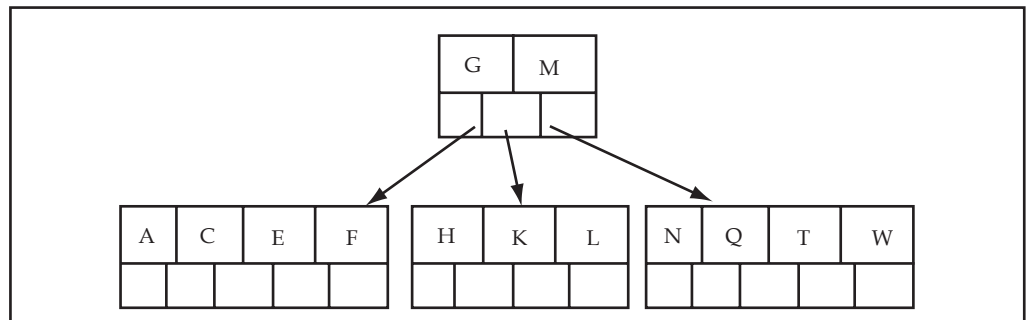


Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.

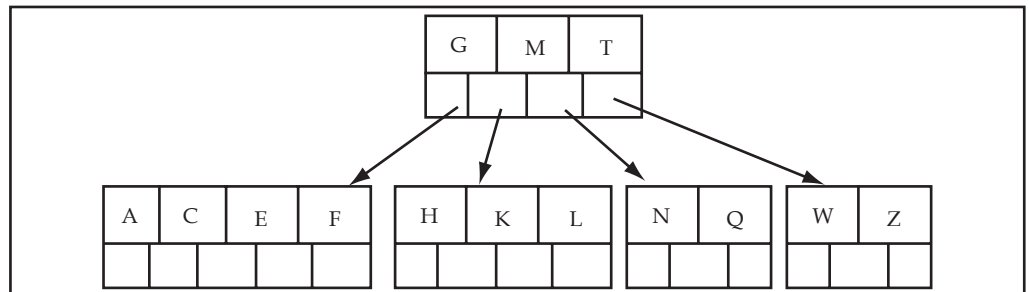
Notes



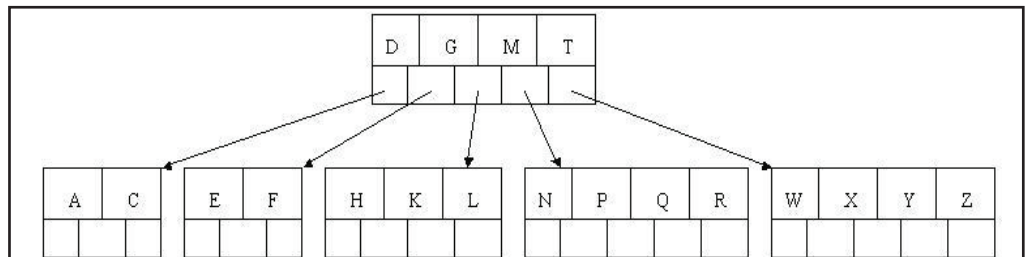
The letters F, W, L, and T are then added without needing any split.



When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

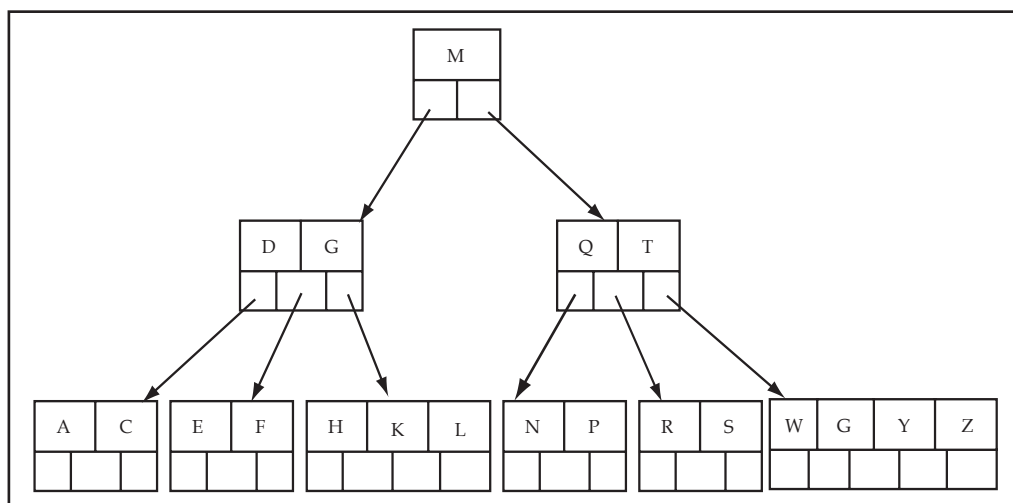


The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.

Notes

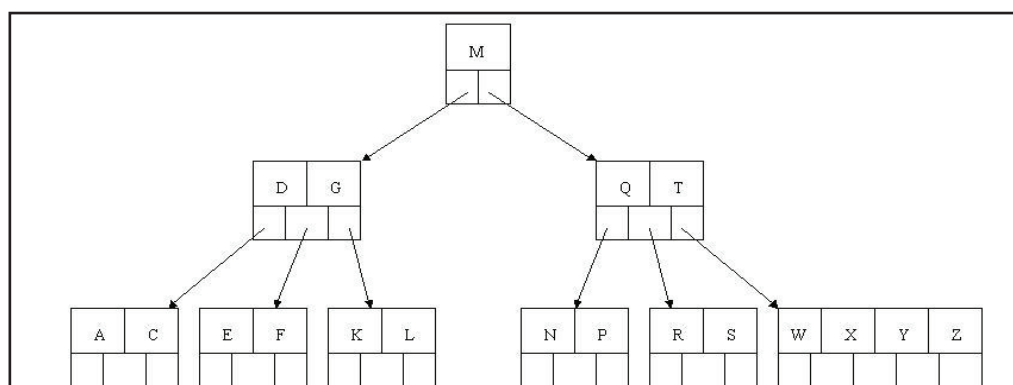


8.6 B-tree-Deleting an Item

Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained. Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be rearranged.

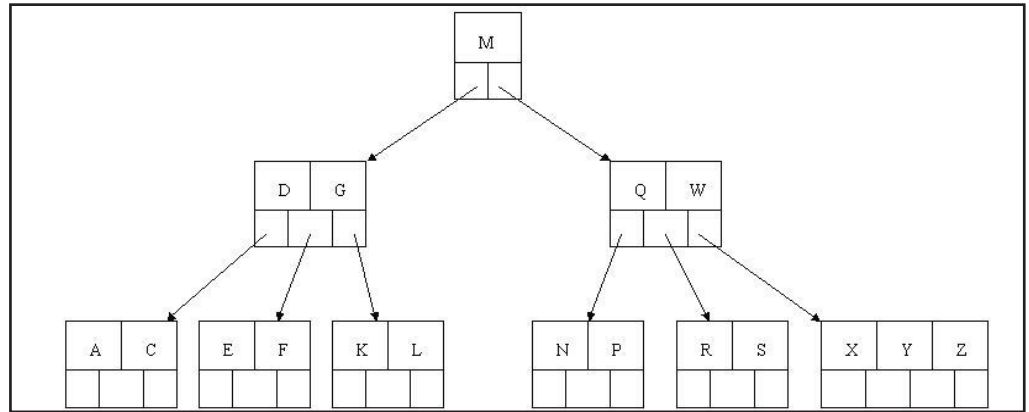
In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been.

This gives:

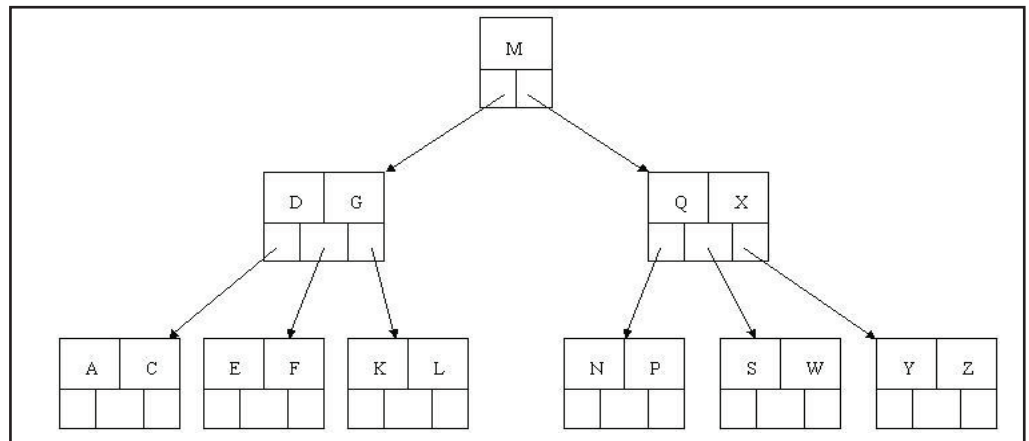


Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method.

Notes

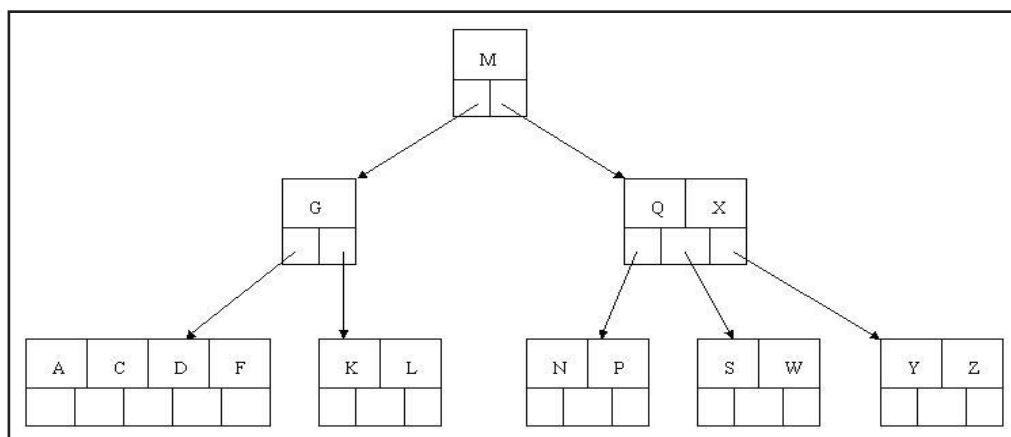


Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

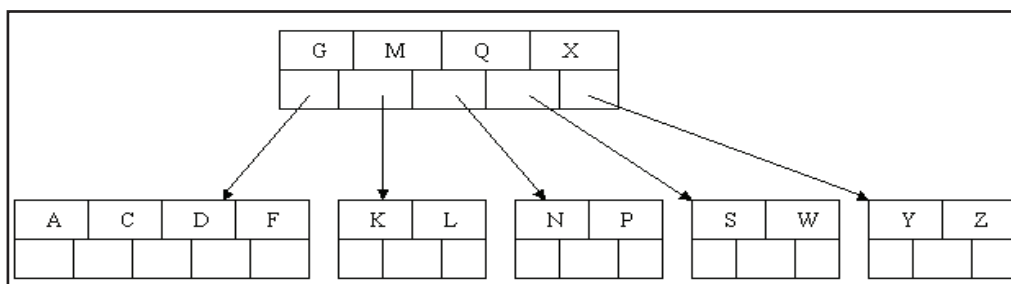


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.

Notes



Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again “borrow” a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M. In other words, the N P node would be attached via the pointer field to the right of M’s new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.



8.7 B-tree Algorithms

A B-tree is a data structure that maintains an ordered set of data and allows efficient operations to find, delete, insert, and browse the data. In this discussion, each piece of data stored in a B-tree will be called a “key”, because each key is unique and can occur in the B-tree in only one location.

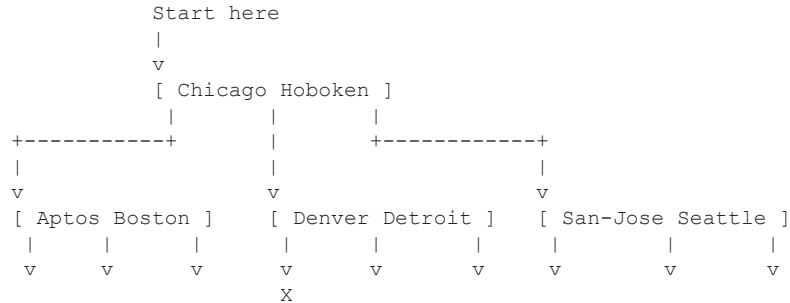
A B-tree consists of “node” records containing the keys, and pointers that link the nodes of the B-tree together.

Every B-tree is of some “order n”, meaning nodes contain from n to 2n keys, and nodes are thereby always at least half full of keys. Keys are kept in sorted order within each node. A corresponding list of pointers are effectively interspersed between keys to indicate where to search for a key if it isn’t in the current node. A node containing k keys always also contains k+1 pointers.

Notes

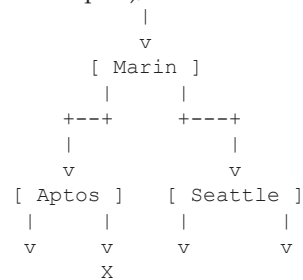


Example: Here is a portion of a B-tree with order 2 (nodes have at least 2 keys and 3 pointers). Nodes are delimited with [square brackets]. The keys are city names, and are kept sorted in each node. On either side of every key are pointers linking the key to subsequent nodes:



To find the key "Dallas", we begin searching at the top "root" node. "Dallas" is not in the node but sorts between "Chicago" and "Hoboken", so we follow the middle pointer to the next node. Again, "Dallas" is not in the node but sorts before "Denver", so we follow that node's first pointer down to the next node (marked with an "X"). Eventually, we will either locate the key, or encounter a "leaf" node at the bottom level of the B-tree with no pointers to any lower nodes and without the key we want, indicating the key is nowhere in the B-tree.

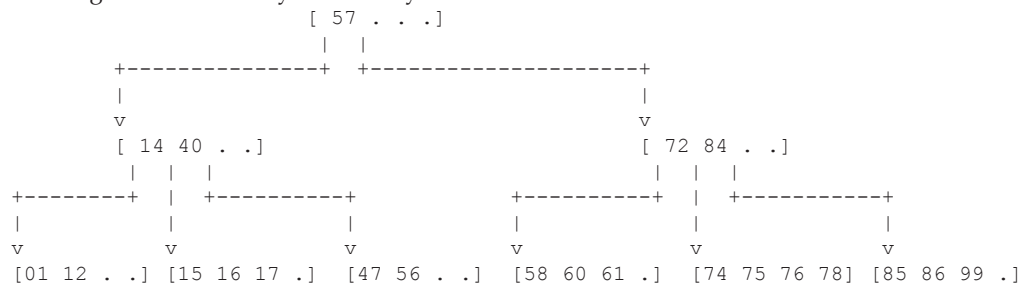
Below is another fragment of an order 1 B-tree (nodes have at least 1 key and 2 pointers). Searching for the key "Chicago" begins at "Marin", follows the first pointer to "Aptos" (since Chicago sorts before Marin), then follows that node's second pointer down to the next level (since Chicago sorts after Aptos), as marked with an "X".



Searching a B-tree for a key always begins at the root node and follows pointers from node to node until either the key is located or the search fails because a leaf node is reached and there are no more pointers to follow.

B-trees grow when new keys are inserted. Since the root node initially begins with just one key, the root node is a special exception and the only node allowed to have less than n keys in an order n B-tree.

Here is an order 2 B-tree with integer keys. Except for the special root node, order 2 requires every node to have from 2 to 4 keys and 3 to 5 pointers. Empty slots are marked with ".", showing where future keys have not yet been stored in the nodes:



To insert the key “59”, we first simply search for that key. If 59 is found, the key is already in the tree and the insertion is superfluous. Otherwise, we must end up at a leaf node at the bottom level of the tree where 59 would be stored. In the above case, the leaf node contains 58, 60, 61, and room for a fourth key, so 59 is simply inserted in the leaf node in sorted order:

```
[58 59 60 61]
```

Now you’ll insert the key “77”. The initial search leads us to the leaf node where 77 would be inserted, but the node is already full with 4 keys: 74, 75, 76, and 78. Adding another key would violate the rule that order 2 B-trees can’t have more than 4 keys. Because of this “overflow” condition, the leaf node is split into two leaf nodes. The leftmost 2 keys are put in the left node, the rightmost 2 keys are put in the right node, and the middle key is “promoted” by inserting it into the parent node above the leaf. Here, inserting 77 causes the 74-75-76-78 node to be split into two nodes, and 76 is moved up to the parent node that contained 72 and 84:

```

Before inserting 77          After inserting 77
[ 72 84 . .]                [ 72 76 84 .]
| | |                        | | | |
+-+ | +-                     --+ | | +--
|                                 | |
|                                 +-----+ +-----+
|                                 | |
v                                 v             v
[74 75 76 78]                 [74 75 . .]   [77 78 . .]

```

In this case, the parent node contained only 2 keys (72 and 84), leaving room for 76 to be promoted and inserted. But if the parent node was also already full with 4 keys, then it too would have to split. Indeed, splitting may propagate all the way up to the root node. When the root splits, the B-tree grows in height by one level, and a new root with a single promoted key is formed. (A situation when an order n root node sometimes has fewer than n keys, just like the situation described earlier when the root node stores the very first key placed in the B-tree.)

B-trees shrink when keys are deleted. To delete a key, first perform the usual search operation to locate the node containing the key. (If the key isn’t found, it isn’t in the tree and can’t be deleted.)

If the found key is not in a leaf, move it to a leaf by swapping the key with the logical “next” key. In a B-tree, the “next” key is always the first key in the leftmost leaf of the right subtree.



Example: In this B-tree we want to delete “37”, which is not in a leaf. “xx” indicates key values that don’t matter:

```

[ xx 37 xx xx ]
|
|
+-->[ xx xx xx xx ]
|
|
+-->[ xx xx xx xx ]
|
|
+-->[41 43 . .]

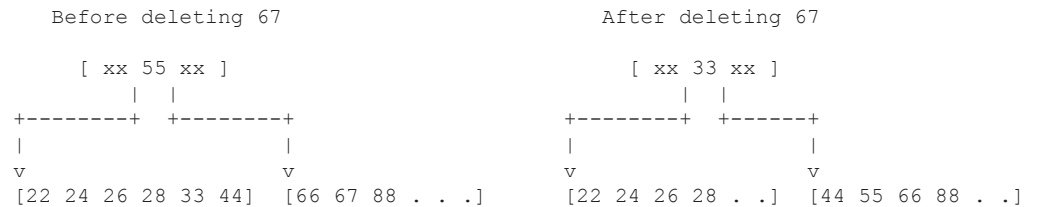
```

You follow the pointer immediately to the right of 37 to find 37’s right subtree, then follow the leftmost pointers in each subnode until we reach a leaf. The first key in the leaf is “41”, the logical “next” key after 37 in the list of all keys in the tree. By swapping 37 and 41, we can move 37 to a leaf node to set up a deletion without violating the key order or pointer order of the overall B-tree.

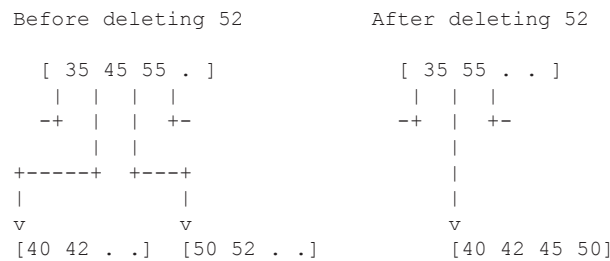
Once the key we want is in a leaf, we can delete it. If at least n keys remain in the node, we’re done, otherwise it is an “underflow”, since every node (except the root) must have at least n keys.

Notes

If a node underflows, we may be able to “redistribute” keys by borrowing some from a neighboring node. For example, in the order 3 B-tree below, the key 67 is being deleted, which causes a node to underflow since it only has keys 66 and 88 left. So keys from the neighbor on the left are “shifted through” the parent node and redistributed so both leaf nodes end up with 4 keys:



But if the underflow node and the neighbor node have less than 2n keys to redistribute, the two nodes will have to be combined. For example, here key 52 is being deleted from the B-tree below, causing an underflow, and the neighbor node can’t afford to give up any keys for redistribution. So one node is discarded, and the parent key moves down with the other keys to fill up a single node:



In the above case, moving the key 45 out of the parent node left two keys (35 and 55) remaining. But if the parent node only had n keys to begin with, then the parent node also would underflow when the parent key was moved down to combine with the leaf key. Indeed, underflow and the combining of nodes may propagate all the way up to the root node. When the root underflows, the B-tree shrinks in height by one level, and the nodes under the old root combine to form a new root.

The payoff of the B-tree insert and delete rules are that B-trees are always “balanced”. Searching an unbalanced tree may require traversing an arbitrary and unpredictable number of nodes and pointers.

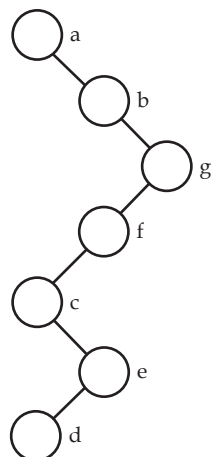


Searching a balanced tree means that all leaves are at the same depth. There is no runaway pointer overhead. Indeed, even very large B-trees can guarantee only a small number of nodes must be retrieved to find a given key. For example, a B-tree of 10,000,000 keys with 50 keys per node never needs to retrieve more than 4 nodes to find any key.



Task

Consider the following binary search tree:



Splay this tree at each of the following keys in turn:

d b g f a d b d

8.8 Applications

1. **Databases:** A database is a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data. The data can consist of anything, including, but not limited to names, addresses, pictures, and numbers. Databases are commonplace and are used everyday. For example, an airline reservation system might maintain a database of available flights, customers, and tickets issued. A teacher might maintain a database of student names and grades.

Because computers excel at quickly and accurately manipulating, storing, and retrieving data, databases are often maintained electronically using a database management system. Database management systems are essential components of many everyday business operations. Database serve as a foundation for accounting systems, inventory systems, medical recordkeeping systems, airline reservation systems, and countless other important aspects of modern businesses.

It is not uncommon for a database to contain millions of records requiring many gigabytes of storage. In order for a database to be useful and usable, it must support the desired operations, such as retrieval and storage, quickly. Because databases cannot typically be maintained entirely in memory, b-trees are often used to index the data and to provide fast access. For example, searching an unindexed and unsorted database containing n key values will have a worst case running time of $O(n)$; if the same data is indexed with a b-tree, the same search operation will run in $O(\log n)$. To perform a search for a single key on a set of one million keys (1,000,000), a linear search will require at most 1,000,000 comparisons. If the same data is indexed with a b-tree of minimum degree 10, 114 comparisons will be required in the worst case. Clearly, indexing large amounts of data can significantly improve search performance. Although other balanced tree structures can be used, a b-tree also optimizes costly disk accesses that are of concern when dealing with large data sets.

Notes

2. **Concurrent Access to B-trees:** Databases typically run in multiuser environments where many users can concurrently perform operations on the database.

Unfortunately, this common scenario introduces complications.



Example: Imagine a database storing bank account balances. Now assume that someone attempts to withdraw ₹40 from an account containing ₹60. First, the current balance is checked to ensure sufficient funds. After funds are disbursed, the balance of the account is reduced. This approach works flawlessly until concurrent transactions are considered. Suppose that another person simultaneously attempts to withdraw ₹30 from the same account. At the same time the account balance is checked by the first person, the account balance is also retrieved for the second person. Since neither person is requesting more funds than are currently available, both requests are satisfied for a total of ₹70. After the first person's transaction, ₹20 should remain ($₹60 - ₹40$), so the new balance is recorded as ₹20. Next, the account balance after the second person's transaction, ₹30 ($₹60 - ₹30$), is recorded overwriting the ₹20 balance. Unfortunately, ₹70 have been disbursed, but the account balance has only been decreased by ₹30. Clearly, this behavior is undesirable, and special precautions must be taken.

A b-tree suffers from similar problems in a multiuser environment. If two or more processes are manipulating the same tree, it is possible for the tree to become corrupt and result in data loss or errors.

The simplest solution is to serialize access to the data structure. In other words, if another process is using the tree, all other processes must wait.

Although this is feasible in many cases, it can place a unnecessary and costly limit on performance because many operations actually can be performed concurrently without risk. Locking, introduced by Gray and refined by many others, provides a mechanism for controlling concurrent operations on data structures in order to prevent undesirable side effects and to ensure consistency.

8.9 Summary

- B-trees are balanced trees that are optimized for situations when part or the entire tree must be maintained in secondary storage such as a magnetic disk.
- A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large.
- A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small.
- This means that only a small number of nodes must be read from disk to retrieve an item.
- The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

8.10 Keywords

B-Tree Algorithms: A B-tree is a data structure that maintains an ordered set of data and allows efficient operations to find, delete, insert, and browse the data.

B-trees: B-trees are balanced trees that are optimized for situations when part or the entire tree must be maintained in secondary storage such as a magnetic disk.

Notes

8.11 Self Assessment

Choose the appropriate answers:

1. A b-tree has a minimum number of allowable children for each node known as the
.....
 (a) Minimization factor
 (b) Maximization factor
 (c) Operational factor
 (d) Situational factor
2. Binary tree is a special type of tree having degree.
 (a) 3
 (b) 1
 (c) 2
 (d) 4

Fill in the blanks:

3. The search operation on a b-tree is to a search on a binary tree.
4. A is a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data.
5. A B-tree is kept balanced by requiring that all leaf nodes are at the same
6. The operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node.

State whether the following statements are True or False:

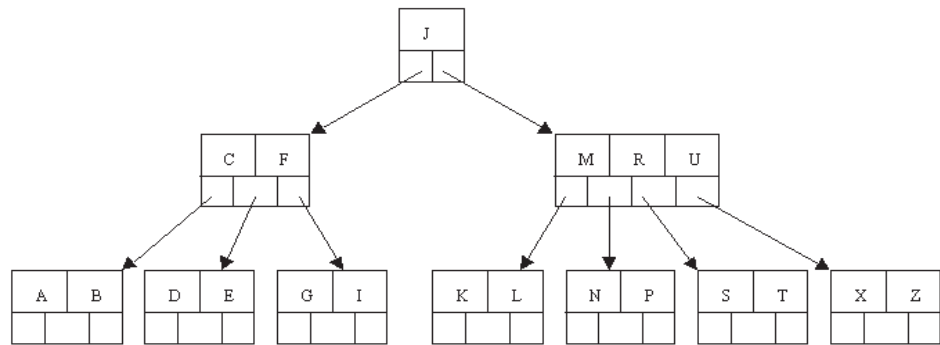
7. Deletion of a key from a b-tree is possible.
8. B-trees do not shrink when keys are deleted.
9. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation.
10. To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similar to B-Tree-Search.

8.12 Review Questions

1. Generalize the amortized analysis given in the text for incrementing four-digit binary integers to n-digit binary integers.
2. Describe the deletion of an item from b-trees.
3. Describe of structure of B-tree. Also explain the operation of B-tree.
4. Explain how will you insert an item in b-trees.

Notes

5. You have a B-tree of order 5 in figure given below. Explain to delete C from it.



6. There are $24 = 4!$ possible ordered sequences of the four keys 1, 2, 3, 4, but only 14 distinct binary trees with four nodes. Therefore, these binary trees are not equally likely to occur as search trees. Find which one of the 14 binary search trees corresponds to each of the 24 possible ordered sequences of 1, 2, 3, 4. Thereby find the probability for building each of the binary search trees from randomly ordered input.
7. "A node containing k keys always also contains $k+1$ pointers." Discuss.
8. "The split operation transforms a full node with $2t - 1$ keys into two nodes with $t - 1$ keys each". Explain
9. Explain the process of deletion of key from b-tree with the help of suitable example.
10. Binary trees are defined recursively; algorithms for manipulating binary trees are usually best written recursively. In programming with binary trees, be aware of the problems generally associated with recursive algorithms. Be sure that your algorithm terminates under any condition and that it correctly treats the trivial case of an empty tree.

Answers: Self Assessment

- | | |
|--------------|------------------|
| 1. (a) | 2. (c) |
| 3. analogous | 4. database |
| 5. depth | 6. B-Tree-Create |
| 7. True | 8. False |
| 9. True | 10. True |

8.13 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Notes

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*, Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 9: Hashing

CONTENTS

Objectives

Introduction

9.1 Hashing

9.1.1 Linear Probing or Linear Open Addressing

9.1.2 Rehashing

9.1.3 Overflow Chaining

9.2 Hash Functions

9.3 Open Hashing

9.4 Closed Hashing

9.5 Rehashing

9.6 Summary

9.7 Keywords

9.8 Self Assessment

9.9 Review Questions

9.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the concept of hashing
- Know of hash functions
- Realise open hashing and closed hashing
- Describe rehashing

Introduction

The search time of each algorithm depend on the number n of elements of the collection S of the data. A searching technique called Hashing or Hash addressing which is essentially independent of the number n .

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms.

A Hash Function is a Unary Function that is used by Hashed Associative Containers: it maps its argument to a result of type `size_t`. A Hash Function must be deterministic and stateless. That is, the return value must depend only on the argument, and equal arguments must yield equal results.

9.1 Hashing

Notes

In many applications we require to use a data object called symbol table. A symbol table is nothing but a set of pairs (name, value). Where value represents collection of attributes associated with the name, and this collection of attributes depends upon the program element identified by the name. For example, if a name x is used to identify an array in a program, then the attributes associated with x are the number of dimensions, lower bound and upper bound of each dimension, and the element type. Therefore a symbol table can be thought of as a linear list of pairs (name, value), and hence you can use a list of data object for realizing a symbol table. A symbol table is referred to or accessed frequently either for adding the name, or for storing the attributes of the name, or for retrieving the attributes of the name. Therefore accessing efficiency is a prime concern while designing a symbol table. Hence the most common way of getting a symbol table implemented is to use a hash table. Hashing is a method of directly computing the index of the table by using some suitable mathematical function called hash function. The hash function operates on the name to be stored in the symbol table, or whose attributes are to be retrieved from the symbol table. If h is a hash function and x is a name, then $h(x)$ gives the index of the table where x along with its attributes can be stored. If x is already stored in the table, then $h(x)$ gives the index of the table where it is stored to retrieve the attributes of x from the table. There are various methods of defining a hash function like a division method. In this method, you take the sum of the values of the characters, divide it by the size of the table, and take the remainder. This gives us an integer value lying in the range of 0 to $(n - 1)$ if the size of the table is n . The other method is a mid square method. In this method, the identifier is first squared and then the appropriate number of bits from the middle of square is used as the hash value. Since the middle bits of the square usually depend on all the characters in the identifier, it is expected that different identifiers will result into different values. The number of middle bits that you select depends on the table size. Therefore if r is the number of middle bits that you use to form hash value, then the table size will be 2^r . Hence when you use this method the table size is required to be power of 2. Another method is folding in which the identifier is partitioned into several parts, all but the last part being of the same length. These parts are then added together to obtain the hash value.

To store the name or to add attributes of the name, you compute hash value of the name, and place the name or attributes as the case may be, at that place in the table whose index is the hash value of the name. For retrieving the attribute values of the name kept in the symbol table, I apply the hash function to the name to obtain index of the table where you get the attributes of the name. Hence you find that no comparisons are required to be done. Hence the time required for the retrieval is independent of the table size. Therefore retrieval is possible in a constant amount of time, which will be the time taken for computing the hash function. Therefore hash table seems to be the best for realization, of the symbol table, but there is one problem associated with the hashing, and it is of collisions. Hash collision occurs when the two identifiers are mapped into the same hash value. This happens because a hash function defines a mapping from a set of valid identifiers to the set of those integers, which are used as indices of the table. Therefore you see that the domain of the mapping defined by the hash function is much larger than the range of the mapping, and hence the mapping is of many to one nature. Therefore when I implement a hash table a suitable collision handling mechanism is to be provided which will be activated when there is a collision.

Collision handling involve finding out an alternative location for one of the two colliding symbols. For example, if x and y are the different identifiers and if $h(x) = h(y)$, x and y are the colliding symbols. If x is encountered before y , then the i th entry of the table will be used for accommodating symbol x , but later on when y comes there is a hash collision, and therefore you have to find out an alternative location either for x or y . This means you find out a suitable alternative location and either accommodate y in that location, or you can move x to that location

Notes

and place y in the i th location of the table. There are various methods available to obtain an alternative location to handle the collision. They differ from each other in the way search is made for an alternative location. The following are the commonly used collision handling techniques:

9.1.1 Linear Probing or Linear Open Addressing

In this method, if for an identifier x , $h(x) = i$, and if the i th location is already occupied then you search for a location close to the i th location by doing a linear search starting from the $(i+1)$ th location to accommodate x . This means you start from the $(i+1)$ th location and do the linear search till you get an empty location, and once you get an empty location I accommodate x there.

9.1.2 Overflow Chaining

This is a method of implementing a hash table, in which collisions gets handled automatically. In this method you use two tables, a symbol table to accommodate identifiers and their attributes, and a hash table which is an array of pointers pointing to symbol table entries. Each symbol table entry is made of three fields, first for holding the identifier, second for holding the attributes, and the third for holding the link or pointer which can be made pointing to any symbol table entry. The insertions into the symbol table are done as follows:

If x is symbol to be inserted, then it will be added to the next available -entry of the symbol table. The hash value of x is then computed, if $h(x) = i$, then the i th hash table pointer is made pointing to the symbol table entry in which x is stored if the i th hash table pointer is not pointing to any symbol table entry. If the i th hash table pointer is already pointing to some symbol table entry, then the link field of symbol table entry containing x is made pointing to that symbol table entry to which i th hash table pointer is pointing to, and make the i th hash table pointer pointing the symbol entry containing x . This is equivalent to building a linked list on the i th index of the hash table. The retrieval of attributes is done as follows:

If x is a symbol, then you obtain $h(x)$, and use this value as the index of the hash table, and traverse the list built on this index to get that entry which contains x . A typical hash table implemented using this technique is shown below:

Let the symbols to be stored are $x_1, y_1, z_1, x_2, y_2, z_2$. The hash function that you use:

$$h(\text{symbol}) = (\text{value of first letter of the symbol}) \bmod n,$$

Where n is the size of table.

if

$$h(x_1) = i$$

$$h(y_1) = j$$

$$h(z_1) = k$$

then

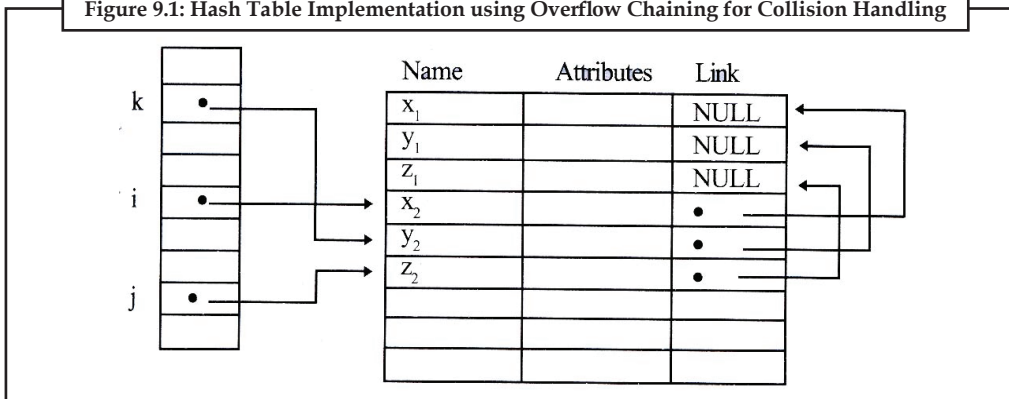
$$h(x_2) = i$$

$$h(y_2) = j$$

$$h(z_2) = k$$

Therefore, the contents of the symbol table will be the one shown in Figure 9.1.

Figure 9.1: Hash Table Implementation using Overflow Chaining for Collision Handling



Consider using division method of hashing store the following values in the hash table of size 11:

25, 45, 96, 101, 102, 162, 197, 201

Use sequential method for resolving the collisions.

Since division method of hashing is to be used the hash function is:

$h(\text{key}) = \text{key} \bmod 11$, where key is the value to be stored.

I start with the value 25, and compute the hash value using 25 as key. The hash value is $h(25) = 25 \bmod 11 = 3$. Therefore store 25 at the index 3 in the table.

For 45, $h(45) = 45 \bmod 11 = 1$, hence place 45 at the index 1.

For 96, $h(96) = 96 \bmod 11 = 8$,

\therefore store 96 at index 8.

For 101, $h(101) = 101 \bmod 11 = 2$,

\therefore store 101 at index 2.

For 102, $h(102) = 102 \bmod 11 = 3$, there is a collision, therefore you find a location closer to location at index 3 which is empty to accommodate 102, you see that the location at index 4 is empty.

\therefore store 102 at index 4.

For 162, $h(162) = 162 \bmod 11 = 8$, again there is a collision, therefore you find a location closer to location at index 8 which is empty to accommodate 162, you see that location at index 9 is empty.

\therefore store 162 at index 9.

For 197, $h(197) = 197 \bmod 11 = 10$, store 197 at index 10.

For 201, $h(201) = 201 \bmod 11 = 3$, again there is a collision, therefore you find a location closer to location at index 3, which is empty to accommodate 201, you see that location at index 5 is empty.

\therefore store 201 at index 5.

Notes

The hash table therefore is the one shown below:

0	
1	45
2	101
3	25
4	102
5	201
6	
7	
8	96
9	162
10	197



Task "A Hash Function must be deterministic and stateless." Discuss.

9.2 Hash Functions

Some of the methods of defining hash function are discussed below:

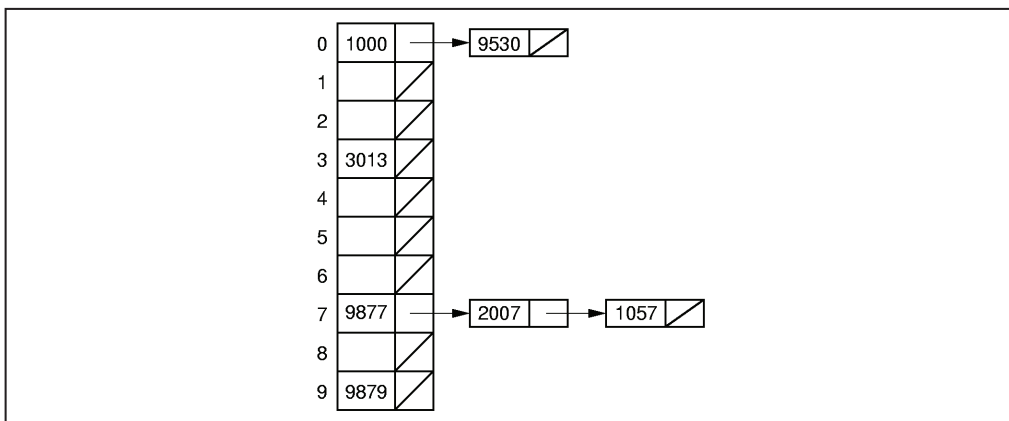
1. **Modular arithmetic:** In this method, first the key is converted to integer, then it is divided by the size of index range, and the remainder is taken to be the hash value. The spread achieved depends very much on the modulus. If modulus is power of small integers like 2 or 10, then many keys tend to map into the same index, while other indices remain unused. The best choice for modulus is often but not always is a prime number, which usually has the effect of spreading the keys quite uniformly.
2. **Truncation:** This method ignores part of key, and use the remainder part directly as hash value. (considering non-numeric fields as their numerical code) If the keys for example are eight digit numbers and the hash table has 1000 entries, then the first, second, and fifth digit from right might make hash value. So 62538194 maps to 394. It is a fast method, but often fails to distribute keys evenly.
3. **Folding:** In this method, the identifier is partitioned into several parts all but the last part being of the same length. These parts are then added together to obtain the hash value. For example an eight digit integer can be divided into groups of three, three, and two digits. The groups are the added together, and truncated if necessary to be in the proper range of indices. Hence 62538149 maps to, $625 + 381 + 94 = 1100$, truncated to 100. Since all information in the key can affect the value of the function, folding often achieves a better spread of indices than truncation.
4. **Mid square method:** In this method, the identifier is squared (considering non-numeric fields as their numerical code), and then the appropriate number of bits from the middle

of the square are used to get the hash value. Since the middle bits of the square usually depend on all the characters in the identifier, it is expected that different identifiers will result in different values. The number of middle bits that we select depends on table size. Therefore if r is the number of middle bits used to form hash value, then the table size will be 2^r , hence when you use mid square method the table size should be a power of 2.

Notes

9.3 Open Hashing

The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. The figure below illustrates a hash table where each slot stores one record and a link pointer to the rest of the list.



Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because I know to stop searching the list once you encounter a key that is greater than the one being searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. Assuming that the table has more slots than there are records to be stored, you can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $\Theta(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

Let:

1. U be the universe of keys:
 - (a) Integers
 - (b) Character strings
 - (c) Complex bit patterns

Notes

2. B the set of hash values (also called the buckets or bins). Let $B = \{0, 1, \dots, m - 1\}$ where $m > 0$ is a positive integer.

A hash function $h: U \rightarrow B$ associates buckets (hash values) to keys.

Two main issues:

Collisions

If x_1 and x_2 are two different keys, it is possible that $h(x_1) = h(x_2)$. This is called a collision. Collision resolution is the most important issue in hash table implementations.

Hash Functions

Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.

9.4 Closed Hashing

1. All elements are stored in the hash table itself
2. Avoids pointers; only computes the sequence of slots to be examined.
3. Collisions are handled by generating a sequence of **rehash** values.

$$h: \underset{\text{universe of primary keys}}{U} \times \underset{\text{probe number}}{U} \rightarrow \{0, 1, 2, \dots, m - 1\}$$

4. Given a key x , it has a hash value $h(x,0)$ and a set of rehash values

$$h(x, 1), h(x, 2), \dots, h(x, m-1)$$

5. I require that for every key x , the probe sequence

$$\langle h(x,0), h(x, 1), h(x,2), \dots, h(x, m-1) \rangle$$

be a permutation of $\langle 0, 1, \dots, m-1 \rangle$.

This ensures that every hash table position is eventually considered as a slot for storing a record with a key value x .

Search (x, T)

Search will continue until you find the element x (successful search) or an empty slot (unsuccessful search).

Delete (x, T)

1. No delete if the search is unsuccessful.
2. If the search is successful, then put the label DELETED (different from an empty slot).

Insert (x, T)

1. No need to insert if the search is successful.
2. If the search is unsuccessful, insert at the first position with a DELETED tag.



Task

“Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list.” Explain.

Notes

9.5 Rehashing

This is another method of collision handling. In this method you find an alternative empty location by modifying the hash function, and applying the modified hash function to the colliding symbol. For example, if x is symbol and $h(x) = i$, and if the i th location is already occupied, then I modify the hash function h to h_1 , and find out $h_1(x)$, if $h_1(x) = j$, and j th location is empty, then I accommodate x in the j th location. Otherwise you once again modify h_1 to some h_2 and repeat the process till the collision gets handled. Once the collision gets handled we revert back to the original hash function before considering the next symbol.

Denote $h(x, 0)$ by simply $h(x)$.

Linear Probing

$$h(x, i) = (h(x) + i) \bmod m$$

Quadratic Probing

$$h(x, i) = (h(x) + C_1 i + C_2 i^2) \bmod m$$

where C_1 and C_2 are constants.

Double Hashing

$$h(x, i) = (h(x) + i \underbrace{h'(x)}_{\text{another hash function}}) \bmod m$$

A Comparison of Rehashing Methods

Linear Probing	m distinct probe sequences	Primary clustering
Quadratic Probing	m distinct probe sequences	No primary clustering; but secondary clustering
Double Probing	m^2 distinct probe sequences	No primary clustering No secondary clustering

9.6 Summary

- Hash functions are mostly used in hash tables, to quickly locate a data record (for example, a dictionary definition) given its search key (the headword).
- Specifically, the hash function is used to map the search key to the index of a slot in the table where the corresponding record is supposedly stored.
- Rehashing schemes use a second hashing operation when there is a collision.

Notes

9.7 Keywords

Folding: In folding the identifier is partitioned into several parts all but the last part being of the same length.

Hash Function: A Hash Function is a Unary Function that is used by Hashed Associative Containers.

Hashing: Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string.

Rehashing: In rehashing find an alternative empty location by modifying the hash function, and applying the modified hash function to the colliding symbol.

9.8 Self Assessment

Fill in the blanks:

1. The simplest form of open hashing defines each slot in the hash table to be the head of a
2. is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list.
3. resolution is the most important issue in hash table implementations.
4. A Hash Function must be and stateless.
5. A is referred to or accessed frequently either for adding the name, or for storing the attributes of the name, or for retrieving the attributes of the name.
6. Hashing is a method of directly computing the index of the table by using some suitable mathematical function called
7. Collision handling involve finding out an alternative location for one of the two colliding
8. method ignores part of key, and use the remainder part directly as hash value.

9.9 Review Questions

1. Describe overflow chaining.
2. Describe various hash functions in detail.
3. Describe rehashing.
4. Devise a simple, easy to calculate hash function for mapping three-letter words to integers between 0 and n-1, inclusive. Find the values of your function on the words
PAL LAP PAM MAP PAT PET SET SAT TAT BAT
for n = 11, 13, 17, 19. Try for as few collisions as possible.
5. Suppose that a hash table contains hash_size = 13 entries indexed from 0 through 12 and that the following keys are to be mapped into the table:
10 100 32 45 58 126 3 29 200 400 0
(a) Determine the hash addresses and find how many collisions occur when these keys are reduced by applying the operation % hash_size.

- (b) Determine the hash addresses and find how many collisions occur when these keys are first folded by adding their digits together (in ordinary decimal representation) and then applying $\% \text{ hash_size}$.
- (c) Find a hash function that will produce no collisions for these keys. (A hash function that has no collisions for a fixed set of keys is called perfect.)
- (d) Repeat the previous parts of this exercise for $\text{hash_size} = 11$. (A hash function that produces no collision for a fixed set of keys that completely fill the hash table is called minimal perfect.)
6. Another method for resolving collisions with open addressing is to keep a separate array called the overflow table, into which are put all entries that collide with an occupied location. They can either be inserted with another hash function or simply inserted in order, with sequential search used for retrieval. Discuss the advantages and disadvantages of this method.
7. With linear probing, it is possible to delete an entry without using a second special key, as follows. Mark the deleted entry empty. Search until another empty position is found. If the search finds a key whose hash address is at or before the just-emptied position, then move it back there, make its previous position empty, and continue from the new empty position. Write an algorithm to implement this method. Do the retrieval and insertion algorithms need modification?
8. In a chained hash table, suppose that it makes sense to speak of an order for the keys, and suppose that the nodes in each chain are kept in order by key. Then a search can be terminated as soon as it passes the place where the key should be, if present. How many fewer probes will be done, on average, in an unsuccessful search? In a successful search? How many probes are needed, on average, to insert a new node in the right place? Compare your answers with the corresponding numbers derived in the text for the case of unordered chains.
9. The hash table itself contained only lists, one for each of the chains. One variant method is to place the first actual entry of each chain in the hash table itself. (An empty position is indicated by an impossible key, as with open addressing.) With a given load factor, calculate the effect on space of this method, as a function of the number of words (except links) in each entry. (A link takes one word.)
10. Distinguish between linear and quadratic probing.
11. Consider using division method of hashing store the following values in the hash table of size 13:
- 29, 45, 106, 136, 162, 172, 297, 301

Answers: Self Assessment

- | | | |
|------------------|-----------------|------------------|
| 1. linked list | 2. Open hashing | 3. Collision |
| 4. deterministic | 5. symbol table | 6. hash function |
| 7. symbols | 8. Truncation | |

9.10 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E. Leiserson & Ronald L., *Rivest: Introduction to Algorithms*, Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 10: Heaps

Notes

CONTENTS

Objectives

Introduction

10.1 Heaps

10.2 Binary Heaps

10.2.1 Complete Trees

10.2.2 Implementation

10.2.3 Putting items into a Binary Heap

10.2.4 Removing items from a Binary Heap

10.3 Applications of Heaps

10.3.1 Discrete Event Simulation

10.3.2 Implementation

10.4 d-Heaps

10.5 Summary

10.6 Keywords

10.7 Self Assessment

10.8 Review Questions

10.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe heaps
- State the concept of binary heaps
- Discuss applications of heaps
- Define d-heaps

Introduction

A heap is a specialized tree-based data structure that satisfies the heap property: if B is a child node of A, then $\text{key}(A) \geq \text{key}(B)$. This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a max-heap. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a min-heap.) The heap is one maximally-efficient implementation of an abstract data type called a priority queue. Heaps are crucial in several efficient graph algorithms.

10.1 Heaps

A heap is a storage pool in which regions of memory are dynamically allocated. For example, in C++ the space for a variable is allocated essentially in one of three possible places: Global

Notes

variables are allocated in the space of initialized static variables; the local variables of a procedure are allocated in the procedure's activation record, which is typically found in the processor stack; and dynamically allocated variables are allocated in the heap. In this unit, the term heap is taken to mean the storage pool for dynamically allocated variables.

I consider heaps and heap-ordered trees in the context of priority queue implementations. While it may be possible to use a heap to manage a dynamic storage pool, typical implementations do not. In this context, the technical meaning of the term heap is closer to its dictionary definition—"a pile of many things."

A binary tree has the **heap property** iff

1. It is empty or
2. The key in the root is larger than that in either child and both subtrees have the heap property.

A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, you are left with two sub-trees and you must *efficiently* re-create a single tree with the heap property.

The value of the heap structure is that you can both extract the highest priority item and insert a new one in $O(\log n)$ time.

10.2 Binary Heaps

A binary heap is a heap-ordered binary tree which has a very special shape called a complete tree. As a result of its special shape, a binary heap can be implemented using an array as the underlying foundational data structure. Array subscript calculations are used to find the parent and the children of a given node in the tree. And since an array is used, the storage overhead associated with the subtree fields contained in the nodes of the trees is eliminated.

10.2.1 Complete Trees

Complete trees and perfect trees are closely related, yet quite distinct. As pointed out in the preceding unit, a perfect binary tree of height h has exactly $n = 2^{h+1} - 1$ internal nodes. Since, the only permissible values of n are

$$0, 1, 3, 7, 15, 31, \dots, 2^{h+1} - 1, \dots,$$

there is no *perfect* binary tree which contains, say 2, 4, 5, or 6 nodes.

However, you want a data structure that can hold an arbitrary number of objects so you cannot use a perfect binary tree. Instead, you use a *complete binary tree*, which is defined as follows:

Definition (Complete Binary Tree)

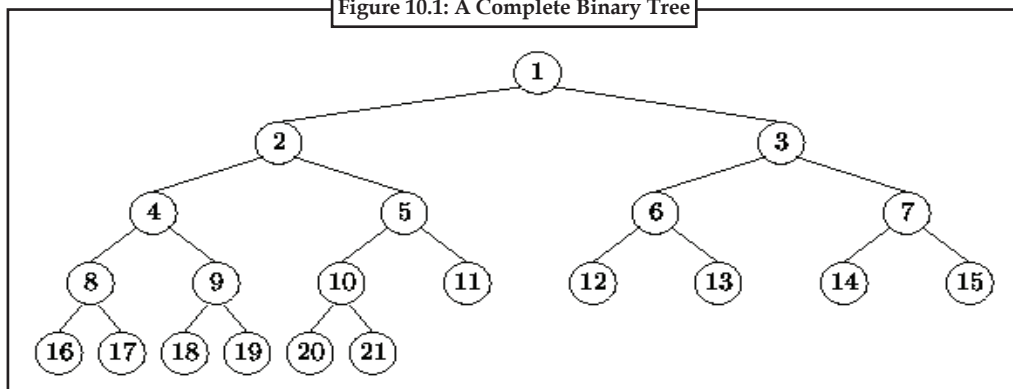
A *complete binary tree* of height $h \geq 0$, is a binary tree $\{R, T_L, T_R\}$ with the following properties.

1. If $h = 0$, $T_L = l$ and $T_R = l$.
2. For $h > 0$ there are two possibilities:
 - (a) T_L is a perfect binary tree of height $h-1$ and T_R is a complete binary tree of height $h-1$;
or
 - (b) T_L is a complete binary tree of height $h-1$ and T_R is a perfect binary tree of height $h-2$.

Figure 10.1 shows an example of a complete binary tree of height four. Notice that the left subtree of node 1 is a complete binary tree of height three; and the right subtree is a perfect binary tree

of height two. Similarly, the left subtree of node 2 is a perfect binary tree of height two; and the right subtree is a complete binary tree of height two.

Figure 10.1: A Complete Binary Tree



Does there exist a complete binary with exactly n nodes for every integer $n > 0$? The following theorem addresses this question indirectly by defining the relationship between the height of a complete tree and the number of nodes it contains.



Task

Discuss how complete and perfect trees are closely related.

Theorem-10.1

A complete binary tree of height $h \geq 0$ contains at least 2^h and at most $2^{h+1} - 1$ nodes.

Proof First, you prove the lower bound by induction. Let m_h be the *minimum* number of nodes in a complete binary tree of height h . To prove the lower bound you must show that $m_h = 2^h$.

Base Case There is exactly one node in a tree of height zero. Therefore, $m_0 = 1 = 2^0$.

Inductive Hypothesis Assume that $m_h = 2^h$ for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider the complete binary tree of height $k+1$ which has the smallest number of nodes. Its left subtree is a complete tree of height k having the smallest number of nodes and its right subtree is a perfect tree of height $k-1$.

From the inductive hypothesis, there are 2^k nodes in the left subtree and there are exactly $2^{(k-1)+1} - 1$ nodes in the perfect right subtree. Thus,

$$\begin{aligned} m_{k+1} &= 1 + 2^k + 2^{(k-1)+1} - 1 \\ &= 2^{k+1} \end{aligned}$$

Therefore, by induction $m_h = 2^h$ for all $h \geq 0$, which proves the lower bound.

Next, I prove the upper bound by induction. Let M_h be the *maximum* number of nodes in a complete binary tree of height h . To prove the upper bound I must show that $M_h = 2^{h+1} - 1$.

Base Case There is exactly one node in a tree of height zero. Therefore, $M_0 = 1 = 2^1 - 1$.

Inductive Hypothesis Assume that $M_h = 2^{h+1} - 1$ for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider the complete binary tree of height $k+1$ which has the largest number of nodes. Its left subtree is a perfect tree of height k and its right subtree is a complete tree of height k having the largest number of nodes.

Notes

There are exactly $2^{k+1} - 1$ nodes in the perfect left subtree. From the inductive hypothesis, there are $2^{k+1} - 1$ nodes in the right subtree. Thus,

$$m_{k+1} = 1 + 2^{k+1} - 1 + 2^{k+1} - 1 = 2^{(k+1)+1} - 1.$$

Therefore, by induction $M_n = 2^{h+1} - 1$ for all $h \geq 0$, which proves the upper bound.

It follows from Theorem 10.1 that there exists exactly one complete binary tree that contains exactly n internal nodes for every integer $n \geq 0$. It also follows from

Theorem 10.1 that the height of a complete binary tree containing n internal nodes is $h = \lceil \log_2 n \rceil$.

Why are interested in complete trees? As it turns out, complete trees have some useful characteristics. For example, in the preceding chapter you saw that the internal path length of a tree, i.e., the sum of the depths of all the internal nodes, determines the average time for various operations. A complete binary tree has the nice property that it has the smallest possible internal path length:

Theorem 10.2

The internal path length of a binary tree with n nodes is at least as big as the internal path length of a complete binary tree with n nodes.

Proof Consider a binary tree with n nodes that has the smallest possible internal path length. Clearly, there can only be one node at depth zero—the root. Similarly, at most two nodes can be at depth one; at most four nodes can be at depth two; and so on. Therefore, the internal path length of a tree with n nodes is always at least as large as the sum of the first n terms in the series

$$0, \underbrace{1}_1, \underbrace{1, 2}_2, \underbrace{2, 2, 2}_4, \underbrace{3, 3, 3, 3, 3, 3, 3, 3}_8, 4, \dots$$

But this summation is precisely the internal path length of a complete binary tree!

Since the depth of the average node in a tree is obtained by dividing the internal path length of the tree by n , Theorem 10.1 tells us that complete trees are the best possible in the sense that the average depth of a node in a complete tree is the smallest possible. But how small is small? That is, does the average depth grow logarithmically with n . The following theorem addresses this question:

Theorem 10.3

The internal path length of a complete binary tree with n nodes is

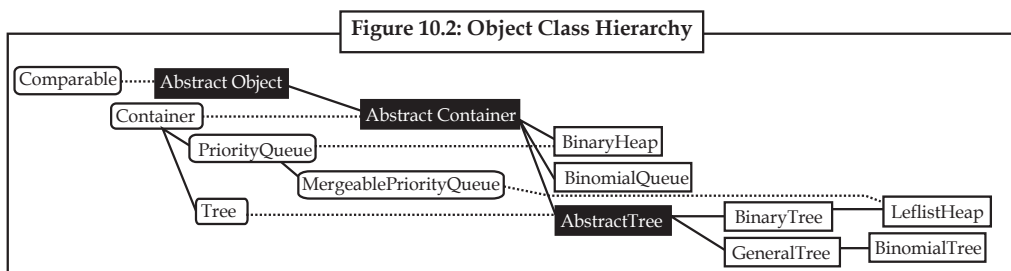
$$\sum_{i=1}^n \lfloor \log_2 i \rfloor = (n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2.$$

Proof The proof of Theorem 10.3 is left as an exercise for the reader. From Theorem 10.3 you may conclude that the internal path length of a complete tree is $O(n \log n)$. Consequently, the depth of the average node in a complete tree is $O(\log n)$.

10.2.2 Implementation

A binary heap is a heap-ordered complete binary tree which is implemented using an array. In a heap the smallest key is found at the root and since the root is always found in the first position of the array, finding the smallest key is a trivial operation in a binary heap.

In this section we will describe the implementation of a priority queue as a binary heap. As shown in Figure 10.2, I define a concrete class called `BinaryHeap` for this purpose.



Program-10.1 Introduces the `BinaryHeap` class. The `BinaryHeap` class extends the `AbstractContainer` class introduced in Program 10.1 and it implements the `PriorityQueue` interface defined in Program 10.1.

```

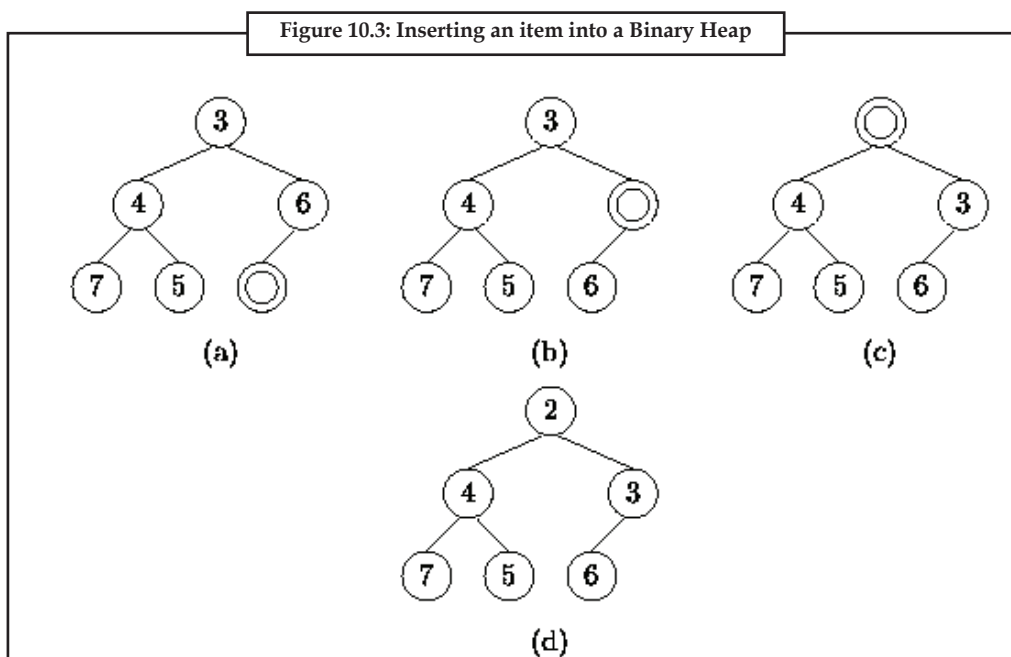
public class BinaryHeap
    extends AbstractContainer
    implements PriorityQueue
{
    protected Comparable [] array;
    // ...
}
  
```

Program 10.1: BinaryHeap Fields

10.2.3 Putting items into a Binary Heap

There are two requirements which must be satisfied when an item is inserted in a binary heap. First, the resulting tree must have the correct shape. Second, the tree must remain heap-ordered. Figure 10.3 illustrates the way in which this is done.

Since the resulting tree must be a complete tree, there is only one place in the tree where a node can be added. That is, since the bottom level must be filled from left to right, the node must be added at the next available position in the bottom level of the tree as shown in Figure 10.3 (a).



Notes

In this example, the new item to be inserted has the key 2. Note that you cannot simply drop the new item into the next position in the complete tree because the resulting tree is no longer heap ordered. Instead, the hole in the heap is moved toward the root by moving items down in the heap as shown in Figure 10.3 (b) and (c). The process of moving items down terminates either when you reach the root of the tree or when the hole has been moved up to a position in which when the new item is inserted the result is a heap.

Program 10.2 gives the code for inserting an item in a binary heap. The `enqueue` method of the `BinaryHeap` class takes as its argument the item to be inserted in the heap. If the priority queue is full an exception is thrown. Otherwise, the item is inserted as described above.

```
public class BinaryHeap
    extends AbstractContainer
    implements PriorityQueue
{
    protected Comparable[] array;
    public void enqueue (Comparable object)
    {
        if (count == array.length - 1)
            throw new ContainerFullException ();
        ++count;
        int i = count;
        while (i > 1 && array [i/2].isGT (object))
        {
            array [i] = array [i/2];
            i/=2;
        }
        array [i] = object;
    }
    //...
}
```

Program 10.2: BinaryHeap class enqueue Method

The implementation of the algorithm is actually remarkably simple. Lines 13-17 move the hole in the heap up by moving items down. When the loop terminates, the new item can be inserted at position i . Therefore, the loop terminates either at the root, $i=1$, or when the key in the parent of i , which is found at position $\lfloor i/2 \rfloor$, is smaller than the item to be inserted.

Notice too that a good optimizing compiler will recognize that the subscript calculations involve only division by two. Therefore, the divisions can be replaced by bitwise right shifts which usually run much more quickly.

Since the depth of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$, the worst case running time for the `enqueue` operation is

$$\lfloor \log_2 n \rfloor, \tau(\text{isGT}) + O(\log n),$$

where $\tau(\text{isGT})$ is the time required to compare to objects. If $\tau(\text{isGT}) = O(1)$, the `enqueue` operation is simply $O(\log n)$ in the worst case.

10.2.4 Removing items from a Binary Heap

Notes

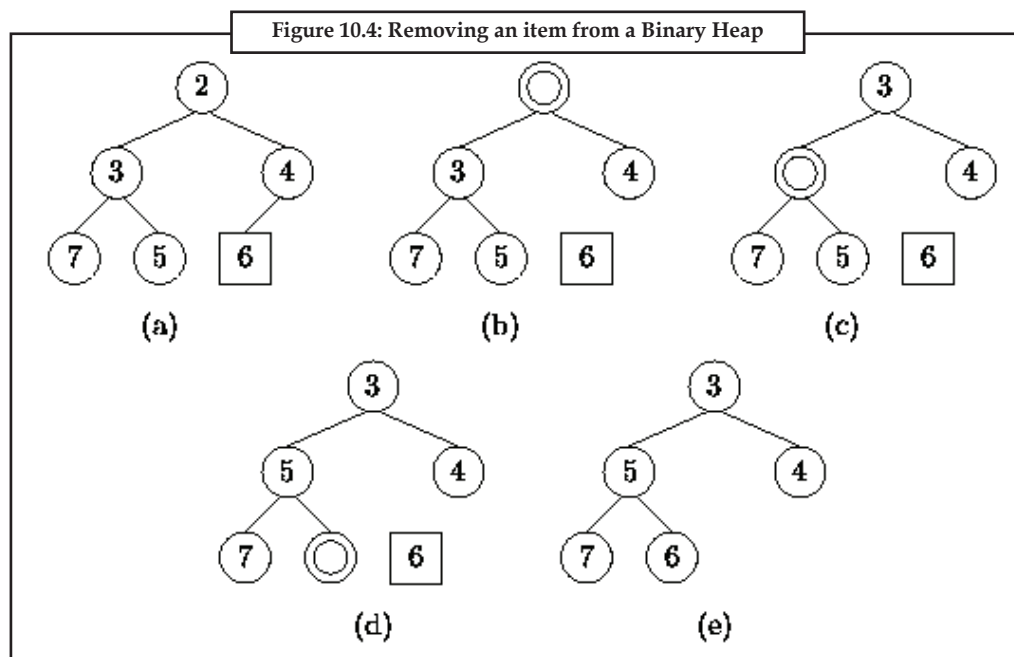
The `dequeueMin` method removes from a priority queue the item having the smallest key. In order to remove the smallest item, it needs first to be located. Therefore, the `dequeueMin` operation is closely related to `findMin`.

The smallest item is always at the root of a min heap. Therefore, the `findMin` operation is trivial. Program 10.3 gives the code for the `findMin` method of the `BinaryHeap` class. Assuming that no exception is thrown, the running time of `findMin` is clearly $O(1)$.

```
public class BinaryHeap
    extends AbstractContainer
    implements PriorityQueue
{
    protected Comparable [] array;
    public Comparable findMin ()
    {
        if (count == 0)
            throw new ContainerEmptyException ();
        return array [1];
    }
    // ...
}
```

Program 10.3: BinaryHeap class findMin Method

Since the bottom row of a complete tree is filled from left to right as items are added, it follows that the bottom row must be emptied from right to left as items are removed. So, you have a problem: The datum to be removed from the heap by `dequeueMin` is in the root, but the node to be removed from the heap is in the bottom row.



Notes

Figure 10.4 (a) illustrates the problem. The `dequeueMin` operation removes the key 2 from the heap, but it is the node containing key 6 that must be removed from the tree to make it into a complete tree again. When key 2 is removed from the root, a hole is created in the tree as shown in Figure 10.4 (b).

The trick is to move the hole down in the tree to a point where the left-over key, in this case the key 6, can be reinserted into the tree. To move a hole down in the tree, you consider the children of the empty node and move up the smallest key. Moving up the smallest key ensures that the result will be a min heap.

The process of moving up continues until either the hole has been pushed down to a leaf node, or until the hole has been pushed to a point where the left over key can be inserted into the heap. In the example shown in Figure 10.4 (b)-(c), the hole is pushed from the root node to a leaf node where the key 6 is ultimately placed is shown in Figure 10.4 (d).

Program 10.4 gives the code for the `dequeueMin` method of the `BinaryHeap` class. This method implements the deletion algorithm described above. The main loop (lines 15-25) moves the hole in the tree down by moving up the child with the smallest key until either a leaf node is reached or until the hole has been moved down to a point where the last element of the array can be reinserted.

```
public class BinaryHeap
    extends AbstractContainer
    implements PriorityQueue
{
    protected Comparable [] array;
    public Comparable dequeueMin()
    {
        if (count == 0)
            throw new ContainerEmptyException ();
        Comparable result = array [1];
        Comparable last = array [count];
        --count;
        int i = 1;
        while (2 * i < count + 1)
        {
            int child = 2 * i;
            if (child + 1 < count + 1
                && array [child + 1].isLT (array [child]))
                child += 1;
            if (last.isLE (array [child]))
                break;
            array [i] = array [child];
            i = child;
        }
        array [i] = last;
        return result;
    }
    // ...
}
```

Program 10.4: BinaryHeap class dequeueMin Method

In the worst case, the hole must be pushed from the root to a leaf node. Each iteration of the loop makes at most two object comparisons and moves the hole down one level.

Therefore, the running time of the `dequeueMin` operation is

$$\lfloor \log_2 n \rfloor (\tau(\text{isLT}) + \tau(\text{isLE})) + O(\log n),$$

where $n = \text{count}$ is the number of items in the heap. If $\tau(\text{isLT}) = O(1)$ and $\tau(\text{isLE}) = O(1)$, the `dequeueMin` operation is simply $O(\log n)$ in the worst case.



Task

Discuss the uses of binary heaps.

10.3 Applications of Heaps

The main applications of heaps are:

1. Discrete Event Simulation
2. Implementation

10.3.1 Discrete Event Simulation

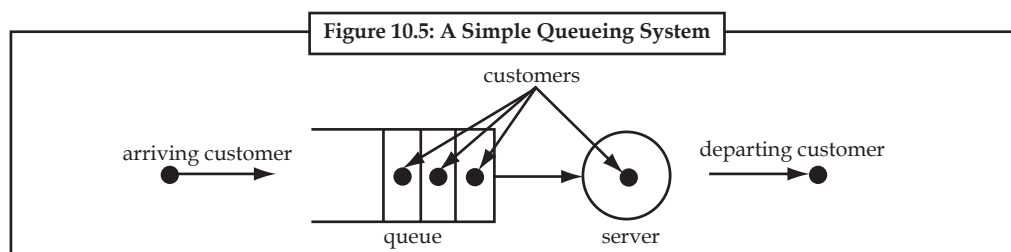
One of the most important applications of priority queues is in *discrete event simulation*. Simulation is a tool which is used to study the behavior of complex systems. The first step in simulation is *modeling*. You construct a mathematical model of the system I wish to study. Then you write a computer program to evaluate the model.

The systems studied using *discrete event simulation* have the following characteristics: The system has a *state* which evolves or changes with time. Changes in state occur at distinct points in simulation time. A state change moves the system from one state to another instantaneously. State changes are called *events*.



Example: Suppose I wish to study the service received by customers in a bank. Suppose a single teller is serving customers. If the teller is not busy when a customer arrives at the bank, the that customer is immediately served. On the other hand, if the teller is busy when another customer arrives, that customer joins a queue and waits to be served.

You can model this system as a discrete event process as shown in Figure 10.5. The state of the system is characterized by the state of the server (the teller), which is either busy or idle, and by the number of customers in the queue. The events which cause state changes are the arrival of a customer and the departure of a customer.



If the server is idle when a customer arrives, the server immediately begins to serve the customer and therefore changes its state to busy. If the server is busy when a customer arrives, that customer joins the queue.

Notes

When the server finishes serving the customer, that customer departs. If the queue is not empty, the server immediately commences serving the next customer. Otherwise, the server becomes idle.

How do you keep track of which event to simulate next? Each event (arrival or departure) occurs at a discrete point in *simulation time*. In order to ensure that the simulation program is correct, it must compute the events in order. This is called the *causality constraint*—events cannot change the past.

In our model, when the server begins to serve a customer you can compute the departure time of that customer. So, when a customer arrives at the server I *schedule* an event in the future which corresponds to the departure of that customer. In order to ensure that events are processed in order, you keep them in a priority queue in which the time of the event is its priority. Since you always process the pending event with the smallest time next and since an event can schedule new events only in the future, the causality constraint will not be violated.

10.3.2 Implementation

This section presents the simulation of a system comprised of a single queue and server as shown in Figure 10.5. Program 10.5 defines the class `Event` which represents events in the simulation. There are two parts to an event, a *type* (either arrival or departure), and a *time*.

```
public class Simulation
{
    static class Event
        extends Association
    {
        public static final int arrival = 0;
        public static final int departure = 1;
        Event (int type, double time)
            {super (new Db1 (time), new Int (type)); }
        double getTime ()
            {return ((Db1) getKey ().doubleValue ());}
        int getType ()
            {return ((Int) getValue ().intValue ());}
    }
    // ...
}
```

Program 10.5: Event Class

An association is an ordered pair comprised of a key and a value. In the case of the `Event` class, the key is the *time* of the event and the value is the *type* of the event. Therefore, the events in a priority queue are prioritized by their times.

Program 10.6 defines the `run` method which implements the discrete event simulation. This method takes one argument, `timeLimit`, which specifies the total amount of time to be simulated.

The `Simulation` class contains a single field, called `eventList`, which is a priority queue. This priority queue is used to hold the events during the course of the simulation.

```
public class Simulation
{
    PriorityQueue eventList = new LeftistHeap ();
    public void run (double timeLimit)
```

```

{
    boolean serverBusy = false;
    int numberInQueue = 0;
    RandomVariable serviceTime = new ExponentialRV (100.);
    RandomVariable interArrivalTime =
        new ExponentialRV (100.);
    eventList.enqueue (new Event.arrival, 0));
    while (!eventList.isEmpty ())
    {
        Event event = (Event) eventList.dequeueMin ();
        double t = event.getTime ();
        if (t > timeLimit)
            {eventList.purge (); break; }
        switch (event.getType ())
        {
        case Event.arrival:
            if (!serverBusy)
            {
                serverBusy = true;
                eventList.enqueue (new Event (Event.departure,
                    t + serviceTime.nextDouble ());
            }
            else
                ++numberInQueue;
            eventList.enqueue (new Event (Event.arrival,
                t + interArrivalTime.nextDouble ());
            break;
        case Event.departure:
            if (numberInQueue == 0)
                serverBusy = false;
            else
            {
                --numberInQueue;
                eventList.enqueue (new Event (Event.departure,
                    t + serviceTime.nextDouble ());
            }
            break;
        }
    }
}
// ...
}

```

Program 10.6: Application of Priority Queues - discrete Event Simulation

Notes

The state of the system being simulated is represented by the two variables `serverBusy` and `numberInQueue`. The first is a `boolean` value which indicates whether the server is busy. The second keeps track of the number of customers in the queue.

In addition to the state variables, there are two instances of the class `ExponentialRV`. It implements the `RandomVariable` interface defined in Program 10.6. This interface defines a method called `nextDouble` which is used to sample the random number generator. Every time `nextDouble` is called, a different (random) result is returned. The random values are exponentially distributed around a mean value which is specified in the constructor. For example, in this case both `serviceTime` and `interArrivalTime` produce random distributions with the mean value of 100 (lines 9-11).

It is assumed that the `eventList` priority queue is initially empty. The simulation begins by enqueueing a customer arrival at time zero (line 12). The `while` loop (lines 13-44) constitutes the main simulation loop. This loop continues as long as the `eventList` is not empty, i.e., as long as there is an event to be simulated.

Each iteration of the simulation loop begins by dequeuing the next event in the event list (line 15). If the time of that event exceeds `timeLimit`, the event is discarded, the `eventList` is purged, and the simulation is terminated. Otherwise, the simulation proceeds.

The simulation of an event depends on the type of that event. The `switch` statement (line 19) invokes the appropriate code for the given event. If the event is a customer arrival and the server is not busy, `serverBusy` is set to `true` and the `serviceTime` random number generator is sampled to determine the amount of time required to service the customer. A customer departure is scheduled at the appropriate time in the future (lines 24-26). On the other hand, if the server is already busy when the customer arrives, I add one to the `numberInQueue` variable (line 29).

Another customer arrival is scheduled after every customer arrival. The `interArrivalTime` random number generator is sampled, and the arrival is scheduled at the appropriate time in the future (lines 30-31).

If the event is a customer departure and the queue is empty, the server becomes idle (lines 34-35). When a customer departs and there are still customers in the queue, the next customer in the queue is served. Therefore, `numberInQueue` is decreased by one and the `serviceTime` random number generator is sampled to determine the amount of time required to service the next customer. A customer departure is scheduled at the appropriate time in the future (lines 38-40).

Clearly the execution of the `Simulation` method given in Program 10.6 mimics the modeled system. Of course, the program given produces no output. For it to be of any practical value, the simulation program should be instrumented to allow the user to study its behavior.



Example: The user may be interested in knowing statistics such as the average queue length and the average waiting time that a customer waits for service. And such instrumentation can be easily incorporated into the given framework.

10.4 d-Heaps

The d-ary heap or d-heap is a priority queue data structure, a generalization of the binary heap in which the nodes have d children instead of 2. Thus, a binary heap is a 2-heap.

The d-ary heap consists of an array of n items, each of which has a priority associated with it. These items may be viewed as the nodes in a complete d-ary tree, listed in breadth first traversal order: the item at position 0 of the array forms the root of the tree, the items at positions 1-d are its children, the next d² items are its grandchildren, etc. Thus, the parent of the item at position i (for any i > 0) is the item at position $\text{ceiling}((i - 1)/d)$ and its children are the items at positions di + 1 through di + d. According to the heap property, in a min-heap, each item has a priority

that is at least as large as its parent; in a max-heap, each item has a priority that is no larger than its parent.

The minimum priority item in a min-heap (or the maximum priority item in a max-heap) may always be found at position 0 of the array. To remove this item from the priority queue, the last item x in the array is moved into its place, and the length of the array is decreased by one. Then, while item x and its children do not satisfy the heap property, item x is swapped with one of its children (the one with the smallest priority in a min-heap, or the one with the largest priority in a max-heap), moving it downward in the tree and later in the array, until eventually the heap property is satisfied. The same downward swapping procedure may be used to increase the priority of an item in a min-heap, or to decrease the priority of an item in a max-heap.

To insert a new item into the heap, the item is appended to the end of the array, and then while the heap property is violated it is swapped with its parent, moving it upward in the tree and earlier in the array, until eventually the heap property is satisfied. The same upward-swapping procedure may be used to decrease the priority of an item in a min-heap, or to increase the priority of an item in a max-heap.

To create a new heap from an array of n items, one may loop over the items in reverse order, starting from the item at position $n - 1$ and ending at the item at position 0, applying the downward-swapping procedure for each item.

To implement Prim's algorithm efficiently, we need a data structure that will store the vertices of S in a way that allows the vertex joined by the minimum cost edge to be selected quickly.

A heap is a data structure consisting of a collection of items, each having a key. The basic operations on a heap are:

1. *insert*(i, k, h). Add item i to heap h using k as the key value.
2. *deletemin*(h). Delete and return an item of minimum key from h .
3. *changekey*(i, k, h). Change the key of item i in heap h to k .
4. *key*(i, h). Return the key value for item i .

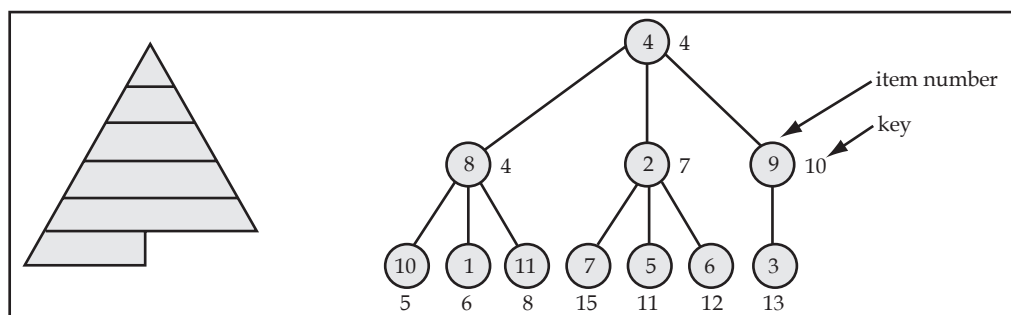
The heap is among the most widely applicable non-elementary data structure.

Heaps can be implemented efficiently, using a heap-ordered tree.

1. each tree node contains one item and each item has a real-valued key
2. the key of each node is at least as large the key of its parent (excepting the root)

For integer $d > 1$, a d -heap is a heap-ordered d -ary tree that is "heapshaped."

1. let T be an infinite d -ary tree, with vertices numbered in breadth-first order
2. a subtree of T is heap-shaped if its vertices have consecutive numbers $1, 2, \dots, n$

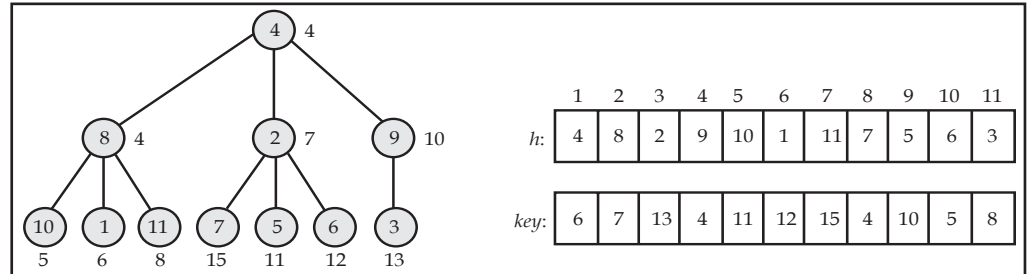


The depth of a d -heap with n nodes is $\leq \lceil \log_d n \rceil$.

Notes

Implementing d-Heaps as Arrays

1. The nodes of a d-heap can be stored in an array in breadth-first order.
 - (a) allows indices for parents and children to be calculated directly, eliminating the need for pointers



2. If i is the index of an item x , then $\lfloor (i-1)/d \rfloor$ is the index of $p(x)$ and the indices of the children of x are in the range $[d(i-1) + 2 .. di + 1]$.
3. When the key of an item is decreased, we can restore heap-order, by repeatedly swapping the item with its parent.
4. Similarly, for increasing an item's key.

d-Heap Operations

```

item function findmin(heap h);
return if h = {} [] null; ! h ≠ {} @h(1) fi;
end;

procedure siftup(item i, integer x, modifies heap h);
integer p;
p := ⌊(x-1)/d⌋;
do p ≠ 0 and key(h(p)) > key(i) =>
h(x) := h(p); x := p; p := ⌊(p-1)/d⌋ ;
od;
h(x) := i;
end;

procedure insert(item i; modifies heap h);
siftup(i, |h| + 1, h);
end;

integer function minchild(integer x, heap h);
integer i, minc;
minc := d(x-1) + 2;
if minc > |h| => return 0; fi;
i := minc + 1;
do i ≤ min { |h|, dx + 1 } =>
if key(h(i)) < key(h(minc)) => minc := i; fi;
i := i + 1;

```

```

od;
return minc;
end;
procedure siftDown(item i, integer x, modifies heap h);
integer c;
c := minchild(x,h);
do c ≠ 0 and key(h(c)) < key(i) =>
h(x) := h(c); x := c; c := minchild(x,h);
od;
h(x) := i;
end;
procedure delete(item i, modified heap h);
item j; j := h(|h|); h(|h|) := null;
if i ≠ j and key(j) ≤ key(i) => siftUp(j, h-1(i), h);
| i ≠ j and key(j) > key(i) => siftDown(j, h-1(i), h);
fi;
end;
item function deleteMin(modifies heap h);
item i;
if h = {} => return null; fi;
i := h(1); delete(h(1), h);
return i;
end;
procedure changeKey(item i, keytype k, modified heap h);
item ki; ki := key(i); key(i) := k;
if k < ki => siftUp(i, h-1(i), h);
| k > ki => siftDown(j, h-1(i), h);
fi;
end;

```

D-Heap Algorithms

A d-heap is a tree with the property that a parent's value is smaller than (or equal to) the value of any of its d children. For example, the min heap we have seen in class is a 2-heap.

Given N the number of elements in the d-heap, in terms of d and N what is the time cost in the worst case (big Oh notation) of each of the following operations.

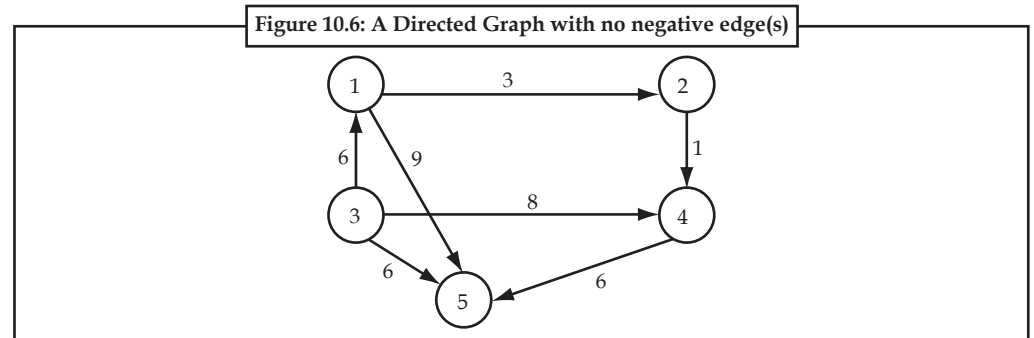
1. **buildHeap**: Builds a d-heap from a list of naturals read from standard input.
2. **insertHeap**: Inserts a new element into the d-heap.
3. **decreaseKey** (p, Δ): lowers the value of the item at position p by a positive amount Δ .
4. **increaseKey** (p, Δ): increases the value of the item at position p by a positive amount Δ .
5. **remove**: removes the node at position p from the d-heap. This is done by performing decreaseKey (p, ∞) and then performing deleteMin ().

Notes

Best example of D-Heap is "DIJKSTRA'S ALGORITHM"

Dijkstra's algorithm (named after its discover, Dutch computer scientist E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination with non-negative weight edge.

It turns out that one can find the shortest paths from a given source to all vertices (points) in a graph in the same time. Hence, this problem is sometimes called the single-source shortest paths problem. Dijkstra's algorithm is a greedy algorithm, which finds shortest path between all pairs of vertices in the graph. Before describing the algorithms formally, let us study the method through an example.



Dijkstra's algorithm keeps two sets of vertices:

S is the set of vertices whose shortest paths from the source have already been determined

Q = V-S is the set of remaining vertices.

The other data structures needed are:

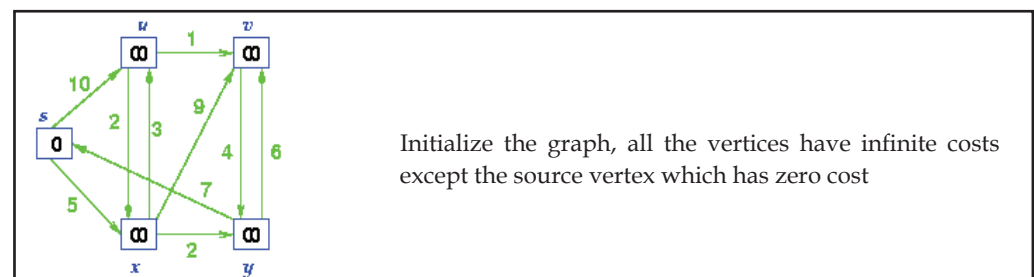
d array of best estimates of shortest path to each vertex from the source pi an array of predecessors for each vertex. predecessor is an array of vertices to which shortest path has already been determined.

The basic operation of Dijkstra's algorithm is edge relaxation. If there is an edge from u to v, then the shortest known path from s to u can be extended to a path from s to v by adding edge (u,v) at the end. This path will have length $d[u]+w(u,v)$. If this is less than $d[v]$, we can replace the current value of $d[v]$ with the new value.

The predecessor list is an array of indices, one for each vertex of a graph. Each vertex entry contains the index of its predecessor in a path through the graph.

Operation of Algorithm

The following sequence of diagrams illustrate the operation of Dijkstra's Algorithm. The bold vertices indicate the vertex to which shortest path has been determined.



Notes

From all the adjacent vertices, choose the closest vertex to the source s.

As we initialized $d[s]$ to 0, it's s. (shown in bold circle)

Add it to S

Relax all vertices adjacent to s, i.e u and x

Update vertices u and x by 10 and 5 as the distance from s.

Choose the nearest vertex, x.

Relax all vertices adjacent to x

Update predecessors for u, v and y.

Predecessor of x = s

Predecessor of v = x, s

Predecessor of y = x, s

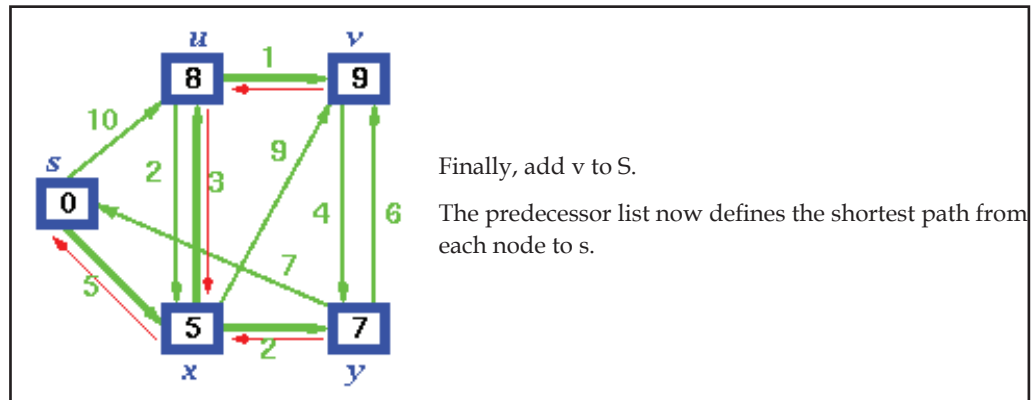
add x to S

Now y is the closest vertex. Add it to S.

Relax v and adjust its predecessor.

u is now closest, add it to S and adjust its adjacent vertex, v.

Notes



Dijkstra's algorithm

* Initialise d and pi* for each vertex v in V(g)

g.d[v] := infinity

g.pi[v] := nil

g.d[s] := 0;

* Set S to empty *

S := { 0 }

Q := V(g)

* While (V-S) is not null*

while not Empty(Q)

1. Sort the vertices in V-S according to the current best estimate of their distance from the source u := Extract-Min (Q);
2. Add vertex u, the closest vertex in V-S, to S, AddNode(S, u);
3. Relax all the vertices still in V-S connected to u relax(Node u, Node v, double w[[]])
 if d[v] > d[u] + w[u][v] then
 d[v] := d[u] + w[u][v]
 pi[v] := u

In summary, this algorithm starts by assigning a weight of infinity to all vertices, and then selecting a source and assigning a weight of zero to it. Vertices are added to the set for which shortest paths are known. When a vertex is selected, the weights of its adjacent vertices are relaxed. Once all vertices are relaxed, their predecessor's vertices are updated (pi). The cycle of selection, weight relaxation and predecessor update is repeated until the shortest path to all vertices has been found.

10.5 Summary

- a heap is a partially sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less (for the sake of simplicity, I will assume that all orderings are from least to greatest) than either of its children.

- Additionally, a heap is a “complete tree” -- a complete tree is one in which there are no gaps between leaves.
- For instance, a tree with a root node that has only one child must have its child as the left node.
- More precisely, a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.

10.6 Keywords

Binary Heap: A binary heap is a heap-ordered binary tree which has a very special shape called a complete tree.

Discrete Event Simulation: One of the most important applications of priority queues is in *discrete event simulation*.

d-Heap: d-heap is a priority queue data structure, a generalization of the binary heap in which the nodes have d children instead of 2.

Heap: A heap is a specialized tree-based data structure that satisfies the heap property: if B is a child node of A, then $\text{key}(A) \geq \text{key}(B)$.

10.7 Self Assessment

Fill in the blanks:

1. A heap is a partially sorted
2. Complete trees and perfect trees are
3. In a heap the is found at the root.
4. The method removes from a priority queue the item having the smallest key.
5. The of an event depends on the type of that event.
6. The minimum priority item in a min-heap may always be found at of the array.
7. The heap is one maximally-efficient implementation of an abstract data type called a
8. calculations are used to find the parent and the children of a given node in the tree.

10.8 Review Questions

1. What do you mean by heaps?
2. Explain complete binary tree.
3. Prove that “A complete binary tree of height $h \geq 0$ contains at least 2^h and at most $2^{h+1} - 1$ nodes.”
4. Prove that “The internal path length of a binary tree with n nodes is at least as big as the internal path length of a *complete* binary tree with n nodes.”
5. Explain the implementation of binary heap.
6. Describe how will you put items into binary heaps.

Notes

7. Write a method and the corresponding recursive function to traverse a binary tree (in whatever order you find convenient) and dispose of all its nodes. Use this method to implement a Binary_tree destructor.
8. Consider a heap of n keys, with x_k being the key in position k (in the contiguous representation) for $0 \leq k < n$. Prove that the height of the subtree rooted at x_k is the greatest integer not exceeding $\lg(n/(k+1))$, for all k satisfying $0 \leq k < n$.
9. "A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted." Discuss.
10. "The enqueue method of the BinaryHeap class takes as its argument the item to be inserted in the heap." Explain.

Answers: Self Assessment

- | | | |
|-------------------|--------------------|-----------------|
| 1. binary tree | 2. closely related | 3. smallest key |
| 4. dequeueMin | 5. simulation | 6. position 0 |
| 7. priority queue | 8. Array subscript | |

10.9 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*. Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 11: Leftist Heaps and Binomial Queues

Notes

CONTENTS

Objectives

Introduction

11.1 Leftist Heaps

11.1.1 Leftist Trees

11.1.2 Implementation

11.1.3 Merging Leftist Heaps

11.1.4 Putting items into a Leftist Heap

11.1.5 Removing items from a Leftist Heap

11.2 Skew Heaps

11.3 Binomial Queues

11.4 Summary

11.5 Keywords

11.6 Self Assessment

11.7 Review Questions

11.8 Further Readings

Objectives

After studying this unit, you will be able to:

- State the concept leftist heaps
- Realise skew heaps
- Explain the binomial queues

Introduction

Numerous data structures have been developed that can support efficient implementations of all the priority-queue operations. Most of them are based on direct linked representation of heap-ordered trees. Two links are needed for moving down the tree (either to both children in a binary tree or to the first child and next sibling in a binary tree representation of a general tree) and one link to the parent is needed for moving up the tree. Developing implementations of the heap-ordering operations that work for any (heap-ordered) tree shape with explicit nodes and links or other representation is generally straightforward. The difficulty lies in dynamic operations such as insert, remove, and join, which require us to modify the tree structure. Different data structures are based on different strategies for modifying the tree structure while still maintaining balance in the tree.

11.1 Leftist Heaps

A leftist heap is a heap-ordered binary tree which has a very special shape called a leftist tree. One of the nice properties of leftist heaps is that it is possible to merge two leftist heaps efficiently. As a result, leftist heaps are suited for the implementation of mergeable priority queues.

Notes

11.1.1 Leftist Trees

A *leftist tree* is a tree which tends to “lean” to the left. The tendency to lean to the left is defined in terms of the shortest path from the root to an external node. In a leftist tree, the shortest path to an external node is always found on the right.

Every node in binary tree has associated with it a quantity called its *null path length* which is defined as follows:

Null Path and Null Path Length

Consider an arbitrary node x in some binary tree T . The *null path* of node x is the shortest path in T from x to an external node of T .

The *null path length* of node x is the length of its null path.

Sometimes it is convenient to talk about the null path length of an entire tree rather than of a node:

Null Path Length of a Tree

The *null path length* of an empty tree is zero and the null path length of a non-empty binary tree $T = \{R, T_L, T_R\}$ is the null path length its root R .

When a new node or subtree is attached to a given tree, it is usually attached in place of an external node. Since the null path length of a tree is the length of the shortest path from the root of the tree to an external node, the null path length gives a lower bound on the cost of insertion.



Example: The running time for insertion in a binary search tree, is at least

$$d\tau_{(\text{compare})} + \Omega(d)$$

where d is the null path length of the tree.

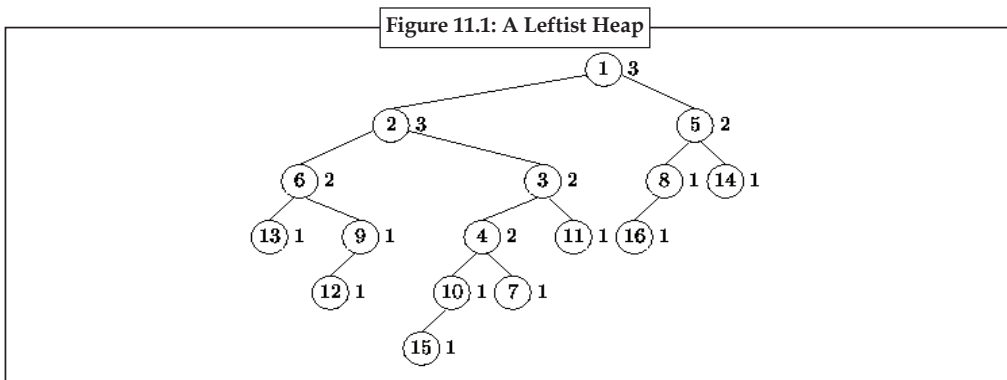
A *leftist tree* is a tree in which the shortest path to an external node is always on the right. This informal idea is defined more precisely in terms of the null path lengths as follows:

Definition (Leftist Tree)

A *leftist tree* is a binary tree T with the following properties:

1. Either $T = l$; or
2. $T = \{R, T_L, T_R\}$, where both T_L and T_R are leftist trees which have null path lengths d_L and d_R , respectively, such that $d_L \geq d_R$.

Figure 11.1 shows an example of a leftist heap. A leftist heap is simply a heap-ordered leftist tree. The external depth of the node is shown to the right of each node in Figure 11.1. The figure clearly shows that it is not necessarily the case in a leftist tree that the number of nodes to the left of a given node is greater than the number to the right. However, it is always the case that the null path length on the left is greater than or equal to the null path length on the right for every node in the tree.



The reason for our interest in leftist trees is illustrated by the following theorems:

Theorem-11.1

Consider a leftist tree T which contains n internal nodes. The path leading from the root of T downwards to the rightmost external node contains at most $\lfloor \log_2(n+1) \rfloor$ nodes.

Proof Assume that T has null path length d . Then T must contain at least 2^{d-1} leaves. Otherwise, there would be a shorter path than d from the root of T to an external node.

A binary tree with exactly l leaves has exactly $l-1$ non-leaf internal nodes. Since T has at least 2^{d-1} leaves, it must contain at least $n \geq 2^d - 1$ internal nodes altogether. Therefore, $d \leq \log_2(n+1)$.

Since T is a leftist tree, the shortest path to an external node must be the path on the right. Thus, the length of the path to the rightmost external is at most $\lfloor \log_2(n+1) \rfloor$.

There is an interesting dichotomy between AVL balanced trees and leftist trees. The shape of an AVL tree satisfies the AVL balance condition which stipulates that the difference in the heights of the left and right subtrees of every node may differ by at most one. The effect of AVL balancing is to ensure that the height of the tree is $O(\log n)$.

On the other hand, leftist trees have an "imbalance condition" which requires the null path length of the left subtree to be greater than or equal to that of the right subtree. The effect of the condition is to ensure that the length of the right path in a leftist tree is $O(\log n)$.

Therefore, by devising algorithms for manipulating leftist heaps which only follow the right path of the heap, you can achieve running times which are logarithmic in the number of nodes.

The dichotomy also extends to the structure of the algorithms. For example, an imbalance sometimes results from an insertion in an AVL tree. The imbalance is rectified by doing rotations. Similarly, an insertion into a leftist tree may result in a violation of the "imbalance condition." That is, the null path length of the right subtree of a node may become greater than that of the left subtree. Fortunately, it is possible to restore the proper condition simply by swapping the left and right subtrees of that node.

11.1.2 Implementation

This section presents an implementation of leftist heaps that is based on the binary tree implementation. Program 11.1 introduces the `LeftistHeap` class. The `LeftistHeap` class extends the `BinaryTree` class introduced in Program 11.1 and it implements the `MergeablePriorityQueue` interface defined in Program 11.1.

Notes

```
public class LeftistHeap
    extends BinaryTree
    implements MergeablePriorityQueue
{
    protected int nullPathLength;
    //...
}
```

Program 11.1: LeftistHeap Fields**11.1.3 Merging Leftist Heaps**

In order to merge two leftist heaps, say h_1 and h_2 , declared as follows

```
MergeablePriorityQueue h1 = new LeftistHeap ();
MergeablePriorityQueue h2 = new LeftistHeap ();
```

I invoke the `merge` method like this:

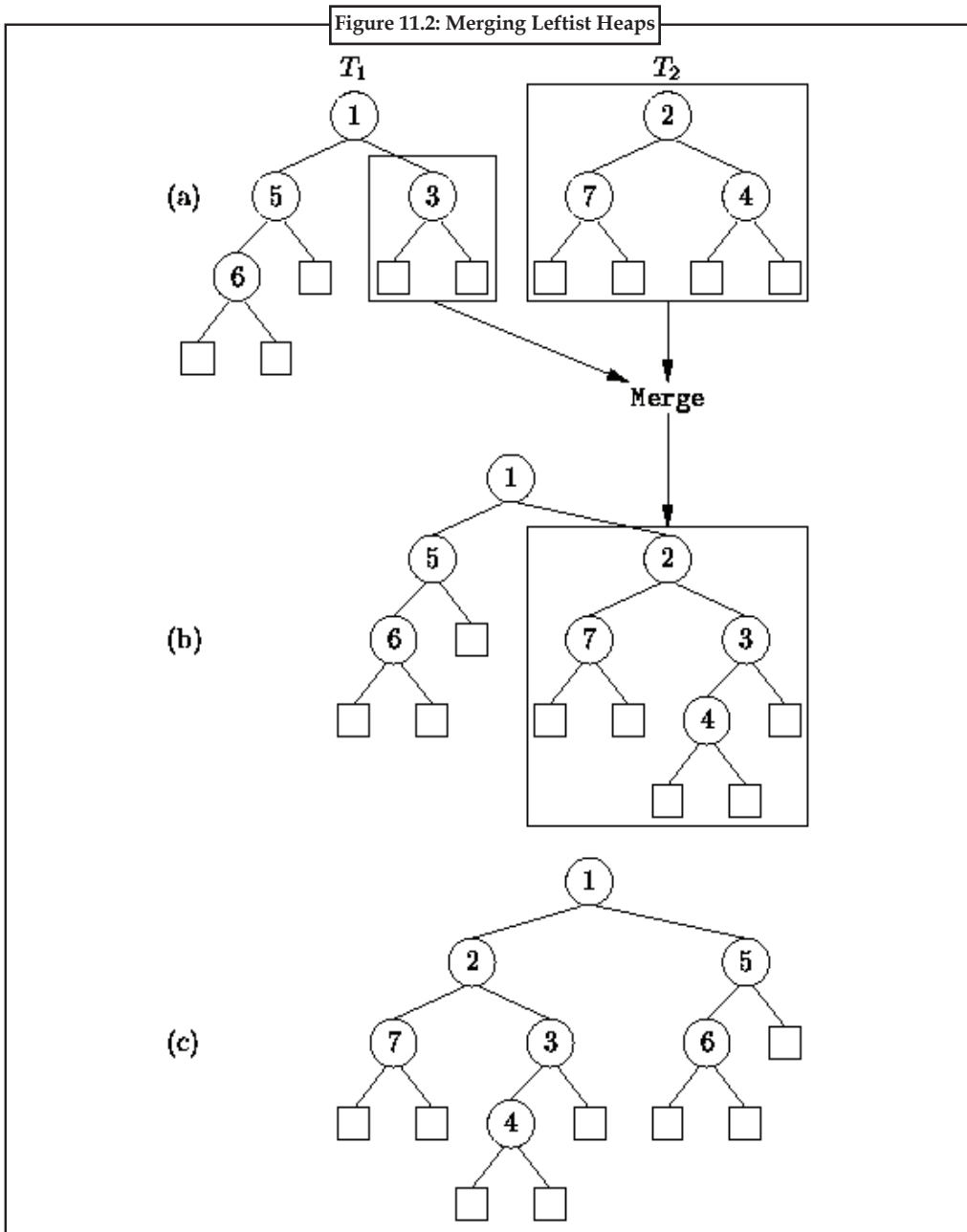
```
h1.merge (h2);
```

The effect of the `merge` method is to take all the nodes from h_2 and to attach them to h_1 , thus leaving h_2 as the empty heap.

In order to achieve a logarithmic running time, it is important for the `merge` method to do all its work on the right sides of h_1 and h_2 . It turns out that the algorithm for merging leftist heaps is actually quite simple.

To begin with, if h_1 is the empty heap, then you can simply swap the contents of h_1 and h_2 . Otherwise, let us assume that the root of h_2 is larger than the root of h_1 . Then you can merge the two heaps by recursively merging h_2 with the *right* subheap of h_1 . After doing so, it may turn out that the right subheap of h_1 now has a larger null path length than the left subheap. This you rectify by swapping the left and right subheaps so that the result is again leftist. On the other hand, if h_2 initially has the smaller root, I simply exchange the roles of h_1 and h_2 and proceed as above.

Figure 11.2 illustrates the merge operation. In this example, I wish to merge the two trees T_1 and T_2 shown in Figure 11.2 (a). Since T_2 has the larger root, it is recursively merged with the right subtree of T_1 . The result of that merge replaces the right subtree of T_1 as shown in Figure 11.2 (b). Since the null path length of the right subtree is now greater than the left, the subtrees of T_1 are swapped giving the leftist heap shown in Figure 11.2 (c).



Program 11.2 gives the code for the merge method of the LeftistHeap class. The merge method makes use of two other methods, swapContents and swapSubtrees. The swapContents method takes as its argument a leftist heap, and exchanges all the contents (key and subtrees) of this heap with the given one. The swapSubtrees method exchanges the left and right subtrees of this node. The implementation of these routines is trivial and is left as a project for the reader. Clearly, the worst-case running time for each of these routines is $O(1)$.

The merge method only visits nodes on the rightmost paths of the trees being merged. Suppose you are merging two trees, say T_1 and T_2 , with null path lengths d_1 and d_2 , respectively. Then the running time of the merge method is

$$(d_1 - 1 + d_2 - 1)\tau(\text{isGT}) + O(d_1 + d_2)$$

Notes

where $\tau(\text{isGT})$ is time required to compare two keys. If you assume that the time to compare two keys is a constant, then you get $O(\log n_1 + \log n_2)$, where n_1 and n_2 are the number of internal nodes in trees T_1 and T_2 , respectively.

```
public class LeftistHeap
    extends BinaryTree
    implements MergeablePriorityQueue
{
    protected int nullPathLength;
    public void merge (MergeablePriorityQueue queue)
    {
        LeftistHeap arg = (LeftistHeap) queue;
        if (isEmpty ())
            swapContents (arg);
        else if (!arg.isEmpty ())
        {
            if (((Comparable) getKey ()) .isGT (
                ((Comparable) arg.getKey ()) )
                swapContents (arg);
            getRightHeap ().merge (arg);
            if (getLeftHeap ().nullPathLength <
                getRightHeap ().nullPathLength)
                swapSubtrees ();
            nullPathLength = 1 + Math.min (
                getLeftHeap ().nullPathLength,
                getRightHeap ().nullPathLength);
        }
    }
    // ...
}
```

Program 11.2: LeftistHeap Class merge Method

11.1.4 Putting items into a Leftist Heap

The enqueue method of the LeftistHeap class is used to put items into the heap. enqueue is easily implemented using the merge operation. That is, to enqueue an item in a given heap, I simply create a new heap containing the one item to be enqueued and merge it with the given heap. The algorithm to do this is shown in Program 11.3.

```
public class LeftistHeap
    extends BinaryTree
    implements MergeablePriorityQueue
{
    protected int nullPathLength;
    public void enqueue (Comparable object)
        { merge (new LeftistHeap (object)); }
    // ...
}
```

Program 11.3: LeftistHeap Class enqueue Method

The expression for the running time for the `insert` operation follows directly from that of the merge operation. That is, the time required for the `insert` operation in the worst case is

$$(d - 1)\tau(\text{isGT}) + O(d),$$

where d is the null path length of the heap into which the item is inserted. If you assume that two keys can be compared in constant time, the running time for `insert` becomes simply $O(\log n)$, where n is the number of nodes in the tree into which the item is inserted.



Task

Discuss the use of enqueue method.

11.1.5 Removing items from a Leftist Heap

The `findMin` method locates the item with the smallest key in a given priority queue and the `dequeueMin` method removes it from the queue. Since the smallest item in a heap is found at the root, the `findMin` operation is easy to implement. Program 11.4 shows how it can be done. Clearly, the running time of the `findMin` operation is $O(1)$.

```
public class LeftistHeap
    extends BinaryTree
    implements MergeablePriorityQueue
{
    protected int nullPathLength;
    public Comparable findMin ()
    {
        if (isEmpty ())
            throw new ContainerEmptyException ();
        return (Comparable) getKey ();
    }
    // ...
}
```

Program 11.4: LeftistHeap Class findMin Method

Since the smallest item in a heap is at the root, the `dequeueMin` operation must delete the root node. Since a leftist heap is a binary heap, the root has at most two children. In general when the root is deleted, you are left with two non-empty leftist heaps. Since you already have an efficient way to merge leftist heaps, the solution is to simply merge the two children of the root to obtain a single heap again! Program 11.5 shows how the `dequeueMin` operation of the `LeftistHeap` class can be implemented.

```
public class LeftistHeap
    extends BinaryTree
    implements MergeablePriorityQueue
{
    protected int nullPathLength;
    public Comparable dequeueMin ()
    {
        if (isEmpty ())
```

Notes

```

        throw new ContainerEmptyException ();
    Comparable result = (Comparable) getKey ();
    LeftistHeap oldLeft = getLeftHeap ();
    LeftistHeap oldRight = getRightHeap ();
    purge ();
    swapContents (oldLeft);
    merge (oldRight);
    return result;
}
// ...
}

```

Program 11.5: LeftistHeap Class dequeueMin Method

The running time of Program 11.5 is determined by the time required to merge the two children of the root (line 17) since the rest of the work in `dequeueMin` can be done in constant time. Consider the running time to delete the root of a leftist heap T with n internal nodes. The running time to merge the left and right subtrees of T

$$(d_L - 1 + d_R - 1)\tau(\text{isGT}) + O(d_L + d_R),$$

where d_L and d_R are the null path lengths of the left and right subtrees T , respectively. In the worst case, $d_R = 0$ and $d_L = \lfloor \log_2 n \rfloor$.

11.2 Skew Heaps

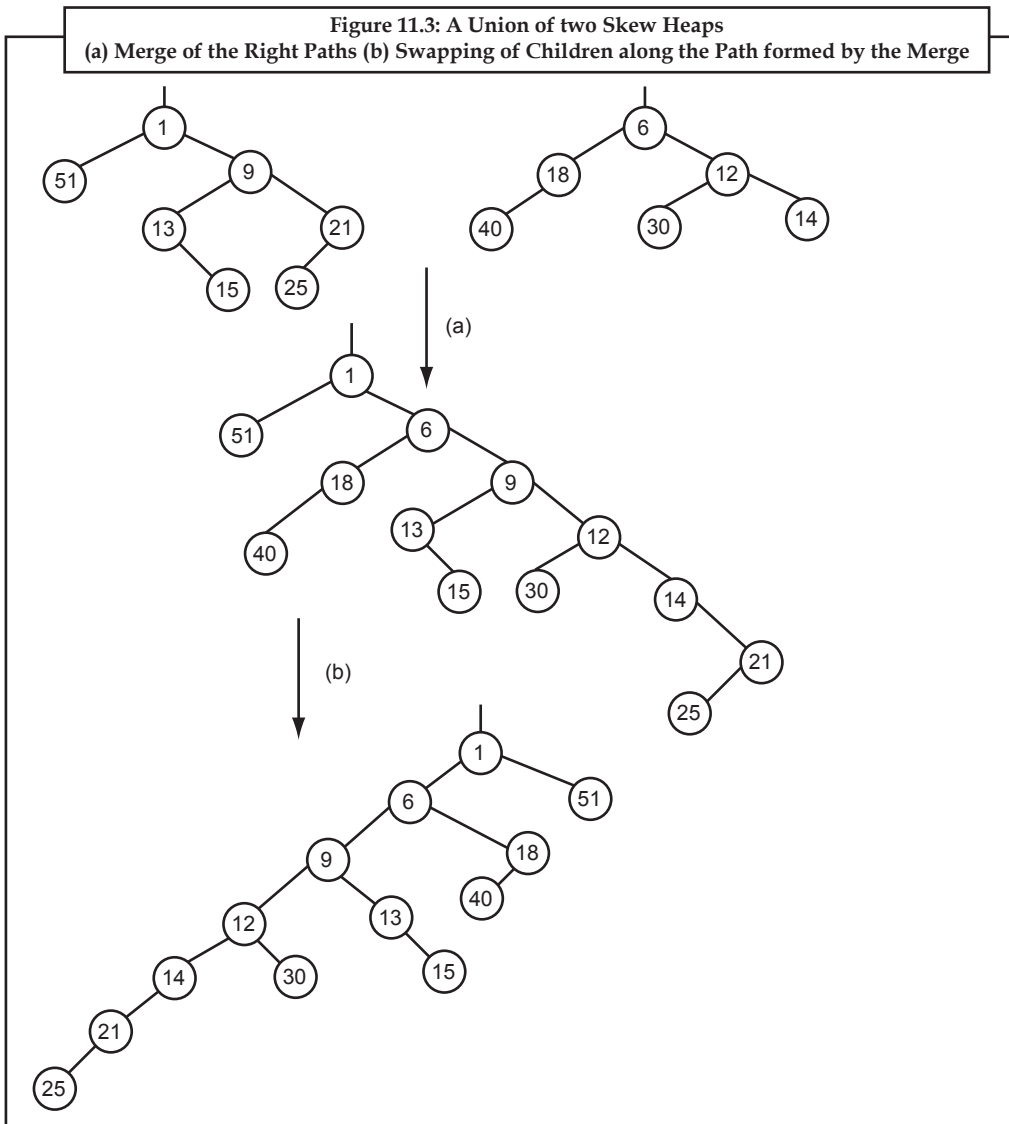
There are several ways to implement heaps in a self-adjusting fashion. The one I shall discuss is called skew heaps as proposed by Sleator and Tarjan, and is analogous to leftist heaps. A skew heap is a heap-ordered binary tree. That is, it is a binary tree with one key in each node so that for each node x other than the root, the key at node x is no less than the key at the parent of x . To represent such a tree you store in each node x its associated key, denoted $\text{key}(x)$ and two pointers $\text{left}(x)$ and $\text{right}(x)$, to its left child and right child, respectively. If x has no left child I define $\text{left}(x) = L$; if x has no right child I define $\text{right}(x) = L$. Access to the tree is by a pointer to its root; you represent an empty tree by a pointer to L .

With this representation you can carry out the various heap operations as follows. I perform `makeheap(h)` in $O(1)$ time by initializing h to L . Since heap order implies that the root is a minimum key in the tree, you can carry out `findmin(h)` in $O(1)$ time by returning the key at the root; returning null if the heap is empty. You perform `insert` and `deletemin` using `union`. To carry out `insert(k, h)`, I make k into a one-node heap and `Union` it with h . To carry out `deletemin(h)`, if h is not empty I replace h by the `Union` of its left and right subtrees and return the key at the original root. (If h is originally empty you return null.)

To perform `union(h1, h2)`, you form a single tree by traversing down the right paths of $h1$ and $h2$, merging them into a single right path with keys in nondecreasing order. First assume the left subtrees of nodes along the merge path do not change. (Figure 11.3(a).) The time for the `Union` operation is bounded by a constant times the length of the merge path. To make `Union` efficient, you must keep right paths short. In leftist heaps this is done by maintaining the invariant that, for any node x , the right path descending from x is a shortest path down to a missing node. Maintaining this invariant requires storing at every node the length of a shortest path down to a missing node; after the merge you walk back up the merge path, updating the shortest path lengths and swapping left and right children as necessary to maintain the leftist property. The length of the right path in a leftist heap of n nodes is at most $\log n$, implying an $O(\log n)$

worst-case time bound for each of the heap operations, where n is the number of nodes in the heap or heaps involved.

Notes



In our self-adjusting version of this data structure, I perform the Union operation by merging the right paths of the two trees and then swapping the left and right children of every node on the merge path except the lowest. (Figure 11.3(b)) This makes the potentially long right path formed by the merge into a left path you call the resulting data structure a skew heap.



Task

“A skew heap is a heap-ordered binary tree.” Explain.

11.3 Binomial Queues

A binomial queue is a priority queue that is implemented not as a single tree but as a collection of heap-ordered trees. A collection of trees is called a *fores*. Each of the trees in a binomial queue

Notes

has a very special shape called a binomial tree. Binomial trees are general trees. i.e., the maximum degree of a node is not fixed.

The remarkable characteristic of binomial queues is that the merge operation is similar in structure to binary addition, i.e., the collection of binomial trees that make up the binomial queue is like the set of bits that make up the binary representation of a non-negative integer. Furthermore, the merging of two binomial queues is done by adding the binomial trees that make up that queue in the same way that the bits are combined when adding two binary numbers.

Binomial Trees

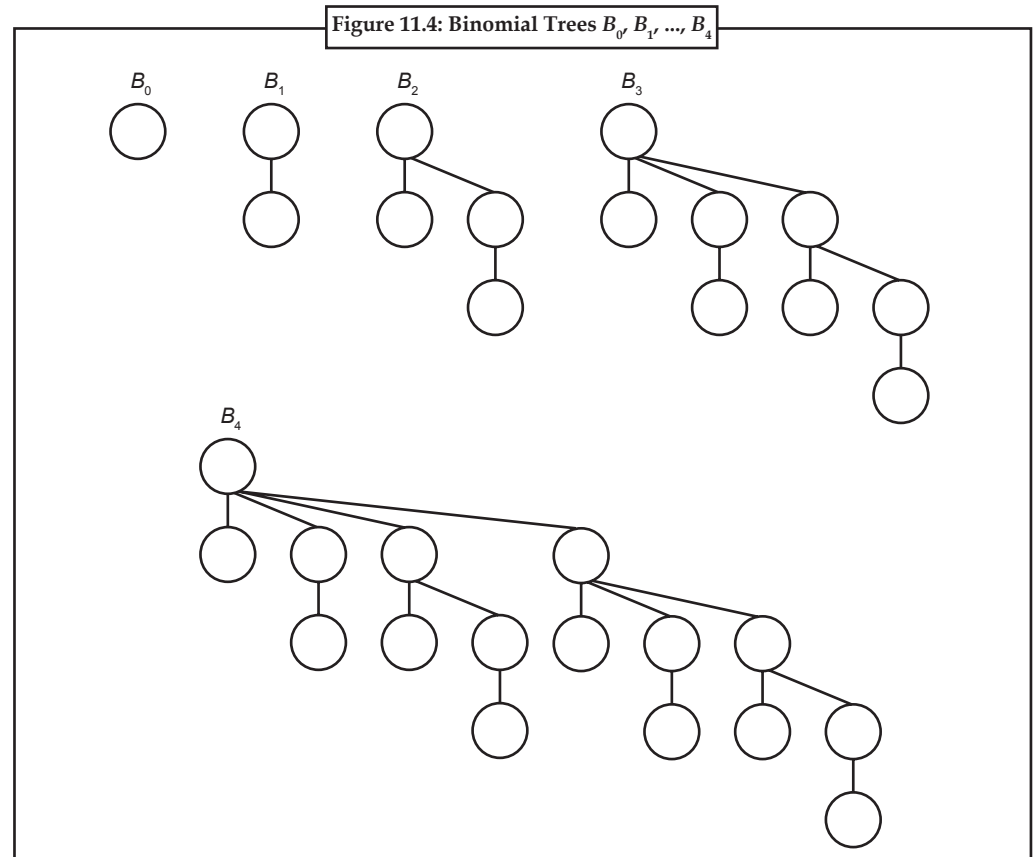
A binomial tree is a general tree with a very special shape:

Definition (Binomial Tree)

The binomial tree of order $k \geq 0$ with root R is the tree B_k defined as follows

1. If $k = 0$, $B_k = B_0 = \{R\}$ i.e., the binomial tree of order zero consists of a single node, R .
2. If $k > 0$, $B_k = \{R, B_0, B_1, \dots, B_{k-1}\}$ i.e., the binomial tree of order $k > 0$ comprises the root R , and k binomial subtrees, B_0, B_1, \dots, B_{k-1} .

Figure 11.4 shows the first five binomial trees, $B_0 - B_4$. It follows directly from the root of B_k , the binomial tree of order k , has degree k . Since k may arbitrarily large, so too can the degree of the root. Furthermore, the root of a binomial tree has the largest fanout of any of the nodes in that tree.



The number of nodes in a binomial tree of order k is a function of k :

Theorem-11.3

The binomial tree of order k , B_k , contains 2^k nodes.

Proof (By induction). Let n_k be the number of nodes in B_k , a binomial tree of order k .

Base Case: By definition, B_0 consists of a single node. Therefore $n_0 = 1 = 2^0$.

Inductive Hypothesis: Assume that $n_k = 2^k$ for $k = 0, 1, 2, \dots, l$, for some $l \geq 0$. Consider the binomial tree of order $l + 1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \dots, B_l\}.$$

Therefore the number of nodes in B_{l+1} is given by

$$\begin{aligned} n_{l+1} &= 1 + \sum_{i=0}^l n_i \\ &= 1 + \sum_{i=0}^l 2^i \\ &= 1 + \frac{2^{l+1} - 1}{2 - 1} \\ &= 2^{l+1} \end{aligned}$$

Therefore, by induction on l , $n_k = 2^k$ for all $k \geq 0$.

It follows from Theorem 11.3 that binomial trees only come in sizes that are a power of two. i.e., $n_k \in \{1, 2, 4, 8, 16, \dots\}$. Furthermore, for a given power of two, there is exactly one shape of binomial tree.

Theorem-11.4

The height of B_k , the binomial tree of order k , is k .

Proof (By induction). Let h_k be the height of B_k , a binomial tree of order k .

Base Case: By definition, B_0 consists of a single node. Therefore $h_0 = 0$.

Inductive Hypothesis: Assume that $h_k = k$ for $k = 0, 1, 2, \dots, l$, for some $l \geq 0$. Consider the binomial tree of order $l + 1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \dots, B_l\}.$$

Therefore the height B_{l+1} is given by

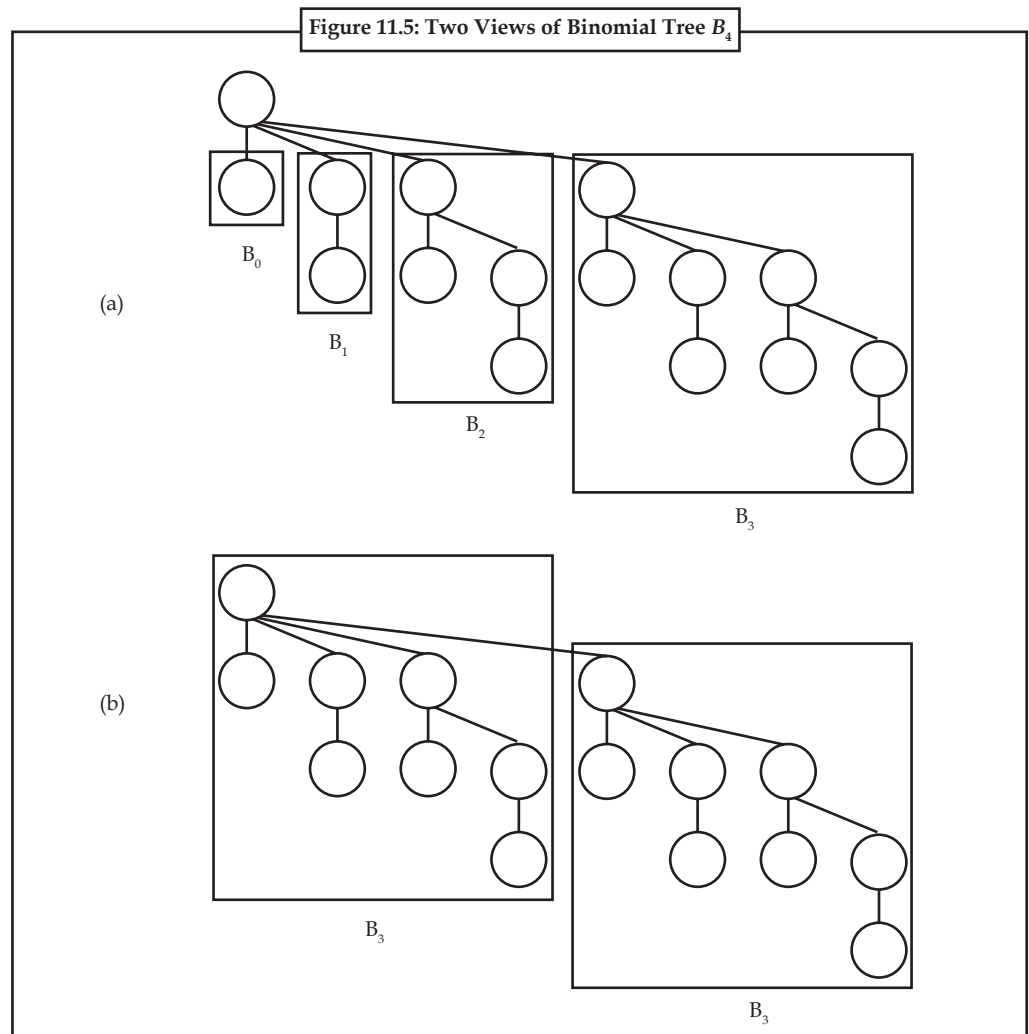
$$\begin{aligned} h_{l+1} &= 1 + \max_{0 \leq i \leq l} h_i \\ &= 1 + \max_{0 \leq i \leq l} i \\ &= l + 1. \end{aligned}$$

Therefore, by induction on l , $h_k = k$ for all $k \geq 0$.

Theorem 11.4 tells us that the height of a binomial tree of order k is k and tells us that the number of nodes is $n_k = 2^k$. Therefore, the height of B_k is exactly $O(\log n)$.

Figure 11.5 shows that there are two ways to think about the construction of binomial trees, i.e., binomial B_k consists of a root node to which the k binomial trees B_0, B_1, \dots, B_{k-1} are attached as shown in Figure 11.5 (a).

Notes



Alternatively, you can think of B_k as being comprised of two binomial trees of order $k-1$.



Example: Figure 11.5 (b) shows that B_4 is made up of two instances of B_3 . In general, suppose you have two trees of order $k-1$, say B_{k-1}^1 and B_{k-1}^2 , where $B_{k-1}^1 = \{R^1, B_{0'}^1, B_{1'}^1, B_{2'}^1, \dots, B_{k-2'}^1\}$. Then you can construct a binomial tree of order k by combining the trees to get

$$B_k = \{R^1, B_{0'}^1, B_{1'}^1, B_{2'}^1, \dots, B_{k-2'}^1, B_{k-1}^2\}.$$

Why do you call B_k a *binomial tree*? It is because the number of nodes at a given depth in the tree is determined by the *binomial coefficient*. And the binomial coefficient derives its name from the *binomial theorem*. And the binomial theorem tells us how to compute the n^{th} power of a *binomial*. And a binomial is an expression which consists of two terms, such as $x+y$. That is why it is called a binomial tree!

Task "Binomial tree is a general tree with a very special shape" Discuss.

11.4 Summary

- A power-of-2 heap is a left-heap-ordered tree consisting of a root node with an empty right subtree and a complete left subtree.
- The tree corresponding to a power-of-2 heap by the left-child, right-sibling correspondence is called a binomial tree.
- Binomial trees and power-of-2 heaps are equivalent. I work with both representations because binomial trees are slightly easier to visualize, whereas the simple representation of power-of-2 heaps leads to simpler implementations.

11.5 Keywords

Binomial Queue: A binomial queue is a priority queue that is implemented not as a single tree but as a collection of heap-ordered trees.

Leftist Heap: A leftist heap is a heap-ordered binary tree which has a very special shape called a leftist tree.

Leftist Tree: A *leftist tree* is a tree which tends to “lean” to the left. The tendency to lean to the left is defined in terms of the shortest path from the root to an external node.

Skew Heap: A skew heap is a heap-ordered binary tree.

11.6 Self Assessment

Fill in the blanks:

1. A collection of trees is called a
2. Every node in binary tree has associated with it a quantity called its
3. The *null path length* of an empty tree is
4. A *leftist tree* is a tree in which the shortest path to an external node is always on the
5. The method of the `LeftistHeap` class is used to put items into the heap.
6. The method locates the item with the smallest key in a given priority queue.
7. Skew heaps as proposed by
8. Each of the trees in a binomial queue has a very special shape called a

11.7 Review Questions

1. What do you mean by leftist heaps?
2. Describe null path and null path length.
3. Write the methods to implement queues by the simple but slow technique of keeping the front of the queue always in the first position of a linear array.
4. Prove that “Consider a leftist tree T which contains n internal nodes. The path leading from the root of T downwards to the rightmost external node contains at most $\lfloor \log_2(n+1) \rfloor$ nodes.”

Notes

5. Write methods to implement queues in a circular array with one unused entry in the array. That is, we consider that the array is full when the rear is two positions before the front; when the rear is one position before, it will always indicate an empty queue.
6. Prove that "The binomial tree of order k , B_k , contains 2^k nodes. extbfProof (By induction). Let n_k be the number of nodes in B_k , a binomial tree of order k ."
7. Write a menu driven demonstration program for manipulating a deque of characters, similar to the Extended_queue demonstration program.
8. Write the class definition and the method implementations needed to implement a deque in a linear array.
9. Write the methods needed to implement a deque in a circular array. Consider the class Deque as derived from the class Queue.
10. Write a method to implement queues, where the implementation does not keep a count of the entries in the queue but instead uses the special conditions
 rear = -1 and front = 0
 to indicate an empty queue.

Answers: Self Assessment

- | | | | |
|------------------|---------------------|-----------------------|----------|
| 1. fores | 2. null path length | 3. zero | 4. right |
| 5. enqueue | 6. findMin | 7. Sleator and Tarjan | |
| 8. binomial tree | | | |

11.8 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C, Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*, Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org
www.web-source.net
www.webopedia.com

Unit 12: Sorting

Notes

CONTENTS

Objectives

Introduction

12.1 Internal Sorting

12.2 Insertion Sort

12.2.1 Algorithm of Insertion Sort

12.2.2 Complexity Analysis

12.3 Shell Sort

12.4 Heap Sort

12.5 Merge Sort

12.6 Merging of two Sorted Lists

12.7 Quick Sort

12.8 Bucket Sort

12.9 External Sorting

12.10 Summary

12.11 Keywords

12.12 Self Assessment

12.13 Review Questions

12.14 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss internal sorting
- Explain heap sort, merge sort and quick sort
- Describe external sorting
- Discuss the implementation of heaps

Introduction

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods. Sorting algorithms can be characterised in the following two ways:

1. Simple algorithms which require the order of n^2 (written as $O(n^2)$) comparisons to sort n items.
2. Sophisticated algorithms that require the $O(n \log_2 n)$ comparisons to sort n items.

Notes

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order already present in the data.

There are two basic categories of sorting methods: Internal Sorting and External Sorting. Internal sorting is applied when the entire collection of data to be sorted is small enough so that the sorting can take place within the main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices. Read and write access times are a major concern in determining sorting performances of such methods.

Searching is the process of looking for something: Finding one piece of data that has been stored within a whole group of data. It is often the most time-consuming part of many computer programs. There are a variety of methods, or algorithms, used to search for a data item, depending on how much data there is to look through, what kind of data it is, what type of structure the data is stored in, and even where the data is stored - inside computer memory or on some external medium.

Till now, we have studied a variety of data structures, their types, their use and so on. In this chapter, we will concentrate on some techniques to search a particular data or piece of information from a large amount of data. There are basically two types of searching techniques, Linear or Sequential Search and Binary Search.

Searching is very common task in day-to-day life, where we are involved some or other time, in searching either for some needful at home or office or market, or searching a word in dictionary. In this chapter, we see that if the things are organised in some manner, then search becomes efficient and fast.

All the above facts apply to our computer programs also. Suppose we have a telephone directory stored in the memory in an array which contains Name and Numbers. Now, what happens if we have to find a number? The answer is search that number in the array according to name (given). If the names were organised in some order, searching would have been fast.

12.1 Internal Sorting

The function of sorting or ordering a list of objects according to some linear order is so fundamental that it is ubiquitous in engineering applications in all disciplines. There are two broad categories of sorting methods: Internal sorting takes place in the main memory, where we can take advantage of the random access nature of the main memory; external sorting is necessary when the number and size of objects are prohibitive to be accommodated in the main memory.

Problem

1. Given records r_1, r_2, \dots, r_n , with key values k_1, k_2, \dots, k_n , produce the records in the order $r_{i_1}, r_{i_2}, \dots, r_{i_n}$, such that
$$k_{i_1} \leq k_{i_2} \leq \dots \leq k_{i_n}$$
2. The complexity of a sorting algorithm can be measured in terms of
 - (a) number of algorithm steps to sort n records
 - (b) number of comparisons between keys (appropriate when the keys are long character strings)
 - (c) number of times records must be moved (appropriate when record size is large)

Any sorting algorithm that uses comparisons of keys needs at least $O(n \log n)$ time to accomplish the sorting.

Notes

Sorting Methods

Internal	External
(In memory)	(Appropriate for secondary storage)
quick sort	
heap sort	merge sort
bubble sort	radix sort
insertion sort	poly-phase sort
selection sort	
shell sort	

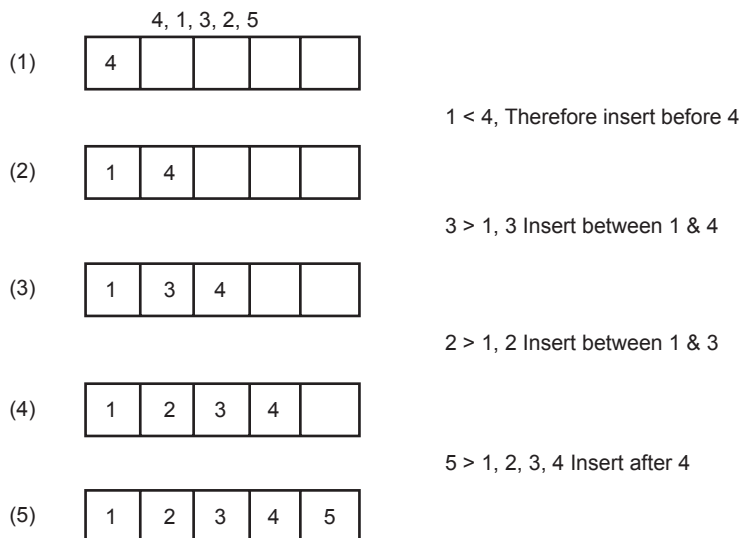
12.2 Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example (refer to Figure 9.1) before presenting the formal algorithm.



Example: Sort the following list using the insertion sort method:



Insertion Sort

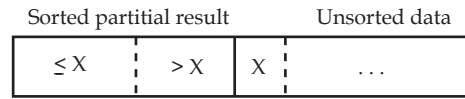
Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

12.2.1 Algorithm of Insertion Sort

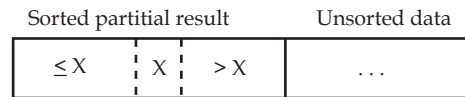
Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains first element of the array and unsorted one contains the rest. At every step, algorithm takes first element in the

Notes

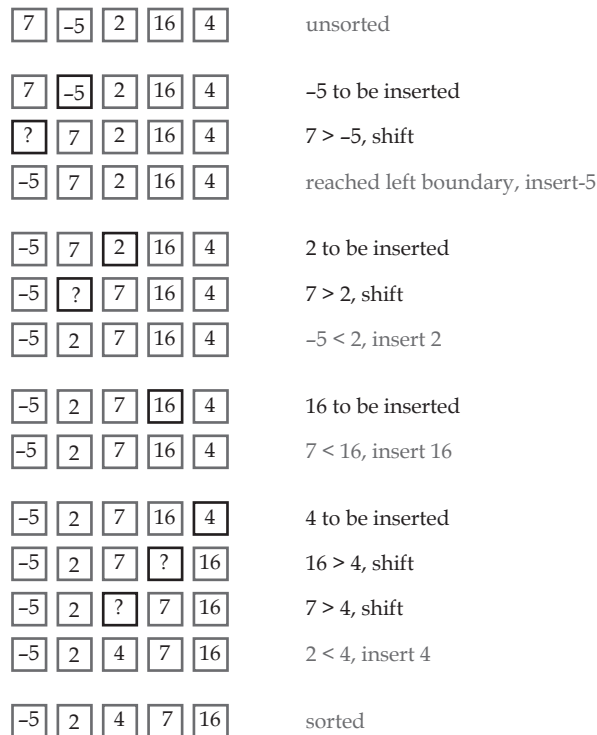
unsorted part and inserts it to the right place of the sorted one. When unsorted part becomes empty, algorithm stops. Sketchy, insertion sort algorithm step looks like this:



becomes



Example: Sort {7, -5, 2, 16, 4} using insertion sort.



The Ideas of Insertion

The main operation of the algorithm is insertion. The task is to insert a value into the sorted part of the array. Let us see the variants of how we can do it.

“Sifting down” using Swaps

The simplest way to insert next element into the sorted part is to sift it down, until it occupies correct position. Initially the element stays right after the sorted part. At each step algorithm compares the element with one before it and, if they stay in reversed order, swap them. Let us see an illustration.


1	3	7	9	16	5	16 > 5, swap
1	3	7	9	5	16	9 > 5, swap
1	3	7	5	9	16	7 > 5, swap
1	3	5	7	9	16	3 < 5 < 7, sifting is done

This approach writes sifted element to temporary position many times. Next implementation eliminates those unnecessary writes.

Shifting Instead of Swapping

We can modify previous algorithm, so it will write sifted element only to the final correct position. Let us see an illustration.

1	3	7	9	16	5	16 > 5, swap
1	3	7	9	?	16	9 > 5, swap
1	3	7	?	9	16	7 > 5, swap
1	3	5	7	9	16	3 < 5 < 7, sifting is done



It is the most commonly used modification of the insertion sort.

Using Binary Search

It is reasonable to use binary search algorithm to find a proper place for insertion. This variant of the insertion sort is called binary insertion sort. After position for insertion is found, algorithm shifts the part of the array and inserts the element. This version has lower number of comparisons, but overall average complexity remains $O(n^2)$. From a practical point of view this improvement is not very important, because insertion sort is used on quite small data sets.

12.2.2 Complexity Analysis

Insertion sort's overall complexity is $O(n^2)$ on average, regardless of the method of insertion. On the almost sorted arrays insertion sort shows better performance, up to $O(n)$ in case of applying insertion sort to a sorted array. Number of writes is $O(n^2)$ on average, but number of comparisons may vary depending on the insertion algorithm. It is $O(n^2)$ when shifting or swapping methods are used and $O(n \log n)$ for binary insertion sort.

From the point of view of practical application, an average complexity of the insertion sort is not so important. As it was mentioned above, insertion sort is applied to quite small data sets (from 8 to 12 elements).

12.3 Shell Sort

Most of the sorting algorithms seen so far such as insertion sort and selection sort have a run time complexity of $O(N^2)$. We also know that no sorting algorithm can have time complexity less than $O(N)$, since we need to scan the list at least once. For example, insertion sort best case complexity is $O(N)$. Can we get a better performance and get run time complexity between $O(N^2)$ and $O(N)$? The answer is Yes and there are many algorithms with a complexity in this range, with varying tradeoffs. In this note, we describe one approach: Shell Sort. Shell Sort has evolved as a trial and error algorithm. Analytical results are not available but empirically it has been found that the

Notes

Shell Sort improves the efficiency and decreases the run time complexity to $O(N^{1.25})$. Donald Shell discovered the Shell sort and thence it is known as Shell Sort.

Algorithm

The algorithms for shell sort can be defined in two steps:

Step 1: Divide the original list into smaller lists.

Step 2: Sort individual sub lists using any known sorting algorithm (like bubble sort, insertion sort, selection sort, etc).

```
void shellsort (int[] a, int n)
{
    int i, j, k, h, v;
    int[] cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
                 1968, 861, 336, 112, 48, 21, 7, 3, 1}
    for (k=0; k<16; k++)
    {
        h=cols[k];
        for (i=h; i<n; i++)
        {
            v=a[i];
            j=i;
            while (j>=h && a[j-h]>v)
            {
                a[j]=a[j-h];
                j=j-h;
            }
            a[j]=v;
        }
    }
}
```

Many questions arise

1. How should I divide the list?
2. Which sorting algorithm to use?
3. How many times I will have to execute steps 1 and 2?
4. And the most puzzling question if I am anyway using bubble, insertion or selection sort then how I can achieve improvement in efficiency? I shall discuss each of these questions one by one.

For dividing the original list into smaller lists, we choose a value K, which is known as increment. Based on the value of K, we split the list into K sub lists. For example, if our original list is $x[0], x[1], x[2], x[3], x[4] \dots x[99]$ and we choose 5 as the value for increment, K then we get the following sub lists.

$$\text{first_list} = x[0], x[5], x[10], x[15] \dots x[95]$$

second_list = x[1], x[6], x[11], x[16].....x[96]
 third_list = x[2], x[7], x[12], x[17].....x[97]
 forth_list = x[3], x[8], x[13], x[18].....x[98]
 fifth_list = x[4], x[9], x[14], x[19]x[99]

Notes

So the i^{th} sub list will contain every K^{th} element of the original list starting from index $i-1$.

According to the algorithm mentioned above, for each iteration, the list is divided and then sorted. If we use the same value of K , we will get the same sub lists and every time we will sort the same sub lists, which will not result in the ordered final list. Note that sorting the five sub lists independently do not ensure that the full list is sorted! So we need to change the value of K (increase or decrease?) for every iteration. To know whether the array is sorted, we need to scan the full list. We also know that number of sub lists we get are equal to the value of K . So if we decide to reduce the value of K after every iteration we will reduce the number of sub lists also in every iteration. Eventually, when K will be set to 1, we will have only one sub list. Hence we know the termination condition for our algorithm is $K = 1$. Since for every iteration we are decreasing the value of the increment (K) the algorithm is also known as “diminishing increment sort”.

Any sorting algorithm or a combination of algorithms can be used for sorting the sub lists, e.g. some shell sort implementation use bubble sort for the last iteration and insertion sort for other iterations. We use insertion sort in this tutorial. How does shell sort give better performance if the sorting is ultimately done by algorithms like insertion sort and bubble sort? The rationale is as follows.

If the list is either small or almost sorted then insertion sort is efficient since less number of elements will be shifting. In Shell Sort as we have already seen, the original list is divided into smaller lists based on the value of increment and sorted.

Initially value of K is fairly large giving a large number of small sub lists. We know that simple sorting algorithms like insertion sort are effective for small lists, since the data movements are over short distances. As K reduces in value, the length of the sub lists increases. But since the earlier sub lists have been sorted, we expect the full list to look more and more sorted. Again note that algorithms such as insertion sort are fairly efficient when they work with nearly sorted lists. Thus the inefficiency arising out of working with larger lists is partly compensated by lists being increasingly sorted. This is the intuitive explanation for the performance of shell sort.

How to choose the value of increment K ?

Till date there is no convincing answer to what should be the optimal sequence for the increment, K . But it has been empirically found that larger number of increments gives more efficient results. Also it is advisable not to use empirical sequences like 1,2,4,8... or 1,3,6,9... if we do so, for different iteration, we may get almost the same elements in the sub lists to compare and sort which will not result in better performance. For example, consider a list $X = 12\ 6\ 2\ 5\ 8\ 10\ 1\ 15\ 31\ 23$ and increment sequences 6 and 3.

When the increment is 6, you will get the following sub lists

(12,1)

(6,15)

(2,31)

(5,23)

(8)

(10)

Notes

After first iteration

X=1 6 2 5 8 10 12 15 31 23

When the increment reduces to 3, you will get the following lists

(1, 5, 12, 23)

(6, 8, 15)

(2, 10, 31)

Now if we analyse the first sub lists where increment was 6 and then second sub lists where increment was 3, we can see that in the second set of lists, we are comparing and sorting 1, 12 again which we had already sorted before in the previous sub list. So, we should choose the value of K in such a way that in every iterations we get almost different elements in the sub lists to compare and sort.

There are proven and recommended ways for generating the increment sequences.

Some of them have been discussed below.

Donald Shell has suggested

$$H_t = N/2$$

$$H_k = h_{k+1} / 2$$

Which give the increment series as: N/2, N/4,1

Hibbard suggested the sequence of increment as 1, 3, 7, 2^k-1 .

Another way for calculating increment K suggested by Knuth and we have used this method for generating the increments in this tutorial.

$$h_i = 1$$


$$h_i = h_i * 3 + 1 \text{ and stops at } h_i, \text{ When } h_i + 2 \geq N.$$

Which gives the increment series as 1, 4, 13..... h_i .



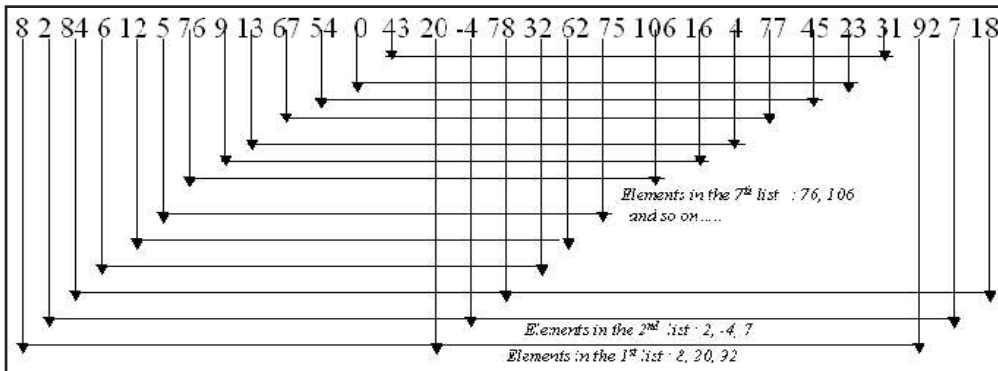
Task Discuss insertion sorting techniques.

Other than these, there are many series which have been empirically found and perform well.

 *Example:* We will take an example to illustrate Shell sort. Let the list be 8 2 84 6 12 5 76 9 13 67 54 0 43 20 -4 78 32 62 75 106 16 4 77 45 23 31 92 7 18 and the increment values using Knuth formula we get are 13, 4, 1.

When increment is 13, we get the following 13 sub lists and the elements are divided among the sub lists in the following ways i.e. every K^{th} element of the list starting with the index i will be in the sub list i.

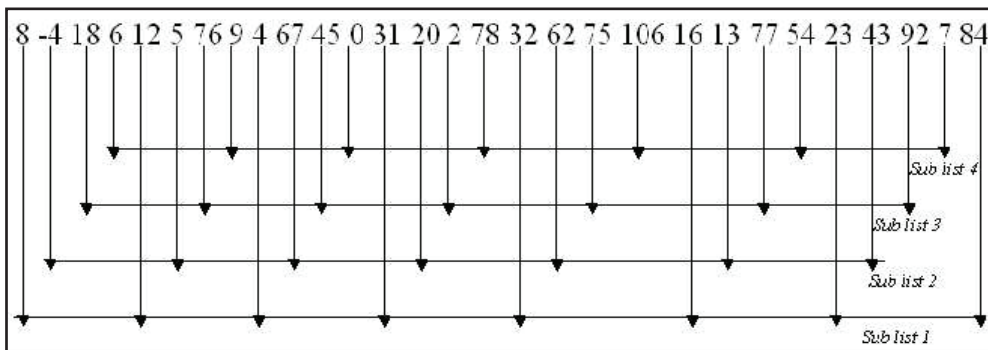
Notes



After sorting the sub lists, the resulting list we get is as follows.

8 -4 18 6 12 5 7 6 9 4 6 7 4 5 0 3 1 2 0 2 7 8 3 2 6 2 7 5 1 0 6 1 6 1 3 7 7 5 4 2 3 4 3 9 2 7 8 4

You now, reduce the increment value to 4 and get the following 4 sub lists.



After sorting these sub lists, the resulting list is as follows:

4 -4 2 0 8 5 18 6 12 13 4 5 7 16 20 7 5 9 2 3 4 3 7 6 5 4 3 1 6 2 7 7 7 8 3 2 6 7 9 2 1 0 6 8 4

You further reduce increment value to 1, which is our last increment value. Now there is only one list, which after sorting, we get:

-4 0 2 4 5 6 7 8 9 12 13 16 18 20 23 31 32 43 45 54 62 67 75 76 77 78 84 92 106

Now try to understand and analyse what is happening? How the elements are moved? We can see, after the first iteration when the increment was 13, the element -4 of the list, which was at 15th position in the original list has reached to the 2nd position. The element 18 was at 29th position and has reached the 3rd position. Using insertion sort on the whole list, we know that if -4 has to reach from the 15th position to the 2nd position, the required number of movements of the elements in the list will be very high. Shell Sort performs better than insertion sort by reducing the number of movements of elements in the lists. This is achieved by large strides the element take in the beginning cycle.

Upper Bound

Theorem: With the h-sequence 1, 3, 7, 15, 31, 63, 127, ..., $2^k - 1$, ... Shellsort needs $O(n \cdot n)$ steps for sorting a sequence of length n (Papernov/Stasevic [PS 65]).

Proof: Let h_t be the h closest to n. We analyze the behavior of Shellsort separately for the elements h_k with $k \leq t$ and with $k > t$.

Let $k \leq t$. Since $h_k = 2^k - 1$ we have the conditions mentioned above that h_{k+1} and h_{k+2} are relatively prime and in $O(h_k)$. Therefore, $O(n \cdot h_k)$ sorting steps suffice for h_k -sorting the data sequence. Since

Notes

the h_k form a geometric series, the sum of all h_k with $k = 1, \dots, t$ is in $O(ht) = O(n)$. Thus $O(n \cdot n)$ sorting steps are needed for this part where $k < t$.

Now let $k > t$. When the sequence is arranged as an array with h_k columns there are n/h_k elements in each column. Thus, $O((n/h_k)^2)$ sorting steps are needed to sort each column, since Insertion Sort has quadratic complexity. There are h_k columns, therefore the number of sorting steps for h_k -sorting the entire data sequence is in $O((n/h_k)^2 \cdot h_k) = O(n \cdot n/h_k)$. Again, the n/h_k form a geometric series whose sum is in $O(n/ht) = O(n)$. Therefore, again $O(n \cdot n)$ steps are needed for this part where $k > t$.

It can be shown that for this h -sequence the upper bound is tight. But there is another h -sequence that leads to a more efficient behavior of Shellsort.

Theorem: With the h -sequence 1, 2, 3, 4, 6, 8, 9, 12, 16, ..., $2p3q$, ... Shellsort needs $O(n \cdot \log(n)^2)$ steps for sorting a sequence of length n (Pratt [Pra 79]).

Proof: If $g = 2$ and $h = 3$, then $\gamma(g, h) = (g-1) \cdot (h-1) - 1 = 1$, i.e. in a 2,3-sorted sequence to the right of each element only the next element can be smaller. Therefore, $O(n)$ sorting steps suffice to sort the sequence with Insertion Sort. Considering elements with odd and with even index separately, it becomes clear that again $O(n)$ sorting steps suffice to make a 4,6-sorted sequence 2-sorted. Similarly, $O(n)$ sorting steps suffice to make a 6,9-sorted sequence 3-sorted and so on.

The above h -sequence has the property that for each h_k also $2h_k$ and $3h_k$ occurs, so $O(n)$ sorting steps suffice for each h_k . Altogether there are $\log(n)^2$ elements in the h -sequence; thus the complexity of Shellsort with this h -sequence is in $O(n \cdot \log(n)^2)$.

The h -sequence of Pratt performs best asymptotically, but it consists of $\log(n)^2$ elements. Particularly, if the data sequence is presorted, a h -sequence with less elements is better, since the data sequence has to be scanned (by the for- i -loop in the program) for each h_k , even if only a few sorting steps are performed.

By combining the arguments of these two theorems h -sequences with $O(\log(n))$ elements can be derived that lead to a very good performance in practice, as for instance the h -sequence of the program. But unfortunately, there seems to be no h -sequence that gives Shellsort a worst case performance of $O(n \cdot \log(n))$. It is an open question whether possibly the average complexity is in $O(n \cdot \log(n))$.

12.4 Heap Sort

Heapsort is a sorting technique which sorts a contiguous list of length n with $O(n \log^2(n))$ comparisons and movement of entries, even in the worst case. Hence it achieves worst-case bounds better than those of quicksort, and for contiguous list it is better than mergesort, since it needs only a small and constant amount of space apart from the list being sorted.

Heapsort proceeds in two phases. First, all the entries in the list are arranged to satisfy heap property, and then top of the heap is removed and another entry is promoted to take its place repeatedly. Therefore we need a procedure which builds an initial heap to arrange all the entries in the list to satisfy heap property. The procedure which builds an initial heap uses a procedure which adjust the i th entry in the list whose entries at $2i$ and $2i + 1$ positions already satisfy heap property in such a manner entry at i th position in the list will also satisfy heap property.

The following C code implements the algorithm.

```
void adjust(list x, int i, int n)
{
    int j, k;
    Boolean flag;
```



```
k = list[i];
flag = true;
j = 2 * i;
while(j <= n && flag)
{
    if((j < n) && (list[j] < list[j+ 1]))
        j = j + 1 ;
    if(k >= list[j])
        flag = false;
    else
    {
        list[j div 2] = list[j];
        j = j * 2;
    }
}
list[j div 2] = k;
}
void build_initial_heap(list x, int n)
{
    int i;
    for(i = n div 2; i >= 1; i--)
        adjust(x, i, n);
}
void heapsort(list x, int n)
{
    int i;
    build_initial_heap(x, n);
    for(i = n-1; i >= 1; i--)
    {
        exchange(list[1], list[i+1]);
        adjust(x, 1, i);
    }
}
void exchange(int &a, int &b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Notes

```

void adjust(int x[], int i, int n)
{
    int j, k;
    int flag;
    k = x[i];
    flag = 1;
    j = 2 * i;
    while(j <= n && flag == 1)
    {
        if((j < n) && (x[j] < x[j+ 1]))
            j++;
        if(k >= x[j])
            flag = 0;
        else
        {
            x[j / 2] = x[j];
            j = j * 2;
        }
    }
    x[j / 2] = k;
}

void build_initial_heap(int x[], int n)
{
    int i;
    for(i = n / 2; i >= 1; i--)
        adjust(x, i, n);
}

void heapsort(int x[], int n)
{
    int i;
    build_initial_heap(x, n);
    for(i = n-1; i >= 1; i--)
    {
        exchange(x[1], x[i+ 1]);
        adjust(x,1,i);
    }
}

```

Analysis of Heapsort

In each pass of while loop in procedure adjust(x,i,n), the position i is doubled, hence the number of passes cannot exceed (log_e n div i). Therefore the computation time of adjust is O(log n div

i). The procedure build-initial-heap calls the adjust procedure for values of i ranging from n/2 down to 1. Hence the total number of iterations will be:

$$\log(n) + \log(n/2) + \dots + \log(n/n/2) = \sum_{i=1}^{n/2} \log(n/i) = n/2 \log(n) - \log(!n/2)$$

This comes out to be some constant times n. Hence the computation time of build_initial_heap is O(n). The heapsort procedure calls adjust (x, 1, i) (n-1) times, hence the total number of iterations made in the heap sort will be:

$$\sum_{i=1}^{n-1} \log(i/1)$$

$$\sum_{i=1}^{n-1} \log(i) = \log(1) + \log(2) + \dots + \log(n-1)$$

which comes out to be approx. n log(n). Hence the computing time of heapsort is O(n log(n)) + O(n).

The only additional space needed by heap sort is space for one record to carry out exchange.

Consider a list given below when heapsort is applied to it we get following:

42	97
32	58
48	75
20	53
58	42
53	53
75	48
53	32
97	20
list to be sorted	initial heap

Output after each pass of heapsort

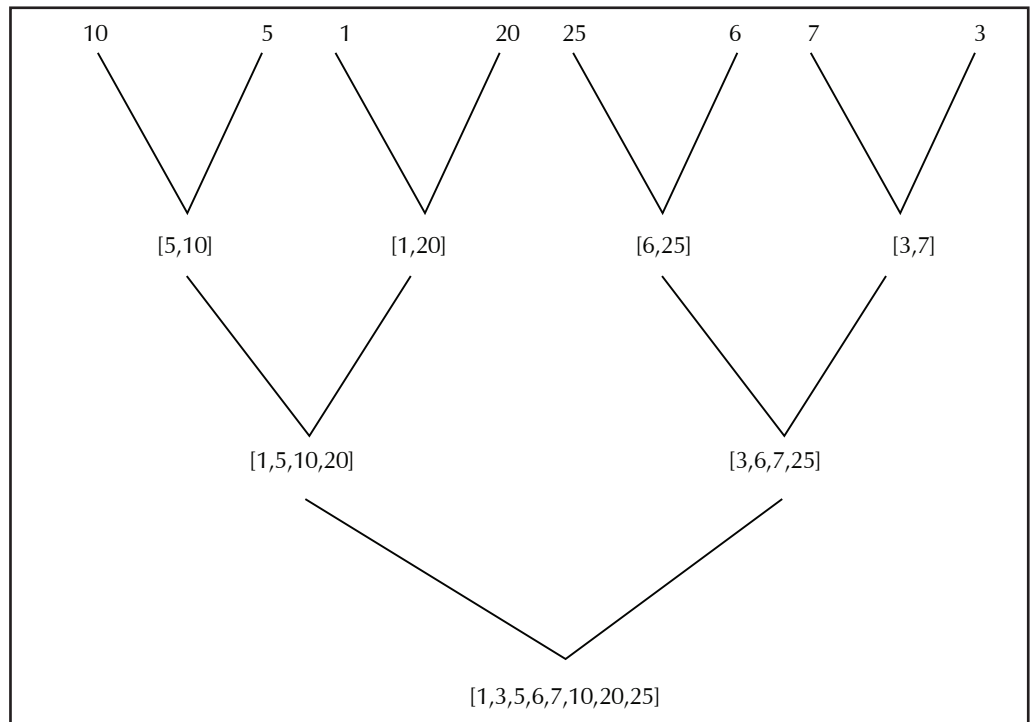
75	58	53	53	48	42	32	20
58	53	48	48	42	32	20	32
53	53	53	20	20	20	42	42
53	32	32	32	32	48	48	48
42	42	42	42	53	53	53	53
20	20	20	53	53	53	53	53
48	48	58	58	58	58	58	58
32	75	75	75	75	75	75	75
97	97	97	97	97	97	97	97

12.5 Merge Sort

This is another sorting technique having the same average and worst case time complexity, requiring an additional list of size n. The technique that we use is the merging of the two sorted lists of size m and n respectively to form a single sorted list of size (m+n). Given a list of size n to be sorted, instead of viewing it to be one single list of size n, we start by viewing it to be n lists

Notes

each of size 1, and merge the first list with the second list to form a single sorted list of size 2. Similarly we merge the third and the fourth lists to form a second single sorted list of size 2, and so on this completes the one pass. We then consider the first sorted list of size 2 and second sorted list of size 2, and merge them to form a single sorted list of size 4. Similarly we merge the third and the fourth sorted lists each of size 2 to form the second single sorted list of size 4, and so on; this completes the second pass. In the third pass we merge these adjacent sorted lists each of size 4 to form sorted lists of size 8. We continue this process till finally we end up with a single sorted list of size n as shown in figure.



To carry out the above task, we require a procedure to merge the two sorted lists of size m and n respectively to form a single sorted list of size (m+n), we also require a procedure to carry out one pass of the list merging the adjacent sorted lists of the specified size. This is because we have to carry out the repeated passes of the given list. In the first pass, we merge the adjacent lists of size 1. In the second pass, we merge the adjacent lists of size 2, and so on. Therefore, we will call this procedure by varying the size of the lists to be merged.

Here is a C implementation of the same.

```
void merge(int s[], int y[], int l, int m, int n)
{
    int i,j,k;
    i =1;
    j = m+1;
    k= 1;
    while ((i <= m) && (j <= n))
    {
        if(x[i] <= x[j])
        {
```

```
        y[k] =x[i];
        i++;
        k++;
    }
    else
    {
        y[k] = x[j];
        j++;
        k++;
    }
}
while (i <= m)
{
    y[k] =x[i];
    i++;
    k++;
}
while (j <= n)
{
    y[k] = x[j];
    j++;
    k++;
}
}
void mpass(int x[],int y[],int l, int n)
{
    int i,j;
    i = l;
    while(i <= n - 2 * l + 1)
    {
        merge(x, y, i, i+l-1, i+ 2 * l-1);
        i = i + 2 * l;
    }
    if((i+l-1) < n)
        merge(x, y, i, i+l-1, n);
    else
        while (i <= n )
        {
            y[i] =x[i];
            i++;
        }
}
```

Notes

```
    }  
}  
void msort(int x[], int n)  
{  
    int l;  
    int y[];  
    l = 1;  
    while(l < n)  
    {  
        mpass(x, y, l, n);  
        l = l * 2;  
        mpass(y, x, l, n);  
        l = l * 2;  
    }  
}
```

The merging of two sub-lists, the first running from the index 1 to m , and the second running from the index $(m+1)$ to n requires no more than $(n-1+1)$ iterations. Hence if $l = 1$, then no more than n iterations, where n is the size of the list to be sorted. Therefore if n is the size of the list to be sorted, every pass that a merge routine performs requires a time proportional to $O(n)$, and since the number of passes required to be performed are $\log_2 n$. The time complexity of the algorithm is $O(n \log_2(n))$, both average and worst case. The merge sort requires an additional list of size n .

Multi-way Merge Sort

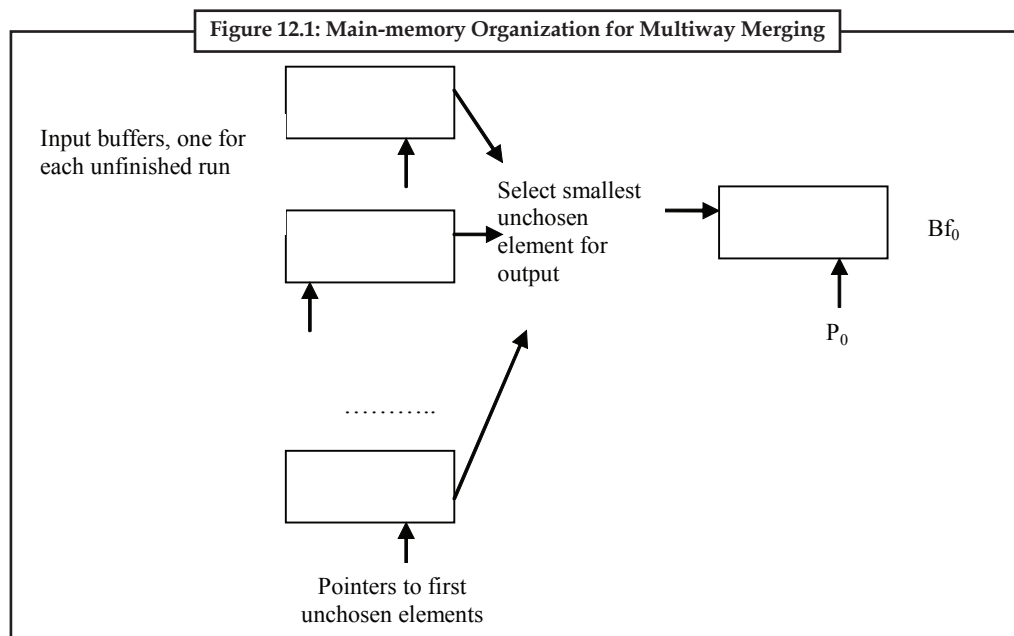
The two-phase, multiway merge-sort algorithm is similar to the external-memory merge-sort algorithm presented in the previous section. Phase 1 is the same, but, in phase 2, the main loop is performed only once merging all $[N/M]$ runs into one run in one go. To achieve this, multiway merging is performed instead of using the TWOWAY-MERGE algorithm.

The idea of multiway merging is the same as for the two-way merging, but instead of having 2 input buffers (Bf_1 and Bf_2) of B elements, we have $[N/M]$ input buffers, each B elements long. Each buffer corresponds to one unfinished (or active) run. Initially, all runs are active. Each buffer has a pointer to the first unchosen element in that buffer (analogous to p_1 and p_2 in TWOWAY-MERGE).

The multiway merging is performed by repeating these steps:

1. Find the smallest element among the unchosen elements of all the input buffers. Linear search is sufficient, but if the CPU cost is also important, minimum priority queue can be used to store pointers to all the unchosen elements in input buffers. In such a case, finding the smallest element is logarithmic in the number of the active runs.
2. Move the smallest element to the first available position of the output buffer.
3. If the output buffer is full, write it to the disk and reinitialize the buffer to hold the next output page.
4. If the buffer, from which the smallest element was just taken is now exhausted of elements, read the next page from the corresponding run. If no pages remain in that run, consider the run finished (no longer active).

When only one active run remains the algorithm finishes up as shown in lines 30 and 32 of TWOWAY-MERGE-it just copies all the remaining elements to the end of file X. Figure 12.1 visualizes multiway merging.



It is easy to see that phase 2 of the two-phase, multiway merge-sort algorithm performs only $\theta(n)$ I/O operations and this is also the running time of the whole algorithm. In spite of this, the algorithm has a limitation-it can not sort very large files.

If phase 1 of the algorithm produces more than $m - 1$ runs ($N/M > m - 1$), all runs can not be merged in one go in phase 2, because each run requires a one-page input buffer in main-memory and one page of main-memory is reserved for the output buffer. How large should the file be for this to happen?

Multiway Merge Sort of Very Large Files

Sometimes there may be a need to sort extremely large files or there is only a small amount of available main memory. As described in the previous section, two-phase, multiway merge sort may not work in such situations.

A natural way to extend the two-phase, multiway merge sort for files of any size is to do not one but many iterations in phase 2 of the algorithm. That is, we employ the external memory merge-sort algorithm from Section 3, but instead of using TWOWAY-MERGE, we use the multiway merging (as described in the previous section) to merge $m - 1$ runs from file Y into one run in file X. Then, in each iteration of the main loop of phase 2, we reduce the number of runs by a factor of $m - 1$.

What is the running time of this algorithm, which we call simply multiway merge sort. Phase 1 and each iteration of the main loop of phase 2 takes $\theta(n)$ I/O operations. After phase 1, we start up with $[N/M] = \lceil n/m \rceil$ runs, each iteration of the main loop of phase 2 reduces the number of runs by a factor of $m - 1$, and we stop when we have just one run. Thus, there are $\log_{m-1}(n/m)$ iterations of the main loop of phase 2. Therefore, the total running time of the algorithm is $\theta(n \log_{m-1}(n/m)) = \theta(n \log_m n - n \log_m m) = \theta(n \log_m n - n) = \theta(n \log_m n)$.

Remember that the cost of the external-memory merge-sort algorithm from Section 3 is $\theta(n \log_2(n/m))$. Thus, multiway merge sort is faster by a factor of $(1 - 1/\log_m n) \log_2 m$. Actually,

Notes

$\theta(n \log_m n)$ is a lower bound for the problem of external-memory sorting. That is, multiway merge sort is an asymptotically optimal algorithm.

12.6 Merging of two Sorted Lists

Assume that two lists to be merged are sorted in descending order. Compare the first element of the first list with the first element of the second list. If the element of the first list is greater, then place it in the resultant list. Advance the index of the first list and the index of the resultant list so that they will point to the next term. If the element of the first list is smaller, place the element of the second list in the resultant list. Advance the index of the second list and the index of the resultant list so that they will point to the next term.

Repeat this process until all the elements of either the first list or the second list are compared. If some elements remain to be compared in the first list or in the second list, place those elements in the resultant list and advance the corresponding index of that list and the index of the resultant list.

Suppose the first list is 10 20 25 50 63, and the second list is 12 16 62 68 80. The sorted lists are 63 50 25 20 10 and 80 68 62 16 12.

The first element of the first list is 63, which is smaller than 80, so the first element of the resultant list is 80. Now, 63 is compared with 68; again it is smaller, so the second element in the resultant list is 68. Next, 63 is compared with 50. In this case it is greater, so the third element of the resultant list is 63.

Repeat this process for all the elements of the first list and the second list. The resultant list is 80 68 63 62 50 25 20 16 12 10.



Lab Exercise Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void read(int *,int);
    void dis(int *,int);
    void sort(int *,int);
    void merge(int *,int *,int *,int);
    int a[5],b[5],c[10];
    clrscr();
    printf("Enter the elements of first list \n");
    read(a,5);    /*read the list*/
    printf("The elements of first list are \n");
    dis(a,5);    /*Display the first list*/
    printf("Enter the elements of second list \n");
    read(b,5);    /*read the list*/
    printf("The elements of second list are \n");
    dis(b,5);    /*Display the second list*/
```


Notes

```
sort(a,5);
printf("The sorted list a is:\n");
dis(a,5);
sort(b,5);
printf("The sorted list b is:\n");
dis(b,5);
merge(a,b,c,5);
printf("The elements of merged list are \n");
dis(c,10); /*Display the merged list*/
getch();
}
void read(int c[],int i)
{
    int j;
    for(j=0;j<i;j++)
        scanf("%d",&c[j]);
    fflush(stdin);
}
void dis(int d[],int i)
{
    int j;
    for(j=0;j<i;j++)
        printf("%d ",d[j]);
    printf("\n");
}
void sort(int arr[] ,int k)
{
    int temp;
    int i,j;
    for(i=0;i<k;i++)
    {
        for(j=0;j<k-i-1;j++)
        {
            if(arr[j]<arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}
```

Notes

```
    }  
  }  
  void merge(int a[],int b[],int c[],int k)  
  {  
    int ptra=0,ptrb=0,ptrc=0;  
    while(ptra<k && ptrb<k)  
    {  
      if(a[ptra] < b[ptrb])  
      {  
        c[ptrc]=a[ptra];  
        ptra++;  
      }  
      else  
      {  
        c[ptrc]=b[ptrb];  
        ptrb++;  
      }  
      ptrc++;  
    }  
    while(ptra<k)  
    {  
      c[ptrc]=a[ptra];  
      ptra++;ptrc++;  
    }  
    while(ptrb<k)  
    {  
      c[ptrc]=b[ptrb];  
      ptrb++; ptrc++;  
    }  
  }  
}
```

Input: Enter the elements of the first list

10 20 25 50 63

Output: The elements of first list are

20 25 50 63

Input: Enter the elements of the second list

16 62 68 80

Output: The elements of second list are

12 16 62 68 80

The sorted list a is

63 50 25 20 10

The sorted list b is

80 68 62 16 12

The elements of the merged list are

80 68 63 62 50 25 20 16 12 10

Notes



Task

Write a C program for merge sort.

12.7 Quick Sort

In this method an array $a[1], \dots, a[n]$ is sorted by picking some value in the array as a key element, we then swap the first element of the list with the key element so that the key will come in the first position, we then find out the proper place of key in the list. The proper place is that position in the list where if key is placed then all elements to the left of it are smaller than the key, and all the elements to the right of it are greater than the key. To obtain the proper position of the key we traverse the list in both the directions using the indices i and j respectively. We initialize i to that index which is one more than the index of the key element, i.e. if the list to be sorted is having the indices running from m to n , then the key element is the at index m , hence we initialize i to $(m+1)$. The index i is incremented till we get an element at the i th position greater than the key value. Similarly we initialize j to n and go on decrementing j till we get an element having the value less than the key value. We then check whether i and j have crossed each other. If not then we interchange the elements at the i th and j th position, and continue the process of incrementing i and decrementing j till i and j crosses each other. When i and j crosses each other we interchange the elements at the key position (i.e. at m th position) and the elements at the j th position. This brings the key element at the j th position, and we find that the elements to left are less than and the elements to the right of it are greater than it. Therefore we can split the given list into two sub-lists. The first one made of elements from m th position to the $(j-1)$ th position, and the second one made of elements from the $(j+1)$ th position to n th position, and repeat the same procedure with each of the sub-lists separately.

Here is a C implementation.

```
void qsort(int x[], int m, int n)
{
    int key, i, j, k;
    if (m < n)
    {
        k = getkeyposition(x, m, n);
        interchange(&x[m], &x[k]);
        key = x[m];
        i = m + 1;
        j = n;
        while (i < j)
        {
            while ((i <= n) && (x[i] <= key))
                i++;
            while (j >= m) && (x[j] > key)
                j--;
            interchange(&x[i], &x[j]);
        }
    }
}
```

Notes

```

        j++;
        if( i < j)
            interchange(&x[i], &x[j]);
    }
    interchange(&x[m], &x[j]);
    qsort(x, m, j-1);
    qsort(x, j+1, n);
}
}
void interchange(int *i, int *j)
{
    int temp;
    temp = *i;
    *i = *j;
    *j = temp;
}

```

Choice of the Key

We can choose any entry in the list as the key. The choice of first entry is often a poor choice for key, since if the list is already sorted, then there will be no element less than the first element selected as key, and so one of the sub-lists will be empty. Hence we choose a key near the center of the list, in the hope that our choice will partition the list in such a manner that about half comes on each side of key.

Therefore the function getkeyposition is:

```

int getkeyposition(int i, int j)
{
    return(i+j mod 2);
}

```

The choice of the key near the center is also arbitrary, and hence it is not necessary that it will always divide the list nicely in to half, it may also happen that one sub-list is much larger than other. Hence some other method of selecting a key should be used. A good way to choose a key is to use a random number generator to choose the position of next key in each activation of quicksort.

Therefore the function getkeyposition is:

```

int getkeyposition(int i, int j)
{
    return a random number in the range of i to j.
}

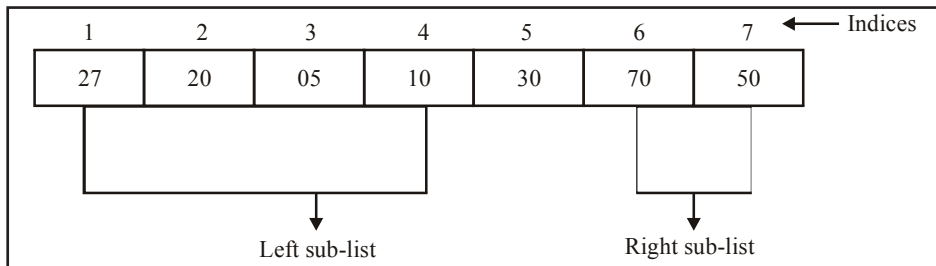
```

Consider the following list:

1	2	3	4	5	6	7	← Indices
30	20	50	10	27	70	05	

Notes

When qsort is activated first time key = 30, and i =2, and j =7, i is incremented till it becomes 3, because at position 3, the value is greater than key, j is not decremented, because at position 7, the value that we have is less than the key. Since $i < j$, we interchange the 3rd element and 7th element. Then i is incremented till it becomes 6, and j is decremented till it becomes 5. Since $i > j$, we interchange the key element that is the element at position 1, with the element at position 5, and call qsort recursively with the left sub-list made of elements from position 1 to 4, and right sub-list made of elements from position 6 to 7 as shown below:



By continuing in this fashion, we finally get the sorted list.

The average case time complexity of the quick sort algorithm can be decided as follows:

We assume that every time the list gets splitted into two approximately equal sized sub-lists. If the size of a given list is n , then it gets splitted into two sub lists of size approximately $n/2$. Each of these sub-lists further gets splitted into two sub-lists of size $n/4$, and this is continued till the size becomes 1. When the quick sort works with a list of size n it places the key element (which we take the first element of the list under consideration) at its proper position in the list. This requires no more than n iterations. After placing the key element at its proper position in the list of size n , quick sort activates itself two times to work with left and right sub-lists, each assumed to be of size $n/2$. Therefore if $T(n)$ is a time required to sort a list of size n . Since the time required to sort the list of size n is equal to the sum of the time required to place the key element at its proper position in the list of size n and the time required to sort the left and right sub-lists each assumed to be of size $n/2$, $T(n)$ comes out to be:

$$T(n) = c*n + 2*T(n/2)$$

Where c is a constant and $T(n/2)$ is the time required to sort the list of size $n/2$.

Similarly the time required to sort the list of size $n/2$ is equal to the sum of the time required to place the key element at its proper positions in the list of size $n/2$ and the time required to sort the left and right sub-lists each assumed to be of size $n/4$. $T(n/2)$ comes out to be:

$$T(n/2) = c*n/2 + 2*T(n/4)$$

Where $T(n/4)$ is the time required to sort the list of size $n/4$.

$$T(n/4) = c*n/4 + 2*T(n/8), \text{ and so on and finally we get } T(1) = 1.$$

$$T(n) = c*n + 2(c*n(n/2) + 2T(n/4))$$

$$T(n) = c*n + c*n + 4T(n/4) = 2*c*n + 4T(n/4) = 2*c*n + 4(c*(n/4) + 2T(n/8))$$

$$T(n) = 2*c*n + c*n + 8T(n/8) = 3*c*n + 8T(n/8)$$

$$T(n) = (\log n)*c*n + nT(n/n) = (\log n)*c*n + nT(1) = n + n*(\log n)*c$$

$$T(n) \text{ nlog}(n)$$

Therefore we conclude that the average time complexity of the quick sort algorithm is $O(n \log D)$. But the worst case time complexity is of the $O(n^2)$.

The reason for this is in the worst case one of the two sub-lists will always be empty, and the other will be of the size $(n-1)$. Where n is the size of the original list. Therefore in the worst case

Notes

T(n) comes out to be:

$$\begin{aligned}
 T(n) &= c*n + T(n-1) \\
 &= c*n + c*(n-1) + T(n-2) \\
 &= 2*c*n - c + T(n-2) \\
 &= 2*c*n - c + c*(n-2) + T(n-3) \\
 &= 3*c*n - 3*c + T(n-3) \\
 &= n*c*n - n*c + T(1) \\
 &= n^2c - nc + 1
 \end{aligned}$$

Therefore T(n) $\sim n^2$, hence the order is $O(n^2)$.

Space Complexity

The average case space complexity is $\log_2 n$, because the space complexity depends on the maximum number of activations that can exist. We find that if we assume that every time the list gets splitted into approximately two lists of equal size then the maximum number of activations that will exist simultaneously will be $\log_2 n$.

In the worst case, there exist n activations because the depth of the recursion is n. Hence, the worst case space complexity is $O(n)$.

Algorithms of Quick Sort

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

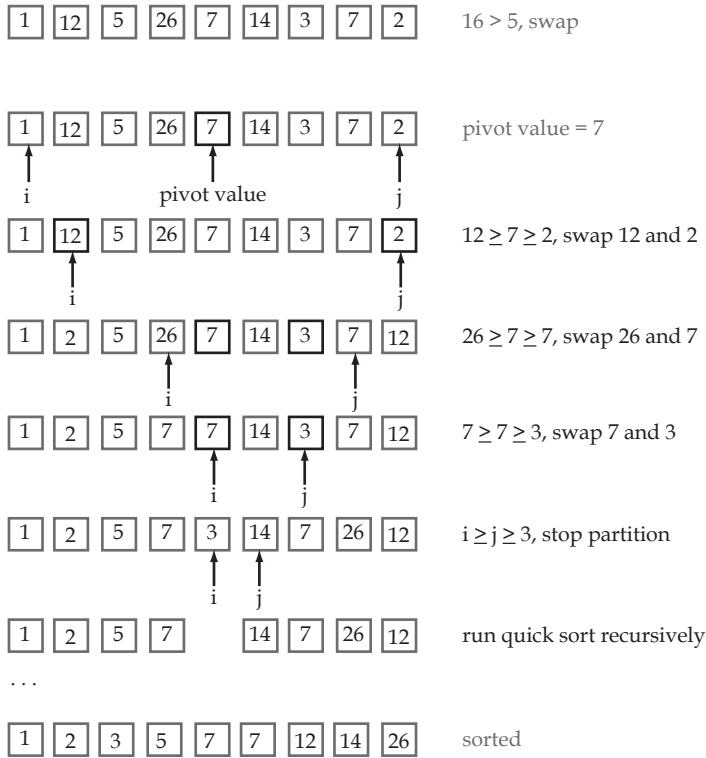
Partition Algorithm in Detail

There are two indices i and j and at the very beginning of the partition algorithm i points to the first element in the array and j points to the last one. Then algorithm moves i forward, until an element with value greater or equal to the pivot is found. Index j is moved backward, until an element with value lesser or equal to the pivot is found. If $i \leq j$ then they are swapped and i steps to the next position (i + 1), j steps to the previous one (j - 1). Algorithm stops, when i becomes greater than j.

After partition, all values before i-th element are less or equal than the pivot and all values after j-th element are greater or equal to the pivot.



Example: Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.



Notice, that we show here only the first recursion step, in order not to make example too long. But, in fact, {1, 2, 5, 7, 3} and {14, 7, 26, 12} are sorted then recursively.

Why does it work?

On the partition step algorithm divides the array into two parts and every element a from the left part is less or equal than every element b from the right part. Also a and b satisfy $a \leq \text{pivot} \leq b$ inequality. After completion of the recursion calls both of the parts become sorted and, taking into account arguments stated above, the whole array is sorted.



Example: Consider the following list to be sorted in ascending order. 'ADD YOUR MAN'. (Ignore blanks) N = 10

	0	1	2	3	4	5	6	7	8	9
A[] =	A	D	D	Y	O	U	R	M	A	N

Quicksort (A, 0, 9)

1. $9 > 0$
2. $V = [9] = 'N'$
 $L = 1-1 = 0$
 $R = I = 9$
4. $A[3] = 'Y' > V$; There fore, $L = 3$
5. $A[8] = 'A' > V$; There fore, $R = 8$

Notes

6. $L < R$
7. SWAP (A,3,8) to get

	0	1	2	3	4	5	6	7	8	9
A[] =	A	D	D	A	O	U	R	M	Y	N

8. $A[4] = 'O' > V$, There fore, $L = 4$
9. $A[7] = 'M' < V$, There fore, $R = 7$
10. $L < R$
11. SWAP (A,4,7) to get

	0	1	2	3	4	5	6	7	8	9
A[] =	A	D	D	A	M	U	R	O	Y	N

12. $A[5] = 'U' > V;.. L = 5$
13. $A[4] = 'M' < V;R = 4$
14. $L < R,.. break$
15. SWAP (A,5,9) to get.

	0	1	2	3	4	5	6	7	8	9
A[] =	A	D	D	A	M	N	R	O	Y	U

at this point 'N' is in its correct place.

$A[5]$, $A[0]$ to $A[4]$ constitutes sub list 1.

$A[6]$ to $A[9]$ constitutes sublist2. Now

16. Quick sort (A, 0, 4)
17. Quick sort (A, 5, 9)

The Quick sort algorithm uses the $O(N \log_2 N)$ comparisons on average. The performance can be improved by keeping in mind the following points.

1. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.
2. Use a better dividing element I in the implementations. We have always used $A[N]$ as the dividing element. A useful method for the selection of a dividing element is the Median-of three method.

Select any 3 elements from the list. Use the median of these as the dividing element.

12.8 Bucket Sort

Bucket sort runs in linear time on the average. It assumes that the input is generated by a random process that distributes elements uniformly over the interval $[0, 1]$.

The idea of Bucket sort is to divide the interval $[0, 1]$ into n equal-sized subintervals, or buckets, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over $(0, 1)$, we don't expect many numbers to fall into each bucket. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each.

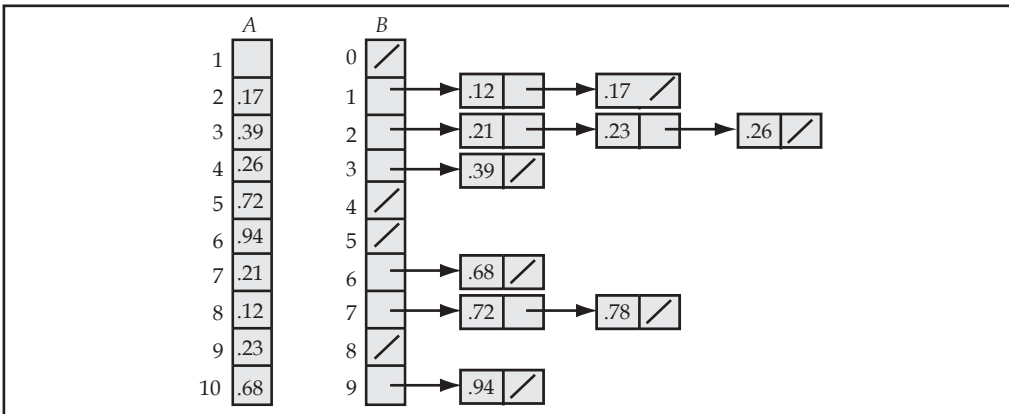
The code assumes that input is in n -element array A and each element in A satisfies $0 \leq A[i] \leq 1$. We also need an auxiliary array $B[0 \dots n-1]$ for linked-lists (buckets).

BUCKET_SORT (A)

1. $n \leftarrow \text{length}[A]$
2. For $i = 1$ to n do
3. Insert $A[i]$ into list $B[nA[i]]$
4. For $i = 0$ to $n-1$ do
5. Sort list B with Insertion sort
6. Concatenate the lists $B[0], B[1], \dots B[n-1]$ together in order.



Example: Given input array $A[1..10]$. The array $B[0..9]$ of sorted lists or buckets after line 5. Bucket i holds values in the interval $[i/10, (i+1)/10]$. The sorted output consists of a concatenation in order of the lists first $B[0]$ then $B[1]$ then $B[2]$... and the last one is $B[9]$.



Analysis

All lines except line 5 take $O(n)$ time in the worst case. We can see inspection that total time to examine all buckets in line 5 is $O(n-1)$ i.e., $O(n)$.

The only interesting part of the analysis is the time taken by Insertion sort in line 5. Let n_i be the random variable denoting the number of elements in the bucket $B[i]$. Since the expected time to sort by INSERTION_SORT is $O(n^2)$, the expected time to sort the elements in bucket $B[i]$ is

$$E[O(n_i^2)] = O(E[n_i^2])$$

Therefore, the total expected time to sort all elements in all buckets is

$$\sum_{i=0}^{n-1} O(E[n_i^2]) = O \sum_{i=0}^{n-1} (E[n_i^2]) \quad \dots(1)$$

In order to evaluate this summation, we must determine the distribution of each random variable n_i .

We have n elements and n buckets. The probability that a given element falls in a bucket $B[i]$ is $1/n$ i.e., Probability = $p = 1/n$.



Note

This problem is the same as that of "Balls-and-Bin" problem.

Notes

Therefore, the probability follows the binomial distribution, which has

Mean: $E[n_i] = np = 1$

Variance: $\text{Var}[n_i] = np(1 - p) = 1 - 1/n$

For any random variable, we have

$$\begin{aligned} E^2[n_i] &= \text{Var}[n_i] + E^2[n_i] \\ &= 1 - 1/n + 1^2 \\ &= 2 - 1/n \\ &= \Theta(1) \end{aligned}$$


Putting this value in equation A above, (do some tweaking) and we have a expected time for INSERTION_SORT, $O(n)$.

Now back to our original problem

In the above Bucket sort algorithm, we observe

$$\begin{aligned} T(n) &= [\text{Time to insert } n \text{ elements in array } A] + [\text{Time to go through auxiliary array } B[0 \dots n-1] * \\ &\quad (\text{Sort by INSERTION_SORT}) \\ &= O(n) + (n-1) \quad (n) \\ &= O(n) \end{aligned}$$

Therefore, the entire Bucket sort algorithm runs in linear expected time.



Task Discuss bubble sort with suitable example.

12.9 External Sorting

External sorting refers to the sorting of a file that is on disk (or tape). Internal sorting refers to the sorting of an array of data that is in RAM. The main concern with external sorting is to minimize disk access since reading a disk block takes about a million times longer than accessing an item in RAM (according to Shaffer - see the reference at the end of this document).

Perhaps the simplest form of external sorting is to use a fast internal sort with good locality of reference (which means that it tends to reference nearby items, not widely scattered items) and hope that your operating system's virtual memory can handle it. (Quicksort is one sort algorithm that is generally very fast and has good locality of reference.) If the file is too huge, however, even virtual memory might be unable to fit it. Also, the performance may not be too great due to the large amount of time it takes to access data on disk.

Methods

Most external sort routines are based on mergesort. They typically break a large data file into a number of shorter, sorted "runs". These can be produced by repeatedly reading a section of the data file into RAM, sorting it with ordinary quicksort, and writing the sorted data to disk. After the sorted runs have been generated, a merge algorithm is used to combine sorted files into longer sorted files. The simplest scheme is to use a 2-way merge: merge 2 sorted files into one sorted file, then merge 2 more, and so on until there is just one large sorted file. A better scheme is a multiway merge algorithm: it might merge perhaps 128 shorter runs together.

12.10 Summary

Notes

- Arranging objects in a specified order is called sorting.
- Bubble sort, insertion sort, selection sort, quick sort, heap sort, radix sort are some of very common search algorithms.
- The comparison starts with the first element (or at the last element) and continues sequentially till either we find a match or the end of the list is encountered. Linear search makes as many comparisons as there are elements in the array. Even if the array is sorted (either in ascending or descending order) the number of comparisons remains the same.

12.11 Keywords

Bubble sort: A sorting technique in which the largest element of the remaining list bubbles up to its proper ordering position in each pass through the list.

Heap sort: A sorting technique in which all the entries in the list are arranged to satisfy heap property, and then top of the heap is removed and another entry is promoted to take its place repeatedly.

Merge sort: A sorting technique in which the given list is broken down into smaller lists repeatedly until the list become easy to sort, then the sorted lists are merged to obtain the final sorted list.

Quick sort: A sorting technique in which an array is sorted by picking some value in the array as a key element, then swapping the first element of the list with the key element so that the key comes in the first position. The proper place of key in the list is found out repeatedly.

Sorting: A technique to arrange the elements of a list in some pre-specified order.

12.12 Self Assessment

Choose the appropriate answers:

1. Which one is not the method of internal sorting?
 - (a) Heap sort
 - (b) Merge sort
 - (c) Quick sort
 - (d) Bubble sort
2. Polyphase sort is a
 - (a) Internal sort
 - (b) External sort
 - (c) Both of the above
 - (d) None
3. is a sorting technique which sorts a contiguous list of length n with $O(n \log_2(n))$ comparisons and movement of entries, even in the worst case.
 - (a) Heap sort
 - (b) Merge sort
 - (c) Quick sort

Notes

- (d) Bubble sort
- 4. Heap sort proceeds in
 - (a) Five phase
 - (b) Three phases
 - (c) One phase
 - (d) Two phase

Fill in the blanks:

- 5. Arranging objects in a specified order is called
- 6. The average time complexity of the quick sort algorithm is
- 7. The computing time of heapsort is
- 8. The time complexity of the algorithm is both average and worst case.

State whether the following statements are true or false:

- 9. The order of linear search in worst case is $O(n/2)$.
- 10. Linear search is more efficient than Binary search.
- 11. For Binary search, the array has to be sorted in ascending order only.
- 12. A **file** is a collection of records and a record is in turn a collection of fields.

12.13 Review Questions

- 1. What is sorting? Explain insertion sorting in details.
- 2. Distinguish between quick and heap sort.
- 3. Explain 2-way merge sort.
- 4. Distinguish between linear and binary search.
- 5. Explain the application of searching.
- 6. Consider the list given below whose elements are arranged in an ascending order. Assume that a binary search technique is used. Find out the number of probes required to find each entry in the list.
 - 16
 - 22
 - 48
 - 53
 - 55
 - 71
 - 80
- 7. Sort the list given below by applying the heapsort method.
 - 81
 - 52
 - 53

96

22

53

83

04

Notes

8. Consider an unsorted array $A[n]$ of integer elements that may have many elements present more than once. It is required to store only the distinct elements of the array A in a separate array B . The information about the number of times each element is replicated is maintained in a third array C . For example, $C[0]$ would indicate the number of times the element $B[0]$ occurs in array A . Write a C program to generate the arrays B and C , given an array A .
9. Write a C program that finds the largest and the second largest elements in an unsorted array A . The program should make just a single scan of the array.
10. Distinguish between internal and external method of sorting.

Answers: Self Assessment

- | | | | |
|---------------------|------------------|--------------------------|-----------|
| 1. (b) | 2. (b) | 3. (a) | 4. (d) |
| 5. sorting | 6. $O(n \log D)$ | 7. $O(n \log(n)) + O(n)$ | |
| 8. $O(n \log_2(n))$ | 9. False | 10. False | 11. False |
| 12. True | | | |

12.14 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C*, Second Ed., Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*. Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 13: Graphs

CONTENTS

Objectives

Introduction

13.1 Defining Graph

13.2 Basic Graph Terminology

13.3 Representations of Graphs

13.3.1 Adjacent Matrix

13.3.2 Adjacency List Representation

13.4 Shortest Path Algorithms

13.5 Summary

13.6 Keywords

13.7 Self Assessment

13.8 Review Questions

13.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Define graph
- Realise basic graph terminology
- Explain representation of graphs
- Discuss shortest path algorithms

Introduction

In this unit, we introduce you to an important mathematical structure called Graph. Graphs have found applications in subjects as diverse as Sociology, Chemistry, Geography and Engineering Sciences. They are also widely used in solving games and puzzles. In computer science, graphs are used in many areas one of which is computer design. In day-to-day applications, graphs find their importance as representations of many kinds of physical structure.

We use graphs as models of practical situations involving routes: the vertices represent the cities and edges represent the roads or some other links, specially in transportation management, Assignment problems and many more optimization problems. Electric circuits are another obvious example where interconnections between objects play a central role. Circuits elements like transistors, resistors, and capacitors are intricately wired together. Such circuits can be represented and processed within a computer in order to answer simple questions like “Is everything connected together?” as well as complicated questions like “If this circuit is built, will it work?”

13.1 Defining Graph

Notes

A Graph G consists of a set V of vertex (nodes) and a set E of edges (arcs). We write $G=(V,E)$. V is a finite and non empty set of vertices. E is a set of pairs of vertices; these pairs are called edges. Therefore

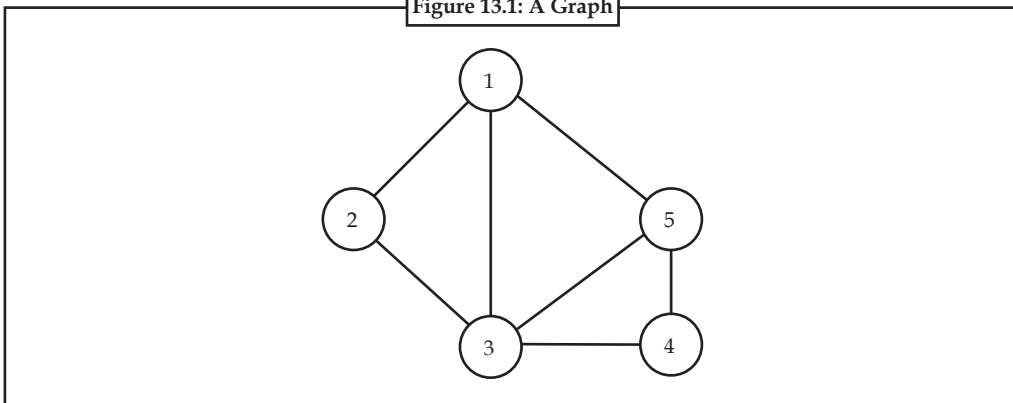
$V(G)$, read as V of G , is set of vertices,

and $E(G)$, read as E of G , is set of edges.

An edge $e = (v,w)$, is a pair of vertices v and w , and is said to be incident with v and w .

A graph may be pictorially represented as given in Figure 13.1.

Figure 13.1: A Graph



We have numbered the nodes as 1,2,3,4 and 5. Therefore

$V(G) = (1, 2, 3, 4, 5)$

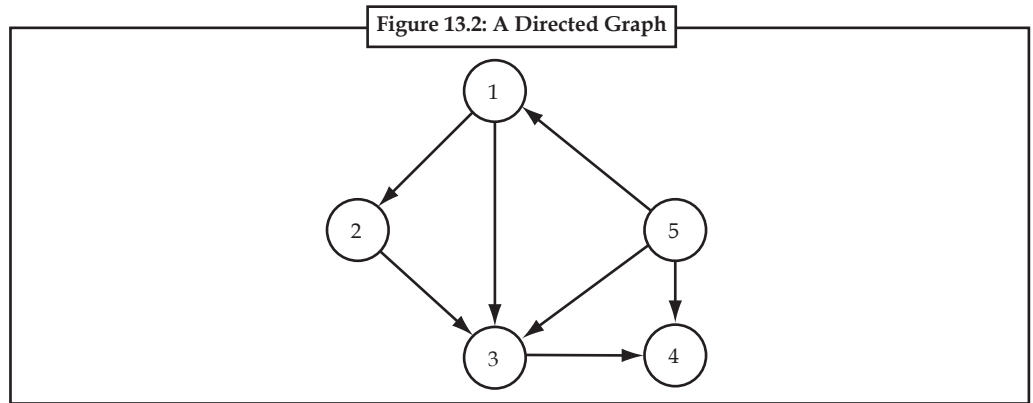
and $E(G) = \{(1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (1, 3), (3, 5)\}$

You may notice that the edge incident with node 1 and node 5 is written as $(1,5)$; we could also have written $(5,1)$ instead of $(1,5)$. The same applies to all the other edges. Therefore, we may say that ordering of vertices is not significant here. This is true for an undirected graph.

In an undirected graph, pair of vertices representing any edge is unordered. Thus (v,w) and (w,v) represent the same edge. In a directed graph each edge is an ordered pair of vertices, i.e. each edge is represented by a directed pair. If $e = (v,w)$, then v is tail or initial vertex and w is head or final vertex. Subsequently (v,w) and (w,v) represent two different edges.

A directed graph may be pictorially represented as given in Figure 13.2.

Notes



The direction is indicated by an arrow. The set of vertices for this graph remains the same as that of the graph in the earlier example, i.e.

$$V(G) = \{1,2,3,4,5\}$$

However the set of edges would be

$$E(G) = \{(1,2), (2,3), (3,4), (5,4), (5,1), (1,3), (5,3)\}$$

Do you notice the difference?



Note Arrow is always from tail vertex to head vertex. In our further discussion on graphs, we would refer to directed graph as digraph and undirected graph as graph.

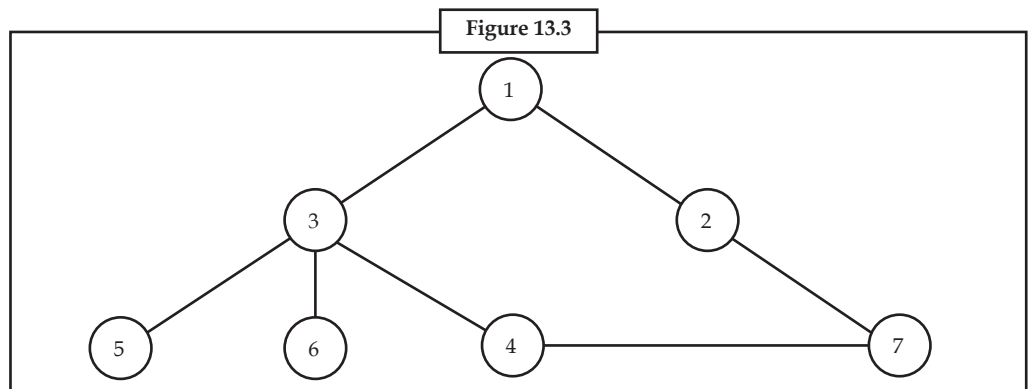
13.2 Basic Graph Terminology

A good deal of nomenclature is associated with graphs. Most of the terms have straight forward definitions, and it is convenient to put them in one place even though we would not be using some of them until later.

Adjacent Vertices

Vertex v_1 is said to be adjacent to a vertex v_2 if there is an edge

$$(v_1, v_2) \text{ or } (v_2, v_1)$$



Vertices adjacent to node 3 are 1,5,6 and 4 and that to node 2 are 1 and 7.

Find out the vertices adjacent to remaining nodes of the graph.

Path: A path from vertex v to vertex w is a sequence of vertices, each adjacent to the next.

Consider the above example again. 1,3,4 is a path

1,3,6 is a path

1,2,7 is a path,

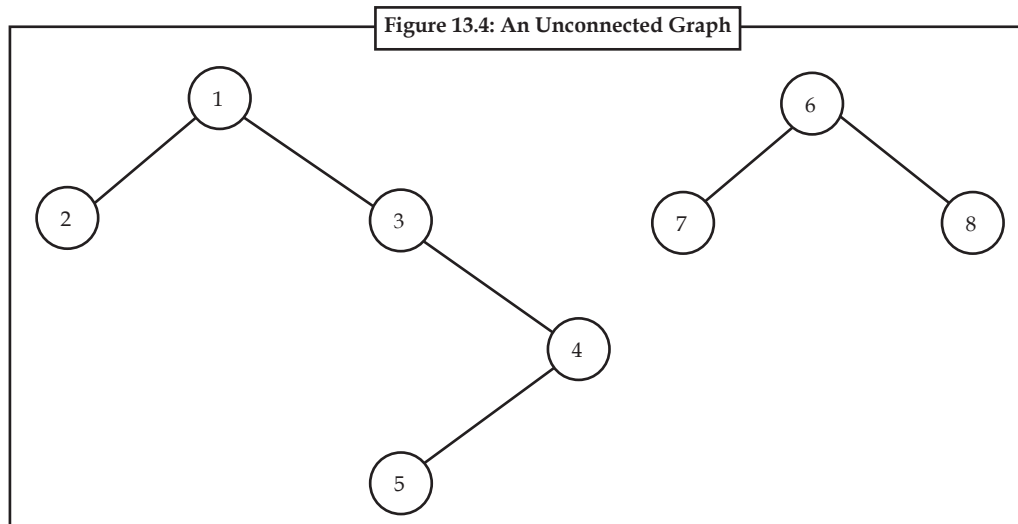
Is 1,4,7 a path?

How many paths are there from vertex 1 to vertex 7?

You may notice that there is a path existing in the above example which starts at vertex 1 and finishes at vertex 1, i.e. path 1,3,4,7,2,1. Such a path is called a cycle.

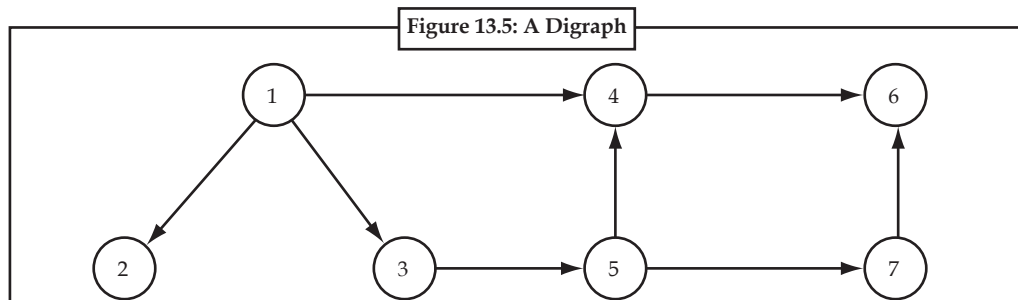
A cycle is a path in which first and last vertices are the same.

DO we have a path from any vertex to any other vertex in the above example? If you see it carefully, you may find the answer to the above question as YES. Such a graph is said to be connected graph. A graph is called connected if there exists a path from any vertex to any other vertex. There are graphs which are unconnected. Consider the graph in Figure 13.4.



It is an unconnected graph. You may say that these are two graphs and not one. Look at the figure in its totality and apply the definition of graph. Does it satisfy the definition of a graph? It does. Therefore, it is one graph having two unconnected components. Since there are unconnected components, it is an unconnected graph.

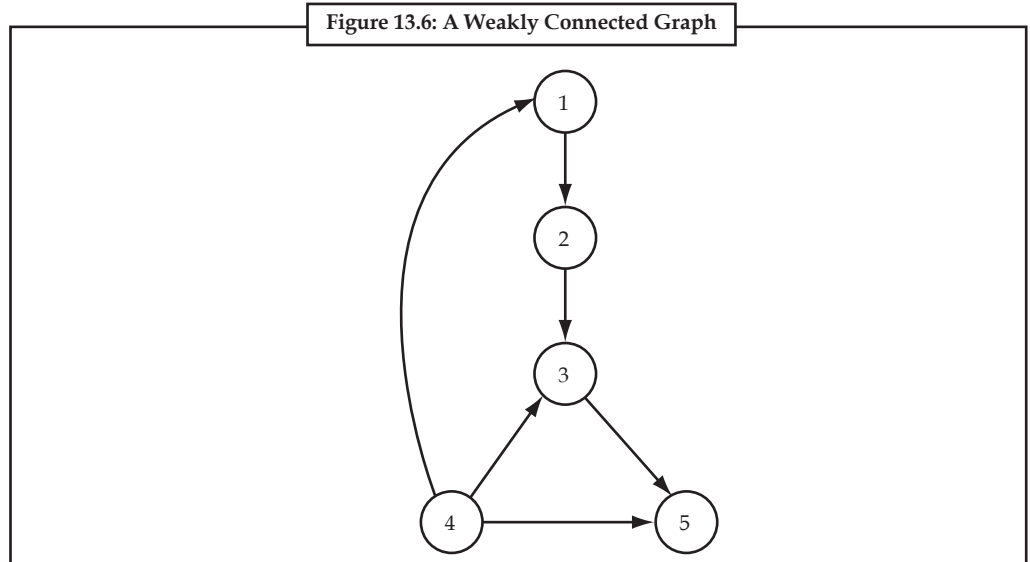
So far we have talked of paths, cycles and connectivity of undirected graph. In a Digraph the path is called a directed path and a cycle as directed cycle.



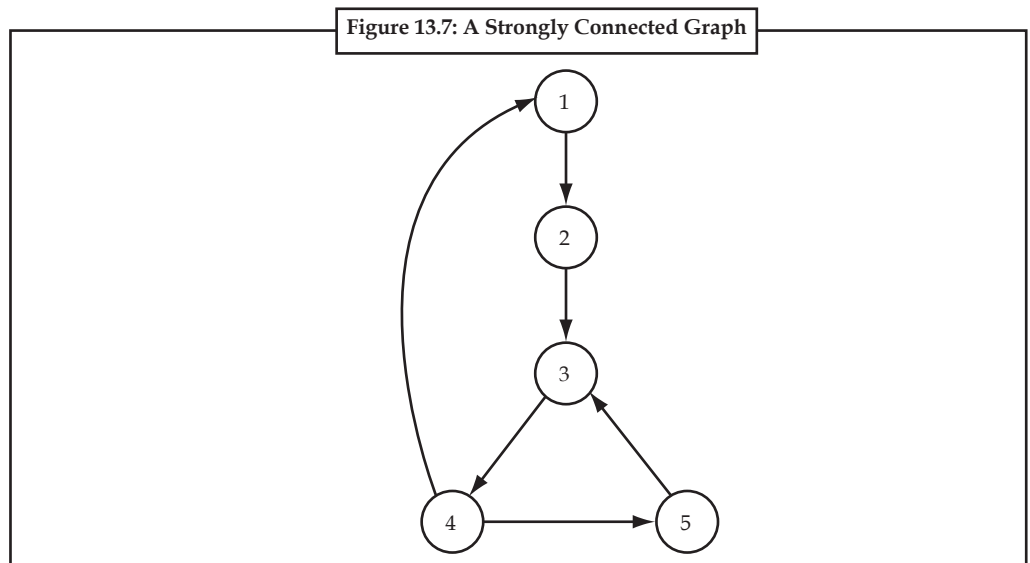
Notes

In Figure 13.4, 5,1,2 is a directed path; 1,3,5,7,6 is a directed path 1,4,5 is not a directed path. There is no directed cycle in the above graph. You may verify the above statement. A digraph is called strongly connected if there is a directed path from any vertex to any other vertex.

Consider the digraph given in Figure 13.6.



There does not exist a directed path from vertex 1 to vertex 4; also from vertex 5 to other vertices; and so on. Therefore, it is a weakly connected graph. Let us make it strongly connected.

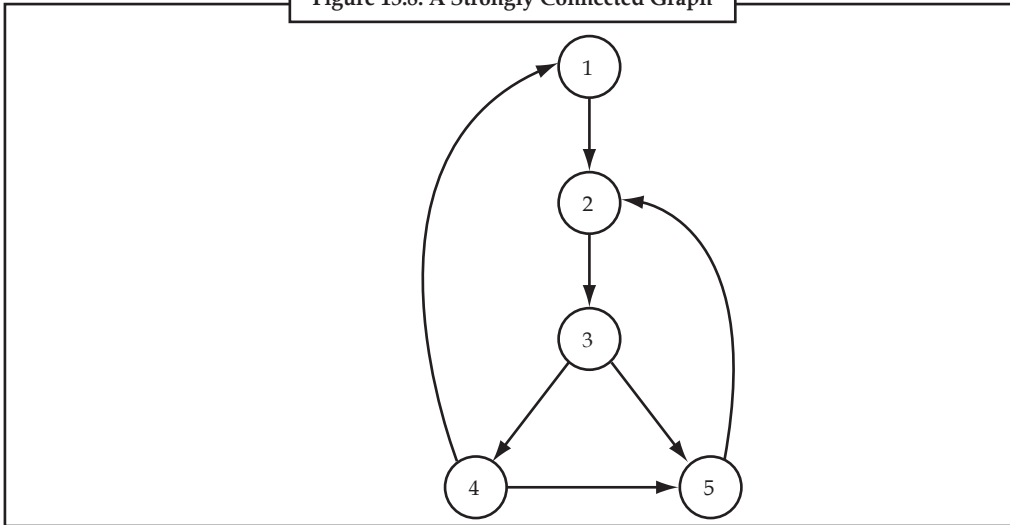


The graph in Figure 13.7 is a strongly connected graph. You may notice that we have added just one arc from vertex 5 to vertex 3.

An alternative could be as given in Figure 13.8.

Notes

Figure 13.8: A Strongly Connected Graph

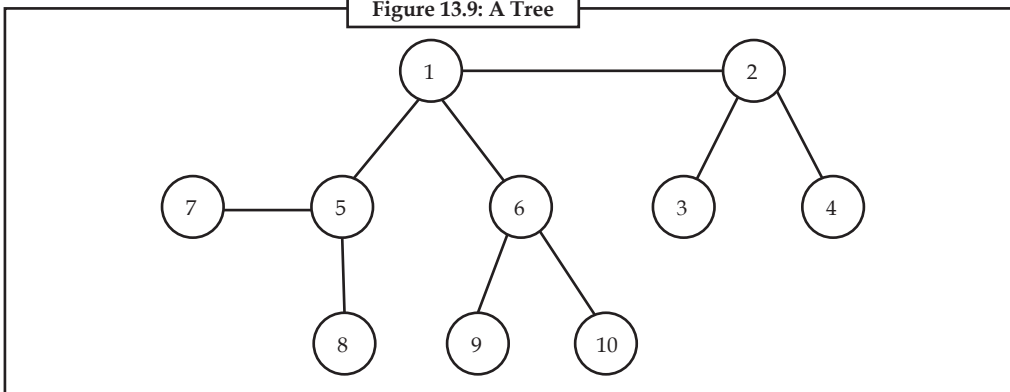


There may exist more alternate structures. Make at least one more alternate structure for the same diagram.

You must have observed that there is no limitation of number of edges incident on one vertex. It could be none, one, or more. The number of edges incident on a vertex determines its degree.

In a digraph we attach an indegree and an outdegree to each of the vertices. In Figure 13.8, the indegree of vertex 5 is 2 and outdegree is 1. Indegree of a vertex v is the number of edges for which vertex v is a head and outdegree is the number of edges for which vertex is a tail.

Figure 13.9: A Tree

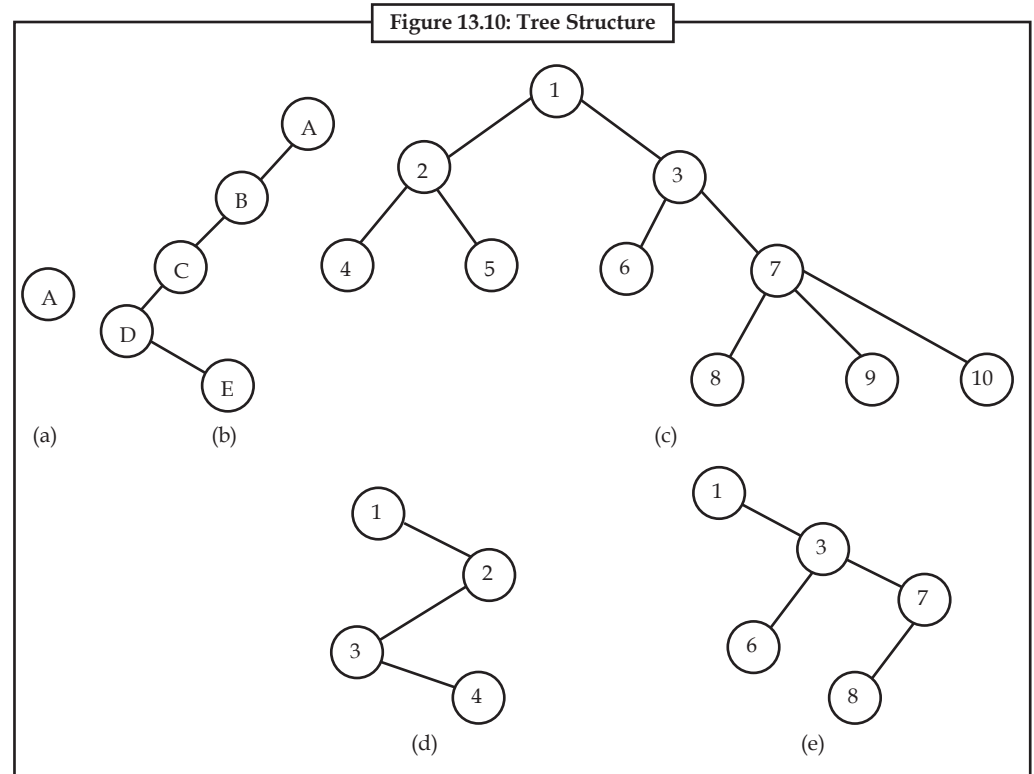


Let us define a special type of graph called Tree. A graph is a tree if it has two properties:

It is connected, and there are no cycles in the graph.

Notes

Graph depicted in Figure 13.9 is a tree and so are the ones depicted in Figure 13.10(a) to 13.10(e).



Because of their special structure and properties, trees occur in many different applications in computer science.



Did u know? In Figure 13.8 how many outdegree present in vertex 4.

13.3 Representations of Graphs

Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus, this mathematical structure must be represented as some kind of data structures. Two such representations are, commonly used. These are:

1. Adjacent Matrix
2. Adjacency List representation.

The choice of representation depends on the application and function to be performed on the graph.

13.3.1 Adjacent Matrix

The adjacency matrix A for a graph $G = (V,E)$ with n vertices, is an $n \times n$ matrix of bits, such that A

$$A_{ij} = 1, \text{ iff there is an edge from } v_i \text{ to } v_j \text{ and}$$

$$A_{ij} = 0, \text{ if there is no such edge.}$$

Table 13.1(a) shows the adjacency matrix for the graph given in Figure 13.1.

Table 13.1: Adjacency Matrix for the Graph in Figure 13.1

Vertice	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	-	1	1
4	0	0	1	0	1
5	1	0	1	1	0

You may observe that the adjacency matrix for an undirected graph is symmetric, as the lower and upper triangles are same. Also all the diagonal elements are zero., since we consider graphs without any self loops. Let us find adjacency matrix for a digraph given in Figure 13.5.

Table 13.2: Adjacency Matrix for Digraph in Figure 13.5

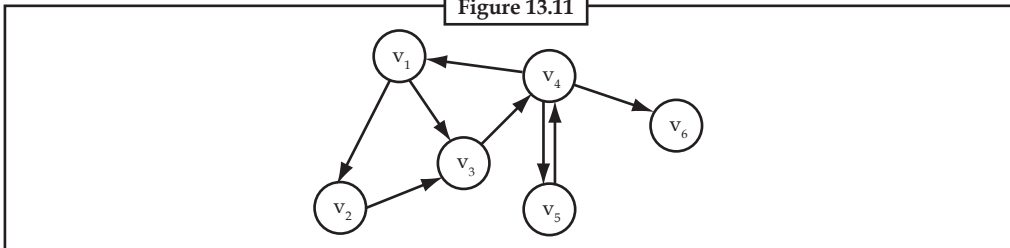
Vertice	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

The total number of 1's account for the number of edges in the digraph. The number of 1's in each row tells the outdegree of the corresponding vertex.

13.3.2 Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all the vertices in a list and then for each vertex, we have a linked list of its adjacent vertices. Let us see it through an example. Consider the graph given in Figure 13.11.

Figure 13.11



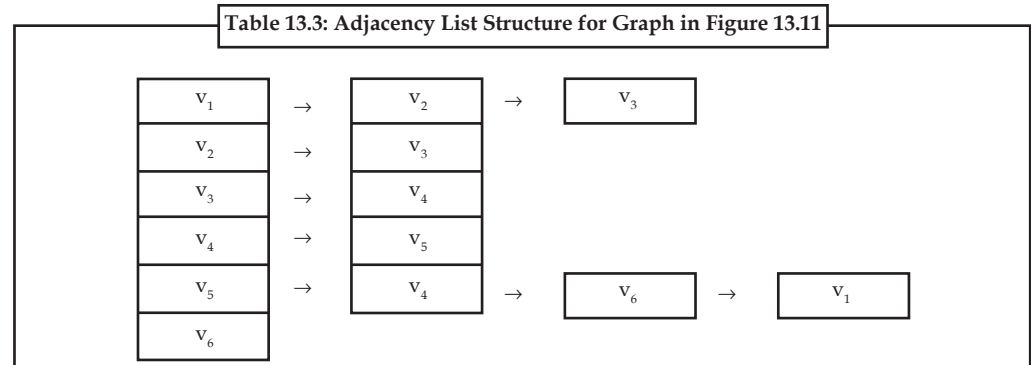
The adjacency list representation needs a list of all of its nodes, i.e.

v ₁
v ₂
v ₃
v ₄
v ₅

And for each node a linked list of its adjacent nodes.

Notes

Therefore we shall have:



Note That adjacent vertices may appear in the adjacency list in arbitrary order. Also an arrow from v₂ to v₃ in the list linked to v₁ does not mean that V₂ and V₃ are adjacent.

The adjacency list representation is better for sparse graphs because the space required is $O(V + E)$, as contrasted with the $O(V^2)$ required by the adjacency matrix representation.

13.4 Shortest Path Algorithms

E.W. Dijkstra developed an algorithm to determine the shortest path between two nodes in a graph. It is also possible to find the shortest paths from a given source node to all nodes in a graph at the same time, hence this problem is sometimes called the single-source shortest paths problem.

The shortest path problem may be expressed as follows:

Given a connected graph $G = (V, E)$, with weighted edges and a fixed vertex s in V , to find a shortest path from s to each vertex v in V . The weights assigned to the edges may represent distance, cost, effort or any other attribute that needs to be minimized in the graph.

A solution to this problem could be found by finding a spanning tree of the graph. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex.

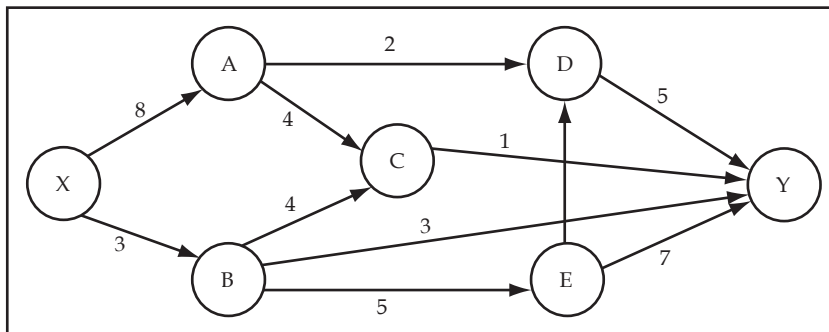
The algorithm finds the routes, by cost precedence. Let's assume that every cost is a positive number. The algorithm is equally applicable to a graph, a digraph, or even to a mixed graph with only some of its sides directed. If we consider a digraph, then every other case is fully covered as well since a no directed side can be considered a 2 directed sides of equal cost for every direction.

The algorithm is based on the fact that every minimal path containing more than one side is the expansion of another minimal path containing a side less. This happens because all costs are considered as positive numbers. In this way the first route $D(1)$ found by the algorithm will be one arc route, that is from the starting point to one of the sides directly connected to this starting point. The next route $D(2)$ will be a one arc route itself, or a two arc route, but in this case will be an expansion of $D(1)$.

Here is the algorithm.

Notes

1. Let V be the set of all the vertices of the graph and S be the set of all the vertices considered for the determination of the minimal path.
2. Set $S = \{ \}$.
3. While there are still vertices in $V - S$.
 - (a) Sort the vertices in $V - S$ according to the current best estimate of their distance from the source.
 - (b) Add u , the closest vertex in $V - S$, to S .
 - (c) Re-compute the distances for the vertices in $V - S$
 - (d) Consider the following example for illustration. Find the shortest path from node X to node Y in the following graph. A label on an edge indicates the distance between the two nodes the edge connects.



Applying Dijkstra algorithm:

1. $S = \{X\}$

Distances of all the nodes from the nodes in the S :

$$XA = 8 \quad XB = 3 \quad XC = \infty$$

$$XD = \infty \quad XE = \infty \quad XY = \infty$$

2. Since, minimum distance from S to $V-S$ is 3 (XB), $S = \{X, B\}$ and $E = \{XB\}$

3. Distances of all the nodes from the nodes in the S :

$$XA = 8 \quad XC = \infty \quad XD = \infty \quad XE = \infty \quad XY = \infty$$

$$XBA = \infty \quad XBC = 7 \quad XBD = \infty \quad XBE = 8 \quad XBY = 6$$

4. Since, minimum distance from S to $V - S$ is 6 (XBY), $S = \{X, B, Y\}$ and $E = \{XBY\}$.

5. Distances of all the nodes from the nodes in the S :

$$XA = 8 \quad XC = \infty \quad XD = \infty \quad XE = \infty$$

$$XBA = \infty \quad XBC = 7 \quad XBD = \infty \quad XBE = 8$$

$$XBYA = \infty \quad XBYC = \infty \quad XBYD = \infty \quad XBYE = \infty$$

Continuing in similar manner, we find that the shortest path between nodes X and Y is XBY with cost value 6.

Notes

13.5 Summary

- Graphs provide in excellent way to describe the essential features of many applications.
- Graphs are mathematical structures and are found to be useful in problem solving. They may be implemented in many ways by the use of different kinds of data structures.
- Graph traversals, Depth First as well as Breadth First, are also required in many applications.
- Existence of cycles makes graph traversal challenging, and this leads to finding some kind of acyclic subgraph of a graph. That is we actually reach at the problems of finding a shortest path and of finding a minimum cost spanning tree problems.

13.6 Keywords

Adjacent: Two vertices in an undirected graph are called adjacent

Edges: A graph G consists of a set V , whose members are called the vertices of G , together with a set E of pairs of distinct vertices from V . These pairs are called the edges of G .

Free Tree: A free tree is defined as a connected undirected graph with no cycles.

Graph: Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers.

Undirected Graph: If the pairs are unordered, then G is called an undirected graph

13.7 Self Assessment

Fill in the blanks:

1. A graph may have many
2. The weight of a tree is just the sum of weights of its
3. The ends when no more paths are found.
4. In an undirected graph, pair of vertices representing any edge is
5. In a Digraph the path is called a directed path and a cycle as

State whether the following statements are true or false:

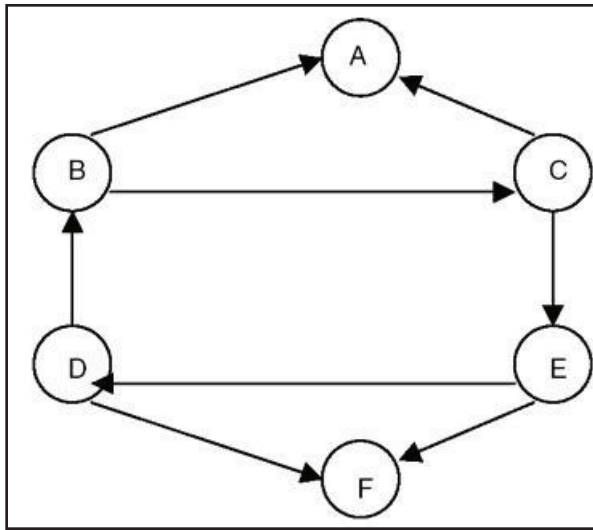
6. E.W. Dijkstra developed an algorithm to determine the shortest path between two nodes in a graph.
7. The algorithm is not equally applicable to a graph, a digraph, or even to a mixed graph with only some of its sides directed.
8. A cycle is not a path in which first and last vertices are the same.
9. A good deal of nomenclature is associated with graphs.
10. In a Digraph the path is called a directed path and a cycle as directed cycle.

13.8 Review Questions

1. What do you mean by shortest path?
2. Find out the minimum number of edges in a strongly connected digraph on n vertices.

3. Test the program for obtaining the depth first spanning tree for the following graph:

Notes



4. "A graph may have many spanning trees; for instance the complete graph on four vertices has sixteen spanning trees". Explain
5. Define vertices of a graph.
6. A graph is regular if every vertex has the same valence (that is, if it is adjacent to the same number of other vertices). For a regular graph, a good implementation is to keep the vertices in a linked list and the adjacency lists contiguous. The length of all the adjacency lists is called the degree of the graph.
7. The topological sorting functions as presented in the text are deficient in error checking. Modify the (a) depth-first and (b) breadth-first functions so that they will detect any (directed) cycles in the graph and indicate what vertices cannot be placed in any topological order because they lie on a cycle.
8. How can we determine a maximal spanning tree in a network?
9. Write Digraph methods called write that will write pertinent information specifying a graph to the terminal. The graph is to be implemented with
- An adjacency table;
 - A linked vertex list with linked adjacency lists;
 - A contiguous vertex list of linked adjacency lists.
10. Implement and test the method for determining shortest distances in directed graphs with weights.

Answers: Self Assessment

- | | |
|-------------------|--------------|
| 1. spanning trees | 2. edges |
| 3. algorithm | 4. unordered |
| 5. directed cycle | 6. True |
| 7. False | 8. False |
| 9. True | 10. True |

13.9 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E. Leiserson & Ronald L., *Rivest: Introduction to Algorithms*. Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 14: Network Flows

Notes

CONTENTS

Objectives

Introduction

14.1 Network Flow

14.1.1 Ford Fulkerson Method

14.1.2 Comparison Networks

14.2 Network Flow Problem

14.3 Minimum Spanning Tree

14.3.1 Kruskal's Algorithm

14.3.2 Prim's Algorithm

14.4 Summary

14.5 Keywords

14.6 Self Assessment

14.7 Review Questions

14.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain network flow
- Describe problem of network flow
- Know minimum spanning tree

Introduction

We use graphs as models of practical situations involving routes: the vertices represent the cities and edges represent the roads or some other links, specially in transportation management, Assignment problems and many more optimization problems. Electric circuits are another obvious example where interconnections between objects play a central role. Circuits elements like transistors, resistors, and capacitors are intricately wired together. Such circuits can be represented and processed within a computer in order to answer simple questions like "Is everything connected together?" as well as complicated questions like "If this circuit is built, will it work?"

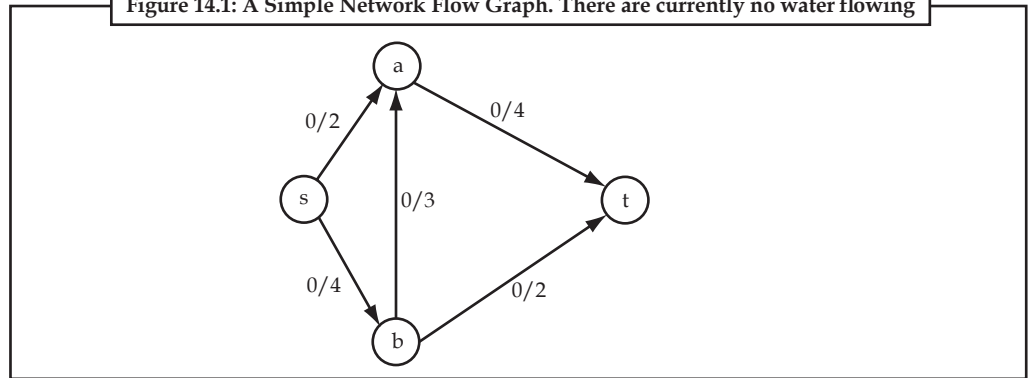
14.1 Network Flow

Network flow is an advanced branch of graph theory. The problem resolves around a special type of weighted directed graph with two special vertices: the source vertex, which has no incoming edge, and the sink vertex, which has no outgoing edge. By convention, the source vertex is usually labelled s and the sink vertex labelled t .

Notes

There are a bunch of junctions (nodes in the graph) and a bunch of pipes (edges in the graph) connecting the junction. The pipe will only allow water to flow one way (the graph is directed). Each pipe has also has a capacity (the weight of the edge), representing the maximum amount of water that can flow through the pipe. Finally, we pour an infinite amount of water into the source vertex. The problem is to find the maximum flow of the graph - the maximum amount of water that will flow to the sink. Below is an example of a network flow graph.

Figure 14.1: A Simple Network Flow Graph. There are currently no water flowing



It is fairly easy to see that the maximum flow in the Figure 14.1 is 6. We can flow 2 units of water from $s \rightarrow a \rightarrow t$, 2 units of water from $s \rightarrow b \rightarrow a \rightarrow t$, and 2 units of water from $s \rightarrow b \rightarrow t$. This gives a flow of 6 and since all incoming edge to the sink are saturated, this is indeed the maximum flow.



Note

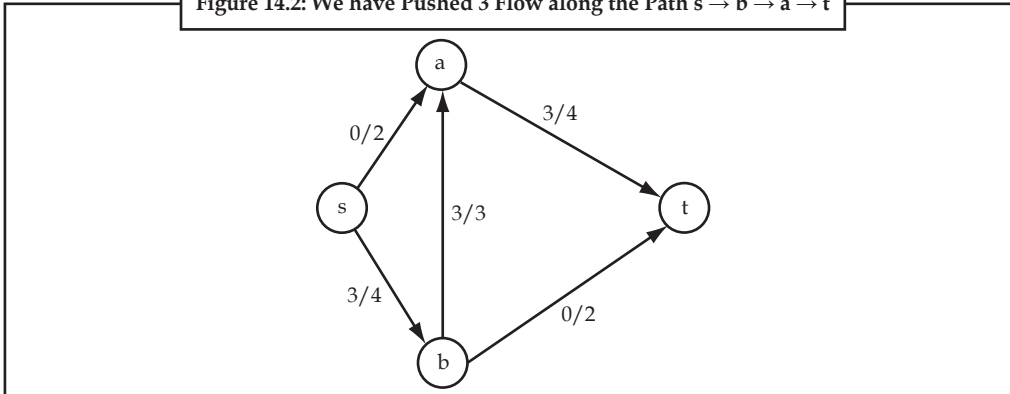
In the example, not all pipe are saturated with water.

14.1.1 Ford Fulkerson Method

It was easy to see the solution in the above example, but how do we find the solution in general? One idea is to keep finding path from s to t along pipes which still has some capacities remaining and push as much flow from s to t as possible. We will then terminate once we can't find any more path. This idea seem to work since it is exactly how we found the maximum flow in the example. However, there is one problem - we cannot guarantee which path we'll find first. In fact, if we picked the wrong path, the whole algorithm will go wrong. For example, what happens if

the first path we found was $s \rightarrow b \rightarrow a \rightarrow t$. If we push as much flow as possible, then we end up with the following:

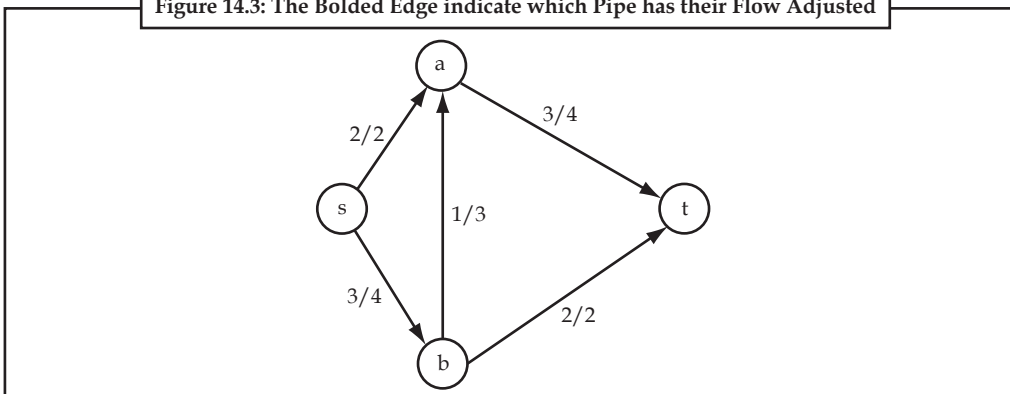
Figure 14.2: We have Pushed 3 Flow along the Path $s \rightarrow b \rightarrow a \rightarrow t$



Now we have run into a problem: our only options left are to push 1 unit of water along the path $s \rightarrow a \rightarrow t$ and 1 unit of water along $s \rightarrow b \rightarrow t$. After that, we won't be able to find any more path from s to t ! Yet, we have only found 5 flow, which is not the maximum flow. Thus, the idea is not optimal since it depends on how we picked our path. While we can try to find a "good" path-picking algorithm, it would be nice if the algorithm is not dependent on the paths we chose.

A crucial observation is that there is actually another path from s to t other than the two that we mentioned above! Suppose we redirect 2 units of water from $b \rightarrow a \rightarrow t$ to $b \rightarrow t$, this will decrease the amount of water running through the pipe $(a; t)$ to 0. Now we have a path from $s \rightarrow a \rightarrow t$ in which we can flow 2 units of water! The graph now looks as follows:

Figure 14.3: The Bolded Edge indicate which Pipe has their Flow Adjusted

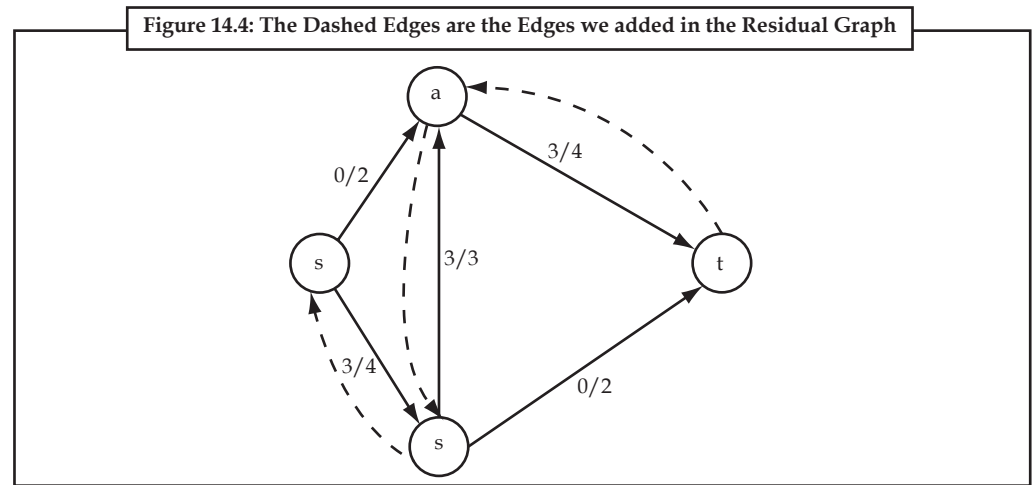


It is now easy to see that we can push 1 unit of water along $s \rightarrow b \rightarrow a \rightarrow t$ to obtain the maximum flow of 6. If we carefully study in the Figure 14.3, what we have essentially done is to flow 2 unit of water along $s \rightarrow a$, and then push back 2 unit of water from $a \rightarrow b$, and finally redirect the pushed back flow along $b \rightarrow t$ to the sink. So the key to completing the algorithm is the idea of pushing back flow - if we have x units of water flowing in the pipe $(u; v)$, then we can pretend there is a pipe $(v; u)$ with capacity x when we are trying to find a path from s to t .

This is the concept of residual graph. The residual graph of a network flow is essentially the network graph except that for every edge $(u; v)$ that currently carries x unit of water, there is

Notes

an edge $(v; u)$ with capacity x in the residual graph. Figure 14.4 shows the residual graph after finding our first path:



The capacity of the dashed edge is the same as the amount of water carried by the solid edge in the opposite direction.

So here is an algorithm to find the maximum flow: First construct the residual graph (in the beginning, the residual graph is the same as the network graph). While there exists a path from s to t in the residual graph, we flow water through one of the path (a path in the a residual graph is called an augmenting path). We then adjust the residual graph accordingly. Once we can no longer find any more path in the residual graph, we have found the maximum flow.

14.1.2 Comparison Networks

Suppose we have a directed network $G = (V, E)$ defined by a set V of nodes (or vertexes) and a set E of arcs (or edges). Each arc (i, j) in E has an associated nonnegative capacity u_{ij} . Also we distinguish two special nodes in G : a source node s and a sink node t . For each i in V we denote by $E(i)$ all the arcs emanating from node i . Let $U = \max u_{ij}$ by (i, j) in E . Let us also denote the number of vertexes by n and the number of edges by m .

We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes. Representing the flow on arc (i, j) in E by x_{ij} we can obtain the optimization model for the maximum flow problem:

$$\text{Maximize } f(x) = \sum_{(i,j) \in E(s)} x_{ij}$$

subject to

$$\sum_{(j,i) \in E} x_{ij} - \sum_{(j,i) \in E} x_{ij} = 0 \quad \forall i \in V \setminus \{s, t\}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i, j) \in E$$

Vector (x_{ij}) which satisfies all constraints is called a *feasible solution* or, a *flow* (it is not necessary maximal). Given a flow x we are able to construct the residual network with respect to this flow according to the following intuitive idea. Suppose that an edge (i, j) in E carries x_{ij} units of flow. We define the residual capacity of the edge (i, j) as $r_{ij} = u_{ij} - x_{ij}$. This means that we can send an additional r_{ij} units of flow from vertex i to vertex j . We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to i over the arc (i, j) .

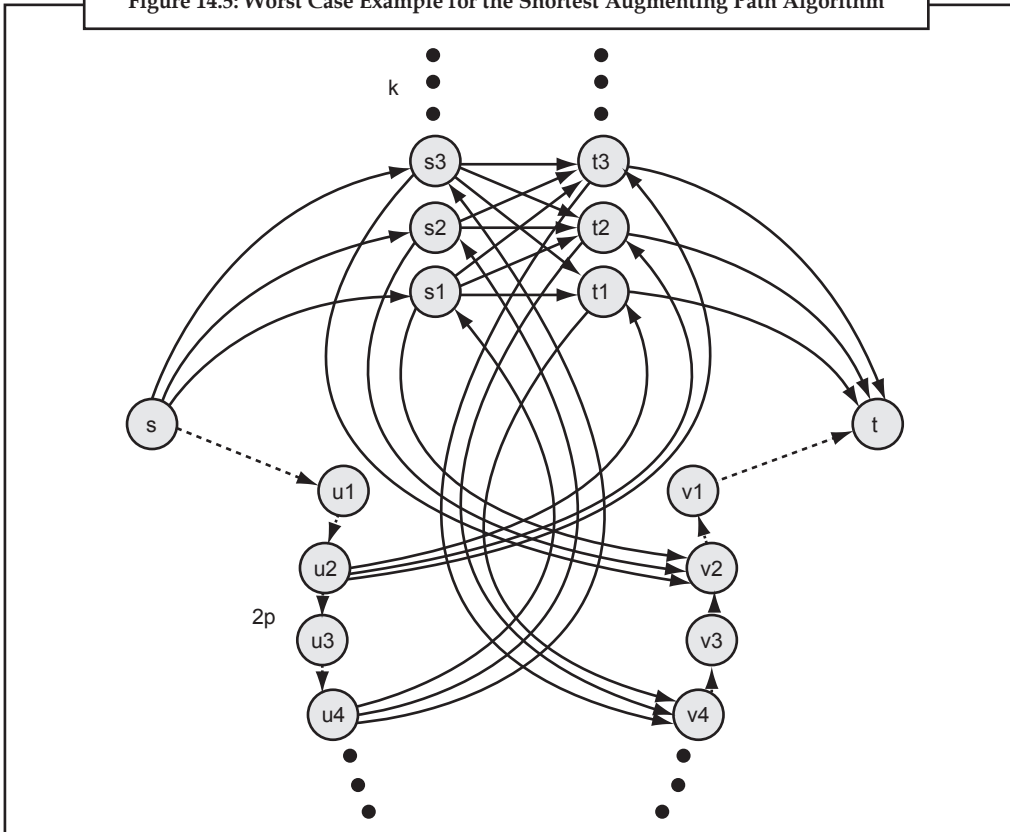
So, given a feasible flow x we define the residual network with respect to the flow x as follows. Suppose we have a network $G = (V, E)$. A feasible solution x engenders a new (residual) network,

which we define by $G_x = (V, E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x .

What is E_x ? We replace each arc (i,j) in E by two arcs $(i,j), (j,i)$: the arc (i,j) has (residual) capacity $r_{ij} = u_{ij} - x_{ij}$ and the arc (j,i) has (residual) capacity $r_{ji} = x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity.

In the worst case both improved and unimproved algorithms will perform $O(n^3)$ augmentations, if $m \sim n^2$. Norman Zadeh developed some examples on which this running time is based. Using his ideas we compose a somewhat simpler network on which the algorithms have to perform $O(n^3)$ augmentations and which is not dependent on a choice of next path.

Figure 14.5: Worst Case Example for the Shortest Augmenting Path Algorithm



All vertices except s and t are divided into four subsets: $S = \{s_1, \dots, s_k\}$, $T = \{t_1, \dots, t_k\}$, $U = \{u_1, \dots, u_p\}$ and $V = \{v_1, \dots, v_p\}$. Both sets S and T contain k nodes while both sets U and V contain $2p$ nodes. k and p are fixed integers. Each bold arc (connecting S and T) has unit capacity. Each dotted arc has an infinite capacity. Other arcs (which are solid and not straight) have capacity k .

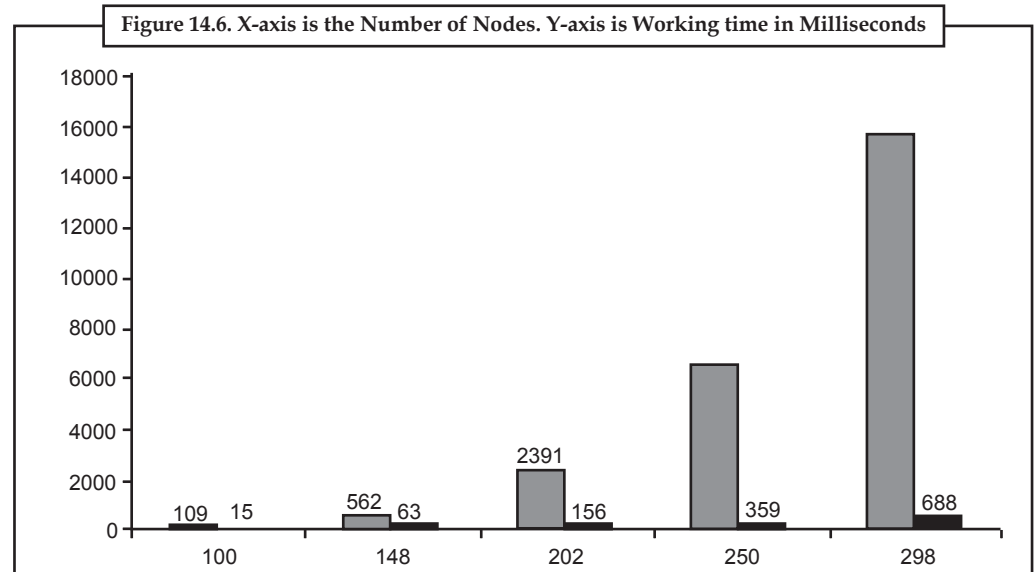
First, the shortest augmenting path algorithm has to augment flow k^2 time along paths (s, S, T, t) which have length equal to 3. The capacities of these paths are unit. After that the residual network will contain reversal arcs (T, S) and the algorithm will chose another k^2 augmenting paths $(s, u_1, u_2, T, S, v_2, v_1, t)$ of length 7. Then the algorithm will have to choose paths $(s, u_1, u_2, u_3, u_4, S, T, v_4, v_3, v_2, v_1, t)$ of length 11 and so on...

Now let's calculate the parameters of our network. The number of vertexes is $n = 2k + 4p + 2$. The number of edges is $m = k^2 + 2pk + 2k + 4p$. As it easy to see, the number of augmentations is $a = k^2(p + 1)$.

Notes


Consider that $p = k - 1$. In this case $n = 6k - 2$ and $a = k^3$. So, one can verify that $a \sim n^3/216$. In Zadeh presents examples of networks that require $n^3/27$ and $n^3/12$ augmentations, but these examples are dependent on a choice of the shortest path.

We made 5 worst-case tests with 100, 148, 202, 250 and 298 vertexes and compared the running times of the improved version of the algorithm against the unimproved one. As you can see on Figure 14.6, the improved algorithm is much faster. On the network with 298 nodes it works 23 times faster. Practice analysis shows us that, in general, the improved algorithm works $n/14$ times faster.



Blue colour indicates the shortest augmenting path algorithm and red does it improved version.

However, our comparison is not definitive, because we used only one kind of networks. We just wanted to justify that the $O(n^2m)$ algorithm works $O(n)$ times faster than the $O(nm^2)$ on a dense network.



Task "Network flow is an advanced branch of graph theory." Discuss.

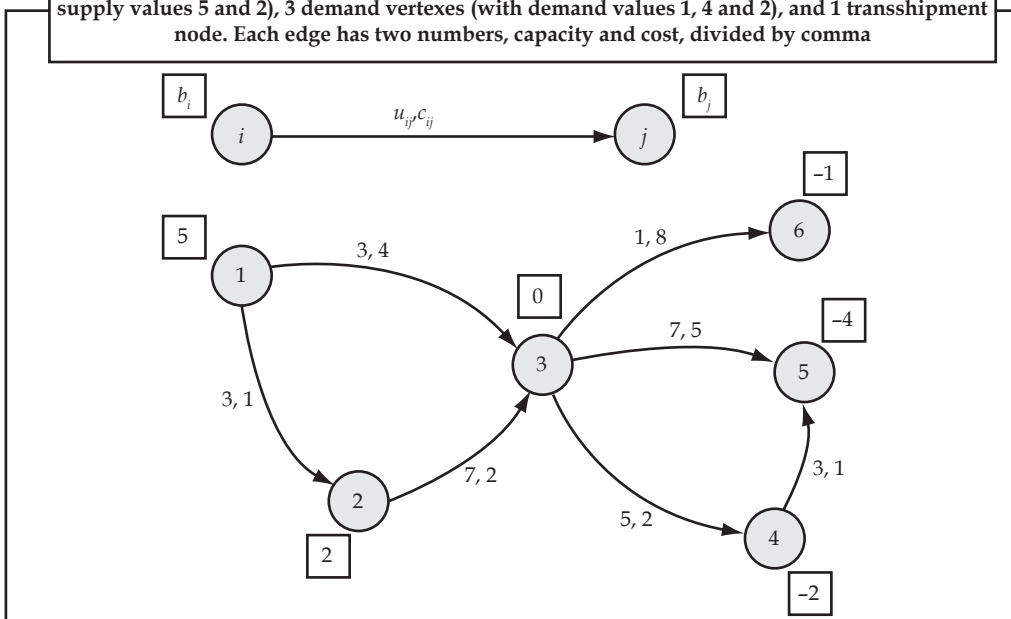
14.2 Network Flow Problem

Let $G = (V, E)$ be a directed network defined by a set V of vertexes (nodes) and set E of edges (arcs). For each edge $(i, j) \in E$ we associate a capacity u_{ij} that denotes the maximum amount that can flow on the edge. Each edge $(i, j) \in E$ also has an associated **cost** c_{ij} that denotes the cost per unit flow on that edge.

We associate with each vertex $i \in V$ a number b_i . This value represents supply/demand of the vertex. If $b_i > 0$, node i is a supply node; if $b_i < 0$, node i is a demand node (its demand is equal to $-b_i$). We call vertex i a transshipment if b_i is zero.

For simplification, let's call G a transportation network and write $G = (V, E, u, c, b)$ in case we want to show all the network parameters explicitly.

Figure: 14.7: An example of the transportation network. In this we have 2 supply vertexes (with supply values 5 and 2), 3 demand vertexes (with demand values 1, 4 and 2), and 1 transshipment node. Each edge has two numbers, capacity and cost, divided by comma



Representing the flow on arc $(i, j) \in E$ by x_{ij} , we can obtain the optimization model for the minimum cost flow problem:

$$\text{Minimize } z(x) = \sum_{(i,j) \in E} c_{ij}x_{ij}$$

subject to

$$\begin{aligned} \sum_{\{j:(i,j) \in E\}} x_{ij} - \sum_{\{j:(j,i) \in E\}} x_{ji} &= b_i && \text{for all } i \in V, \\ 0 \leq x_{ij} &\leq u_{ij} && \text{for all } (i, j) \in E. \end{aligned}$$

The first constraint states that the total outflow of a node minus the total inflow of the node must be equal to mass balance (supply/demand value) of this node. This is known as the mass balance constraints. Next, the flow bound constraints model physical capacities or restrictions imposed on the flow's range. As you can see, this optimization model describes a typical relationship between warehouses and shops, for example, in a case where we have only one kind of product. We need to satisfy the demand of each shop by transferring goods from the subset of warehouses, while minimizing the expenses on transportation.

This problem could be solved using simplex-method, but in this article we concentrate on some other ideas related to network flow theory. Before we move on to the three basic algorithms used to solve the minimum cost flow problem, let's review the necessary theoretical base.

Finding a Solution

When does the minimum cost flow problem have a feasible (though not necessarily optimal) solution? How do we determine whether it is possible to transport the goods or not?

If $\delta^{def} = \sum_{i \in V} b_i \neq 0$ then the problem has no solution, because either the supply or the demand dominates in the network and the mass balance constraints come into play.

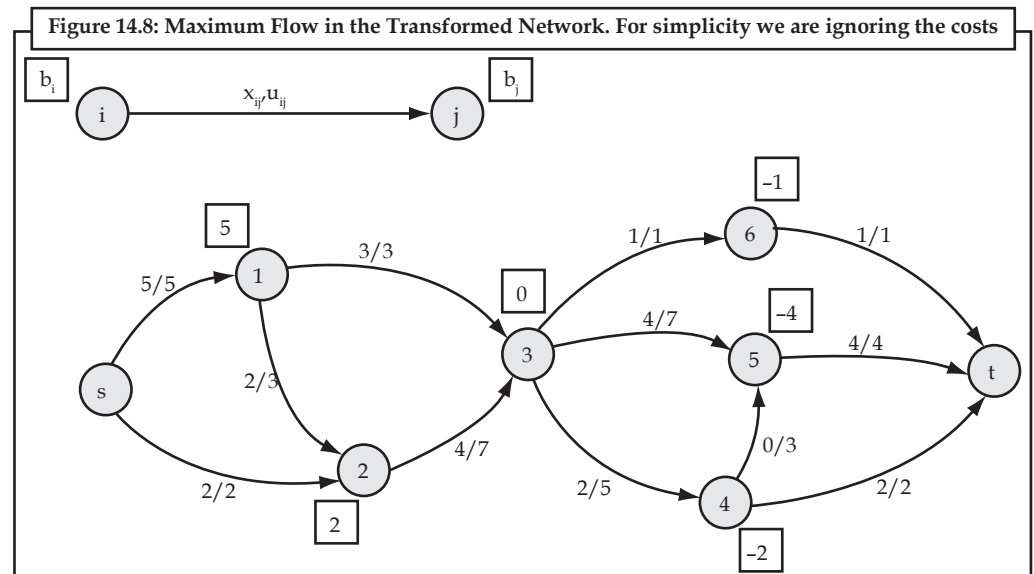
Notes

We can easily avoid this situation, however, if we add a special node r with the supply/demand value $b_r = -\delta$. Now we have two options: If $\delta > 0$ (supply dominates) then for each node $i \in V$ with $b_i > 0$ we add an arc (i, r) with infinite capacity and zero cost; otherwise (demand dominates), for each node $i \in V$ with $b_i < 0$, we add an arc (r, i) with the same properties. Now we have a new network with $\sum_{i \in V \cup \{r\}} b_i = 0$ - and it is easy to prove that this network has the same optimal value as the objective function.

Consider the vertex r as a rubbish or scrap dump. If the shops demand is less than what the warehouse supplies, then we have to eject the useless goods as rubbish. Otherwise, we take the missing goods from the dump. This would be considered shady in real life, of course, but for our purposes it is very convenient. Keep in mind that, in this case, we cannot say what exactly the "solution" of the corrected (with scrap) problem is. And it is up to the reader how to classify the emergency uses of the "dump." For example, we can suggest that goods remain in the warehouses or some of the shop's demands remain unsatisfied.

Even if we have $\delta = 0$ we are not sure that the edge's capacities allow us to transfer enough flow from supply vertexes to demand ones. To determine if the network has a feasible flow, we want to find any transfer way what will satisfy all the problem's constraints. Of course, this feasible solution is not necessarily optimal, but if it is absent we cannot solve the problem.

Let us introduce a source node s and a sink node t . For each node $i \in V$ with $b_i > 0$, we add a source arc (s, i) to G with capacity b_i and cost 0. For each node $i \in V$ with $b_i < 0$, we add a sink arc (i, t) to G with capacity $-b_i$ and cost 0.



The new network is called a transformed network. Next, we solve a maximum flow problem from s to t . If the maximum flow saturates all the source and sink arcs, then the problem has a feasible solution; otherwise, it is infeasible. As for why this approach works, we'll leave its proof to the reader.

Having found a maximum flow, we can now remove source, sink, and all adjacent arcs and obtain a feasible flow in G . How do we detect whether the flow is optimal or not? Does this flow minimize costs of the objective function z ? We usually verify "optimality conditions" for the answer to these questions, but let us put them on hold for a moment and discuss some assumptions.

Now, suppose that we have a network that has a feasible solution. Does it have an optimal solution? If our network contains the negative cost cycle of infinite capacity then the objective

function will be unbounded. However, in some tasks, we are able to assign finite capacity to each uncapacitated edge escaping such a situation.

Notes

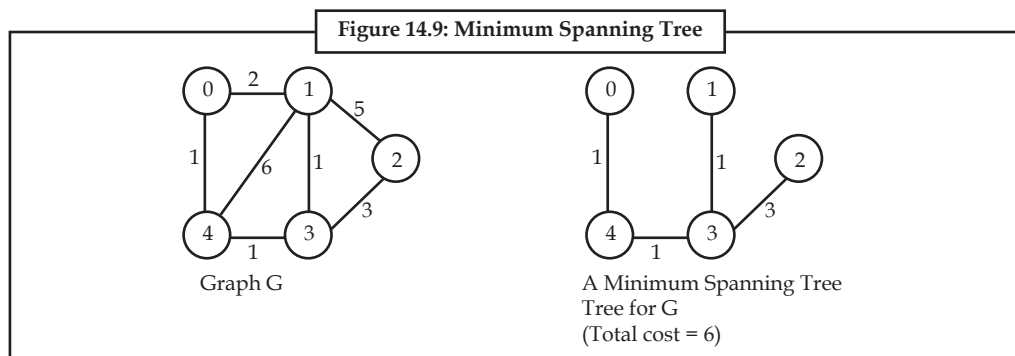
14.3 Minimum Spanning Tree

A minimum spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

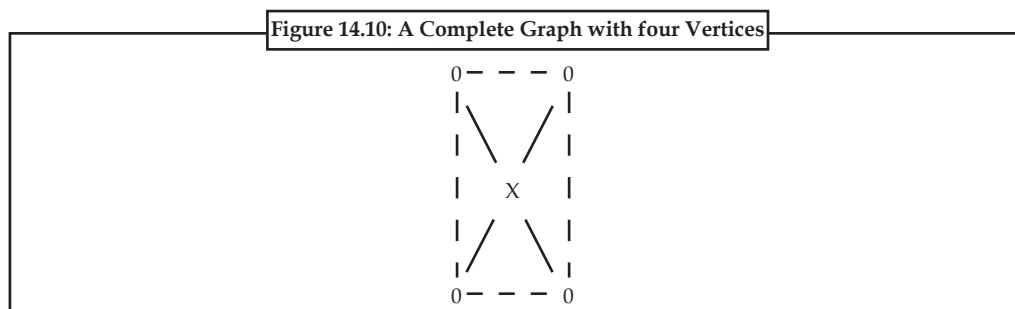
One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights.

A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house.

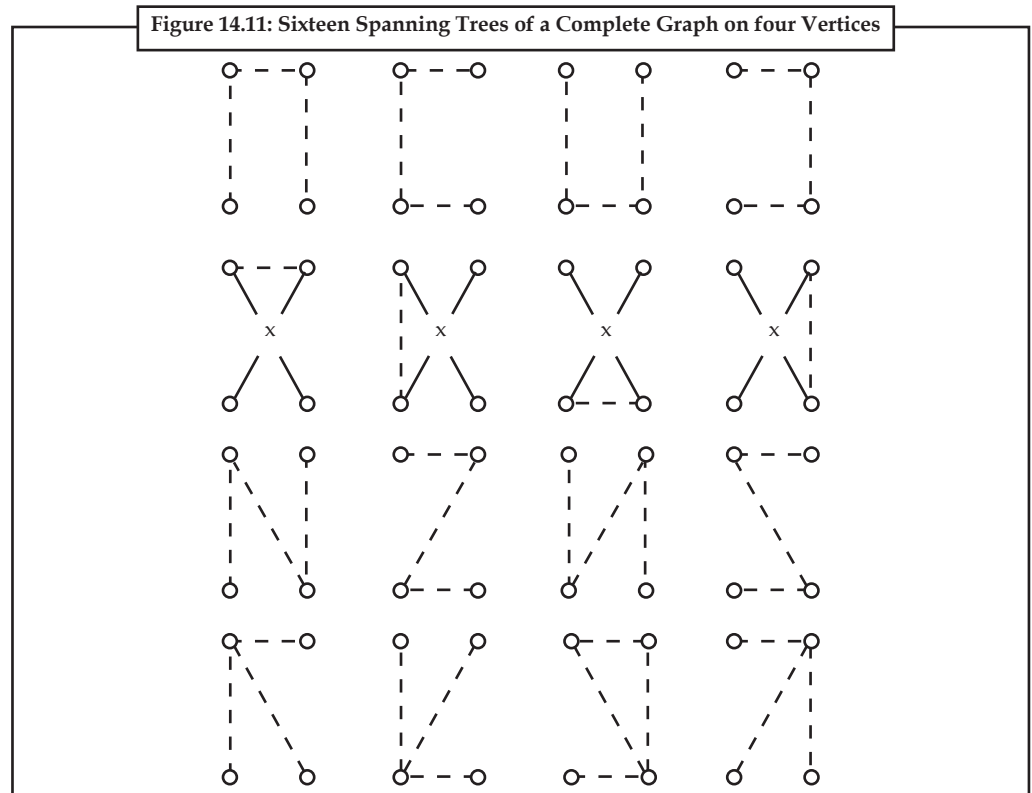
There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.



A graph may have many spanning trees; for instance the complete graph on four vertices has sixteen spanning trees:



Notes



Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. The problem: how to find the minimum length spanning tree?

This problem can be solved by many different algorithms. It is the topic of some very recent research. There are several “best” algorithms, depending on the assumptions you make:

A randomized algorithm can solve it in linear expected time.

It can be solved in linear worst case time if the weights are small integers. Otherwise, the best solution is very close to linear but not exactly linear. The exact bound is $O(m \log \beta(m,n))$ where the beta function has a complicated definition: the smallest i such that $\log(\log(\log(\dots \log(n)\dots)))$ is less than m/n , where the logs are nested i times.

These algorithms are all quite complicated, and probably not that great in practice unless you’re looking at really huge graphs. Here we’ll go through two simple classical algorithms.

14.3.1 Kruskal’s Algorithm

We’ll start with Kruskal’s algorithm, which is easiest to understand and probably the best one for solving problems by hand.

Kruskal’s algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

Kruskal’s algorithm is an example of a greedy algorithm. This algorithm first appeared in Proceedings of the American Mathematical Society in 1956, and was written by Joseph Kruskal.

Algorithm**Notes**

Let $G = (V, E)$ which is represented by an adjacency list Adj. Some support data structures:

1. F is the forest – a set of all (partial) trees.
2. MST is the minimum spanning tree, represented by a set of edges.
3. Q is a priority queue of edges.

Kruskal(G)

1. Let F be a set of singleton set of all vertices in G.
2. MST \leftarrow {}
3. Q \leftarrow E
4. while Q not empty do
5. (u, v) \leftarrow ExtractMin(Q) // Q is modified
6. if FIND-SET(u) \neq FIND-SET(v) then // FIND-SET(i) returns the set in F
 - // which vertex i belongs to.
 - // This effectively does cycle check.
 - // If ACCEPTED,
7. begin
8. merge(FIND-SET(u), FIND-SET(v)) in F
9. MST \leftarrow MST Union {(u, v)}
10. end
11. return MST.

Proof of Correctness

Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P.

It remains to show that the spanning tree Y is minimal:

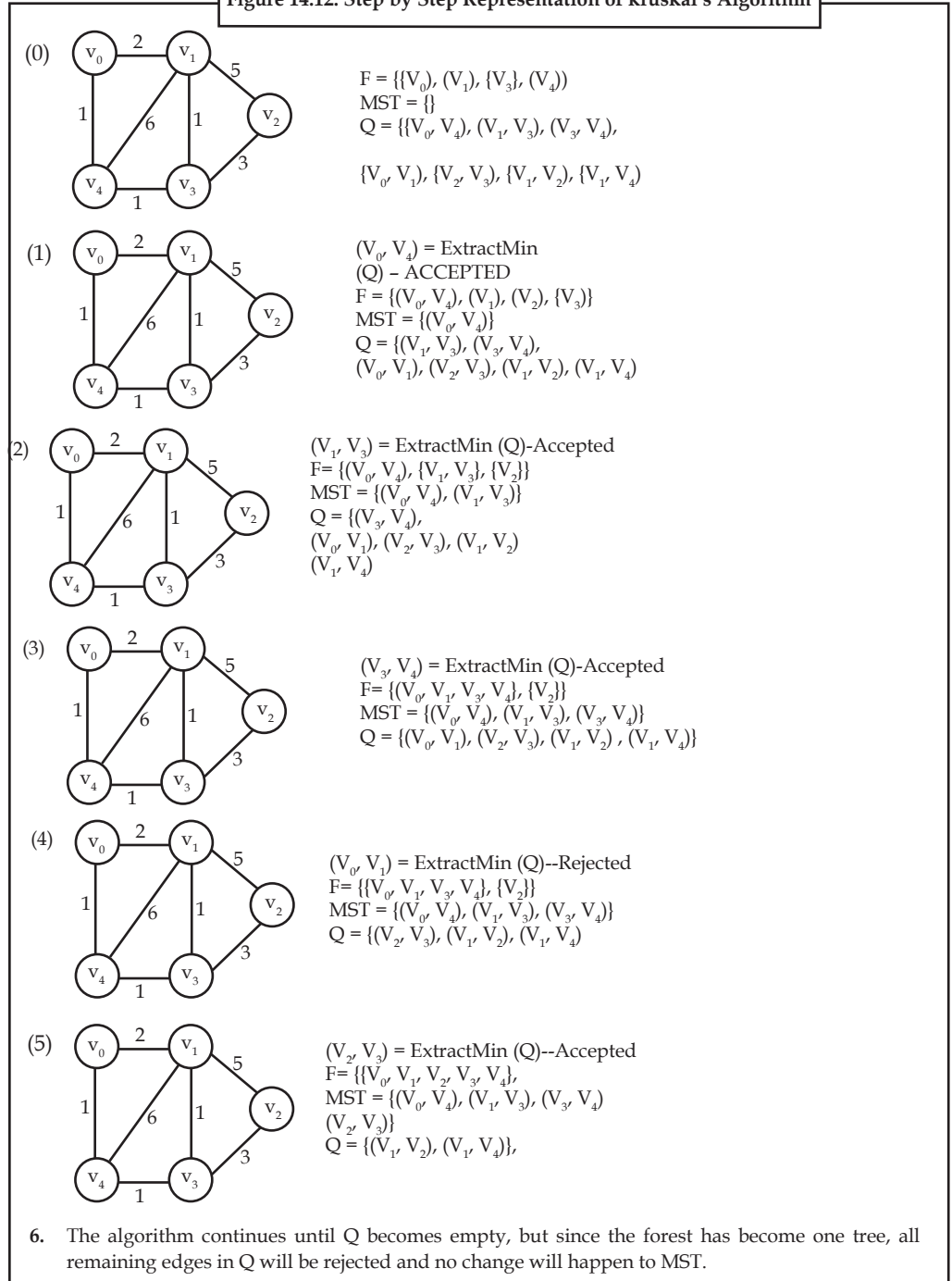
Let Y_1 be a minimum spanning tree. If $Y = Y_1$ then Y is a minimum spanning tree. Otherwise, let e be the first edge considered by the algorithm that is in Y but not in Y_1 . $Y_1 \cup e$ has a cycle, because you cannot add an edge to a spanning tree and still have a tree. This cycle contains another edge f which at the stage of the algorithm where e is added to Y, has not been considered. This is because otherwise e would not connect different trees but two branches of the same tree. Then $Y_2 = Y_1 \cup e/f$ is also a spanning tree.

Its total weight is less than or equal to the total weight of Y_1 . This is because the algorithm visits e before f and therefore $w(e) \leq w(f)$. If the weights are equal, we consider the next edge e which is in Y but not in Y_1 . If there is no edge left, the weight of Y is equal to the weight of Y_1 although they consist of a different edge set and Y is also a minimum spanning tree.

Notes

In the case where the weight of Y_2 is less than the weight of Y_1 we can conclude that Y_1 is not a minimum spanning tree, and the assumption that there exist edges e, f with $w(e) < w(f)$ is incorrect. And therefore Y is a minimum spanning tree (equal to Y_1 or with a different edge set, but with same weight).

Figure 14.12: Step by Step Representation of Kruskal's Algorithm



- Notes
- 13. end
 - 14. end
 - 15. end

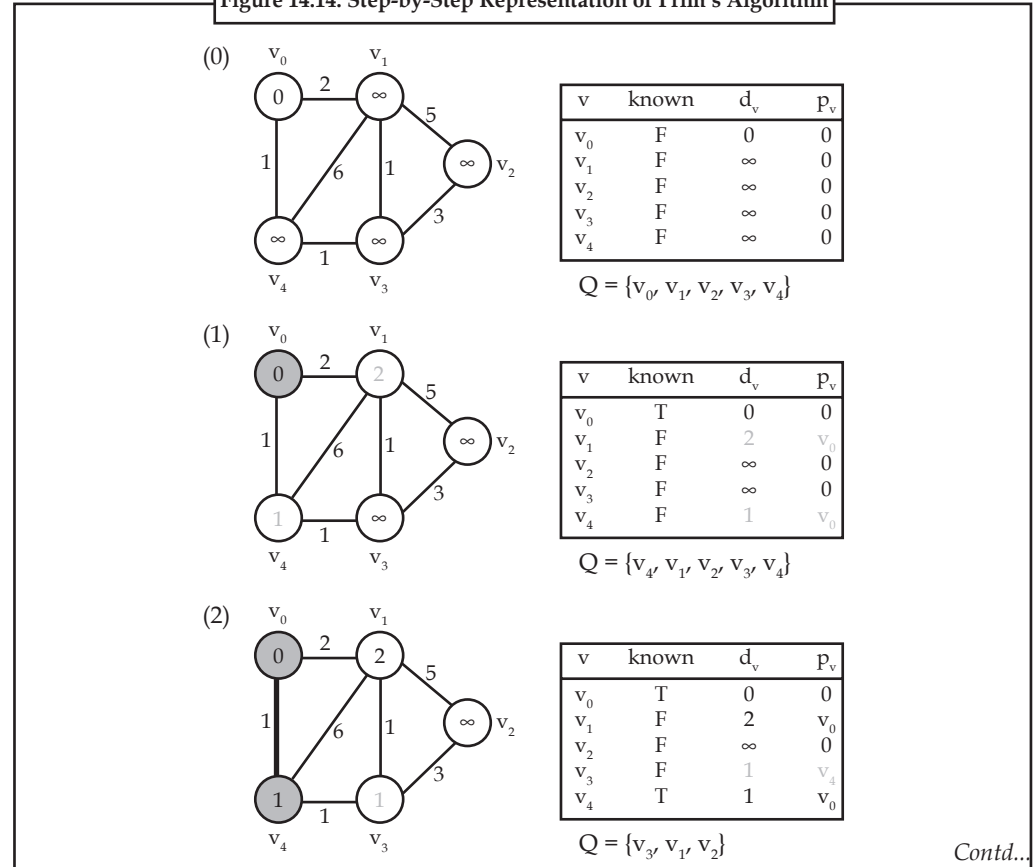
Proof of Correctness

Let P be a connected, weighted graph. At every iteration of Prim’s algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim’s algorithm is a tree, because the edge and vertex added to Y are connected. Let Y_1 be a minimum spanning tree of P . If $Y_1 = Y$ then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of Y that is not in Y_1 , and V be the set of vertices connected by the edges added before e . Then one endpoint of e is in V and the other is not. Since Y_1 is a spanning tree of P , there is a path in Y_1 joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in V to one that is not in V . Now, at the iteration when e was added to Y , f could also have been added and it would be added instead of e if its weight was less than e . Since f was not added, we conclude that

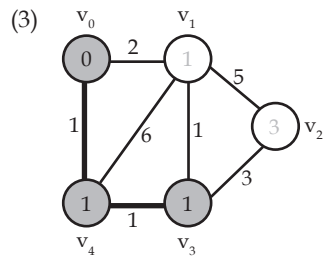
$$w(f) \leq w(e).$$

Let Y_2 be the graph obtained by removing f and adding e from Y_1 . It is easy to show that Y_2 is connected, has the same number of edges as Y_1 , and the total weights of its edges is not larger than that of Y_1 , therefore it is also a minimum spanning tree of P and it contains e and all the edges added before it during the construction of V . Repeat the steps above and we will eventually obtain a minimum spanning tree of P that is identical to Y_1 . This shows Y is a minimum spanning tree.

Figure 14.14: Step-by-Step Representation of Prim’s Algorithm

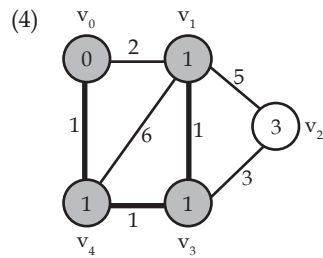


Notes



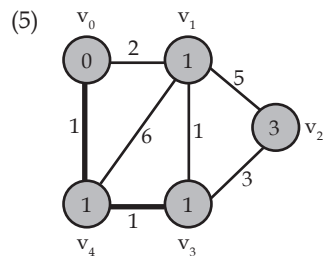
v	known	d_v	p_v
v_0	T	0	0
v_1	F	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{v_1, v_2\}$



v	known	d_v	p_v
v_0	T	0	0
v_1	T	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{v_2\}$



v	known	d_v	p_v
v_0	T	0	0
v_1	F	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{\}$



Note

v_0 is the source vertex, and $d[i]$ for each vertex i is also indicated in its circle:

14.4 Summary

- Perhaps the best safeguard in the use of powerful tools is to insist on regularity; that is, to use the powerful tools only in carefully defined and well-understood ways. Because of the generality of graphs, it is not always easy to impose this discipline on their use. In this world, nonetheless, irregularities will always creep in, no matter how hard we try to avoid them. It is the bane of the systems analyst and programmer to accommodate these irregularities while trying to maintain the integrity of the underlying system design. Irregularity even occurs in the very systems that we use as models for the data structures we devise, models such as the family trees whose terminology we have always used.

14.5 Keywords

Network Flow: Network flow is an advanced branch of graph theory.

Zero-One Principle: If a sorting network sorts every sequence of 0's and 1's, then it sorts every arbitrary sequence of values.

Prim's algorithm: Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.

Notes

Minimum Spanning Tree: A minimum spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

Kruskal's algorithm: Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.

14.6 Self Assessment

State whether the following statements are true or false:

1. Network flow is an advanced branch of graph theory.
2. Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.
3. Kruskal's algorithm is not an example of a greedy algorithm.
4. Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.
5. A graph may not have many spanning trees.

Fill in the blanks:

6. Prism algorithm was discovered in 1930 by mathematician
7. developed some examples on which the running time is based.
8. A can solve it in linear expected time.
9. Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected graph.
10. Prism's algorithms sometimes called the

14.7 Review Questions

1. What do you mean by network flow?
2. Explain Ford Fulkerson's method of network flow.
3. What do you mean by minimum spanning tree?
4. Describe proof of correctness of Kruskal's algorithm
5. Explain prim's algorithm
6. Dijkstra's algorithm to compute a minimal spanning tree in a network works by considering all edges in any convenient order. As in Kruskal's algorithm, we select edges for a spanning tree, by adding edges to an initially empty set. However, each edge is now selected as it is considered, but if it creates a cycle together with the previously selected edges, the most expensive edge in this cycle is deselected. Prove that the edges chosen by Dijkstra's algorithm also form a minimal spanning tree of a connected network.
7. Kruskal's algorithm to compute a minimal spanning tree in a network works by considering all edges in increasing order of weight. We select edges for a spanning tree, by adding edges to an initially empty set. An edge is selected if together with the previously selected edges it creates no cycle. Prove that the edges chosen by Kruskal's algorithm do form a minimal spanning tree of a connected network.
8. Explain how Prim's algorithm for minimal spanning trees differs from Kruskal's algorithm.

9. Graphs may be implemented in many ways by the use of different kinds of data structures. Postpone implementation decisions until the applications of graphs in the problem-solving and algorithm-development phases are well understood.
10. How can we determine a maximal spanning tree in a network?

Notes

Answers: Self Assessment

- | | |
|-----------------|-------------------------|
| 1. True | 2. True |
| 3. False | 4. True |
| 5. False | 6. Vojtich Jarník |
| 7. Norman Zadeh | 8. randomized algorithm |
| 9. weighted | 10. DJP algorithm |

14.8 Further Readings



Books

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Burkhard Monien, *Data Structures and Efficient Algorithms*, Thomas Ottmann, Springer.

Kruse, *Data Structure & Program Design*, Prentice Hall of India, New Delhi.

Mark Allen Weles, *Data Structure & Algorithm Analysis in C Second Ed.*, Addison-Wesley Publishing.

RG Dromey, *How to Solve it by Computer*, Cambridge University Press.

Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific.

Shi-kuo Chang, *Data Structures and Algorithms*, World Scientific.

Sorenson and Tremblay, *An Introduction to Data Structure with Algorithms*.

Thomas H. Cormen, Charles E, Leiserson & Ronald L., *Rivest: Introduction to Algorithms*. Prentice-Hall of India Pvt. Limited, New Delhi.

Timothy A. Budd, *Classic Data Structures in C++*, Addison Wesley.



Online links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

LOVELY PROFESSIONAL UNIVERSITY

Jalandhar-Delhi G.T. Road (NH-1)

Phagwara, Punjab (India)-144411

For Enquiry: +91-1824-300360

Fax.: +91-1824-506111

Email: odl@lpu.co.in

