ADVANCED DATA STRUCTURE

SYLLABUS

UNIT I

Introduction: Mathematics Review – A brief introduction to recursion. Algorithm analysis, Mathematic background - model - what to analyze – Running time calculations.

Lists, Stacks, Queues: Abstract data types (ADT) – The List ADT – The stack ADT – The Queue ADT

UNIT II

Trees: Implementation of trees, tree travels with an application – Binary trees – The search tree ADT – Binary Search Trees

Hashing: General Idea - Hash function - Separate Chaining.

Priority Queues (Heaps): Model - Simple implementations - Binary Heap.

UNIT III

Sorting: Preliminaries – Insertion sort – Shell Sort – Heap Sort – Merge Sort – Quick Sort.

UNIT IV

Graph Algorithms: Definition Topological sort shortest – Path Algorithms – Network How Problems – Minimum Spanning Tree – Applications of Depth – First Search.

UNIT V

Algorithm Design Techniques: Greedy Algorithms – Divide and Conquer – Running Time of divide and Conquer Algorithms – Closest-Points Problems – The Selection Problem – Theoretical improvements for Arithmetic Problems.

UNIT I

LESSON

1

FUNDAMENTAL OF DATA STRUCTURES

CONTENTS

1.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Discuss the concept of recursion
- Describe algorithm analysis which includes mathematic background, model, what to analyze and running time calculations

1.1 INTRODUCTION

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 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.

In this lesson, we begin with an explanation of what an algorithm and a data structure are. We introduce the concept of abstract data types. We then describe the pseudo-code, which would be used for writing the algorithms throughout. Finally, the important aspect of analysis of algorithms is briefly touched upon with an introduction to the big 'O' notation.

We will also discuss recursion in this lesson. Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. Recursion may require maintaining a stack, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme programming language standard requires implementations to recognize and optimize tail recursion.

1.2 RECURSION

Recursion is a wonderful, powerful way to solve problems. It is an important concept in computer science. Many algorithms can be best described in terms of recursion. Recursion defines a function in terms of itself. That is, in the course of the function definition there is a call to that very same function. At first this may seem like a never ending loop, or like a dog chasing its tail. It can never catch it. So too it seems our method will never finish. This might be true is some cases, but in practice we can check to see if a certain condition is true and in that case exit (return from) our method. The case in which we end our recursion is called a base case. Additionally, just as in a loop, we must change some value and incrementally advance closer to our base case.

Consider this function:

```
void myMethod( int counter)
{
if(counter == 0)
return;
else
{
System.out.println(""+counter);
myMethod(--counter);
return;
}
}
```

This recursion is not infinite, assuming the method is passed a positive integer value.

Consider this method:

```
void myMethod( int counter)
{
if(counter == 0)
return;
else
{
System.out.println("hello" + counter);
myMethod(--counter);
System.out.println(""+counter);
return;
}
}
```

The above recursion is essentially a loop like a for loop or a while loop. When do we prefer recursion to an iterative loop? We use recursion when we can see that our problem can be reduced to a simpler problem that can be solved after further reduction.

Every recursion should have the following characteristics:

- A simple base case which we have a solution for and a return value.
- A way of getting our problem closer to the base case, i.e., a way to chop out part of the problem to get a somewhat simpler problem.
- A recursive call which passes the simpler problem back into the method.

The key to thinking recursively is to see the solution to the problem as a smaller version of the same problem. The key to solving recursive programming requirements is to imagine that your method does what its name says it does even before you have actually finish writing it. You must pretend the method does its job and then use it to solve the more complex cases. Here is how.

Identify the base case(s) and what the base case(s) do. A base case is the simplest possible problem (or case) your method could be passed. Return the correct value for the base case. Your recursive method will then be comprised of an if-else statement where the base case returns one value and the non-base case(s) recursively call(s) the same method with a smaller parameter or set of data.

Thus you decompose your problem into two parts:

1. The simplest possible case which you can answer (and return for)
2. All other more complex cases which you will solve by returning the result of a second calling of your method.

This second calling of your method (recursion) will pass on the complex problem but reduced by one increment. This decomposition of the problem will actually be a complete, accurate solution for the problem for all cases other than the base case.

Thus, the code of the method actually has the solution on the first recursion.

Let's consider writing a method to find the factorial of an integer. For example 7! equals 7*6*5*4*3*2*1 . But we are also correct if we say 7! equals 7*6!. In seeing the factorial of 7 in this second way we have gained a valuable insight. We now can see our problem in terms of a simpler version of our problem and we even know how to make our problem progressively more simple. We have also defined our problem in terms of itself, i.e., we defined 7! in terms of 6!. This is the essence of recursive problem solving. Now all we have left to do is decide what the base case is. What is the simplest factorial? 1!. 1! equals 1.

Let's write the factorial function recursively.

```
int myFactorial( int integer)
{
if( integer == 1)
```

```
return 1;
else
{
return(integer*(myFactorial(integer-1);
}
}
```

Note that the base case (the factorial of 1) is solved and the return value is given. Now let us imagine that our method actually works. If it works we can use it to give the result of more complex cases. If our number is 7 we will simply return 7 * the result of factorial of 6. So we actually have the exact answer for all cases in the top level recursion. Our problem is getting smaller on each recursive call because each time we call the method we give it a smaller number. Try to run this program in your mind with the number 2. Does it give the right value? If it works for 1 then it must work for two since 2 merely returns 2 * factorial of 1. Now will it work for 3? Well, 3 must return 3 * factorial of 2. Now since we know that factorial of 2 works, factorial of 3 also works. We can prove that 4 works in the same way, and so on and so on.

However, in fact, your code won't run forever like an infinite loop, instead, you will eventually run out of stack space (memory) and get a run-time error or exception called a stack overflow. There are several significant problems with recursion.

Mostly it is hard (especially for inexperienced programmers) to think recursively, though many AI specialists claim that in reality recursion is closer to basic human thought processes than other programming methods (such as iteration). There also exists the problem of stack overflow when using some forms of recursion (head recursion.) The other main problem with recursion is that it can be slower to run than simple iteration. Then why use it? It seems that there is always an iterative solution to any problem that can be solved recursively. Is there a difference in computational complexity? No.

Is there a difference in the efficiency of execution? Yes, in fact, the recursive version is usually less efficient because of having to push and pop recursions on and off the run-time stack, so iteration is quicker. On the other hand, you might notice that the recursive versions use fewer or no local variables.

So why use recursion? The answer to our question is predominantly because it is easier to code a recursive solution once one is able to identify that solution. The recursive code is usually smaller, more concise, more elegant, possibly even easier to understand, though that depends on ones thinking style. But also, there are some problems that are very difficult to solve without recursion. Those problems that require backtracking such as searching a maze for a path to an exit or tree based operations are best solved recursively.

Tail Recursion

Tail recursion is defined as occurring when the recursive call is at the end of the recursive instruction. This is not the case with my factorial solution above. It is useful to notice when ones algorithm uses tail recursion because in such a case, the algorithm can usually be rewritten to use iteration instead. In fact, the compiler will (or at least should) convert the recursive program into an iterative one. This eliminates the potential problem of stack overflow.

This is not the case with head recursion, or when the function calls itself recursively in different places. Of course, even in these cases we could also remove recursion by using our own stack and essentially simulating how recursion would work.

In the example of factorial above, the compiler will have to call the recursive function before doing the multiplication because it has to resolve the (return) value of the function before it can complete the multiplication. So the order of execution will be "head" recursion, i.e. recursion occurs before other operations.

To convert this to tail recursion we need to get all the multiplication finished and resolved before recursively calling the function. We need to force the order of operation so that we are not waiting on multiplication before returning. If we do this the stack frame can be freed up.

The proper way to do a tail-recursive factorial is this:

```
int factorial(int number) {
if(number == 0) {
return 1;
}
factorial_i(number, 1);
}
int factorial_i(int currentNumber, int sum) {
if(currentNumber == 1) {
return sum;
} else {
return factorial_i(currentNumber - 1, sum*currentNumber);
}
Recursion
}
```

Notice that in the call return factorial_i(currentNumber - 1, sum*currentNumber); both parameters are immediately resolvable. We can compute what each parameter is without waiting for a recursive function call to return. This is not the case with the previous version of factorial. This streamlining enables the compiler to minimize stack use as explained above.

1.3 ALGORITHM ANALYSIS

Computer science is the study of methods for effectively using a computer to solve problems, or in other words, determining exactly the problem to be solved.

This process entails:

1. Understanding of the problem.
2. Translating descriptions, goals, requests and unstated desires into a precisely formulated conceptual solution.
3. Implementing the solution with a computer program.

This solution typically consists of two parts: data structures and algorithms.

An algorithm is a well-defined list of steps for solving a particular problem. A set of algorithms are always used for performing operations on the data stored by means of data structure. Thus algorithms handle data through data structure. In constructing a solution to a problem, a data structure must be chosen that allows the data to be operated upon easily in the manner required by the algorithm.

Data may be arranged and managed at many levels. Algorithm has to be designed in such a manner so that it can perform the desired operation on the stored data.

An algorithm may need to put new data into an existing collection of data, remove data from a collection, or query a collection of data for a specific purpose.

In the design of many types of programs, the choice of data structures is a primary design consideration. Experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction - data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

The formal algorithm consists of two parts. The first part is a paragraph describing the purpose of the algorithm, identifies the variable which is used in the algorithm and lists of input data. The second part of the algorithm consists of the list of steps that is to be executed.

1.3.1 Problem Solution using Pseudocode

Pseudocode is an outline of a program, written in a form that can easily be converted into real programming statements.

Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step – an important one - in producing the final code. The benefit of pseudocode is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use for the final implementation.

Problem

A shop has started a discount scheme. According to that scheme if the purchased quantity is more 10 then 10% discount DISC has been given to the customer. Operator has to enter the quantity QTY and rate RATE of the item; the program will display the total value TOT_VAL. One way to solve the problem is as follows:

Solution

First Initialize DISC with = 0 and accept the RATE and QTY from the user. Then check whether the QTY is more than 10 or not. If QTY >10 then set DISC = 10. Calculate the total value TOT_VAL and display it.

A formal Algorithm of the stated problem:

Algorithm: (Total Value Calculation)

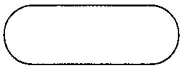
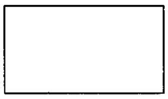
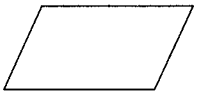
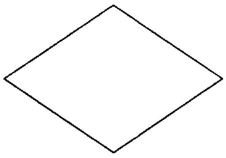
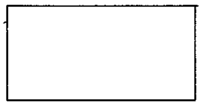
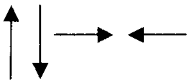
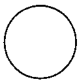
This algorithm accepts the input for QTY and RATE from the user then checks whether the QTY > 10 or not. If QTY >10 then assigns DISC = 10. After that the VAL = (QTY * RATE) - (RATE * DISC/100) is calculated and displayed.

```
Step 1. [Initialize] set DISC: =0.
Step 2. [Accept the rate and quantity of the item].
Accept and assign INPUT for QTY and RATE.
Step 3. [Check QTY > 10]
            If QTY > 10, then:
            set DISC: =10. [Allow discount]
            [End of If structure.]
Step 4. [Calculation of total value]
            Set VAL: = (QTY * RATE) - (RATE * DISC/100).
Step 5. Print VAL.
Step 6. Exit.
```

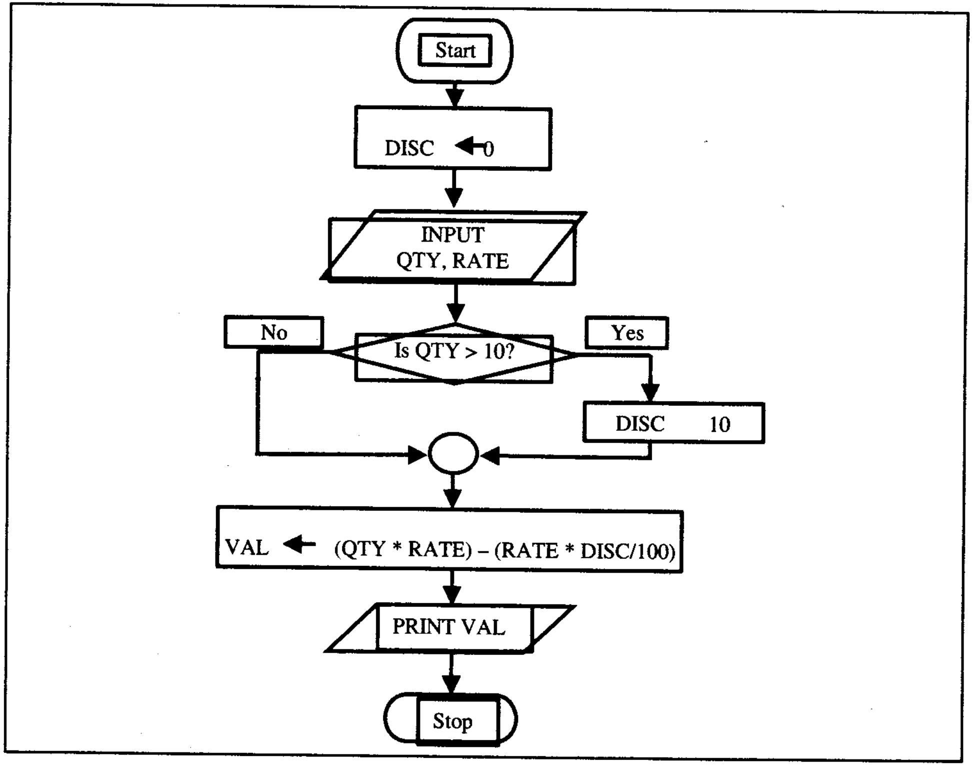
1.3.2 Problem Solution using Flow Chart Diagram

A flow chart is a graphical or symbolic representation of a process. Each step in the process flow is represented by a different symbol and contains a short text description of the process step in the flow chart symbol. The flow chart symbols are linked together with arrow connectors (also known as flow lines).

Table 1.1: Program Flow Chart Symbols

	Terminal box indicates the point at which the algorithm begins or terminates.
	Processing box indicates the straightforward computation, assignment operation.
	Input-Output box used for taking input data and giving output results or messages.
	Decision box used when algorithm has to choose between two or three branches leading to other parts of a flowchart.
	Represents a predefined module. the detailed internal steps of which are defined elsewhere. Subroutines.
	Flow lines used to connect different boxes and indicate direction of flow.
	Connector connects different parts of flow chart.

A Flow chart of the previously stated Total Value Calculation problem is given below:



1.3.3 Mathematic Background

As in C, we shall assume four basic data types - integer, float, boolean and char. Integer and float (also called real in some languages) are available in every programming language. A variable of type boolean can have either true or false as its value. A variable constant or a float constant will be written in the usual form. For example, -78 is an integer constant while -78.5 and .78e-3 are float constants. A character constant will be written by enclosing it within single quotation marks like 'a' or 'A' or '%'. A boolean constant will be written as true or false. Among the constants, we shall also allow 'strings' (a string is a sequence of characters) for example, 'xyz'.

Expressions will be made up of variables and constants connected by means of operators. We shall use the usual arithmetic operators like +, -, * and /. In addition to these, operators like mod will be used to mean remainder of integer division. Thus a mod b will mean the remainder of division of a by b. The operators * and mod will have higher precedence than + and -. Float and integer can be mixed in expressions but the result will be float. Boolean expressions can be obtained by using the relational operators like the following:

= = (equal to)

!= (not equal to)

< (less than)

> (greater than)

< = (less than or equal to)

> = (greater than or equal to)

Boolean variables or expressions can be connected with the logical operators not represented by !, and represented by &&, or represented by | |, to obtain further compound boolean expressions. For example,

(a > = 10) | | (a < = 20)

would mean a should be greater than or equal to 10 or a should be less than or equal to 20. Among the logical operators not will have a higher precedence than and which in turn will have a precedence higher than or.

As regards associativity of operators, we shall use parentheses to avoid confusion. Implicitly, left to right associativity will be assumed. Thus, a * b / c would mean (a*b)/c.

The variables used in the programs should be declared before the executable statements in a section beginning with the keyword var. The format of a variable declaration is as follows:

<data type> <list of variables>;

For example,

int x, y, z;

means that x, y and z are variables of type integer. The data type can be standard type or user-defined.

The enumerated type as available in C are assumed to be included in this pseudo-code. We illustrate this with the help of following example:

enum colour (brown, red, green);

This declaration assigns 0 to brown, 1 to red and 2 to green. The enumerated type is defined with a list of names as shown in the case of colour. These names are the values of a variable of this type we can assume. For example, the statement,

a = brown;

will assign the value of brown to a. Note that brown is a constant of type color and not a variable.

As in every programming language, the implied sequencing of statements will be assumed in the pseudo-code. This means that statements will be executed sequentially in a top-to-bottom manner unless the flow of control is explicitly altered by a control construct such as a loop construct. The statements will be separated from one another by means of a semicolon(;). A group of statements placed within begin and end will be a compound statement and will be treated as a single unit.

There will be assignment statements in the usual format namely,

<variable> = <expression>;

Thus,

a[i] = x-y;

is an example of the assignment statement.

We shall also use the usual if-then-else statement in the following format:

if <condition> then <statement block> else <statement block>

Here, <condition> is a boolean expression. Sometimes it will be expressed using English. Each of the 'then' or 'else' parts will contain one simple or compound statement. We further assume that the 'else' part may be absent. The statements in the 'then' part will be executed if the condition is true otherwise the statements in the 'else' part will be executed. In either case, control will reach the next statement in sequence after the execution of the said statements.

The following are examples of the if-then-else construct:

- if(a[j] < a[k]) {x = a[j]} else {x = a[k]};
- if (first) {x = 3; y = 5;} else {x = 7; y = 9;}

In the above example first is a boolean variable.

As for the loops, we shall use three statements – the while, the do-while and the for statements. The while statement has the following format:

while <condition> do <statement>

The statement within the while construct will be executed repeatedly so long as the condition becomes true. Once the condition is false, the loop is exited. The statements within the loop are not executed at all if the condition is false to start with. The format of the do-while statement is as follows:

do <statements> while <condition>

The statements within the do-while construct will be executed repeatedly as long as the condition remains true. The loop will be executed at least once irrespective of the condition. Note that unlike the while, the do-while allows more than one statement to be written within its scope and no curly races are necessary. The format of the for statement is shown below:

For(<initialization>;<condition>;<modification>) <statement>;

<Initialization> assigns initial values to the loop-control variables. The statement within the loop will be executed repeatedly, with different values of the control variable. Each time the loop is repeated, the values of the control variables are modified in <modification> section. The loop is exited when the <condition> becomes false.

The following are some examples of the while, do-while and for statements:

```
while(j < 100) {sum = sum + a[j]; j = j + 1;}
for(j = 1; j < 100; i = i + 1) sum = sum + a[j];
do {sum = sum + a[j];j = j + 1} while (j < 100)
for(j = 100; j > 1; j = j – 1) sum = sum + a[j];
```

To call a function (or a procedure) the following statement is used:

<function/procedure name> (<parameter list>);

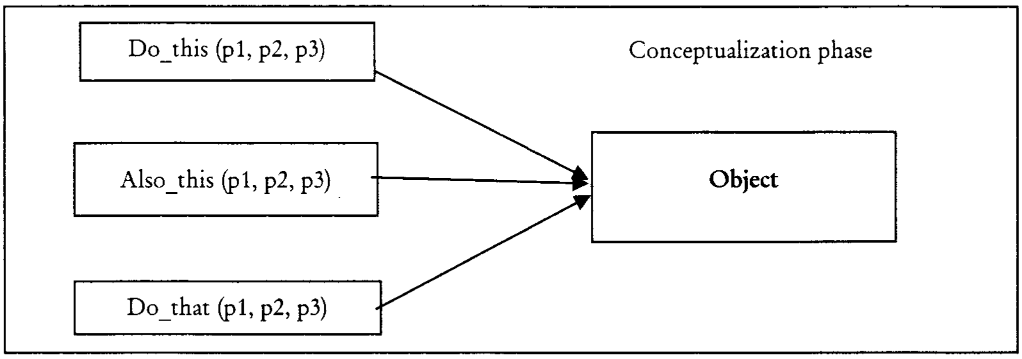
The parameter list consists of expressions that stand for the actual parameters. The expressions will be separated by commas (,). Thus the procedure statement, find(x, y, z + 3); means a call to some procedure named find and x, y, z + 3 are the actual parameters (also called actual arguments). As usual the correspondence between actual parameters and formal parameters (also known as dummy arguments) will be positional.

1.3.4 Model: Abstract Data Type (ADT)

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

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



From this we figure out:

- Object make-up (in terms of data)
- Object interface (what sort of messages it would handle)
- 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 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 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.

1.3.5 What to Analyze?

- Here correct algorithms are analyzed.
- An algorithm is said to be correct if it stops with the correct output, for each input instance.
- Incorrect algorithms
  - may not stop in any way on some input instances
  - may stop with other than the desired answer

1.3.6 Running Time Calculations

An algorithm is a method for solving a class of problems on a computer. The complexity of an algorithm depends on the running time, or storage, or whatever units are relevant for using the algorithm to solve one of those problems.

Computing takes time. Some problems take a very long time, others can be done quickly. Some problems seem to take a long time, and then someone discovers a faster way to do them (a 'faster algorithm'). The study of the amount of computational effort that is needed in order to perform certain kinds of computations is the study of computational complexity.

Naturally, we would expect that a computing problem for which millions of bits of input data are required would probably take longer than another problem that needs only a few items of input. So the time complexity of a calculation is measured by expressing the running time of the calculation as a function f(n) of some measure of the amount of data that is needed to describe the problem to the computer.

For instance, think about this statement: 'I just bought a matrix inversion program, and it can invert an n × n matrix in just $1.2n^3$ minutes.' We see here a typical description of the complexity of a certain algorithm. The running time of the program is being given as a function of the size of the input matrix.

A faster program for the same job might run in $0.8n^3$ minutes for an n × n matrix. If someone were to make a really important discovery, then maybe we could actually lower the exponent, instead of merely shaving the multiplicative constant. Thus, a program that would invert an n × n matrix in only 7n2.8 minutes would represent a striking improvement of the state of the art.

The general rule is that if the running time is at most a polynomial function of the amount of input data, then the calculation is an easy one, otherwise it's hard.

Suppose we are writing an algorithm for searching the occurrence of a particular letter from a given word. If, the letter occurs at the beginning of the word then the f(n) is small. On the other hand, if the particular letter does not appear in the given word then the f(n) is big.

Generally the complexity of an algorithm is measured by three certain cases.

- *Best Case:* The minimum value of f(n) for any possible input.
- *Average Case:* The average value of f(n) for a certain probabilistic distribution for the input.
- *Worst Case:* The maximum value of f(n) for any possible input.

Big-O Notation

It is a theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = O(g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

Formal Definition: f(n) = O(g(n)) means there are positive constants c and k, such that 0 = f(n) = cg(n) for all n = k. The values of c and k must be fixed for the function f and must not depend on n.

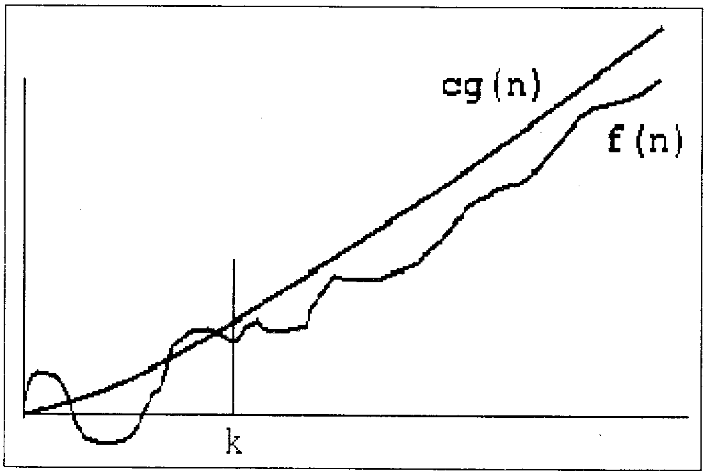


Figure 1.1: Graphical Representation Big O Notation

Performance of an algorithm (and therefore the corresponding program) is often measured in terms of the space and time required to execute it. Generally, time and space required to execute a program are inversely proportion. Accordingly, if the program is required to be executed quickly, the space it will occupy will be more and vice-versa.

A time-critical system, such as real time systems that need very small response time, would achieve its objective by allowing the program to occupy more space in the memory. On the other hand, a space-critical program would take more time to run.

However, an algorithm can almost always be developed that uses the available space and execution time balance in a given system.

Check Your Progress

1. Fill in the blanks:
   (a) Iteration (looping) in functional languages is usually accomplished via ....
   (b) Recursion defines a function in terms of ....
   (c) The case in which we end our recursion is called a .... case.
2. Define algorithm.
3. How many parts are there in a formal algorithm? Mention the parts.
4. Define pseudocode.

1.4 LET US SUM UP

Recursion is a wonderful, powerful way to solve problems. Recursive functions invoke themselves, allowing an operation to be performed over and over. Recursion may require maintaining a stack, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages.

An algorithm is a well-defined list of steps for solving a particular problem. The formal algorithm consists of two parts. The first part is a paragraph describing the purpose of the algorithm, identifies the variable which is used in the algorithm and lists of input data. The second part of the algorithm consists of the list of steps that is to be executed.

The objective of analyzing an algorithm is to obtain quantitative measures for the resources required by the algorithm during its execution. Performance of an algorithm is often measured in terms of the space and time required to execute it. Generally, time and space required to execute a program are inversely proportion. Accordingly, if the program is required to be executed quickly, the space it will occupy will be more and vice-versa.

1.5 KEYWORDS

Recursion: It is a method in which a function calls itself.

Tail Recursion: It is defined as occurring when the recursive call is at the end of the recursive instruction.

Flow Chart : Diagramatic representation of an algorithm.

Pseudocode: An outline of a program, written in a form that can easily be converted into real programming statements.

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

Algorithms: A finite set of instructions which, when followed, accomplishes a particular task, the termination of which is guaranteed under all cases.

ADT (Abstract Data Type): A mathematical model with a collection of operations defined on that model.

Space Complexity: A quantitative measures for the space required by the algorithm during its execution.

Time Complexity: A quantitative measures for the time required by the algorithm during its execution.

1.6 QUESTIONS FOR DISCUSSION

1. What is recursion? How does recursion works?
2. Describe tail recursion.
3. Can you express each of the following algebraic formulae in a recursive form:

   (a) $y = (x_1 + x_2 + ... + x_n)$

   (b) $y = 1 + 2x + 4x_2 + 8x_2 + ... + 2_n x_n$.

   (c) $y = (1 + x)_n$
4. Given $S = 1 + 2^2 + 3^2 + 4^2 + ... + n^2$, where S is the sum of the squares of n numbers, write an algorithm to compute S.
5. Write an algorithm to compute the sums for the first n terms of the following series, where n has to be input by the user :

   (a) $S = 1 + 2 + 3 + ...$

   (b) $S = 1 + 3 + 5 + ...$

   (c) $S = 2 + 4 + 6 + ...$

   (d) $S = 1 + 1/2 + 1/3 + ...$
6. Given a number n, write an algorithm to compute the factorial of n.
7. Write short note on the following:

   (a) ADT

   (b) Mathematical notations

Check Your Progress: Model Answers

1. (a) Recursion

   (b) Itself

   (c) Base
2. An algorithm is a well-defined list of steps for solving a particular problem.
3. The formal algorithm consists of two parts. The first part is a paragraph describing the purpose of the algorithm, identifies the variable which is used in the algorithm and lists of input data. The second part of the algorithm consists of the list of steps that is to be executed.
4. Pseudocode is an outline of a program, written in a form that can easily be converted into real programming statements.

1.7 SUGGESTED READINGS

John R. Hubbard, Schaum's Outline of Data Structures with Java, 2nd Edition, McGraw Hill.

William Collins, Data Structures and the Java Collections Framework, 2nd Edition, McGraw Hill.

Mastering Algorithms with C - by Kyle Loudon - Published by O'Reilly & Associates.

The Art of Computer Programming - by Donald E. Knuth - Published by Addison-Wesley Professional.

Introduction to Algorithms - by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - Published by The MIT Press.

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

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

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

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

LESSON

2

LISTS, STACKS AND QUEUES

CONTENTS

2.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Describe the application of ADT in linked lists
- Explain the application of ADT in stacks
- Identify the application of ADT in queues

2.1 INTRODUCTION

Computers help us in solving many real life problems. We know that computers can work with greater speed and accuracy than human beings. What actually does a computer do? A very simple answer to this question is that a computer stores data and reproduces it as information as and when required. Representation of data should be in a proper format so that accurate information can be produced at high speed. In this lesson, we will study the various ways in which data can be represented. Efficient storage and retrieval of data is important in computing. In this lesson, we will study and implement various data structures.

Organized data is known as information. Let us consider a few examples and try to understand what data structures are. You all must have seen a pile of plates in a restaurant. Whenever a plate is required, the plate on the top is removed. Similarly, if a plate is added to the pile, it is kept on the top of the pile. There is a definite process involved in the storage and retrieval of the plates. If the rack is empty there will be no plates available. Similarly, if the rack is full then there will be no place for more plates. Similar process is applied with stacks. Stack is a data structure which stores data at the top, this operation is known as push. It retrieves data from the top, the operation is known as pop. If the stack is empty then the pop operation raises an error while push operation cannot be performed if the stack is full. Stack is shown in Figure 2.1.

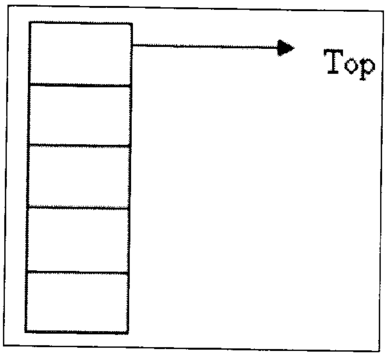


Figure 2.1: Stack (Pile of Plates)

You must have seen a queue at the bus stop. There is a well-defined method for a person to enter the queue as well as leave the queue to get into the bus. To enter a queue a person stands at the end of the queue and the person at the start of the queue leaves the queue to enter the bus. Similarly, we have queue as a data structure in which a data element is added at the rear end and removed from the front end. Figure 2.2 shows the structure of queue.

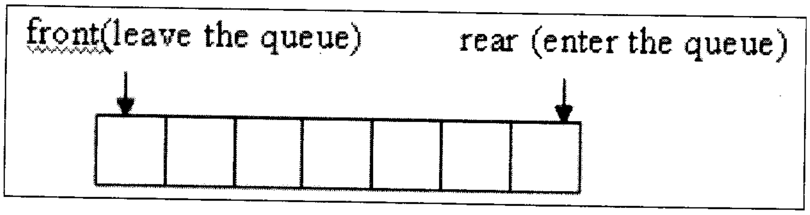


Figure 2.2: Queues

In a train, separate compartments are joined together. A new compartment can be added at the end, at the beginning or in between the existing train compartments. The same method is followed to remove a compartment. Linked lists in data structures follow similar approach. An element can be inserted and deleted from any position in the linked list. A list can be shown as in Figure 2.3.

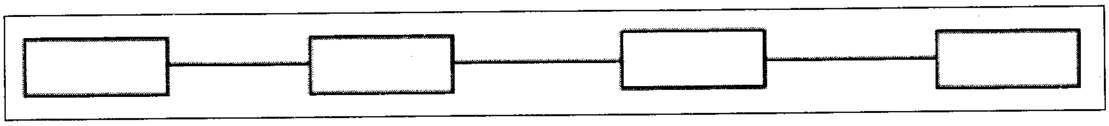


Figure 2.3: Linked List (Train Compartments)

There are other data structures that we will study in this course, for example graphs, trees etc. As we have seen in the above examples data structures stores a given set of data according to certain rules. These rules help in organizing the data.

You have studied programming languages before. This example will use some concepts from it to explain why data structures are essential. Data can be stored in various ways. Usually the representation chosen depends on the nature of the problem and not on the data. Consider a program, which requires storing marks of five students for a single subject. The simplest way to store them will be to use five integer variables. We assume that the roll numbers are from one to five. Now I wish to write a program, which can give me the marks of any roll number given as input. Is it possible to write an efficient program to do this task if the numbers are stored as variables?

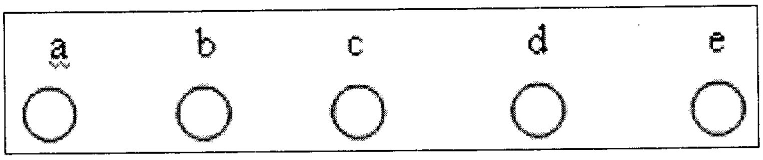


Figure 2.4: Marks Stored in five different Variables

The data, in this case the marks, is stored, but it cannot be reproduced as information efficiently. Now we take an array of integers with five elements. The above data is stored in the array. An array will have a name 'a' and each individual element can be accessed using index. Therefore, the marks for the first roll number will be stored as a [0], for the second roll number as a [1] and so on.

There is a relation between the roll number and the array index. Now it is much easier to access the marks according to the roll number. This could not be achieved using variables, as there was no relation between the data. It was not possible to relate marks and roll number.

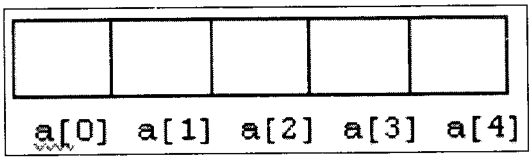


Figure 2.5: Marks Stored as an Array

The above example can be slightly modified so that now we will store the marks of three subjects per roll number. We can take three independent arrays to store them. We can access the individual marks for each roll number from the respective arrays. We find that the arrays contain marks of different subjects for the same roll number. It would be easier to handle the marks if these three arrays were grouped together.

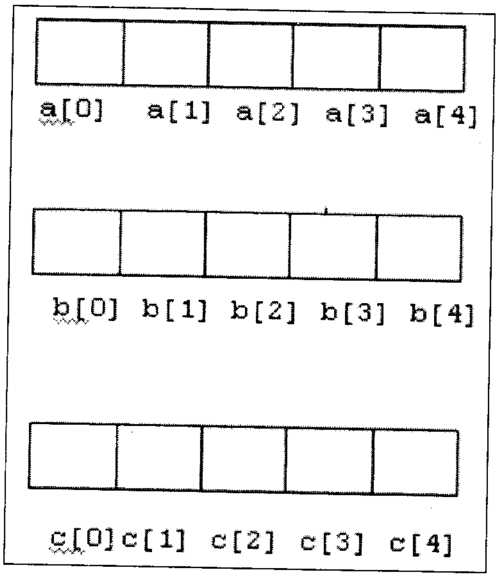


Figure 2.6: Three Arrays to Store the Marks of three Subjects

Now, we need a data structure that stores the above information in logical relation with each other so that it can be retrieved with higher speed. Structures in C/C++ store various fields together, so now we can store the marks of three subjects in the structure and then make an array of five elements. The above example shows that the method in which data should be represented depends on the problem to be solved. The structure definition will be as follows:

```
struct record
{
int a[5];
int b[5];
int c[5];
};
```

In the above problem, we have used three different types of representations for the data. They are all built using the basic data structures. There are certain rules associated with each data structure that restrict the storage and retrieving of data in an ordered format. We have seen examples like stacks and queues, which restrict the entry and exit of its elements according to certain rules. Stacks allow entry and exit of elements at the top position only, while in queues an element is added at the end and removed from the front. Arrays and structures are built in data structures in most of the programming languages. Data can be stored in many other ways using other type of data structures. Thus we can define data structure as a collection of data elements whose organization is characterized by accessing functions that are used to store and retrieve individual data elements. Data structures are represented at the logical level using a tool called Abstract Data Type and actually implemented using algorithms.

Abstract Data Type's show how exactly the data structure behaves. What are the various operations required for the data structure? It can be called as the design of the data structure. Using the Abstract Data Type, it can be implemented using various algorithms. Abstract Data Type clearly states the nature of the data to be stored and the rules of the various operations to be performed on the data. For example, the Abstract Data Type of a stack will clearly state that a new element will be inserted at the top and an element can be removed only from the top of the stack. It does not specify how these rules should be implemented. It specifies what are the requirements of a stack.

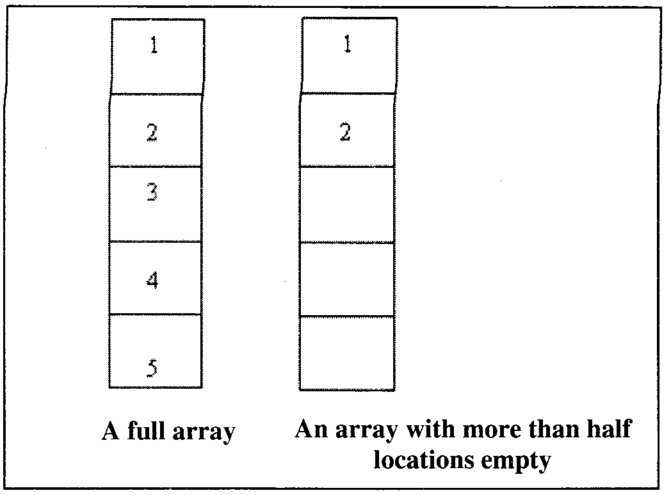
The implementation of a data structure is done with the help of algorithms. An algorithm is a logical sequence of discrete steps that describe a complete solution to a given problem in a finite amount of time. A task can be carried out in various ways, hence there are more than one algorithm which can be used to implement a given data structure. This means that we will have to analyze algorithms to select the best suited for the application.

2.2 SINGLY LINKED LIST

An array is an example of list. Arrays have fixed size, which is declared and fixed at the start of the program, and therefore can not be changed while it is running. For example, suppose an array of size 5 has been declared at the start of the program.

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.



A full array

An array with more than half locations empty

Figure 2.7

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.

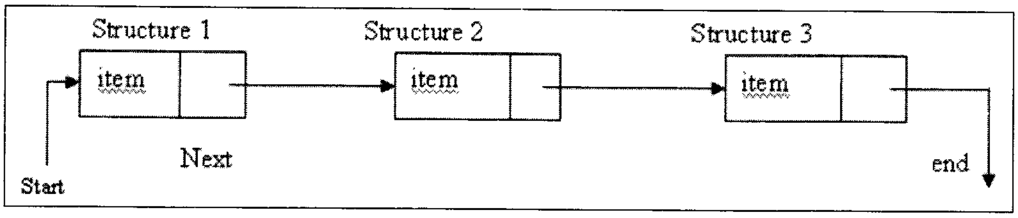


Figure 2.8: A Linked List

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.

2.2.1 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

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. ***Crlist:***

   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.

   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.

2.2.2 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.

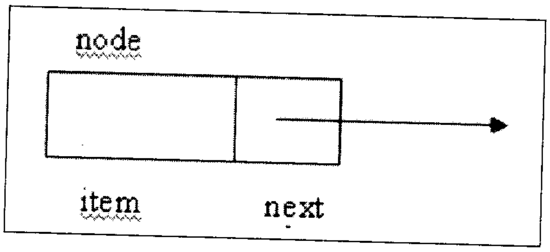
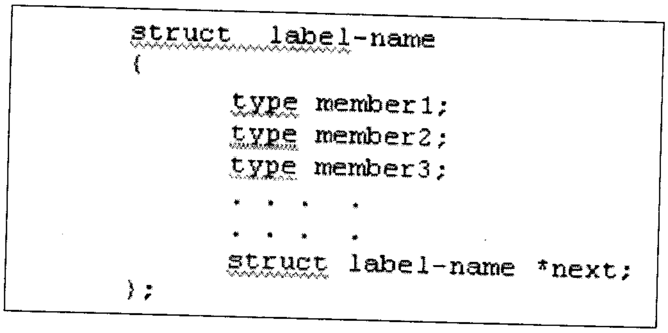


Figure 2.9: Node

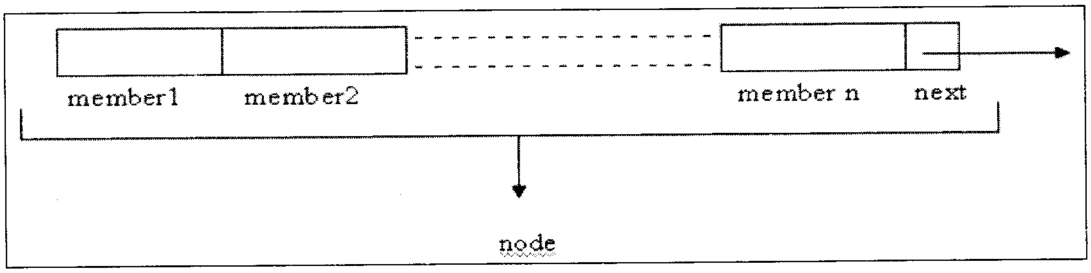
Such structures, which contain a member field that points to the same structure type, are called self-referential structures. A node may be represented in general form as follows:



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

Figure 2.10: Structure Declaration for the Node

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;

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

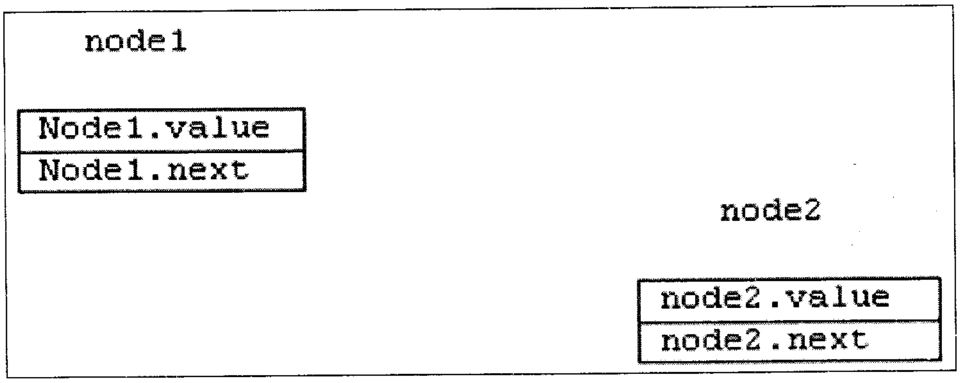


Figure 2.11: Creation of the two Nodes

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:

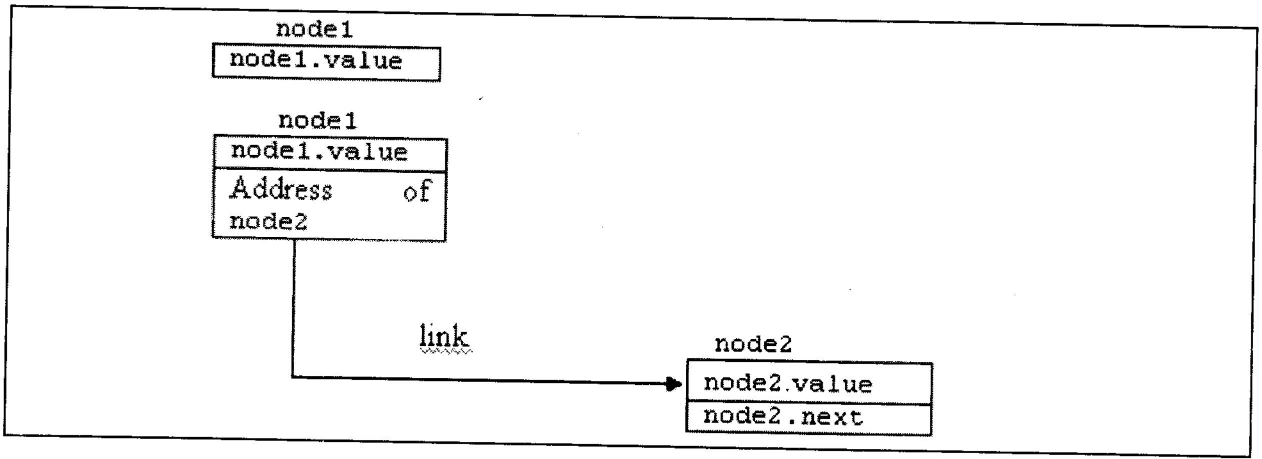


Figure 2.12: Node 1 Pointing to Node 2

Now we can assign values to the field value.

node1.value=30;

node2.value=40;

The result is as follows:

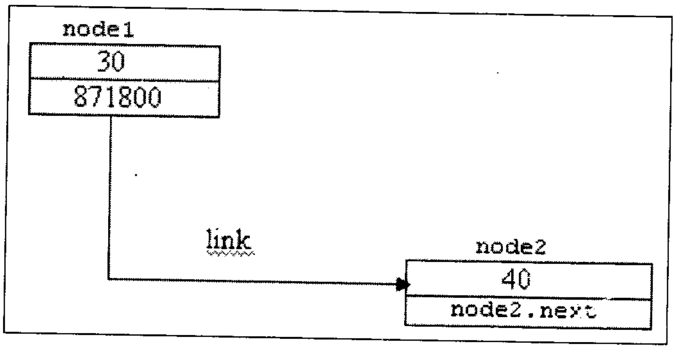


Figure 2.13: The Complete List of two Nodes

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;

2.3 APPLICATION: POLYNOMIAL ADDITION

We will now try to add two polynomials using linked lists. In mathematics a polynomial is written as follows:

$P\ (x) = a_1x^n + a_2x^{(n-1)} + \ldots + a_n$

Each term will be represented as a node. The node will be of fixed size having 3 fields, which represent the coefficient and exponent of a term plus the pointer to the next term.

Assuming that all coefficients are integers, the required declarations in C are as follows:

```
STRUCT LINK_LIST

{

INT COEF;

INT EXPO;

STRUCT LINK_LIST *NEXT;

};
```

Polynomial nodes will be drawn as below:

For example, the polynomial a = $3x^{14} + 2x^{10} + 3x^4$ will be stored as:

In order to add two polynomials together, we examine their starting terms. Two pointers are used to move along the two polynomials. If the exponents of two terms are equal, then the coefficients are added and a new term created for result. If the exponents are not equal, then the term with bigger exponent is attached to the resultant polynomial.

Following is the program and algorithm for the above problem:

/*this program adds two polynomials using linked lists*/

Algorithm

Step 1: Start.

Step 2: Ask the user to enter the first polynomial.

Step 3: Ask the user to enter the value of coefficient and exponent, or '0,0' terminate the polynomial.

Step 4: Ask the user to enter the second polynomial.

Step 5: Repeat step 2

Step 6: If the user enters the value greater than 0, then match the values of different exponents entered by the user and then perform addition of the coefficients with similar exponents.

Step 7: Display the results after the polynomial addition.

Step 8: End.

```
#include <stdlib.h>

#include <conio.h>

#include <iostream.h>
```

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAX 10
int array[MAX];
class polyadd
{
protected:
struct link_list
{
int coef;
int expo;
struct link_list *next;
};typedef struct link_list node;
public:
void crpoly(node *list);
void padd(node *list1, node *list2, node *list3);
void print(node *list);
};
void polyadd::crpoly(node *list) //create the polynomial
{
int x, y;
cout<<endl<<"Input a pair of number as 'coef, expo' :";
cout<<endl<<"input '0,0' to stop entering";
cout<<endl<<"input the coefficient ";
cin >>x;
cout<<endl<<"input the exponent ";
cin >>y;
if(x==0 && y == 0)
{
list->next = NULL;
}
else
if(x != 0 || y == 0)
{
list->next = new node();
```

```
list->next->coef = x;
list->next->expo = y;
crpoly(list->next);
}
return;
}
void polyadd::padd(node *list1, node *list2, node *list3)
{ //add two polynomials
if(list1 != NULL || list2 != NULL)
{
if(list1->expo == list2->expo)
{
list3->next = new node();
list3->next->next = NULL;
list3->next->coef = list1->coef + list2->coef;
list3->next->expo = list1->expo;
padd(list1->next, list2->next, list3->next);
}
else if(list1->expo>list2->expo)
{
list3->next= new node();
list3->next->next = NULL;
list3->next->coef = list1->coef;
list3->next->expo = list1->expo;
padd(list1->next, list2, list3->next);
}
else if(list1->expo< list2->expo)
{
list3->next = new node();
list3->next->next = NULL;
list3->next->coef = list2->coef;
list3->next->expo = list2->expo;
padd(list1, list2->next, list3->next);
}
else if(list1 == NULL && list2!= NULL)
{
```

```
list3->next = new node();
list3->next->next = NULL;
list3->next->coef = list2->coef;
list3->next->expo = list2->expo;
padd(list1, list2->next, list3->next);
}
else if(list2 == NULL && list1!= NULL)
{
list3->next = new node();
list3->next->next = NULL;
list3->next->coef = list1->coef;
list3->next->expo = list1->expo;
padd(list1->next, list2, list3->next);
}
}
return;
}
void polyadd::print(node *list)
{
if(list->next != NULL)
{
cout<<endl<<"Coefficient is "<<list->next->coef;
cout<<endl<<"Exponent is "<<list->next->expo;
print(list->next);
}
return;
}
void main()
{
clrscr();
polyadd *pa = new polyadd();
node * head1;
node * head2;
node * head3;
head1 = new node();
head2 = new node();
```

```
head3 = new node();
cout<<″Enter first polynomial″;
pa->crpoly(head1);
cout<<″Printing first polynomial″;
pa->print(head1);
cout<<″Enter second polynomial″;
pa->crpoly(head2);
cout<<″Printing second polynomial″;
pa->print(head2);
pa->padd(head1->next, head2->next, head3);
cout<<endl<<"printing after addition";
pa->print(head3);
cout<<endl;
}
```

Sample Output:

```
Enter first polynomial
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 2
input the exponent 2
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 1
input the exponent 1
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 0
input the exponent 0
Printing first polynomial
Coefficient is 2
Exponent is 2
Coefficient is 1
Exponent is 1
Enter second polynomial
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 3
input the exponent 3
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 1
input the exponent 1
input a pair of numbers as 'coef,expo':
input '0,0' to stop entering,
input the coefficient 0
input the exponent 0
Printing after addition
Coefficient is 3
Exponent is 3
Coefficient is 2
Exponent is 2
Coefficient is 2
Exponent is 1
```

2.4 ADT OF STACKS

A stack is a dynamic structure i.e. it changes as elements are added to and removed from it. The operation that adds an element to the top of a stack is usually called PUSH and the operation that takes the top element off the stack is called POP. When we start using the stack for the first time, it must be empty, so we need to create an empty stack. We must also have an operation to check whether a stack has something in it to pop or not. For particular implementations one may need to check whether a stack is full or not. ADT of such operations are given as follows:

ADT Specification for Stacks

Value definition: A stack can contain anything of the type its implementing data structure is defined, i.e. integers, characters, complex records etc.

Definition clause: A stack as explained is a list of elements in which the item added most recently is taken out first, i.e. the Last item In is the First one Out. Therefore, a stack can be defined as a LIFO list of elements.

Operations:

1. ***Create:***

   Function: initializes stack to an empty state.

   Precondition: none.

   Postcondition: stack is created and is empty.

2. ***Push:***

   Function: adds new element to the top of the stack.

   Precondition: stack is created and is not full.

   Postcondition: original stack plus new element added on top.

3. ***Pop:***

   Function: removes top element from the stack.

   Precondition: stack is created and is not empty.

   Postcondition: original stack with top element removed.

4. ***Empty:***

   Function: tests whether stack is empty.

   Precondition: stack is created.

   Postcondition: answer as yes or no depending on the status of the stack; no change in stack contents.

5. ***Full:***

   Function: tests whether stack is full.

   Precondition: stack is created.

   Postcondition: answer as yes or no depending on the status of the stack; no change in stack contents.

6. ***Destroy:***

   Function: removes all elements from stack, leaving the stack empty.

   Precondition: stack is created.

   Postcondition: stack is empty.

These are the ADT specifications for some of the operations, which are commonly performed on stacks. Reader can think of more such operations depending on a particular situation and requirement and can write ADT for them on the same lines.

2.5 IMPLEMENTATION

A stack can be implemented using both static and dynamic implementations, i.e. the space can be allocated at compile time itself or at the execution time of the program. Each implementation has its own advantages and disadvantages, which we will consider later.

2.5.1 Implementing a Stack using an Array

Since all elements of the stack are of the same type, an array can be used to contain them. We can store elements in sequential slots in the array, placing the first element pushed in the first array position, the second element in the second array position and so on.

We also need to know how to find the top element when we want to pop and where to put the new element when we push. Although, we can access any element of an array directly, we have to confirm to the stack restriction of "LIFO ". So, we will access the stack elements only through the top, not through the bottom or through the middle.

Therefore, even though the representation of the stack may be a random-access structure such as an array, the stack itself as a logical entity is not randomly accessed. We can use its top element only. In the array representation, a variable 'top' keeps track of the top position of the stack. An empty stack is signaled by stack top being zero and a full stack by top greater than the last storage location.

In the following programs, stack is an array of size 'MAX' and top is a variable, which keeps track of top position of the stack. Before writing the actual code, let us try and write an algorithm for it.

Algorithm

Step 1: Start.

Step 2: Declare a stack of fixed size.

Step 3: Insert elements into the stack and display the status of the stack after each insertion.

Step 4: Insert values into the stack until the stack is full.

Step 5: Start the pop operation and display the status of the stack after each pop operation.

Step 6: Continue wit h the pop operation until the stack is empty.

Step 7: End.

Program

To implement a stack using arrays.

```
#include<iostream.h>
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>
class IntStack
{
protected:
int count;
public:
IntStack(int num)
{
top = 0;
maxelem = num;
s = new int[maxelem];
count =0;
}
int push(int t)
```

```
{
if (top == maxelem)
return maxelem;
s[top++] = t;
count++;
return count;
}
int pop()
{
if (top < 0)
{
return (-1);
}
top = top-1;
cout<<"top elelmnt is " <<s[top];
return (s[top]);
}
void display_pop()
{
if (top < 0)
{
cout << "(empty)\n";
return;
}
for (int t=top;t>=0;t--)
cout << endl<<s[t] << " ";
cout << "\n";
}
void display_push()
{
if (top < 0)
{
cout << "(empty)\n";
return;
}
for (int t=0;t<top;t++)
cout << s[t] << " ";
cout << "\n";
}
int empty()
{
return top == 0;
}
private:
int *s;
int top;
int maxelem;
};
void main()
{
IntStack *s = new IntStack(100);
int d;
int count;
clrscr();
count = s->push(1);
s->display_push();
count = s->push(2);
s->display_push();
count = s->push(3);
s->display_push();
count = s->push(4);
s->display_push();
for(int i=0;i<count;i++)
{
s->pop();
s->display_pop();
}
getch();
}
```

2.5.2 Implementing Stacks using Linked Lists

As we know, the linked lists use the concept of dynamic memory allocation, i.e. memory is allocated as and when needed. Unlike the array version, this implementation has no program imposed maximum stack size. Nodes are allocated dynamically, using the new operator, as and when needed. If the call to new fails, then it returns NULL and it is assumed that memory is full. In case of successful memory allocation, the new structure is logically added. An empty stack is represented by stack top (stack pointer) pointing to NULL. Before writing the actual code, let us write an algorithm for the program.

Algorithm

Step 1: Start.

Step 2: Declare the structure of the linked list.

Step 3: Insert elements through the top of the linked list and increment the top position by 1 after each insertion.

Step 4: Insert the elements into the list until it is full.

Step 5: Perform the pop operation by decrementing the top position by 1 after each pop.

Step 6: Display the list after every pop operation until the list is empty.

Step 7: End.

Program

To implementing stacks using linked lists.

```
/* program to show push and pop operations using linked lists*/
//Implementation of stack using Linked Lists
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
struct node
{
int data;
node *link;
};
class stack
{
private:
node *top;
public:
stack()
{
top=NULL;
}
void push(int item);
int pop();
~stack()
{
if(top==NULL)
return;
node *temp;
while(top!=NULL)
{
temp=top;
top=top->link;
delete temp;
}
}
};
void stack:: push (int item)
{
node *temp;
temp=new node;
if(temp==NULL)
cout<<endl<<"stack is Full";
temp->data=item;
temp->link=top;
top=temp;
}
int stack:: pop()
{
if(top==NULL)
{
cout<<endl<<"Stack is Empty";
return NULL;
}
node *temp;
int item;
temp=top;
```

```
item=temp->data;
top=top->link;
delete temp;
return item;
}
void main()
{
clrscr();
stack s;
s.push(11);
s.push(12);
s.push(13);
s.push(14);
s.push(15);
int i=s.pop();
cout<<endl;
cout<<"Item popped=" <<i<<endl;
i=s.pop();
cout<<"Item Popped="<<i<<endl;
i=s.pop();
cout<<"Item Popped="<<i<<endl;
i=s.pop();
cout<<"Item Popped="<<i<<endl;
i=s.pop();
cout<<"Item Popped="<<i<<endl;
i=s.pop();
cout<<"Item Popped="<<i<<endl;
getchr();
}
```

2.6 ANALYSIS OF STACK IMPLEMENTATIONS

As stated in earlier lesson, an array variable of MAX stack size will take the same amount of memory, no matter how many array slots are actually used. Therefore, we need to reserve space for the maximum possible. If more elements need to be stored, then we may fall short of memory. Also, if elements stored are very less, we may have a lot of unused space. On the other hand, the linked implementation using dynamically allocated storage space only requires space for the number of elements actually present in the stack at run time. But, the elements are larger since we must store the link (the next field) along with the user's data.

Apart from the space requirement, the two implementations can be compared on other criteria also. For example, programming efforts and program complexity. We can compare the efficiency of the two representations with respect to each other in terms of Big Oh notation.

1. ***Create operation:*** In both implementations, operations like creating a stack have measure O(1). In an array implementation, only the stack has to be declared using arrays. Also, in the linked implementation, only the memory for the first node has to be allocated. Therefore, a constant amount of work is involved at all times.
2. ***Empty or full operation:*** In both implementations, operations like checking whether a stack is full or empty have Big Oh measure O(1) because the algorithm has to check for the presence of just one element.
3. ***Push or pop operation:*** In push or pop operations, the number of elements in the stack do not affect the amount of work done by these operations. Because, in both operations we directly access the top of the stack, i.e. only one element. Therefore, push and pop have measure O(1).
4. ***Destroy operation:*** Probably, this is the only operation amongst the basic ones, which differs from one implementation to other. In the array version, we just have to set the top field to zero, so it is an O(1) operation. But, in the linked version, we must process every node in the stack to free the node space.

Therefore, the operation is O(n) where n is the number of nodes in the stack.

In all, the two implementations are almost equivalent in terms of amount of work they do. Since the destroying operation is not widely used, the difference is not significant. The Table 2.1 summarizes the Big Oh measures of various operations on the two implementations.

Table 2.1: Big Oh Measure of Common Stack Operations

Operations	Array Implementation	Linked Implementation
Create	O(1)	O(1)
Empty	O(1)	O(1)
Full	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Destroy	O(1)	O(n)

The decision to use one of the two implementations depends on the situation.

Linked implementation is more flexible and is preferable where the number of stack elements vary greatly. It wastes less space when the stack is small. When stack size is unpredictable, linked implementation is preferable. Array implementation is short, simple and efficient. When we are sure that we will not need to exceed the declared stack size, the array implementation is a good choice.

For example, if a customer database for a bank is to be maintained, then the linked implementation would be preferable because the number of customers is expected to vary greatly. On the contrary, if a fixed list of students of a class is to be maintained, array implementation would be more convenient.

Examples: A stack is an appropriate data structure when information must be saved and then later retrieved in reverse order. Any situation requiring to store a previous step and coming back to it in future may be a good one to use a stack. Following are some examples using stack as their data structure.

Reversing an Input Text Line

Brief

As a simple example of using stacks, let us try to make a function that will read a line of input and will then write it out backward. We can accomplish this task by pushing each character onto the stack as it is read. When we come to end of the input, we will pop characters off the stack and they will come off **in the reverse order.**

Program

To reverse an input text line.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 10 /*defining stack size*/
main() /*main starts here*/
{
int top=0;
char stack[MAX],c; /*declaring a character stack*/
clrscr();
cout<<"\n enter the sequence of characters:";
while((c=getchar())!='\n') /*accepting the text*/
{
stack[top]=c;
top++;
}
cout<<"\n the reversed string is:"; //printing the text in
//reverse order//
while(top!=0)
{
cout<<stack[top-1];
top--;
}
}
/*main ends here*/
```

Validating an Expression by Parenthesis Matching

Brief

Consider a mathematical expression that includes several sets of nested parentheses, for e.g.

(a-((b+c(d))))

We want to make sure that the parentheses are nested correctly, i.e.

1. We want to check that there are equal numbers of right and left parentheses.
2. Every right parenthesis is preceded by a matching left parenthesis.

Expressions such as

((a+b)

violate condition 1, and expressions such as-

)a+b(-c

violate condition 2.

Before writing the actual code for the program let us write an algorithm for it and try to understand the logic.

Algorithm

Step 1: Start.

Step 2: Declare a character array to store opening braces.

Step 3: Start accepting the expression.

Step 4: If the character is an opening brace, e.g. ‘(‘ or ‘{‘ or’[‘, push it onto the stack. If successive opening braces, keep pushing them on the stack.

Step 5: If the character is a closing brace, e.g. ‘)’ or’}’ or’]’,

1. Check if the stack is empty.
2. If the stack is empty, it means there was no corresponding opening brace for the closing brace. Therefore, the expression is invalid.
3. If the stack is not empty, pop an element from the stack.
4. If the popped opening brace corresponds to the closing brace then the expression is valid.
5. Else the expression is invalid.

Step 6: When we come to the end of accepting the expression,

1. If there are still some opening braces left in the stack, it means that the expression is invalid.
2. If the stack is empty, expression is valid.

Step 7: End.

/*Program: To validate an expression by matching left and right parenthesis*/

```
//A PROGRAM TO TEST FOR THE MATCHING PARENTHESIS IN AN //EXPRESSION.
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
```

```
#define MAX 10
class stk
{
public:
char stack[MAX];
char c,ele;
int top;
stk()
{
top=0;
}
int push(char c)
{
stack[top]=c;
return ++top;
}
char pop()
{
ele=stack[top-1];
stack[top-1]=0;
top--;
return ele;
}
};
void main()
{
clrscr();
stk s;
int flag=1,top;
int tFlag =0;
char ret;
char c;
system("Clear");
cout<<"Enter the Expression";
while((c=getchar())!='\n')
{
```

```
if(c=='(' || c=='{' ||c=='[')
{
top = s.push(c);
tFlag = 1;
}
else
if(c==')'|| c==']' ||c=='}')
{
if(top==0)
{
flag=0;
break;
}
else
{
ret = s.pop();
if((ret=='('&&c==')')||(ret=='{'&&c=='}')||(ret=='['&& c==']'))
{
tFlag =0;
}
else
{
flag=0;
break;
}
}
}
}
if(s.stack[0]!=0)
{
flag=0;
}
if(flag==1){
cout<<"Expression is valid";}
else if(flag==0){
cout<<"Expression is invalid.";}
```

```
getche();
}
```

We have tried to keep the above program as simple as possible, because our aim was to illustrate the concept of stacks and not of making an extensive expression evaluator. This is why we have made some assumptions, which are as follows:

Any expression is to be contained in brackets, for e.g. the following expression will not work with the above program:

a+(b+c) – d

Rather, it has to be in the following format:

(a+(b+c) – d)

The reason for using a stack in this problem should be clear. The last parenthesis to be opened should be the first one to be closed. This is simulated by a stack in which the last element to arrive is the first to leave. Each element on the stack represents a parenthesis that has been opened but has not yet been closed. Pushing an item onto the stack corresponds to an opening brace and popping an item from the stack corresponds to closing a parenthesis.

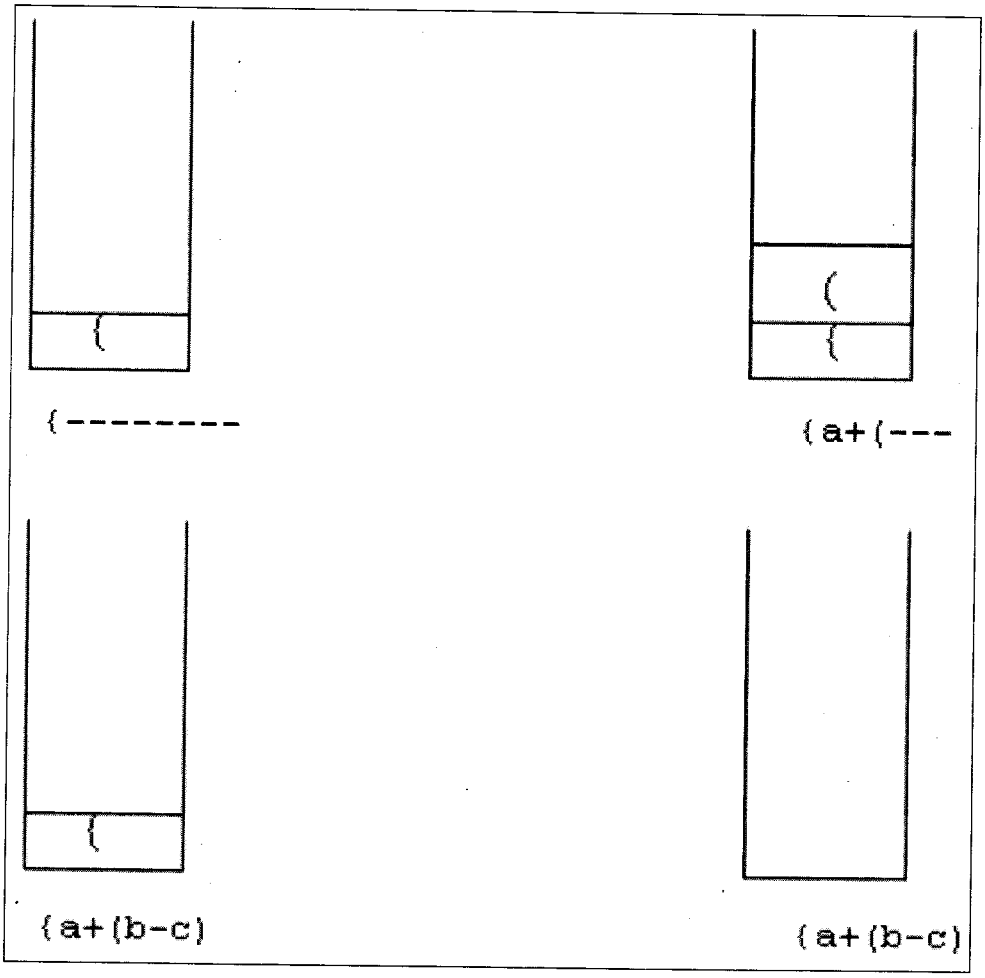


Figure 2.14

The Figure 2.14 depicts the contents of the stack at various stages of processing the for expression:

{a+(b-c)}

Postfix Expression Evaluation

Brief

The sum of 2 and 3 is represented normally as 2+3. This is called infix notation. The same sum can be represented as +23, which is called prefix notation, and 23+, which is called postfix notation.

The prefixes "in", "pre" and "post" refer to the relative position of the operator with respect to the two operands. In prefix notation, operator is before the two operands. In infix notation, operator is in between the two operands. In postfix notation, operator comes immediately after the two operands. Reader should gather more information on various notations in relevant literature.

Postfix notation has some obvious advantages over the most commonly used infix notation.

1. Need for parenthesis is eliminated.
2. Knowledge of operator precedence is not required.

We can try to develop a program for evaluating a postfix expression. Each operator encountered refers to the previous two operands. Each time we come across an operand we push it on to a stack. When we reach an operator, its operands will be the two top elements on the stack. We can pop these two elements, perform the operation on them and push back the result on to the stack.

It is then available for use with the next operator. The following program evaluates a postfix expression using this method. But let us write an algorithm first.

Algorithm

Step 1: Start.

Step 2: Declare the structure of the stack.

Step 3: Accept an expression from the user (ex: 1+2 or 2*2 etc.).

Step 4: Check if the operator is:

1. '+', then perform addition between the values and store as result.
2. '-', then perform subtraction between the values and save as result.
3. '*', then perform multiplication between the values and store as result.
4. '/', then perform division between the values and store as result.

Step 5: Push the result obtained from the expression and store it on the top position of the stack.

Step 6: Display the result on the screen.

Step 7: End.

Sample Output.

Enter the expression: 49*5+

The result is 41.

Stack for Sub-programs

This is one of the most important applications of stacks. What happens within the computer when sub-programs are called? The system (or the program) must remember the place where the call was made, so that it can return there after the sub-program is complete. It must also remember all the local variables, CPU registers, and other data, so that information is not lost while the sub-program is working. We can think of all this information as one large structure, a temporary storage area for each sub-program.

Suppose that we have 3 sub-programs called A, B, and C, and suppose that A invokes B and B invokes C. Then B has not completed its work until C has finished and returned. Similarly, A is the first to start work, but it is the last to be finished, not until after B has finished and returned. Thus the sequence in which this activity proceeds is summed up as the property last in, first out. If we consider the machine's task of assigning temporary storage areas for use by sub-programs, then those areas would be allocated in a list with this same property, that is, in a stack.

The example is represented in the Figure 2.15.

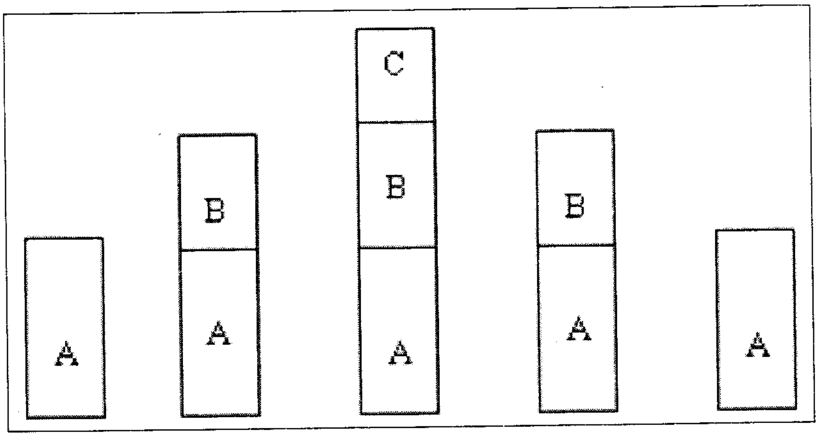


Figure 2.15: Stack for Sub-programs at Various Stages

2.7 ADT OF QUEUES

A queue is an ordered collection of items from which items can be deleted from one end (called the front of the queue) and into which items can be inserted at one end only (called the rear of the queue). As opposed to stacks, queues are FIFO lists. The element that was the First to be In, will be the First to be Out. Put in other words, the element that has spent the longest time in the queue, will be the first to be taken out. This is opposite to a stack, in which we know that the element that has spent the least time, is the first to be out, i.e. LIFO.

There are various examples of queues in the real world. A line at a railway counter or a bus stop is familiar examples of queues. The person first in the queue will be the first to get the ticket. Similarly, any new passenger will have to stand at the back of the queue.

To add elements to a queue we access the rear of the queue. To remove elements we access the front. The middle elements are logically inaccessible, even if we physically store the queue elements in a random access structure such as an array.

The essential property of the queue is its FIFO access.

As it is clear by now, there are two operations that can be applied to a queue. First, new elements are added to the rear of the queue. We will call this operation enterq. We can also take the elements off the front of the queue. We will call this operation exitq.

We are also required to check whether the queue contains anything or is empty.

Theoretically, we can always enter in a queue, for in principle, a queue is not limited in size. But certain implementation, for e.g. an array implementation, requires us to check whether the data structure is full, before entering a new element. Therefore, we can also have an operation for this purpose. Before doing anything, we also need to create a queue and initialize it to an empty state. Also, we might want to delete all elements of the queue, leaving an empty structure.

Following is the ADT representation of some of the common operations that can be performed on queues.

- ***Value definition:*** A queue can contain anything of the type its implementing data structure is defined, i.e. integers, characters, complex records, etc.
- ***Definition clause:*** A queue as explained is a list of elements in which the item added first is taken out first, i.e. the First item In is the First one Out. Therefore, a queue can be defined as a FIFO list of elements.

Operations:

1. ***Create:***

   Function: initializes queue to empty state.

   Precondition: none.

   Postcondition: A queue is created and is empty.

2. ***Enterq:***

   Function: adds new element at the rear of the queue.

   Precondition: queue is created and is not full.

   Postcondition: original queue plus new element added at the rear.

3. ***Exit:***

   Function: removes front element of the queue.

   Precondition: queue is created and is not empty.

   Postcondition: original queue minus the front element.

4. ***Empty:***

   Function: checks whether the queue is empty.

   Precondition: queue is created .

   Postcondition: answers ‘yes’ or ‘no’; original queue unchanged.

5. ***Full:***

   Function: checks whether the queue is full.

Precondition: queue is created.

Postcondition: answers 'yes' or 'no'; original queue unchanged.

Students can try their own ADT specifications for any other operations that they may come across with various implementations.

The enterq operation can always be performed, since there is no limit to the number of elements a queue may contain. So, an overflow situation should never occur. The exit operation, however, can be applied only if the queue is nonempty.

The result of an illegal attempt to remove an element from an empty data structure is called underflow.

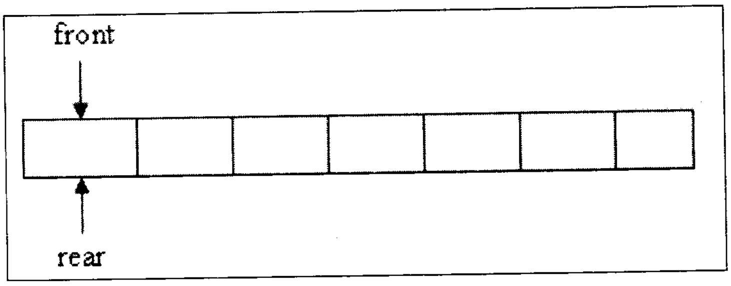
2.8 QUEUE IMPLEMENTATIONS

2.8.1 Array Implementation of Queues

Representation of a queue as arrays is somewhat different than a stack. In addition to a one-dimensional array, we need 2 variables, front and rear. Front points to the first element of the queue and rear to the last element of the queue. Thus, front=rear, when there are no elements in the queue. The initial condition is

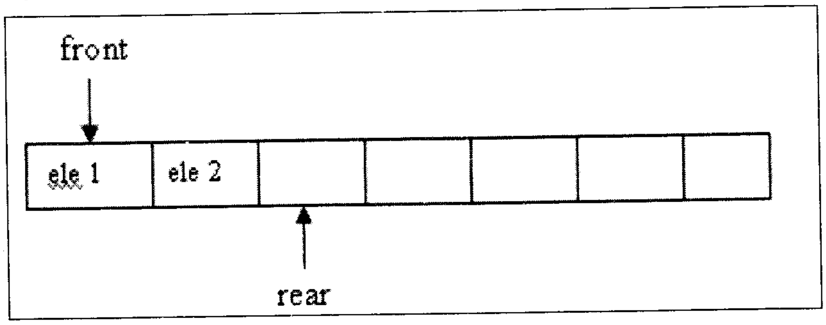
front=rear=0.

Initial Queue



Rear

Queue after Inserting two Elements



Queue after Deleting an Element

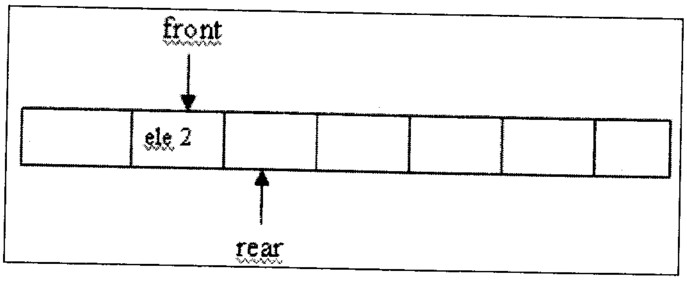


Figure 2.16: Various Stages in Array Implementation of Queues

A full queue is shown by rear, which is equal to the last storage section.

The following program shows insertion and deletion operations on a queue using array. Before writing the actual code, let us try to write an algorithm for the program.

Algorithm

Step 1: Start.

Step 2: Declare the structure of the queue.

Step 3: Insert the elements into the queue from the rear and display the status of queue after each insertion.

Step 4: Continue insertion until the queue is full.

Step 5: Initiate the pop operation by popping the elements from the front.

Step 6: Display the queue after each pop operation.

Step 7: End.

Program

To delete and insert from a queue.

//CREATION OF QUEUES USING ARRAYS

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#define MAX 10
class queue
{
private:
int arr[MAX];
int front,rear;
public:
queue()
{
```

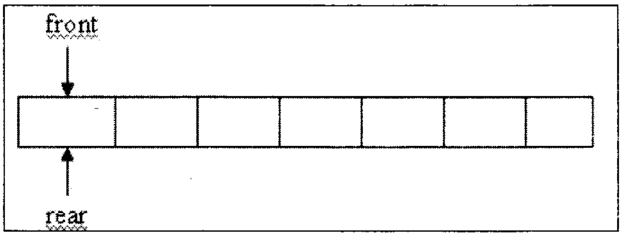
```
front =-1;
rear=-1;
}
void addq(int item)
{
if(rear==MAX-1)
{
cout<<endl<<"Queue is Full";
return;
}
rear++;
arr[rear]=item;
cout<<endl<<"items added"<<arr[rear]<<endl;
if(front==-1)
front=0;
}
int delq()
{
int data;
if(front==-1)
{
cout<<endl<<"Queue is empty";
return NULL;
}
data =arr[front];
if(front==rear)
front=rear=-1;
else
front++;
return data;
}
};
void main()
{ clrscr();
queue a;
a.addq(11);
```

```
a.addq(21);
a.addq(31);
a.addq(41);
a.addq(51);
int i=a.delq();
cout<<endl<<"Item Deleted="<<i<<endl;
i = a.delq();
cout<<endl<<"Item Deleted="<<i<<endl;
i= a.delq();
cout<<endl<<"Item Deleted="<<i<<endl;
i= a.delq();
cout<<endl<<"Item Deleted="<<i<<endl;
i = a.delq();
cout<<endl<<"Item Deleted="<<i<<endl;
getche();
}
```

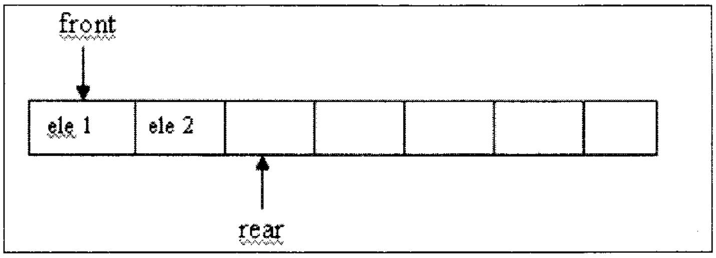
The above design has a shortcoming. As we enter and delete elements from the queue, the front and rear locations also shift forward i.e. as we delete the first item from the queue, the second location becomes the front. Therefore, we loose the first storage location for future storage. As we continue entering and deleting elements, the total storage space available goes on decreasing. Since, we are using arrays as our basic data structure for queues; this can be a serious drawback.

One solution to the above problem can be to shift all remaining elements up every time we remove an element from the queue. But, this increases the overheads. To understand this, take a look at the Figure 2.18.

Initial Queue



Queue after Inserting two Elements



Queue after Deleting an Element

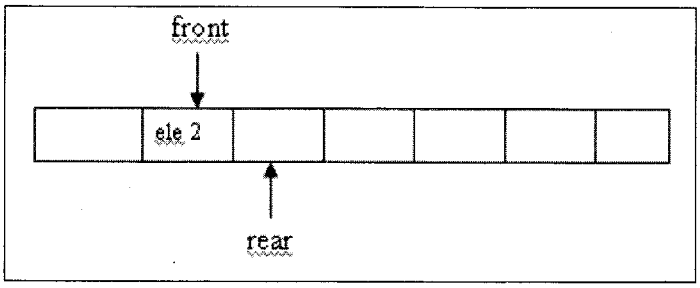


Figure 2.17: Loosing a Storage Location

As you can see above, after taking out the front element of the queue, our front location has incremented to the next element. Thus we have lost the first memory location. Hereafter, we only have 6 locations to store data. Gradually, this space will go on decreasing.

One way of rectifying this problem is to shift the rest of the elements of the queue one space up each time the front element is deleted, as said above. But if a queue is large, this will require a lot of effort.

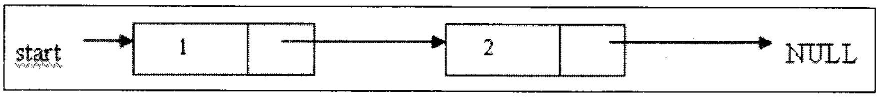
The decision to use this design depends on the final use to which the queue will be put. If the number of elements to be stored in the queue is large, there will be a lot of processing required to move up all the elements. On the other hand, if the queue generally contains only a few elements, this movement may not be much of an overhead. Thus, the complete evaluation of the design depends on the intended use of the program. We will see other implementations to rectify this shortcoming as we proceed.

2.8.2 Linked Implementation of Queues

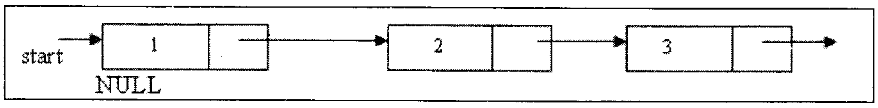
Unlike the array implementation, this version has no program imposed maximum queue size. Nodes are allocated dynamically using new operator, as and when required. If the call to new fails, then it returns NULL and is assumed that memory is full. In case of successful allocation, the new structure is logically added.

To remove a node from a linked list, the node being pointed by the front pointer is removed and its next node becomes the front node. Take a look at the following figure.

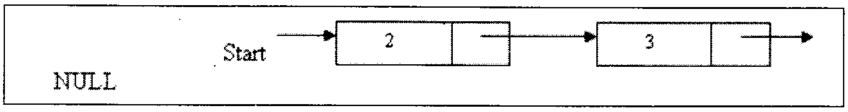
Existing Queue



Inserting another Node at the Rear



Deleting a Node from the Front



2.18: Linked Implementation of Queues

The following program shows how to use linked lists to implement queues. Before writing the actual code, let us write an algorithm for the program.

Algorithm

Step 1: Start.

Step 2: Declare the structure of the linked queue.

Step 3: Insert the elements in the linked queue and increment the rear by 1 after each insertion.

Step 4: Insert the elements until the queue is full.

Step 5: Perform the pop operation by deleting the elements from the front of the linked queue.

Step 6: Display the status of the queue after each pop operation.

Step 7: End.

Program

To implement queues using linked lists.

/*this program uses linked lists to implement queues*/

// IMPLEMENTATION OF LINKED QUEUES

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
node *link;
};
class queue
{
private:
node *front, *rear;
public:
queue()
{
front=rear=NULL;
}
void addq(int item)
{
node *temp;
```

```
temp=new node;
if(temp==NULL)
cout<<endl<<"Queue is Full";
temp->data=item;
temp->link=NULL;
if(front==NULL)
{
rear=front=temp;
return;
}
rear->link=temp;
rear=rear->link;
}
int delq()
{
if(front==NULL)
{
cout<<endl<<"Queue is Empty";
return NULL;
}
node *temp;
int item;
item=front->data;
temp=front;
front=front->link;
delete temp;
return item;
}
~queue()
{
if(front==NULL)
return;
node *temp;
while(front!=NULL)
{
temp=front;
front=front->link;
delete temp;
}
}
};
void main()
{
clrscr();
queue a;
a.addq(11);
a.addq(21);
a.addq(31);
int i=a.delq();
cout<<endl<<"The Item deleted="<<i;
i=a.delq();
cout<<endl<<"The Item deleted="<<i;
i=a.delq();
cout<<endl<<"The Item deleted="<<i;
getche();
}
```

2.9 ANALYSIS OF QUEUE IMPLEMENTATIONS

An array of MAX queue size will take same amount of memory, no matter how many array slots are actually used. The linked implementation using dynamically allocated storage space only requires space for the number of elements actually in the queue at the run time, plus space for the external pointers. However, the node elements are larger, since we must store the link as well as the user data.

We can also compare the relative "efficiency" of the two implementations, in terms of Big Oh. Following are the Big Oh measurements of some of the common operations that are performed on queues.

1. *Create:* In both implementations, create operation has a measure of O(1). It takes fixed amount of work in any case.

2. *Empty and full:* In both implementations, these two operations have measure O(1) only. Because, the algorithm has to do only one operation, that of checking whether the queue contains anything or not.

3. *Enterq and exitq:* These operations are also O(1) in both implementations.

   The number of elements in the queue does not affect the work done by these two operations. In both implementations, we can directly access the front and rear of the queue. Therefore, amount of work done by these two operations is independent of queue size.

4. *Deleting the entire queue:* This operation differs from one implementation to other. The array implementation just sets the front and rear indexes, so it is an O(1) operation. The linked

implementation has to access each node and free it explicitly. Therefore, this operation has the measure O(n), where n is the number of elements in the queue at that time.

As with the array and linked implementation of stacks, these two implementations of queues are more or less equivalent in terms of amount of work they do, possibly differing in one operation. Following table summarizes the Big Oh measures of various operations on the two implementations of queues.

Table 2.2: Big Oh Comparison of Queue Operations

Operations	**Array Implementation**	**Linked Implementation**
Create	O(1)	O(1)
Empty	O(1)	O(1)
Full	O(1)	O(1)
Enterq	O(1)	O(1)
Exitq	O(1)	O(1)
destroy	O(1)	O(n)

The answer to the question, which implementation is better, depends on the requirements of your application.

Set ADT

A set is a collection of bindings. Each binding consists of a key and a value. A key uniquely identifies its binding; a value is data that is somehow pertinent to its key. Programming systems use sets often. For example compilers and assemblers use sets to relate symbols to their meanings.

Set Interface

The Set interface should contain these function declarations:

```
Set_T          Set_new(int iEstimatedLength,
               int (*pfCompare)(const void *pvKey1, const void *pvKey2));
void           Set_free(Set_T oSet);
void           Set_clear(Set_T oSet);
int            Set_getLength(Set_T oSet);
int            Set_put(Set_T oSet, const void *pvKey, void *pvValue);
int            Set_remove(Set_T oSet, const void *pvKey);
const void     *Set_getKey(Set_T oSet, const void *pvKey);
void           *Set_getValue(Set_T oSet, const void *pvKey);
void           Set_map(Set_T oSet,
               void (*pfApply)(const void *pvKey, void **ppvValue, void *pvExtra),
               void *pvExtra);
```

Complex Number ADT

```
typedef struct {
float real;
float imag;
} COMPLEX;
COMPLEX makecomplex (float, float) ;
COMPLEX addc (COMPLEX, COMPLEX);
COMPLEX subc (COMPLEX, COMPLEX);
COMPLEX multc (COMPLEX, COMPLEX);
COMPLEX divc (COMPLEX, COMPLEX);
```

A String ADT

Most languages either have a built in string datatype or a standard library, so rare to create own string ADT.

- A string datatype should have operations to:
- Return the nth character in a string.
- Set the nth character in a string to c.
- Find the length of a string.
- Concatenate two strings. "Alison" + "Cawsey" = "Alison Cawsey"
- Copy a string.
- Delete part of a string. ("Alison Cawsey" "Alison")
- Modify and compare strings in other ways.

String Processing Algorithms

String processing algorithms are algorithms for processing sequences of characters or symbols e.g.,

- File compression – take a sequence, encode it as a shorter sequence.
- Cryptography – take a sequence, encode it so enemies can't read it!
- String search – search for occurrences of one sequence within another.
- Pattern matching – find out if sequence matches some pattern.
- Parsing – Work out structure of a sequence, in terms of a grammar.
- Applications of more complex algorithms might include genome sequencing and analysis.
- We can start by looking at the relevant datatypes or classes for strings and sequences.– Delete part of a string. ("Alison Cawsey" _ "Alison")
- Modify and compare strings in other ways.

String Implementations

There are two main ways that strings may be:

Implemented: As a fixed length array, where the first element denotes the length of the string, e.g., [6,a,l,i,s,o,n,.....]. This is used as the standard string type in Pascal.

As an array, but with the end of the string indicated using a special 'null' character (denoted '_ 0'), e.g., [a,l,i,s,o,n,_ 0,.....].

Memory can be dynamically allocated for the string once we know its length.

First implementation has disadvantages of all fixed length array implementations. But some operations are efficient (e.g., finding length).

Second implementation has advantages of dynamic allocation of space; modifying string also may be more efficient, as needn't recalculate size.

Check Your Progress

Define the following:

1. Stack
2. Linked Implementation

2.10 LET US SUM UP

An ADT is the logical picture of a data type plus the specifications of the operations required in creating and manipulating objects of this data type. Basic operations performed on any type of data structures are insert, modify, delete, search, sort etc. The operations performed on linked list are create, insert, delete, modify etc. A single node of linked list consist of mainly two fields (1) data and (2) pointer to the next node as shown below.

The other type of linked lists are:

- Circular linked lists.
- Doubly or two way linked lists.

In computing paradigm linked lists can be put to use for a variety of purposes. Also they can be used to implement data structures like stacks and queues. An array is a list of elements in which each element is accessible via an index. There are various algorithms available to implement the same task. Hence it is necessary to analyze algorithms. Performance of a program depends on two factors - space complexity and time complexity. Space complexity of a program is the amount of memory required to execute the program successfully. In static allocation memory required by a program is known at compile time whereas in dynamic allocation it can increase during the execution. Time complexity of a program is the amount of time required to execute a program successfully. The order of magnitude of an algorithm is the sum of the frequencies of all of its statements. Asymptotic notations calculate the approximate time and space requirements of an algorithms. There are various types of asymptotic notations viz. Big Oh, Omega, Theta etc. The major goals involved in the study of data structure are:

To identify and develop useful mathematical entities and operations and

- To determine what classes of problems can be solved using these entities and operations.
- Determine representations for abstract entities and implement them.

A stack is an ordered list in which all insertions and deletions are done at one end, called the top. A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. Unlike an array, the definition of stacks and queues provide for the insertion and deletion of items. So, stacks and queues are dynamic, constantly changing objects. Queues provide FIFO storage as opposed to LIFO storage provided by stacks. A stack is a dynamic structure i.e. it changes as elements are added to and removed from it. The operation that adds an element to the top of a stack is usually called PUSH and the operation that takes the top element off the stack is called POP. Even though the representation of the stack may be a random-access structure such as an array, the stack itself as a logical entity is not randomly accessed. Stacks and queues can be implemented using arrays as well as using linked lists. As stated in earlier lessons, an array variable of MAX stack size will take the same amount of memory, no matter how many array slots are actually used. Therefore, we need to reserve space for the maximum possible. On the other hand, the linked implementation using dynamically allocated storage space only requires space for the number of elements actually present in the stack at run time. But the elements are larger since we must store the link (the next field) along with the user's data. Stack is an appropriate data structure when information must be saved and then later retrieved A in reverse order. Any situation requiring to backtrack to some earlier position may be a good one to use a stack. There are 2 operations that can be applied to a queue. First, new elements are added to the rear of the queue. We will call this operation enterq. We can also take the elements off the front of the queue. We will call this operation exitq. The result of an illegal attempt to remove an element from an empty data structure is called underflow.

2.11 KEYWORDS

Stack: Stack is a data structure which stores data at the top.

Data Structure: It is a collection of data elements.

2.12 QUESTIONS FOR DISCUSSION

1. What do you understand by the term Abstract Data Type?
2. Differentiate between the Stacks and Queues on the basis of the storage of data.
3. What is the basic difference between stacks and queues?
4. What is a LIFO list and a FIFO list?
5. What are the various functions that can be performed on stacks and queues?

Check Your Progress: Model Answers

1. A stack is a dynamic structure i.e. it changes as elements are added to and removed from it. The operation that adds an element to the top of a stack is usually called PUSH and the operation that takes the top element off the stack is called POP.
2. Linked implementation is more flexible and is preferable where the number of stack elements vary greatly. It wastes less space when the stack is small. When stack size is unpredictable, linked implementation is preferable.

2.13 SUGGESTED READINGS

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

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

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

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

Jean-Paul, Tremblay, Paul G Sorenson, *Introduction to data structures with application*, McGraw Hill Book Company

Richard F Gilberg, *et al.*, *Data Structures – A Pseudocode Approach with C*, First Edition, Thomson, 2002

Kutti, NS Padhye, P.Y., *Data Structures in C++*, 2nd ed., Prentice Hall 2000

Robert Sedgewick, *Algorithms in C++*, 3rd ed., Addison Wesely 1999

Ellis Horowitz, et al, *Fundamentals of Data Structures in C++*, 1st ed, Galgotia

UNIT II

LESSON

3

TREES

CONTENTS

3.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Define trees
- Identify various characteristics of tress
- Describe the representations in contiguous memory
- Explain the linked tree representation
- Traverse binary tree
- Traverse a binary tree
- Search a binary tree
- Insert into a binary search tree
- Understand and use AVL trees

3.1 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 organisational 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.

3.2 TREES

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 > = 0 disjoint set of nodes $T_1, T_2,...,T_n$ each of which is a tree.

Shown below in Figure 3.1 is a structure, which is tree.

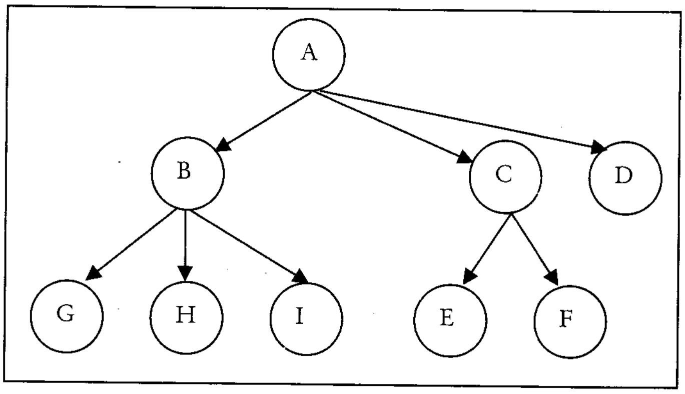


Figure 3.1: A Tree Structure

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 3.2 is a structure, which is not a tree:

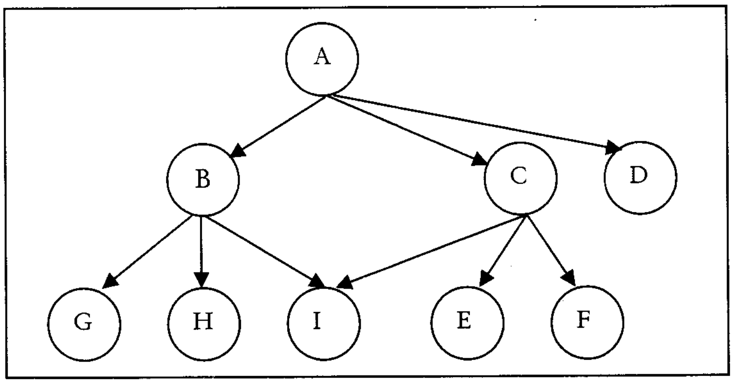


Figure 3.2: A Non-tree Structure

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.

3.2.1 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.

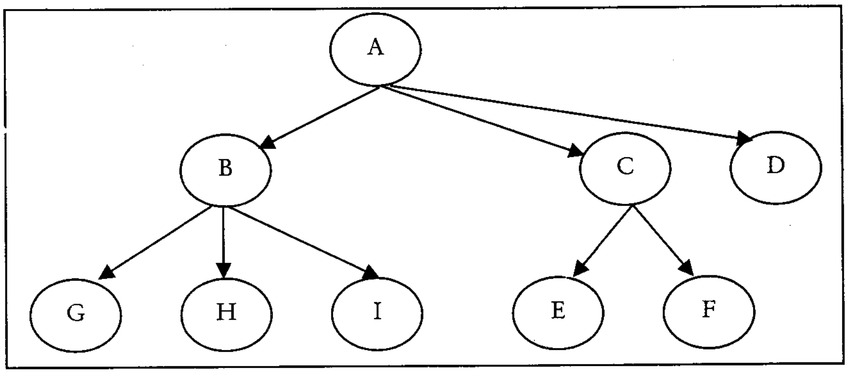
3.2.2 Degree of a Tree

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).

3.2.3 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 descendents of the root nodes will be 2. The level of their descendents 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 below:



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

3.3 N-ARY 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. Note, however, that the degree of the nodes may be less than N. Thus, a tree with degree 2 is called binary tree, with degree 3 is called ternary tree and so on.

3.3.1 Binary Tree

Binary tree is a special type of tree having degree 2. In a binary tree no node of can have degree greater than 2. Therefore, a binary tree is a set of zero or more nodes T such that

(i) There is specially designated node called as root of tree and

(ii) The remaining nodes are partitioned into two disjoint sets T_1 and T_2 each of which is a binary tree. T_1 is called as a left sub-tree and T_2 is called right sub-tree or vice-versa.

A binary tree is shown below (Figure 3.3).

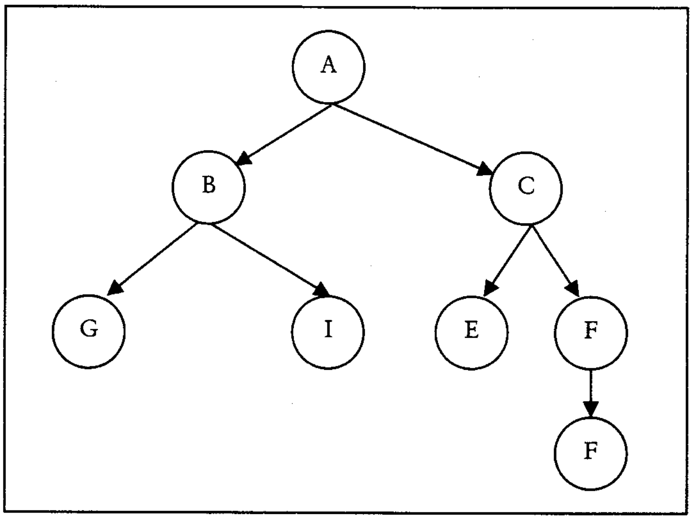


Figure 3.3: Binary Tree Structure

For a binary tree note that,

(i) maximum number of nodes at level i is 2*i-1

(ii) if k is the depth of the tree then the maximum number of nodes the tree can have is

$2^{k-1} + 2^{k-2} + ... + 2^2 + 2^1 = 2^k - 1$

Also there are skewed binary trees like the one shown below in Figure 3.4

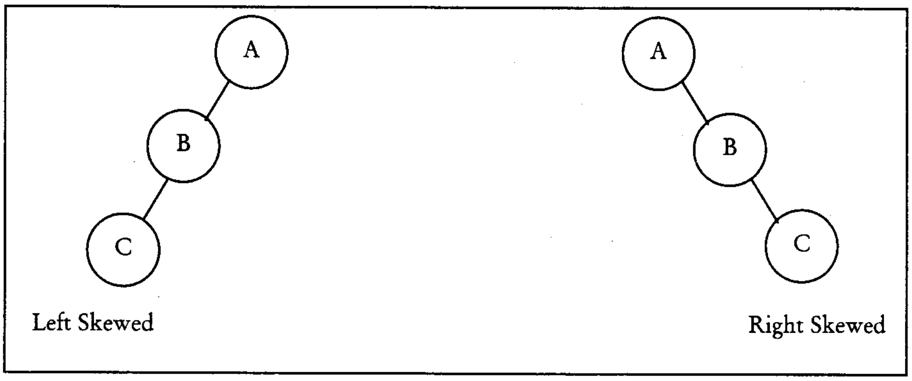


Figure 3.4: Skewed Trees

3.3.2 Full and Complete Binary Tree

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.

For 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 shown below in Figure 3.5.

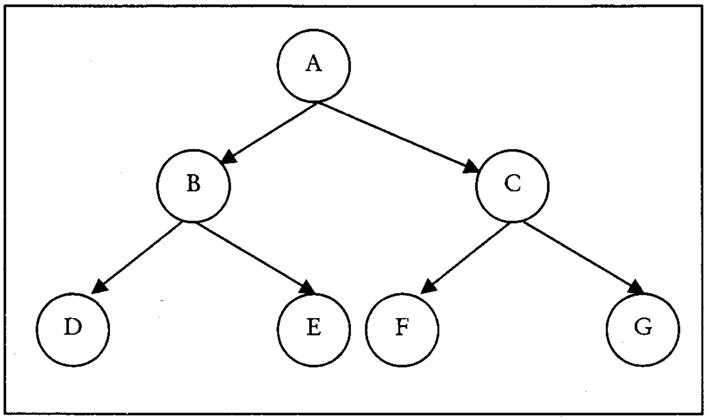


Figure 3.5: A Full Binary Tree

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

A complete binary tree of depth k is a tree with n node 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 below in Figure 3.6.

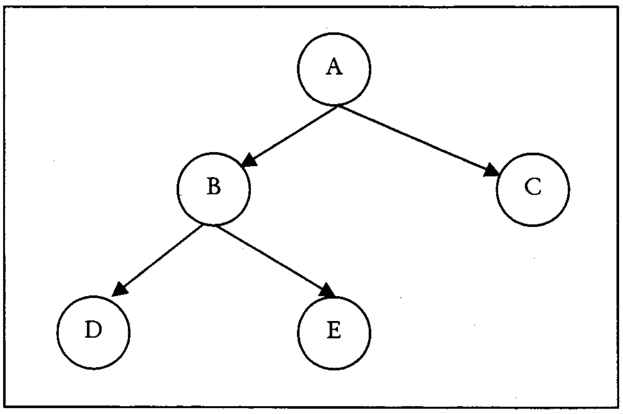


Figure 3.6: A Complete Binary Tree

3.3.3 Representations in Contiguous Memory

If a binary tree is a complete binary tree, then 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 tree is an array of n elements, then 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 i^{th} index in the array, and the parent of node i will get mapped at an index i/2 whereas left child of node i gets mapped at an index 2i and right child gets mapped at an index 2i + 1.

For 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 show below in Figure 3.7.

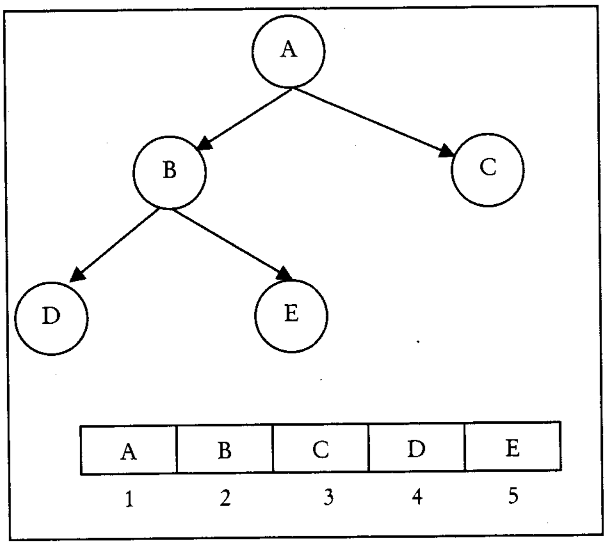


Figure 3.7: An Array Representation of a Complete Binary Tree having 5 Nodes and Depth 3

Shown below in Figure 3.8 is another example of an array representation of a complete binary tree with depth k = 3, having the number of nodes n = 4.

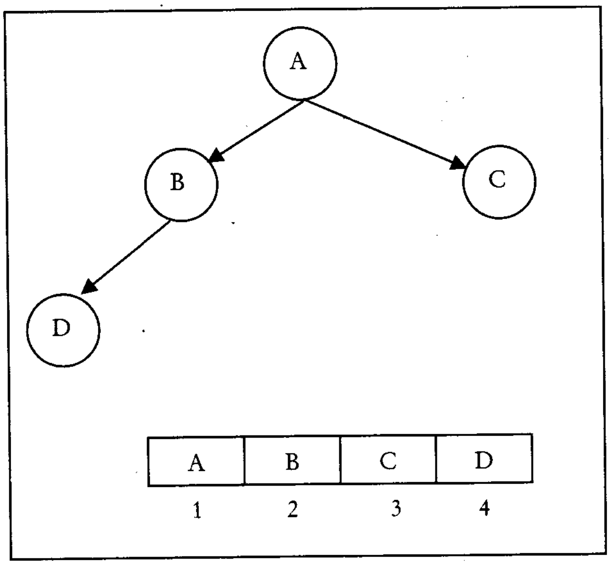
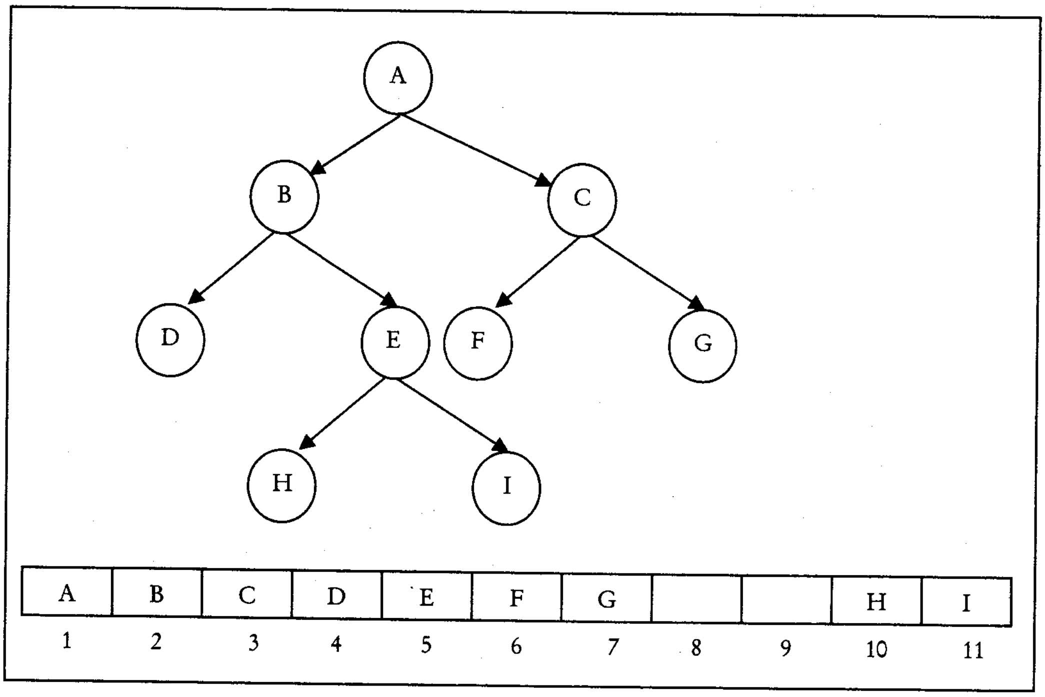


Figure 3.8: An Array Representation of a Complete Binary Tree having 4 Nodes and Depth 3

In general any binary tree can be represented using an array. But we see that an array representation of a complete binary tree does not lead to the wastage of any storage. But if we want to represent a binary tree which is not a complete binary tree using an array representation, then it leads to the wastage of storage as shown in Figure 3.9.

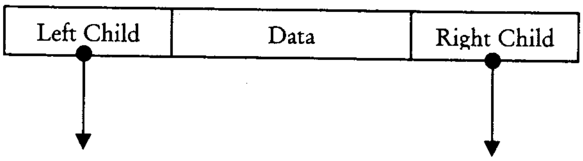


A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Figure 3.9: An Array Representation of Binary Tree

3.4 LINKED TREE REPRESENTATION

An array representation of a binary tree is not suitable for frequent insertions and deletions, and therefore we find that even though no storage is wasted if the binary tree is a complete one when array representation is used, 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 having 3 fields, one for holding data, one for linking it with left sub-tree and the one for linking it with right sub-tree as shown below:



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

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

A tree representation using the above node structure is shown below in Figure 3.10.

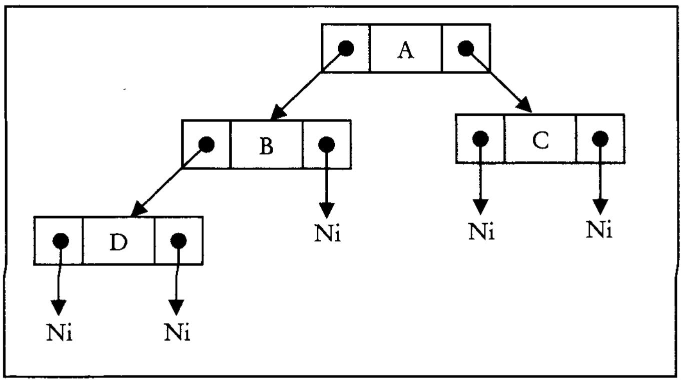


Figure 3.10: Linked Representation of a Binary Tree

3.5 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.

3.5.1 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

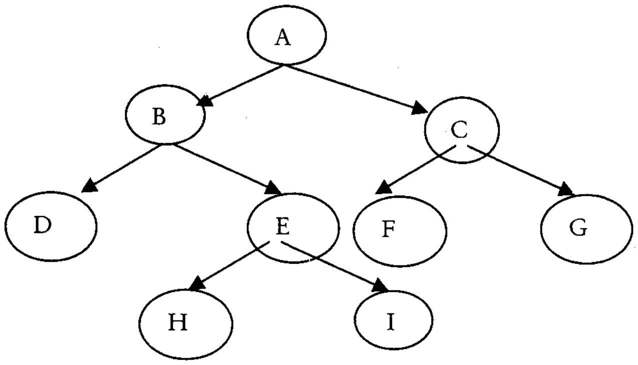
Where,

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 3.11, using the inorder, preorder and postorder is the one shown below in Figure 3.11 itself.



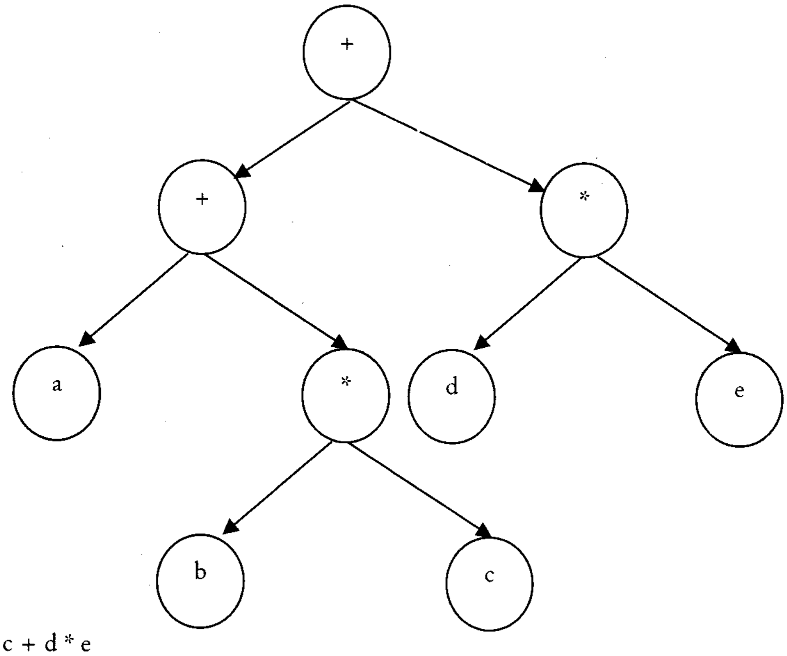
Inorder: DBHEIAFCG

preorder: ABDEHICFG

postorder: DHIEBFGCA

Figure 3.11: A Binary Tree along with its Inorder, Preorder and Postorder

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 posfix expression as shown below in Figure 3.12.



Inorder : a + b * c + d * e
Postorder: abc*+de*+

Figure 3.12: A Binary Tree of an Expression along with its Inorder and Postorder

Given an order of traversal of a tree it is possible to construct a tree. For example consider the following order:

Inorder = DBEAC

We can construct the binary trees shown below in Figure 3.13 using this order of traversal:

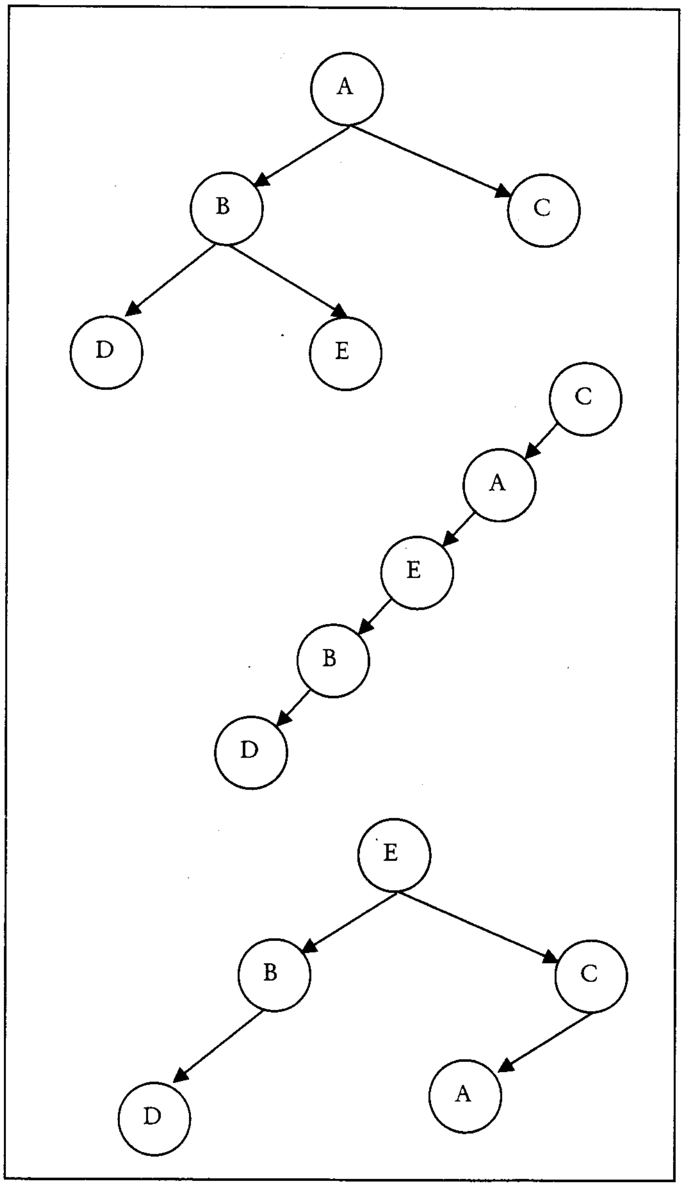


Figure 3.13: Binary Trees Constructed using Given Inorder

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.

For example, consider the following orders:

lnorder = DBEAC

Postorder = DEBCA

We can construct a unique binary tree shown in Figure 3.14 using these orders of traversal.

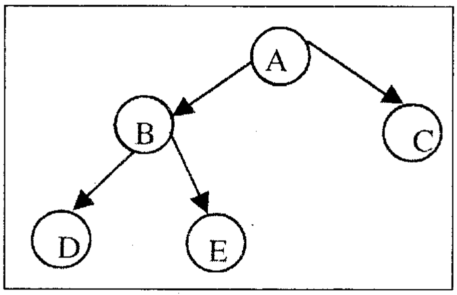


Figure 3.14: A Unique Binary Constructed using the Inorder and Postorder

3.5.2 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 are n + 1. Since every node is placed on the stack once, the statements stack[top] := p and p := stack[top] are executed n times. The test for nil link 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.

3.5.3 Preorder Traversal

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

3.5.4 Postorder Traversal

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

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 preoder, 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 3.15:

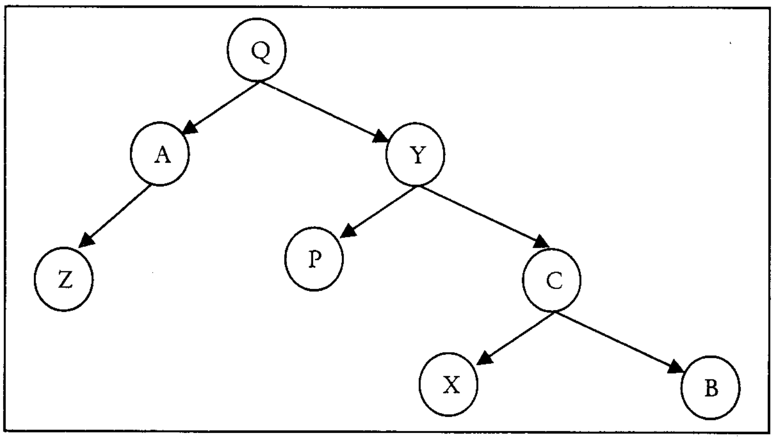


Figure 3.15: A Unique Binary Tree Constructed Using the Inorder and Postorder

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->1child == NULL) && (p->rchild == NULL))
                count = 1;
        else
                count = count(p->1child) + 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->1child);
                swaptree(p->rchild);
                temp = p->1child;
                p->1child = 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->1child,p2->1child))
                                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
               {
               q = new(tnode);
               q->data = p->data;
               q->lchild = copytree(p->lchild);
               q->rchild = copytree(p->rchild);
               return(q);
     }
}
```

3.6 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 3.16 is a binary search tree:

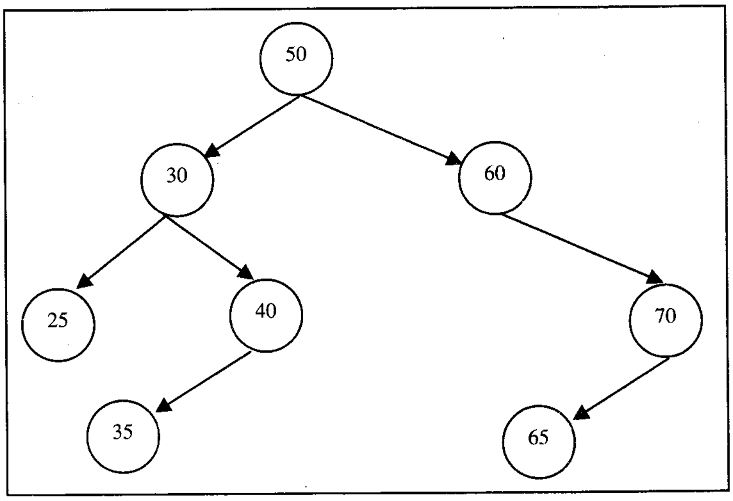


Figure 3.16: A Binary Search Tree

A binary search tree is basically a binary tree, and therefore it can be traversed is in-order, preorder, and post-order. 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 an 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.

3.6.1 Creating a Binary Search Tree

We assume that every node 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 *lchid;
        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 temp 1, it then stores value of temp 1 in another variable temp2, compares the data value of the node pointed to by temp 1 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 temp 1 then it copies the left link of the node pointed by temp 1 into temp 1 (goes to the left), otherwise it copies the right link of the node pointed by temp 1 into temp 1(goes to the right). It repeats this process till temp 1 becomes nil. When temp 1 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->1child = 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;
                }
        }
}
```

3.6.2 Deletion of a Node from Binary Search Tree

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.

3.6.3 Deletion of a Node with two Children

Consider the binary search tree shown below in Figure 3.17:

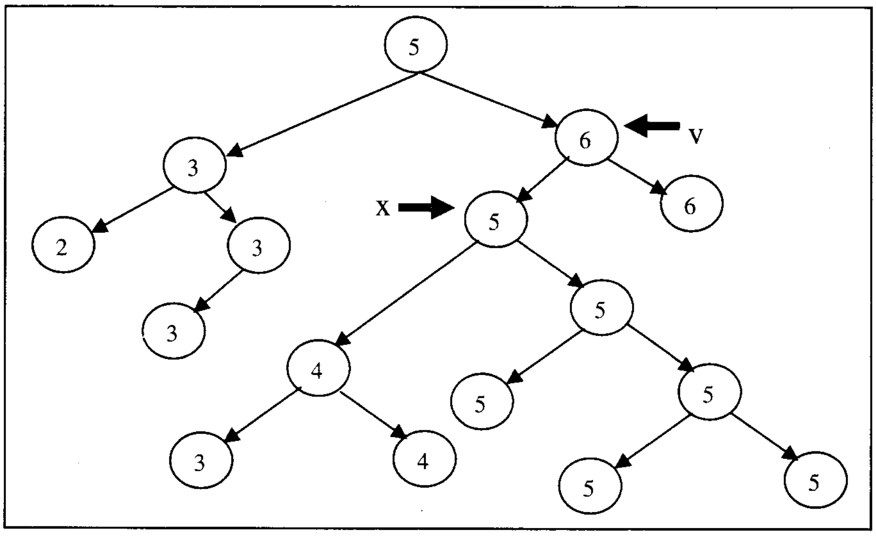


Figure 3.17: A Binary Tree before Deletion of a Node Pointed to by x

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 by x. We store the pointer to the left child of the node pointed by x in a temporary pointer temp. We then make the left child of the node pointed by y to be the left child of the node pointed by x. We then traverse the tree with the root as the node pointed by temp to get its right leaf, and make the right child of this right leaf to be the right child of the node pointed by x, as shown below in Figure 3.18:

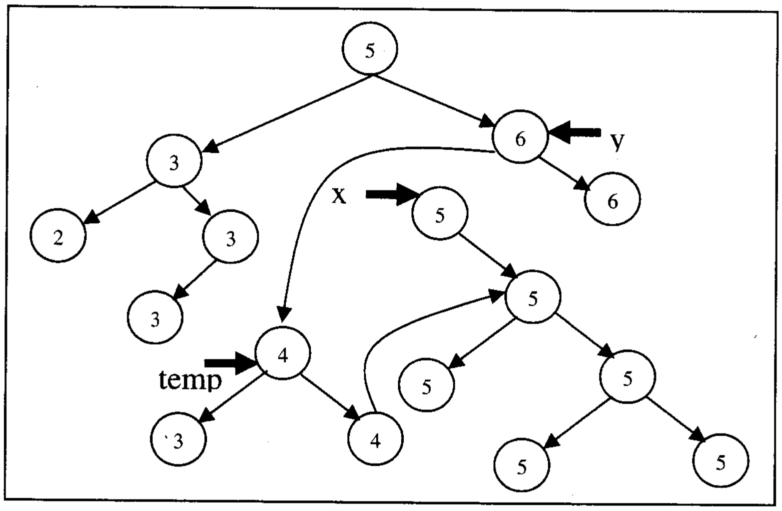


Figure 3.18: A Binary Tree after Deletion of a Node Pointed to by x

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

Another method is store the pointer to the right child of the node pointed by x in a temporary pointer temp. We then make the left child of the node pointed by y to be the right child of the node pointed by x. We then traverse the tree with the root as the node pointed by temp to get its left leaf, and make the left child of this left leaf to be the left child of the node pointed by x, as shown in Figure 3.19:

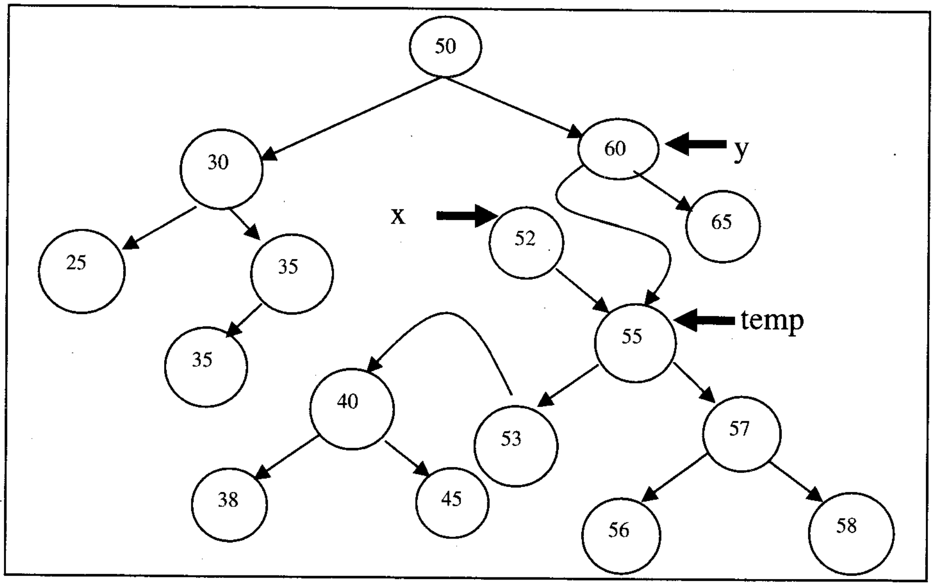


Figure 3.19: A Binary Tree after Deletion of a Node Pointed to by x

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

3.6.4 Deletion of a Node with one Child

Consider the binary search tree shown below in Figure 3.20.

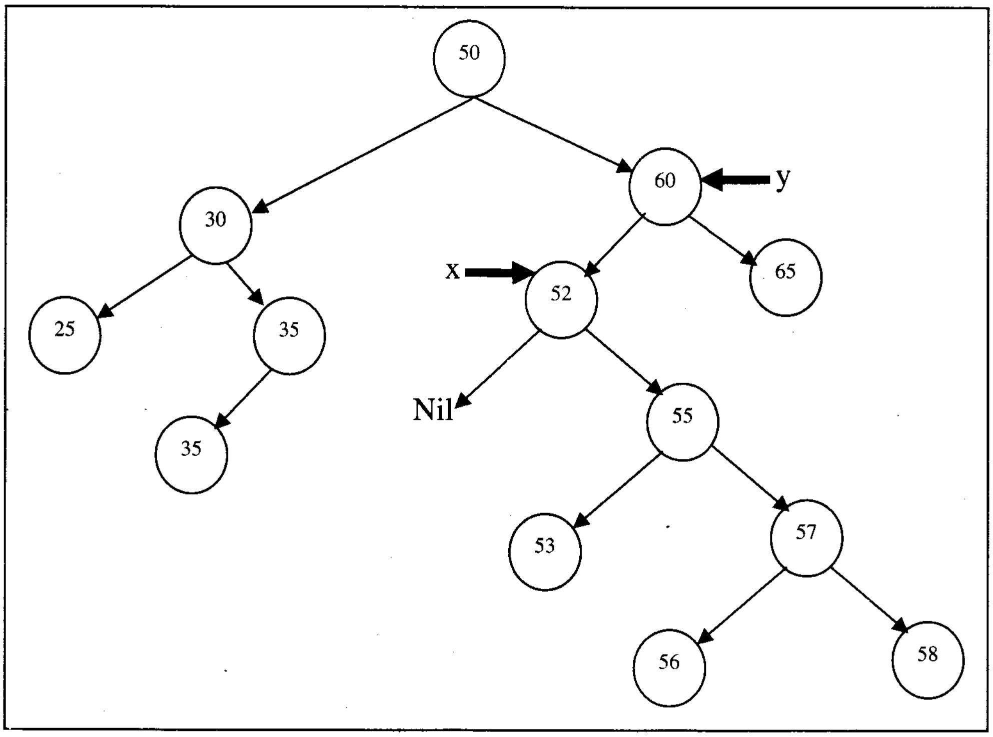


Figure 3.20: A Binary Tree before Deletion of a Node Pointed to by x

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 3.21:

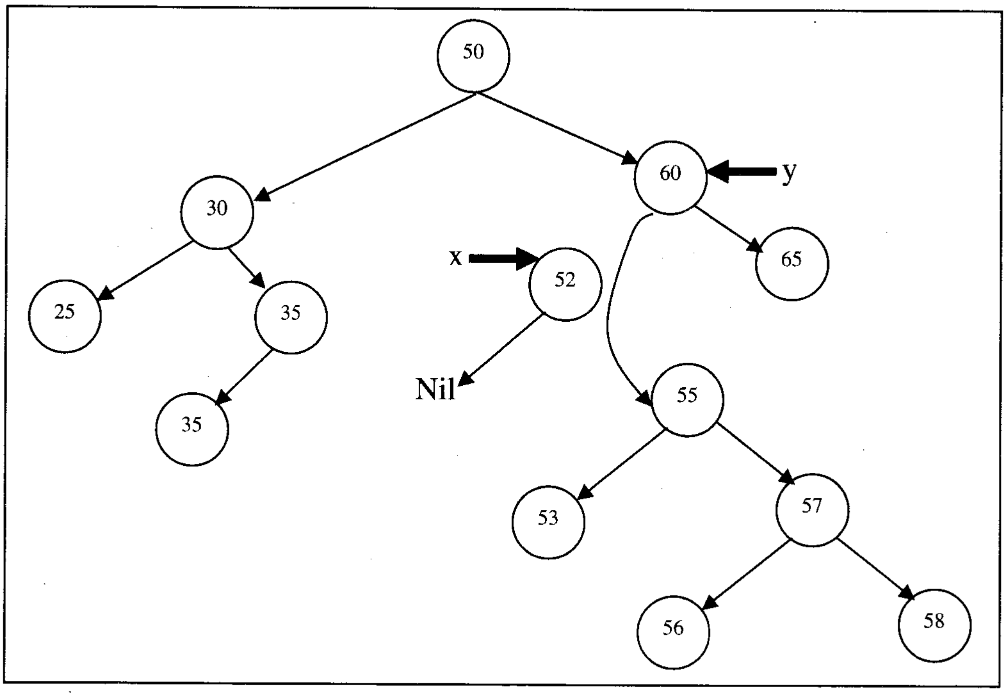


Figure 3.21: A Binary Tree after Deletion of a Node Pointed to by x

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

3.6.5 Deletion of a Node with no Child

Consider the binary search tree shown below in Figure 3.22:

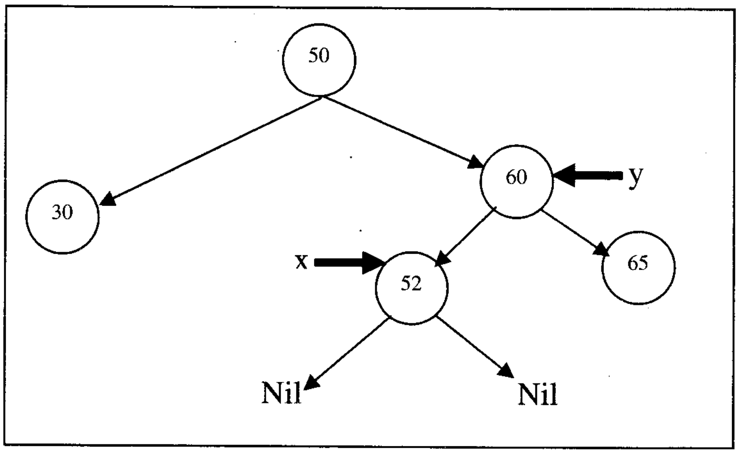


Figure 3.22: A Binary Tree before Deletion of a Node Pointed to by x

Let the left child of the node pointed by y be nil, and dispose node pointed by x as shown below in Figure 3.23.

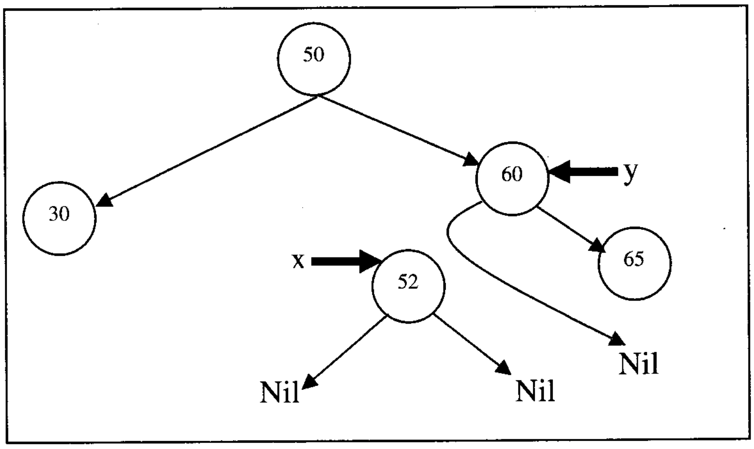


Figure 3.23: A Binary Tree after Deletion of a Node Pointed to by x

3.6.6 Searching for a Target Key in a Binary Search Tree

```
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;
    ]
    }
```

3.6.7 An Application of a Binary Search 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.

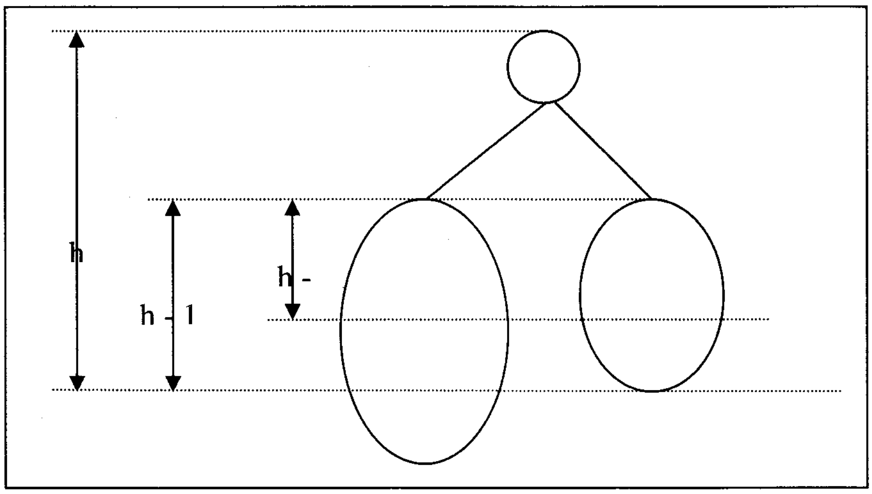
A program to create a binary search tree, given a list of identifiers is given below:

```
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);
}
```

3.7 AVL TREES

An **AVL tree** is a balanced binary search tree. It takes its name from the initials of its inventors - **A**delson, **V**elskii and **L**andis. 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.

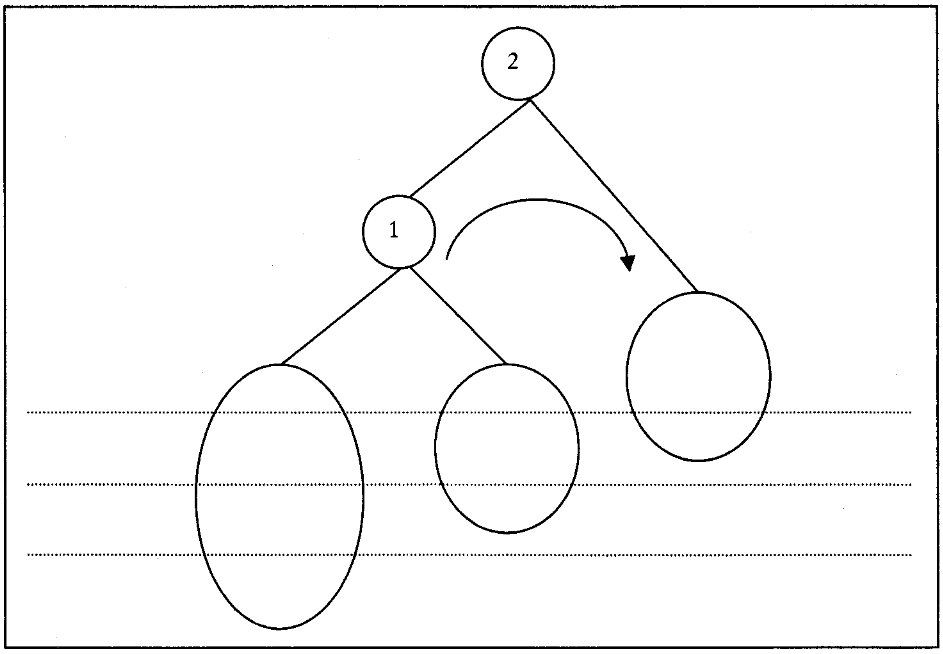


Here, the height of the tree is h. Height of one subtree is h–1 while that of another subtree of the same node is h–2, differing from each other by just 1. Therefore, it is an AVL tree.

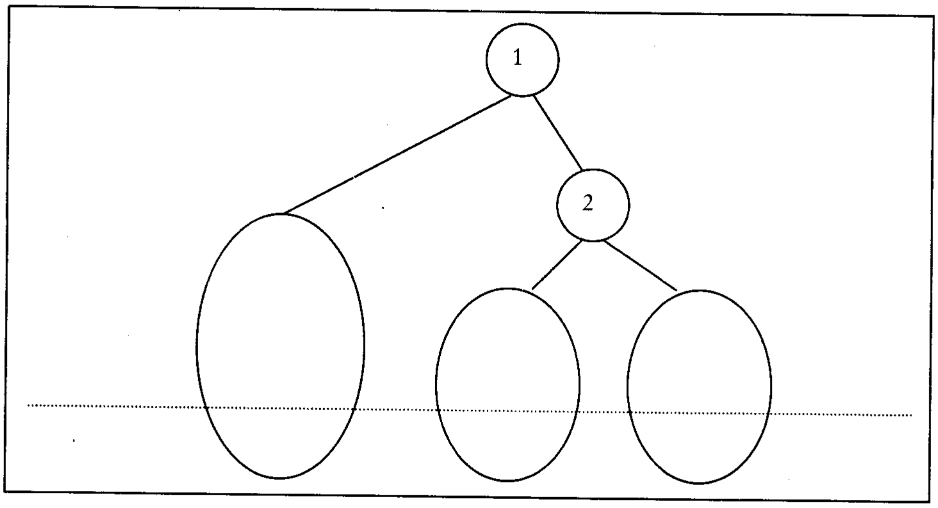
Inserting a Node into AVL Tree

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 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:



Check Your Progress

1. What are the characteristic properties of an AVL tree?
2. Define level of a node.

3.8 LET US SUM UP

- 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 2k-1 number of nodes. If a binary tree has fewer than 2k-1 nodes, it is not a full binary tree.
- An AVL tree is another balanced binary search tree. It takes its name from the initials of its inventors - Adelson, Velskii and Landis.
- In an AVL tree the sub-trees of every node differ in height by at most one level and every sub-tree is an AVL tree.

3.9 KEYWORDS

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.

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

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

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

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

N-ary tree: A tree in whose degree is N.

Binary tree: A tree of degree 2.

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

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

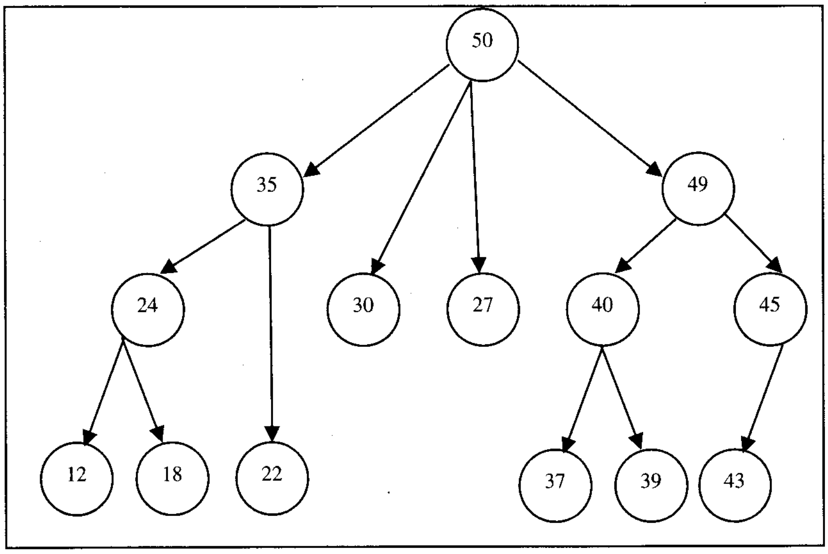
Preorder: A tree traversing method in which the tree is traversed in the order of node, left-tree and then right-tree.

Search tree: A tree constructed and used in searching algorithms.

AVL tree (Adelson, Velskii and Landis Tree): A balanced binary search tree in which the sub-trees of every node differ in height by at most one level and every sub-tree is an AVL tree.

3.10 QUESTIONS FOR DISCUSSION

1. Consider the tree given below:
   (i) Find the degree of each mode of the tree.
   (ii) The degree of the tree.
   (iii) The level of each nodes of the tree.



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

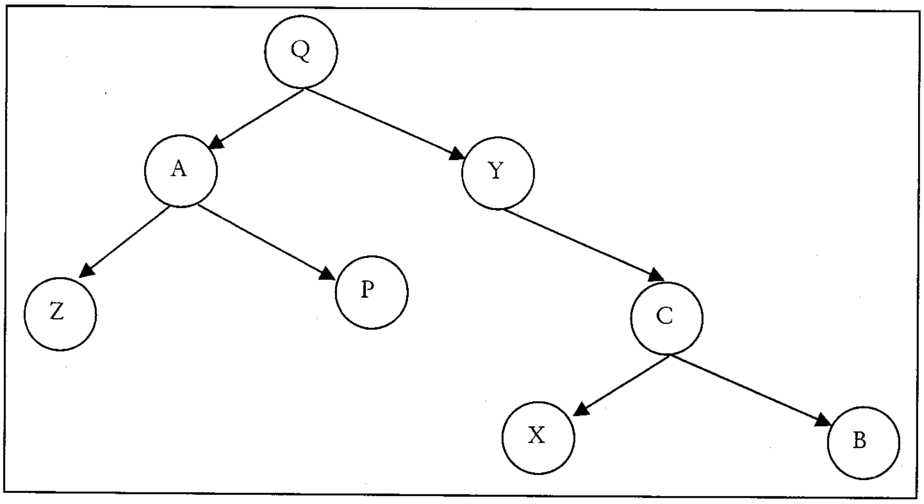
3. How many binary trees are possible with three nodes?

4. 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

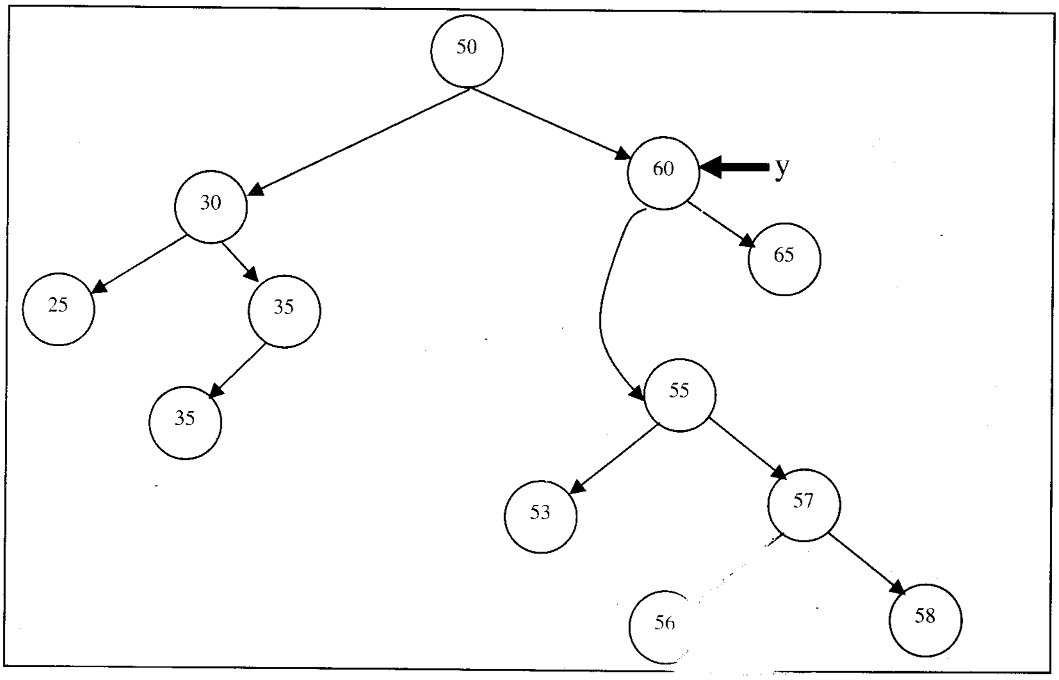
5. Perform inorder, preorder and postorder traversal in the following tree.



6. If the preorder traversal of a tree gives the following sequence of nodes, draw the tree. Also traverse it in inorder and postoder.

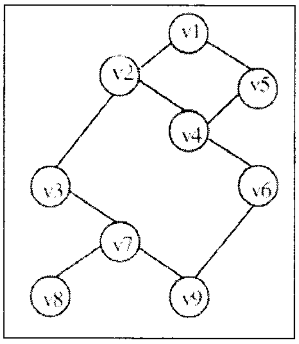
   ABCDEFGH

7. Show the result of deleting node (60) from the following binary search tree.



8. Show the result of inserting node (45) into the above binary search tree.

9. Convert the following graph into a binary tree by removing necessary edges.



12. Where are AVL trees used?

Check Your Progress: Model Answers

1. An **AVL tree** is another balanced binary search tree. It takes its name from the initials of its inventors - **A**delson, **V**elskii and **L**andis. An AVL tree has the following properties:
   i. The sub-trees of every node differ in height by at most one level.
   ii. Every sub-tree is an AVL tree.
2. 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 descendents of the root nodes will be 2. The level of their descendents will be 3 and so on.

3.11 SUGGESTED READINGS

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

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

How to Solve it by Computer; RG Dromey; Cambridge University Press

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

LESSON

4

HASHING AND PRIORITY QUEUES

CONTENTS

4.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define hashing and hashing functions
- Describe the collision handling techniques

4.1 INTRODUCTION

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. This concept has been discussed in this lesson in detail.

A priority queue is a collection of elements such that each element has been assigned a priority. We have discussed priority queues and its implementation in this lesson.

4.2 HASHING

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 we 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 we 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 we select depends on the table size. Therefore if r is the number of middle bits that we use to form hash value, then the table size will be 2^r. Hence when we 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.

4.2.1 Hashing 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 we use mid square method the table size should be a power of 2.

4.2.2 Hash Collision

To store the name or to add attributes of the name, we 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, we apply the hash function to the name to obtain index of the table where we get the attributes of the name. Hence we 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, we 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 we 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, we have to find out an alternative location either for x or y. This means we find out a suitable alternative location and either accommodate y in that location, or we can move x to that location 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:

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 we 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 we start from the $(i+1)^{th}$ location and do the linear search till we get an empty location, and once we get an empty location we accommodate x there.
2. ***Rehashing:*** This is another method of collision handling. In this method, we 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 we 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 we accommodate x in the j^{th} location. Otherwise, we 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.
3. ***Separate Chaining/Overflow chaining:*** This is a method of implementing a hash table, in which collisions gets handled automatically. In this method, we use two tables, a symbol table to accommo-

date 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 holing 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 ith hash table pointer is made pointing to the symbol table entry in which x is stored if the ith hash table pointer is not pointing to any symbol table entry. If the ith 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 ith hash table pointer is pointing to, and make the ith hash table pointer pointing the symbol entry containing x. This is equivalent to building a linked list on the ith index of the hash table. The retrieval of attributes is done as follows:

If x is a symbol, then we 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 we use is:

h(symbol) = (value of first letter of the symbol) mod 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 4.1.

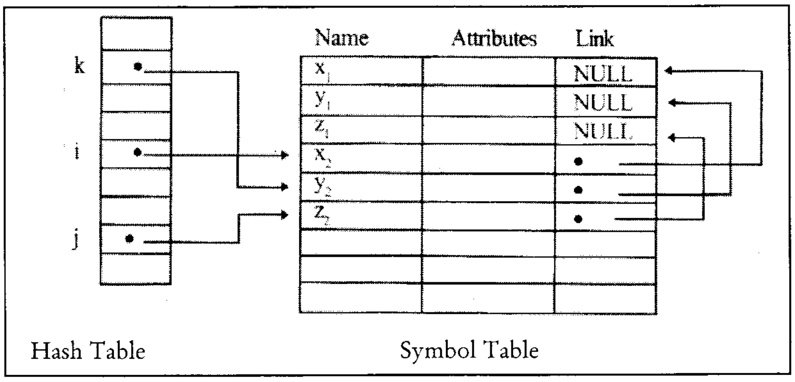


Figure 4.1: Hash Table Implementation using Overflow Chaining for Collision Handling

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 his:

h(key) = key mode 11, where key is the value to be stored.

We start with the value 25, and compute the hash value using 25 as key. The hash value is h(25) = 25 mod 11 = 3. Therefore store 25 at the index 3 in the table.

For 45, h(45) = 45 mod 11 = 1, hence place 45 at the index 1.

For 96, h(96) = 96 mod 11 = 8,

∴ store 96 at index 8.

For 101,h(101) = 101 mod 11 = 2,

∴ store 101 at index 2.

For 102, h(102) = 102 mod 11 = 3, there is a collision, therefore we find a location closer to location at index 3 which is empty to accommodate 102, we see that the location at index 4 is empty.

∴ store 102 at index 4.

For 162, h(162) = 162 mod 11 = 8, again there is a collision, therefore we find a location closer to location at index 8 which is empty to accommodate 162, we see that location at index 9 is empty.

∴ store 162 at index 9.

For 197, h(197) = 197 mod 11 = 10,

∴ store 197 at index 10.

For 201, h(20 1) = 201 mod 11 = 3, again there is a collision, therefore we find a location closer to location at index 3, which is empty to accommodate 201, we see that location at index 5 is empty.

∴ store 201 at index 5.

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

4.3 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;
        }
}
```

Binary Heap

A binary tree that has the following properties (called heap properties) is called a heap tree or binary heap.

1. Either it is empty

   Or

2. The key in the root is larger than that in either child

   And

3. Both subtrees have the heap properties.

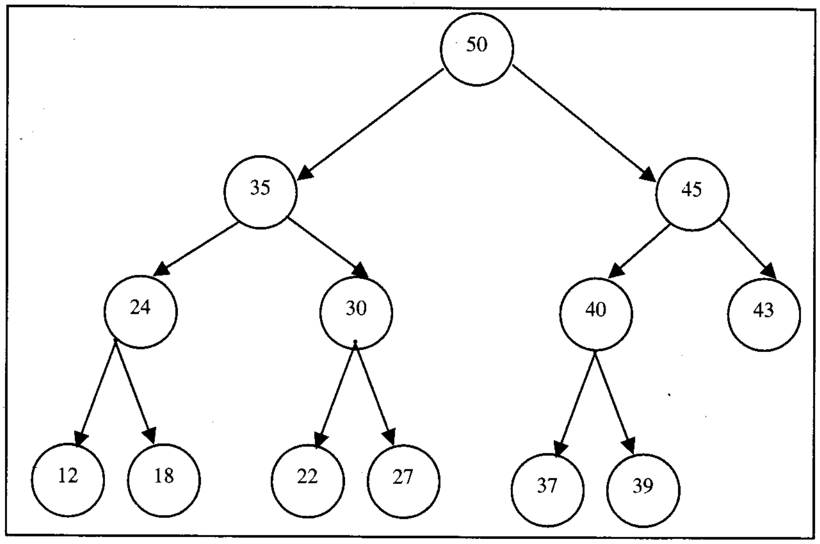
Thus, a heap tree or binary heap can be used as a priority queue where the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must efficiently re-create a single tree with the heap property. Insertion and deletion in a heap tree is very efficient – of the order of O(log n) - as compared to other trees.

Adding a Node to Heap

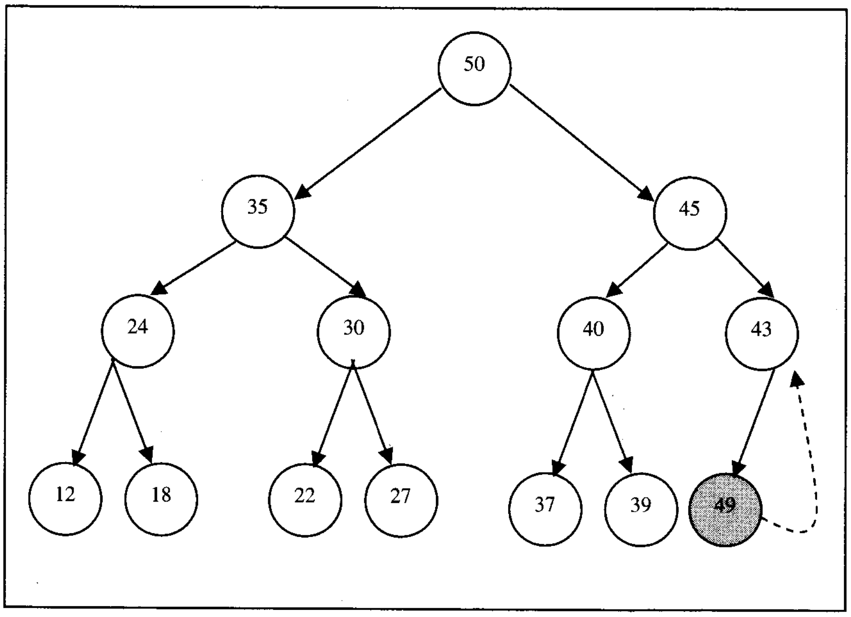
Inserting a node into a heap-tree is relatively straightforward. Because we keep of the largest position n which has been filled so far, we can insert the new element at position n+1 provided there is still room in the array. This will once again give us a complete binary tree, but the heap-tree property might, of course, be violated now. Hence we may have to 'bubble up' the new element. This is done by comparing its priority with that of its parent, and if the new element has higher priority larger, then it is exchanged with its parent. We may have to repeat this process, but once we reach a parent that has bigger or equal priority, we can stop. Inserting a node takes at most log(n) steps because the number of times that we may have to `bubble up' the new element depends on the height of the tree. An algorithm for inserting a new node is listed below:

```
insert(node v)
{
if (n < MAX)
{
n = n + 1;
heap[n] = v;
bubble_up(n);
}
else
report error: out of space;
}

bubble_up(int i)
{
while (not isroot(i) and heap[i] > heap[parent(i)])
{
swap heap[i] and heap[parent(i)];
i = parent(i);
}
}

Boolean isroot(int t)
{
      if (t == 1)
return TRUE;
            else
                  return FALSE;
}
```

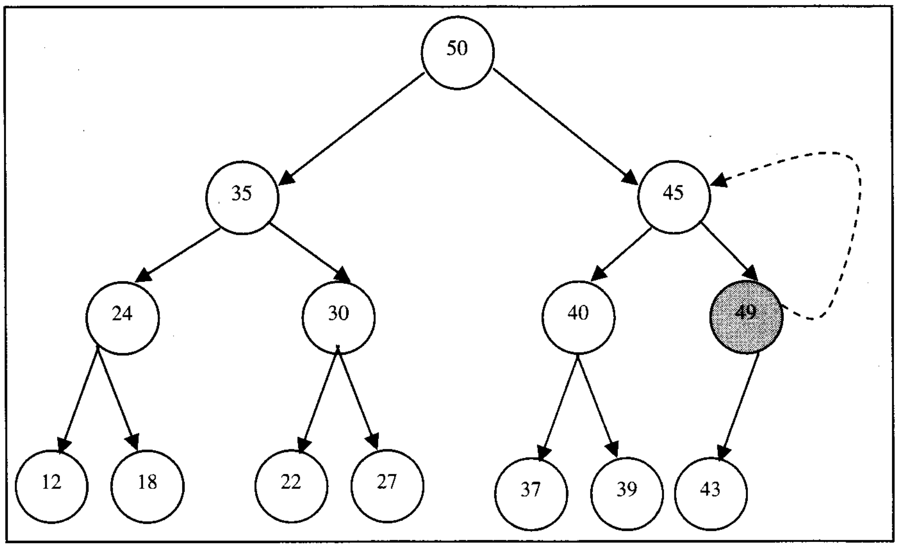
Consider the following heap tree.



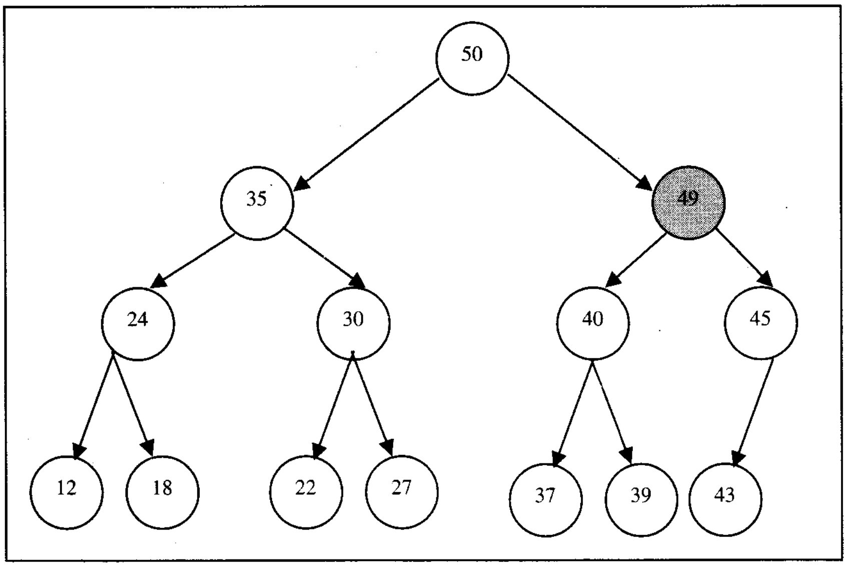
Let us insert a node with data value 49. Since the next position is the left child of the node with value 43, the new node will be added as shown below:



However, this makes the tree violate the heap property. Therefore, 43 must be swapped with 49.



Even now the heap property is not being fulfilled. Therefore, 45 must be swapped with 49. At this point the tree possesses the heap property and thus, we stop.

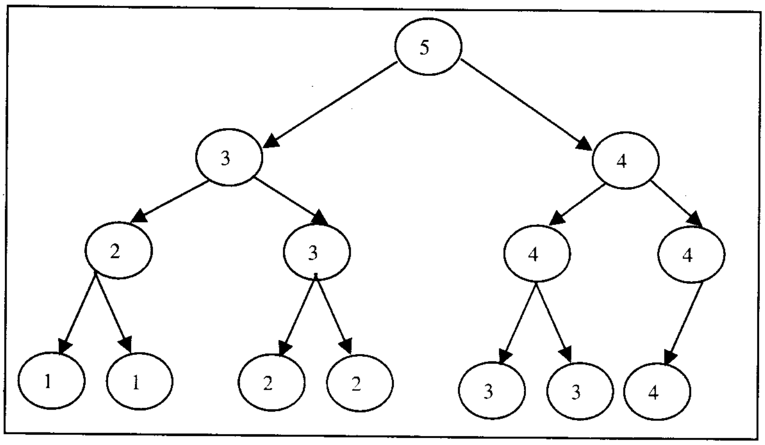


Deleting a Node from Heap

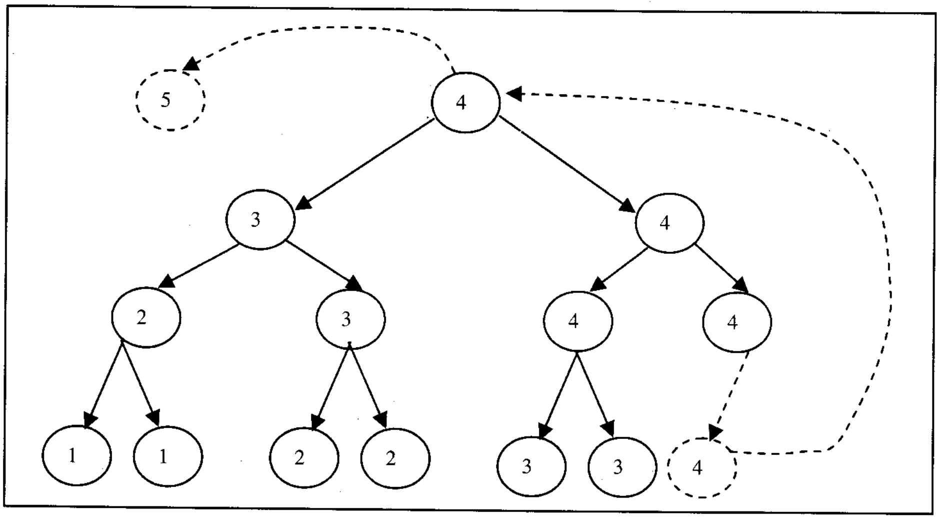
In a heap-tree, only the node with the highest priority (the one at the root) is deleted. We're then left with something which isn't a binary tree at all. We can now 'trickle down' the new root by comparing it to both its children and exchanging it for the largest. This process is then repeated until this element has found its place.

Again, this takes at most log(n) steps. Note that this algorithm does not try to be fair in the sense that if two nodes have the same priority, it is not necessary that the one that has been waiting longer is removed first. A solution to this is to keep some kind of time-stamp on arrivals, or by giving them numbers.

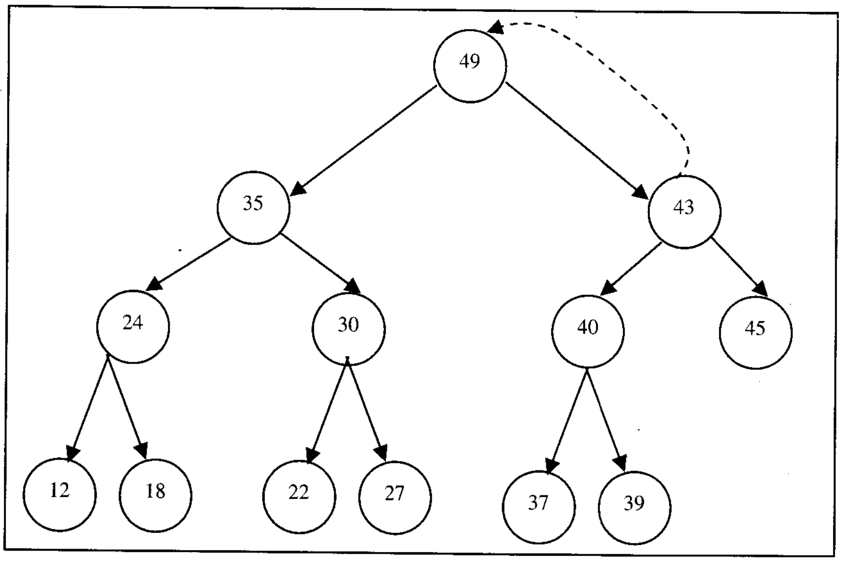
Consider the following heap tree, for illustrating deletion operation.



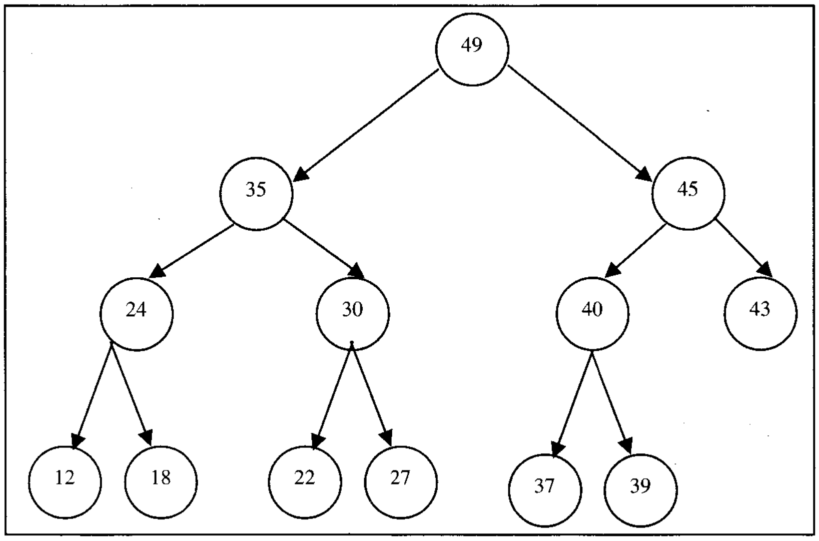
Deleting a node will remove 50 from the root of the heap tree. The empty root must then be filled with the last element of the heap tree (i.e., 43).



However, in doing so, the heap looses its heap property and therefore, it must be rearranged. 43 must be swapped with 49.



Even this does not confirm with the definition of heap. One more swap is necessary - 43 with 45 - resulting in the final heap tree.



Check Your Progress

1. Define rehashing.
2. What is mid-square method?

4.4 LET US SUM UP

- Hashing is a method of directly computing the index of the table by using some suitable mathematical function called hash function.
- 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

Hashing: Hashing is a method of directly computing the index of the table by using some suitable mathematical function called hash function.

Separate Chaining: This is a method of implementing a hash table, in which collisions gets handled automatically.

Priority Queue: A queue in which elements are assigned priorities to determine the order in which they can be retrieved.

4.6 QUESTIONS FOR DISCUSSION

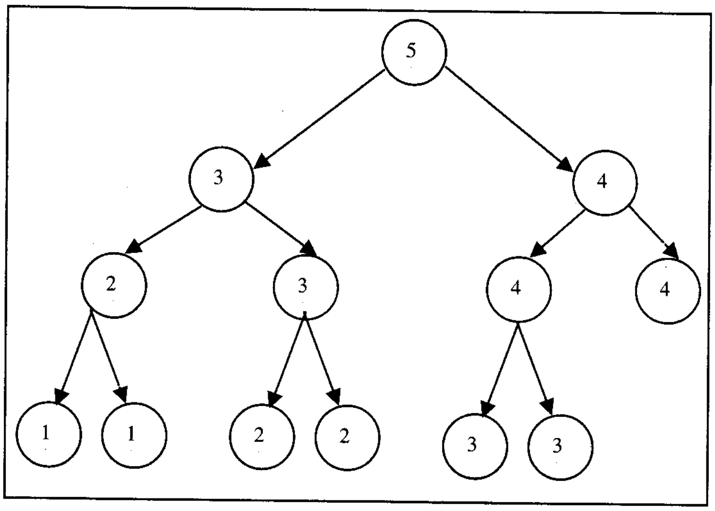
1. Use division method of hashing to store the following values in the hash table of size 12.

   125, 145, 196, 201, 202, 145, 107, 201

Use sequential method for resolving the collisions.

2. How is a heap tree different from a binary tree?

3. Show the step-wise results of inserting a node (36) into the following heap tree.



4. Give an example of a priority queue from everyday life.

Check Your Progress: Model Answers

1. Rehashing is a method of collision handling. In this method we find an alternative empty location by modifying the hash function, and applying the modified hash function to the colliding symbol.

2. In Mid square method 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.

4.7 SUGGESTED READINGS

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

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

How to Solve it by Computer; RG Dromey; Cambridge University Press

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

UNIT III

LESSON

5

SORTING

CONTENTS

5.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss the concept of sorting
- Discuss insertion sort, shell sort, heap sort, merge sort, and quick sort

5.1 INTRODUCTION: SORTING PRELIMINARIES

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data or alphabetically, with character data.

Let P be a list of n elements P_1, P_2, P_3...P_n in memory. Sorting P refers to the operation of rearranging the contents of P so that they are increasing in order (numerically or lexicographically), that is,

$$P_1 \leq P_2 \leq P_3 \ldots \leq P_n$$

Since P has n elements, there are n! ways that the contents can appear in P. These ways correspond precisely to the n! permutations of 1, 2, ...n. Accordingly, each sorting algorithm must take care of these n! possibilities.

Classification of Sorting

Sorting can be classified in two types:

(i) Internal Sorting

(ii) External Sorting

- ***Internal Sorting:*** If the records that have to be sorted are in the internal memory.
- ***External Sorting:*** If the records that have to be sorted are in secondary memory.

Efficiency Considerations

The most important considerations are:

1. The amount of machine time necessary for running the program.
2. The amount of space necessary for the program.

In most of the computer applications one is optimized at the expense of another. The actual time units to sort a file of size n varies from machine to machine, from one program to another, and from one set of data to another.

So, we find out the corresponding change in the amount of time required to sort a file induced by a change in the file size n. The time efficiency is not measured by the number of time units required but rather by the number of critical operations performed.

Critical operations are those that take most of the execution time. For example, key comparisons (that is, the comparisons of the key of two records in the file to determine which is greater), movement of records or pointer to record, or interchange of two records.

5.2 'O' NOTATION

Given two functions f(n) and g(n), we say that f(n) is *of the order of g(n)* or that f(n) is O(g(n)) if there exists positive integers a and b such that

$f(n) \leq a * g(n)$ for $n \geq b$.

For example, if $f(n) = n^2 + 100n$ and $g(n) = n^2$

f(n) is O(g(n)), since $n^2 + 100n$ is less than or equal to $2n^2$ for all n greater than or equal to 100. In this case a = 2 and b = 100.

The same f(n) is also $O(n^3)$, since $n^2 + 100 n$ is less than or equal to $2n^3$ for all n greater than or equal to 8. If f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)). For example, $n^n + 100n$ is $O(n^2)$ and n^2 is $O(n^3)$, then $n^2 + 100n$ is $O(n^3)$ for (a = 1, b = 1). This is called the transitive property.

If the function is C * n then its order will be $O(n^k)$ for any constant c and k. As c * n is less than or equal to $c * n^k$ for any n ^{3} 1 (i.e a = c, b = 1). If f(n) is n^k then its order will be $O(n^{k+j})$ for any j ^{3} O(for a = 1, b = 1).

If f(n) and g(n) are both O(h (n)), the new function f(n) + g(n) is also O(h(n)). If f(n) is any polynomial whose leading power is k [i.e. $f(n) = c_1 * n^k + c_2 * n^{k-1} + .. + c_k * n + c_{k+1}$]

f(n) is $O(n^k)$.

Algorithm Efficiency in Logarithmic Function

Let $\log_m n$ and $\log_k n$ be two functions. Let xm be $\log_m n$ and xk be $\log_k n$ then

$m^{xm} = n$ and $k^{xk} = n$

Since if $m^x = n$

So that $\log_m n = x$

So that $m^{xm} = k^{xk}$

Taking $\log_m$ of both sides.

$xm = \log_m (k^{xk})$

Now it can easily be shown that $\log_z (x^y)$ equals y $*\log_Z x$ for any x, y and z, so that the last equation can be written as

$\log_m n = xk * \log_m k$ [since $xm = \log_m n$]

or as

$\log_m n = (\log_m k) * \log_k n$ [since $xk = \log_k n$]

Thus $\log_m n$ and $\log_k n$ are constant multiples of each other.

If f(n) = c * g(n) then f(n) is O(g(n)) thus $\log_m n$ is $O(\log_k n)$ and $\log_k n$ is $O(\log_m n)$ for any m and k.

The following facts establish an order hierarchy of functions:

- C is O(1) for any constant C.
- C is O(log n), but log n is not O(1).
- C * $\log_k n$ is O(log n) for any constant C, K.
- C * $\log_k n$ is O(n), but n is not O(log n).
- C * n^k is $O(n^k)$ for any constant C, K.
- C * n^k is $O(n^{k+j})$, but n^{k+j} is not $O(n^k)$.
- C * n * $\log_k$ n is O(n log n) for any constant C, K.
- C * n* $\log_k$ n is $O(n^2)$, but n^2 is not O(n log n).
- C * n^j * $\log_k$ n is $O(n^j \log n)$ for any constant c, j, k.
- C * n^j * ($\log_k n$ is $O(n^{j+1})$, but n^{j+1} is not $O(n^j \log n)$.
- C * n^j * ($\log_k n$)l is $O(n^j (\log n)^l)$ for any constant c, j, k, l.
- C * n^j * ($\log_k n$)l is $O(n^{j+1})$ but n^{j+1} is not $O(n^j (\log n)^l)$.

5.3 INSERTION SORT

An insertion sort is one that sorts a set of records by inserting records into an existing sorted file. Suppose an array A with n elements A[1], A[2],.......A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted sub array A[1], A[2],... A[K-1].

Example: Sort the following list using the insertion sort method: 4, 1, 3, 2, 5

i)	4					Place 4 in 1st position
ii)	1	4				1 < 4, therefore insert prior to 4
iii)	1	3	4			3 > 1, insert between 1 & 4
iv)	1	2	3	4		2 > 1, insert between 1 & 3
v)	1	2	3	4	5	5>4, insert after 4

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.

Algorithm to Implement Insertion Sort

```
insert sort (x, n)
int x[ ], n;
{
        int i, k, y;
        for (k = 1; k < n; k++)
        {
                y = x [k];
                for (i = k-1; i > = 0 && y < x [i]; i --)
                x [i+1] = x[i];
                x [i+1] = y;
        }
}
```

Analysis of Insertion Sort

If the initial file is sorted, only one comparison is made on each pass, so that sort is O(n). If the file is initially sorted in reverse order, the sort is O(N^2), since the total number of comparisons are:

$(n - 1) + (n - 2) + ... + 3 + 2 + 1 = (N - 1) * N/2$

which is O(N^2).

The closer the file is to sorted order, the more efficient the simple insertion sort becomes. The space requirements for the sort consists of only one temporary variable, Y. The speed of the sort can be improved somewhat by using a binary search to find the proper position for x[k] in the sorted file.

x[0], x[1] ... x[k - 1]

This reduces the total number of comparisons from $O(N^2)$ to $O(n \log_2 (n))$. However even if the correct position i for x[k] is found in $O(\log_2 (n))$ steps, each of the elements x[i+1] ... x[k - 1] must be moved by one position. This latter operation to be performed N times requires $O(N^2)$ replacement.

5.4 SHELL SORT

More significant improvement on simple insertion sort than binary or list insertion can be achieved using the Shell Sort (Diminishing Increment Sort). This method sorts separate subfiles of the original file.

These subfiles contain every kth element of the original file. The value of k is called an increment. For example, if k is 5, the subfile consisting of x[0], x[5], x[10],... is first sorted. Five subfiles, each containing one fifth of the element of the original file are sorted in this manner. These are:

Subfile 1:	x[0]	x[5]	x[10] ....
Subfile 2:	x[1]	x[6]	x[11] ....
Subfile 3:	x[2]	x[7]	x[12] ....
Subfile 4:	x[3]	x[8]	x[13] ...
Subfile 5:	x[4]	x[9]	x[14] ...

The i^{th} element of the j^{th} subfile is x[(i - 1)* 5+j - 1]. If a different increment is chosen, the k subfiles are divided so that the i^{th} element of the j^{th} subfile is x[(i - 1) * k+j - 1].

After the k subfiles are sorted (usually by simple insertion), a new smaller value of k is chosen and the file is again partitioned into a new set of subfiles. Each of these larger subfiles are sorted and the process is repeated yet again with an even smaller value of k.

Eventually, the value of k is set to 1, so that the subfile consisting of the entire file is sorted.

A decreasing sequence of increments is fixed at the start of the entire process. The last value in this sequence must be 1.

For example, if the original file is:

25 27 48 37 12 92 86 33

and the sequence (5, 3, 1) is chosen, the following subfiles are sorted on each iteration:

First iteration (increment = 5)

(x[0], x[5])

(x[1], x[6]

(x[2], x[7]

(x[3])]

(x[4])

Second iteration (increment = 3)

(x[0], x[3], x[6])

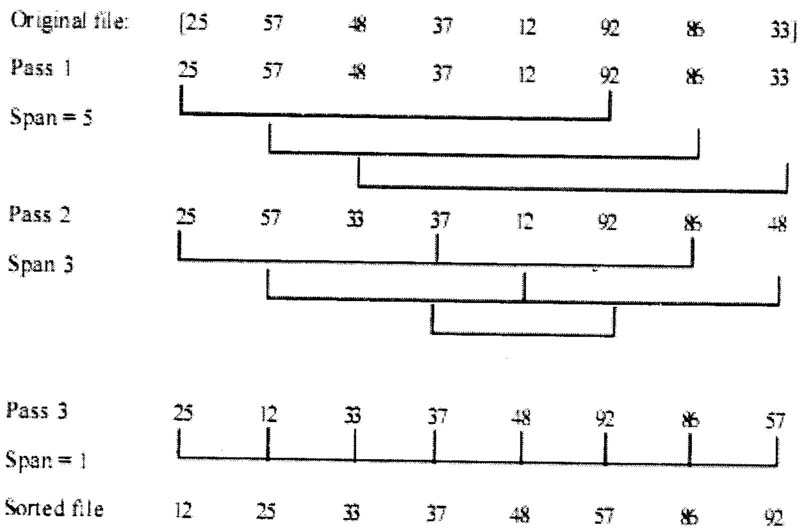
(x[1], x[4], x[7])

(x[2], x[5])

Third iteration (increment = 1)

(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])

Example: The following figure illustrates the shell sort on this sample file:


Original file: [25 57 48 37 12 92 86 33]
Pass 1 25 57 48 37 12 92 86 33
Span = 5
Pass 2 25 57 33 37 12 92 86 48
Span 3
Pass 3 25 12 33 37 48 92 86 57
Span = 1
Sorted file 12 25 33 37 48 57 86 92

Algorithm to Implement Shell Sort

```
void shellsort (int x[ ], int n, int increments [ ], int numeric)
{
      int incr, j, k, span, y;
              for (incr = 0; incr <numeric; incr ++)
              {
                      span = increments [incr];
                      for (j = span; j <n; j++)
                          /* insert element x[j] into
                          /* its proper position with its */
                          /* subfile * /
                          y = x[j];
                      for (x = j-span; k > = 0 && y < x [k]; k-=span)
                         x[k+span] = x[k];
                          x[k+span]=y;
              }     /* end for */
          } /* end for * /
} /* end shell sort */
```

Analysis of Shell Sort

Since the first increment used by the shell sort is large, the individual subfiles are quite small, so that the simple insertion sort on those subfiles are fairly fast. Each sort of a subfile causes the entire file to be more nearly sorted. Thus, although successive passes of the shell sort use smaller increments and therefore, deal with larger subfiles, those subfiles are almost sorted due to the actions of previous passes.

Thus the insertion sort on these subfiles are also quite efficient. The actual time requirement for a specific sort depends on the number of elements in the array increments and on their actual values.

It has been shown that order of the shell sort can be approximated by $O(n * (\log (n^2))$ if an appropriate sequence of increments is used. For other series, the running time $O(n^{1.5})$.

5.5 HEAP SORT

The algorithm which we now formulate is a combination of algorithms by Floyd and Williams. In general a heap which represents a table of n records satisfies the property

$K_j \leq K_i$; for $2 \leq j < n$ and $i = [j/2]$. The binary tree is allocated sequentially such that the indices of the left and right sons (if they exist) of record 1 are 2i and 2i+1 respectively.

A complete binary tree is said to satisfy the "Heap Condition" if the key of each node is greater than or equal to the keys in its children. Thus, the root node will have the largest key value. Trees can be represented as arrays, by first numbering the nodes (starting from the root) from left to right.

The key value of nodes are then assigned to array positions whose index is given by the number of the node.

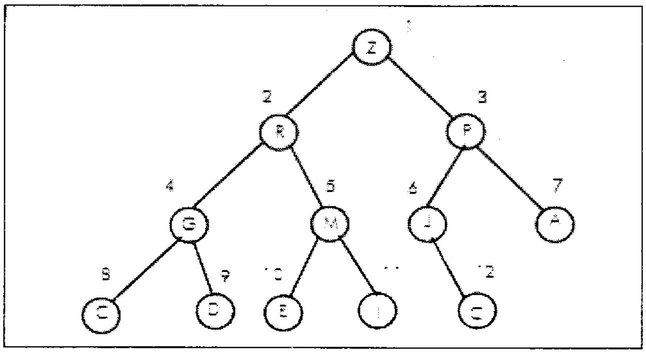
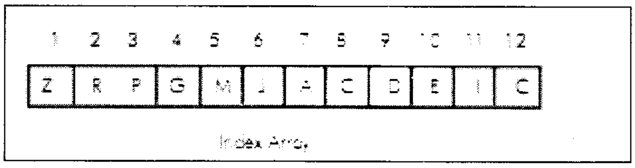


Figure 5.1: Heap 1



1	2	3	4	5	6	7	8	9	10	11	12
Z	R	P	G	M	J	A	C	D	E	I	C

Index Array

The relationships of the node can also be determined from the array representation. If a node is at position j, its children will be at positions 2j and 2j+1. Its parent will be at position j/2. A heap is a complete binary tree in which each node satisfies the heap condition, represented as an array. The operation on a heap works in two steps:

(i) The required node is inserted/deleted/or replaced.

(ii) First step may cause violation of the heap condition so the heap is traversed and modified to rectify any such violations.

5.5.1 Insertion in Heap

Consider an insertion of node R in the heap 1.

1. Initially R is added as the right child of J and given the number 13.
2. But R J, the heap condition is violated.
3. Move R upto position 6 and move J to position 13.
4. R P, therefore, the heap condition is still violated.
5. Swap R and P.
6. The heap condition is now satisfied by all nodes.

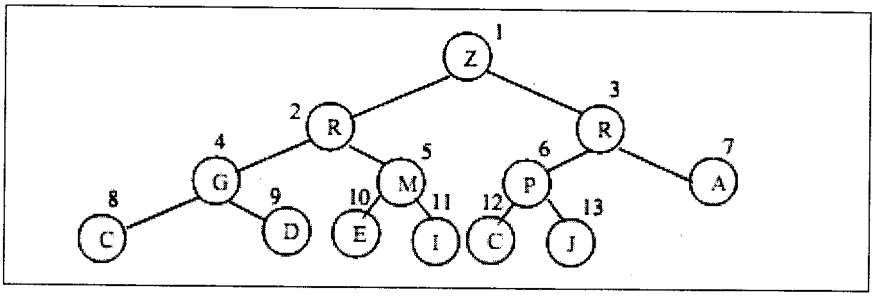


Figure 5.2: Heap 2

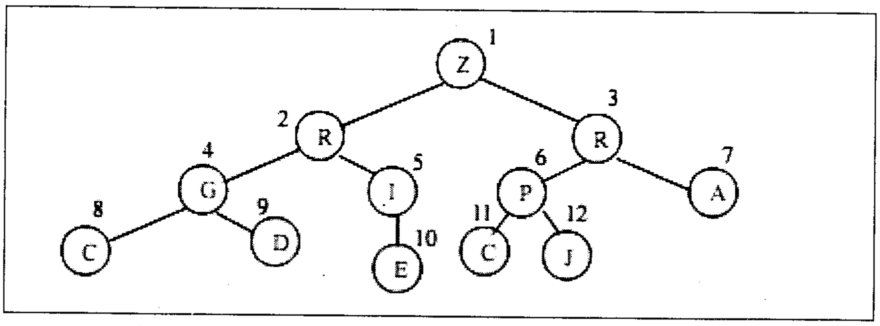


Figure 5.3: Heap 3

A general algorithm for creating a heap is given below:

1. Repeat through step 7 while there still is another record to be placed in the heap.
2. Obtain child to be placed at leaf level.
3. Obtain child to be parent for this child.
4. Repeat through step 6 while the child has a parent & the key of the child is greater than that of its parent.
5. Move parent down to position of child.
6. Obtain position of new parent for the child.
7. Copy child record into its proper place.

More formal procedure Create_Heap (K,N) is given below. K is a given table containing the keys of the N records of a table, this algorithm creates a heap as previously described. The index variable Q controls the number of insertions which are to be performed. The integer variable J denotes the index of the parent of key k[I]. Key contains the key of the record being inserted into an existing heap.

```
1.  [Build Heap]
    Repeat through step 7 for Q =2,3.....N
2.  [Initialize construction phase]
    IfQ
    KEYfK[Q]
3.  [Obtain parent of new record]
    JfTrunc (I/2)
4.  [Place new record in existing heap]
    Repeat through step 6 while I>1 and KEY > K[J]
5.  [Interchange record]
    K[1] fK[J]
6.  [Obtain next parent]
    IfJ
    JfTrunc (I/2)
    if J <1
    then Jf1
7.  [Copy new record into its proper place]
    K [I] fKEY
8.  [Finished]
    Return
```

5.5.2 Deletion from Heap

When we delete any node from the tree, the child with greater value will be promoted to replace that node.

Consider deletion of M from heap 2.

The larger of M's children are promoted to 5.

An efficient sorting method is based on the heap construction and node removal from the heap in order. This algorithm is guaranteed to sort N elements in N log N steps.

5.6 CONSTRUCTION OF HEAP

Two methods of heap construction and then removal in order from the heap to sort the list are as follows:

Top-down Heap Construction

- Insert items into an initially empty heap, keeping the heap condition intact in all steps.

Bottom-up Heap Construction

- Build a heap with the items in order presented.
- From the right most node modify to satisfy the heap condition.

For example, to build a heap of the following using both methods of construction.

Let the input string be "PROFESSIONAL".

5.6.1 Top-down Construction

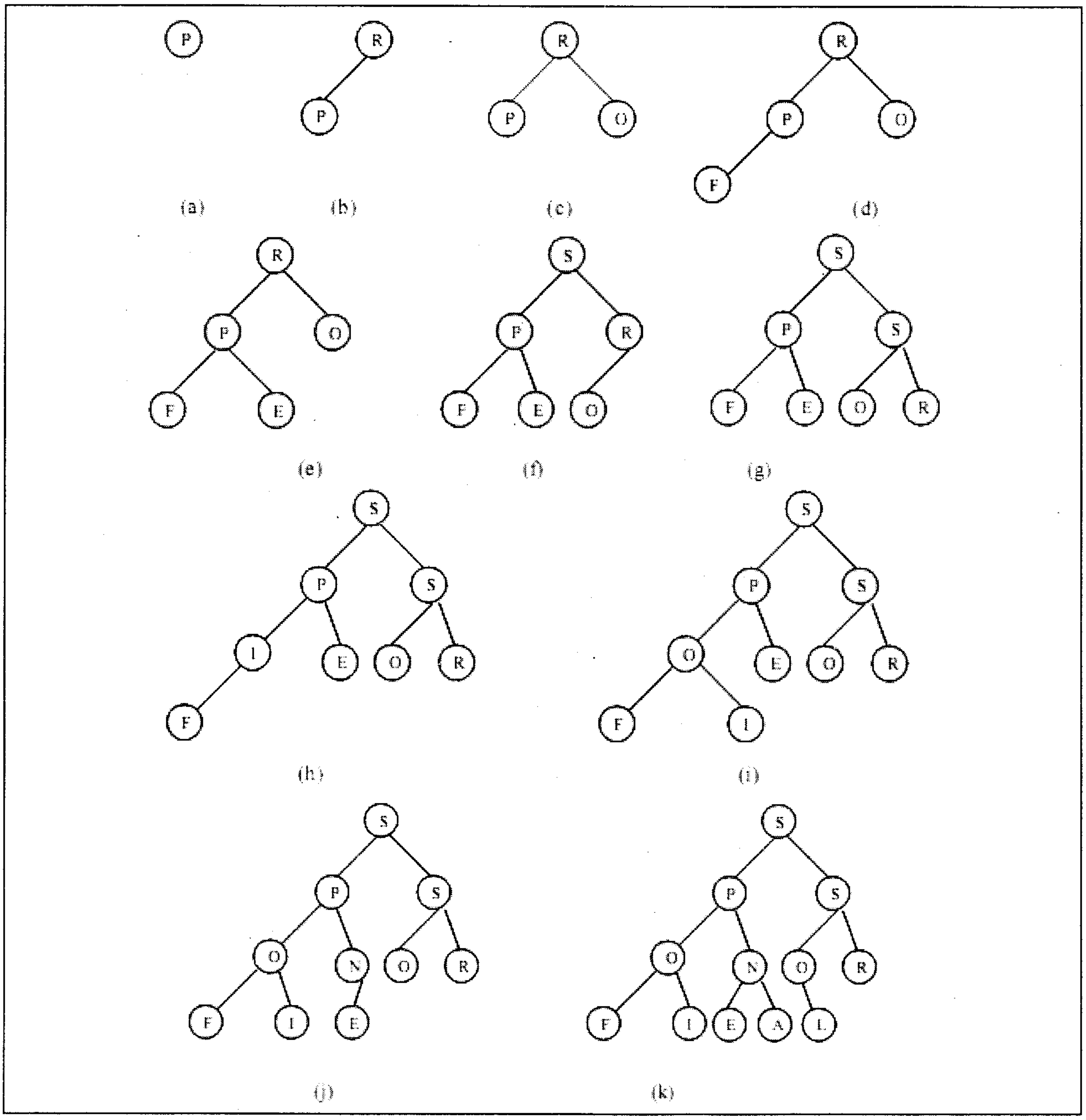


Figure 5.4

5.6.2 Bottom-up Construction

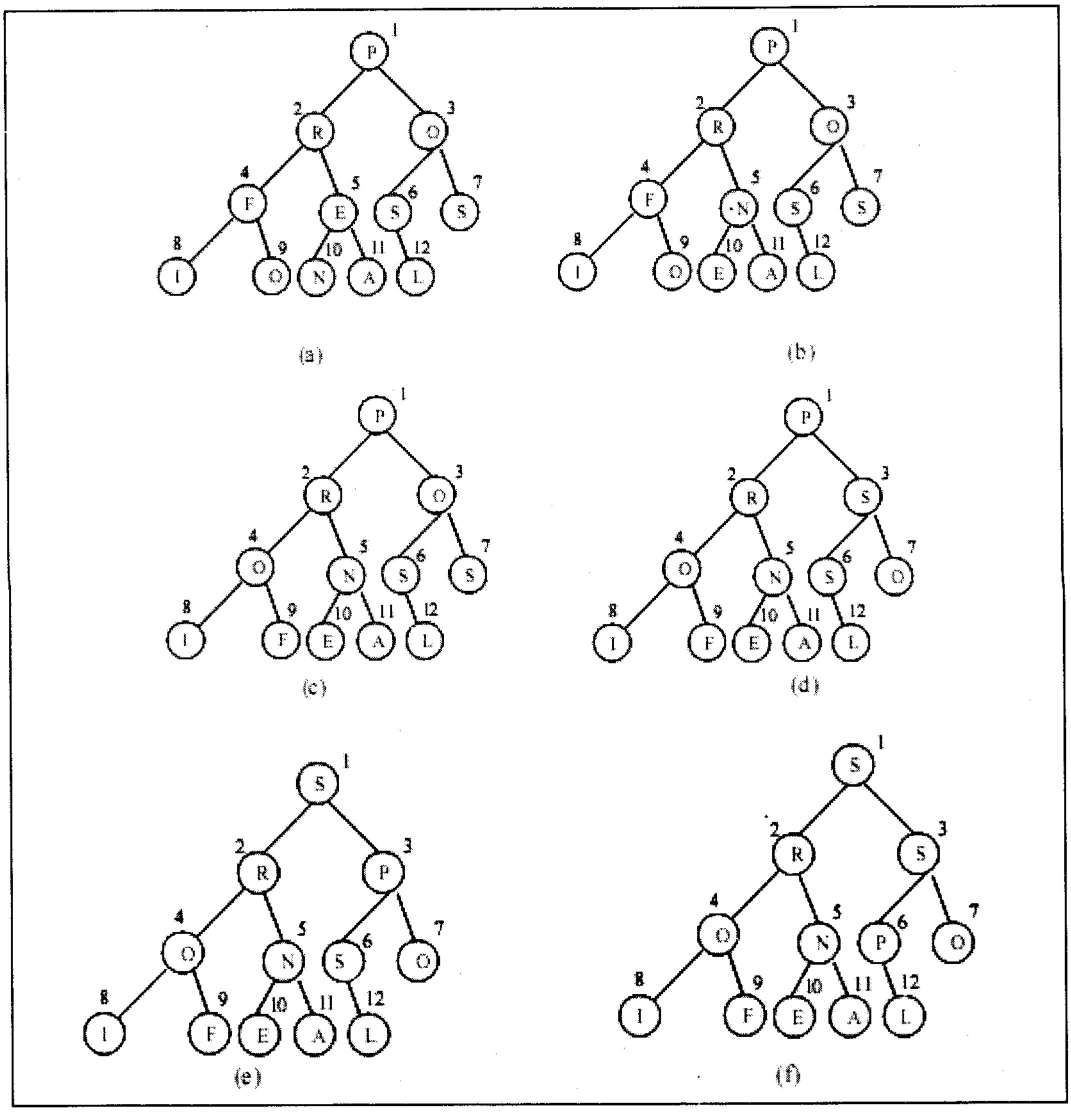
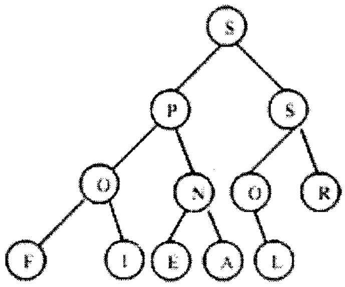


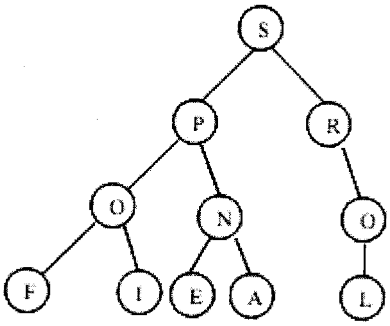
Figure 5.5

5.7 SORTING USING HEAP

The sorted elements will be placed in x[ ] an array of size 12.

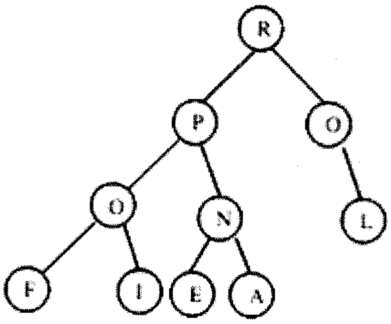


(i) Remove 'S' and store in x[12].



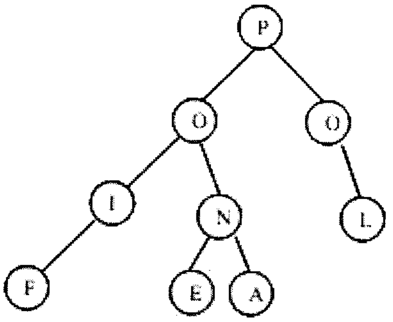
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=												S

(ii) Remove 'S' and store in x[11].



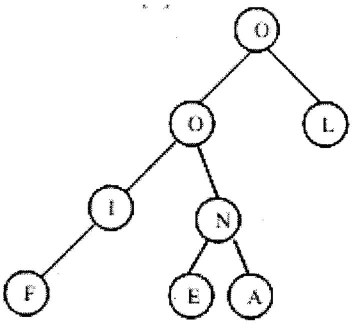
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=											S	S

(iii) Remove 'R' and store in x[10].



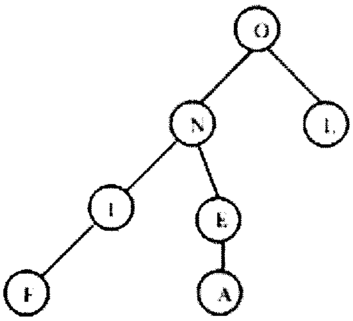
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=										R	S	S

iv) Remove 'P; and store in x [9].



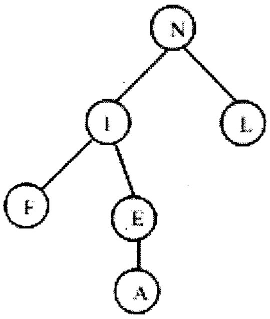
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=									P	R	S	S

(v) Remove 'O' and store in x[8].



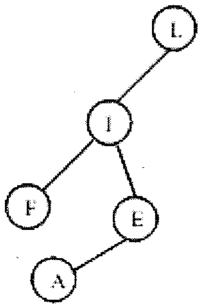
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=								O	P	R	S	S

(vi) Remove 'O' and store in x[7].



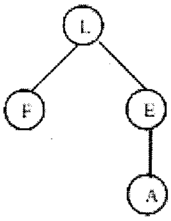
	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=							O	O	P	R	S	S

(vii) Remove 'N' and store in x[6].



	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=						N	O	O	P	R	S	S

(viii)Remove 'L' and store in x[5].



	1	2	3	4	5	6	7	8	9	10	11	12
x[ ]=					L	N	O	O	P	R	S	S

(ix) Similarly, the remaining 3 nodes are removed and the heap modified to get the sorted list AEFILNOOPRSS.

5.8 MERGE SORT

The operation of sorting is closely related to the process of merging. This sorting method uses merging of two ordered lists which can be combined to produce a single sorted list. This process can be accomplished easily by successively selecting the record with the smallest key occurring in either of the lists and placing this record in a new table, thereby creating an ordered list.

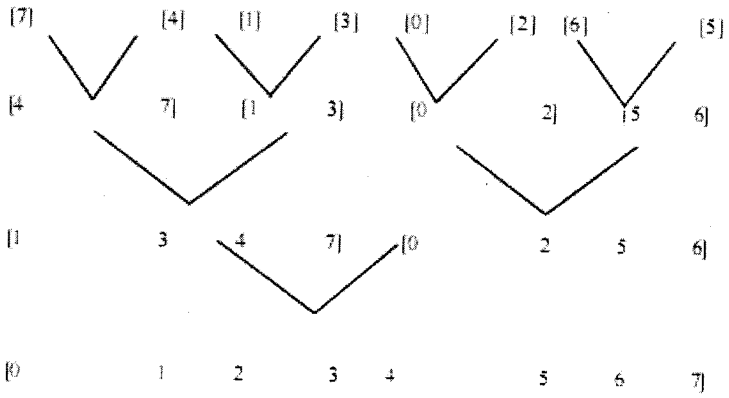
Merge sort is one of the divide and conquer class of algorithm. The basic idea is to:

- Divide the list into a number of sublists.
- Sort each of these sublists.
- Merge them to get a single sorted list.

Two-way merge sort divides the list into two, sorts the sublist and then merges them to get the sorted list, also called concatenate sort.

Multiple merging can also be accomplished by performing a simple merge repeatedly. For example, if we have 16 lists to merge, we can first merge them in pairs. The result of this first step yields eight tables which are again merged in pairs to give four tables. This process is repeated until a single table is obtained. In this example, four separate passes are required to yield a single list. In general, k separate passes are required to merge 2k separate lists into a single list.

Example



Algorithm for Merging Two Sorted Files to Get Third Sorted File

Let (X_1, ..., X_m) and (X_{m+1}, ..., X_n) be two sorted files. Let the merged file is (Z_1, ..., Z_n).

Procedure:

```
merge (x, z, l, m, n)
        int x[ ], z[ ], l, m, n;
{
        /* (x[l], ..., x[m]) and (x[m+1], ..., x[n]) are two sorted lists with
keys, such that x[l] £ ... £ x[m] * /
int i, j, k, t;
        i = l;
        k = l; /* i, j & k are positions in these files * /
        j = m+1;
while (( i £ m) & & (j < = n))
        {
                if (x[i] < = x[j])
                        {
                                z[k] = x[i];
                                i ++;
                        }
                else
                        {
                                z[k] = x [j];
                                j++;
                        }
                        k = k+1;
                }
If (i > m)
{
            for (t = j; t < n; t++)
                  z[k + t - j] = x[t];
      }
else
      {
            for (t = i; t < m; t++)
                  z[k + t - i] = x[t];
      }
}
```

The above algorithm for merge sort has one important property that after pass K, the array A will be positioned into sorted subarrays of exactly L = 2^K elements (except the last subarray).

By dividing n (size of array A) by 2*L, we get the quotient Q which is number of pairs of sorted subarrays, of size L, i.e.

Q = INT (N/2*L)

S=2*L*Q will be the total number of elements in the Q pairs of subarrays. R=N - S denotes the number of remaining elements.

Analysis of Algorithm 'MERGE'

The while loop is iterated at most n - l + 1 times. The if statement moves at most one record per iteration (considering sorting of records), the total time is therefore O(n - l + 1). If records are of length m then this time is O(m(n - l + 1)).

Analysis of 'MSORT'

On the ith pass the files being merged are of size 2^{i-1}. Consequently, a total of [$\log_2$ (N)] passes are made over the data. Since two files can be merged in linear time (algorithm 'MERGE'), each pass of merge sort takes O(N) time. As there are [$\log_2$ (N)] passes, the total computing time is O(N * $\log_2$ N).

5.9 QUICK SORT

Quick sort is also known as partition exchange sort. An element (a) is chosen from a specific position within the array such that x is partitioned and a is placed at position j and the following conditions hold:

1. Each of the elements in position 0 through j-1 is less than or equal to a.
2. Each of the elements in position j+1 through n-1 is greater than or equal to a.

The purpose of the Quick Sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore reduces unnecessary swaps, and moves an item a great distance in one move. A pivotal item near the middle of the array is chosen, and then items on either side are moved so that the data items on one side of the pivot are smaller than the pivot, whereas those on the other side are larger, the middle (pivot) item is in its correct position. The procedure is then applied recursively to the parts of the array, on either side of the pivot, until the whole array is sorted.

Example:

If an initial array is given as:

```
    25    57    48    37    12    92    86    33
```

and the first element (25) is placed in its proper position, the resulting array is:

```
    12    25    57    48    37    92    86    33
```

At this point, 25 is in its proper position in the array (x[1]), each element below that position (12) is less than or equal to 25, and each element above that position (57, 40, 37, 92 86 and 33) is greater than or equal to 25.

Since 25 is in its final position the original problem has been decomposed into the problem of sorting the two subarrays.

(12) and (57 48 37 92 86 33)

First of these subarrays has one element so there is no need to sort it. Repeating the process on the subarray x[2] through x[7] yields:

```
    12    25    (48   37    33)   57    (92   86)
```

and further repetitions yield

```
    12    25    (37   33)   48    57    (92   86)
    12    25    (33)  37    48    57    (92   86)
    12    25    33    37    48    57    (92   86)
    12    25    33    37    48    57    (86)  92
    12    25    33    37    48    57    86    92
```

Algorithm to Implement Quick Sort

```
        qsort (int x[ ], int m, int n)
{
        /* m and n contains upper and lower bounds of array * /
        /* array has to be sorted in non-decreasing order */
        int i, j, k, t;
        if (m < n)
              {
                      i = m;
                      j = n+1;
                      k = x [m];             /* key */
                      while(1)
                            {
                                  do
                                        {
                                              i = i+1;
```

```
                        }
                              while (x [i] < k);
                  do
                        {
                              j = j - 1;
                        }
                              while (x [j] > k);
                  if (i < j)
                        {
                              t = x [i];
                              x [i] = x [j];
                              x [j] = t;
                        }
                  else break;
            }
            }
                        t = x [m];
                        x [m] = x [j];
                        x [j] = t;
                        q sort (x, m, j -1);
                        q sort (x, j+1; +n);
}
```

We shall illustrate the mechanics of this method by applying it to an array of numbers. Suppose, the array A initially appears as:

(15, 20, 5, 8, 95, 12, 80, 17, 9, 55)

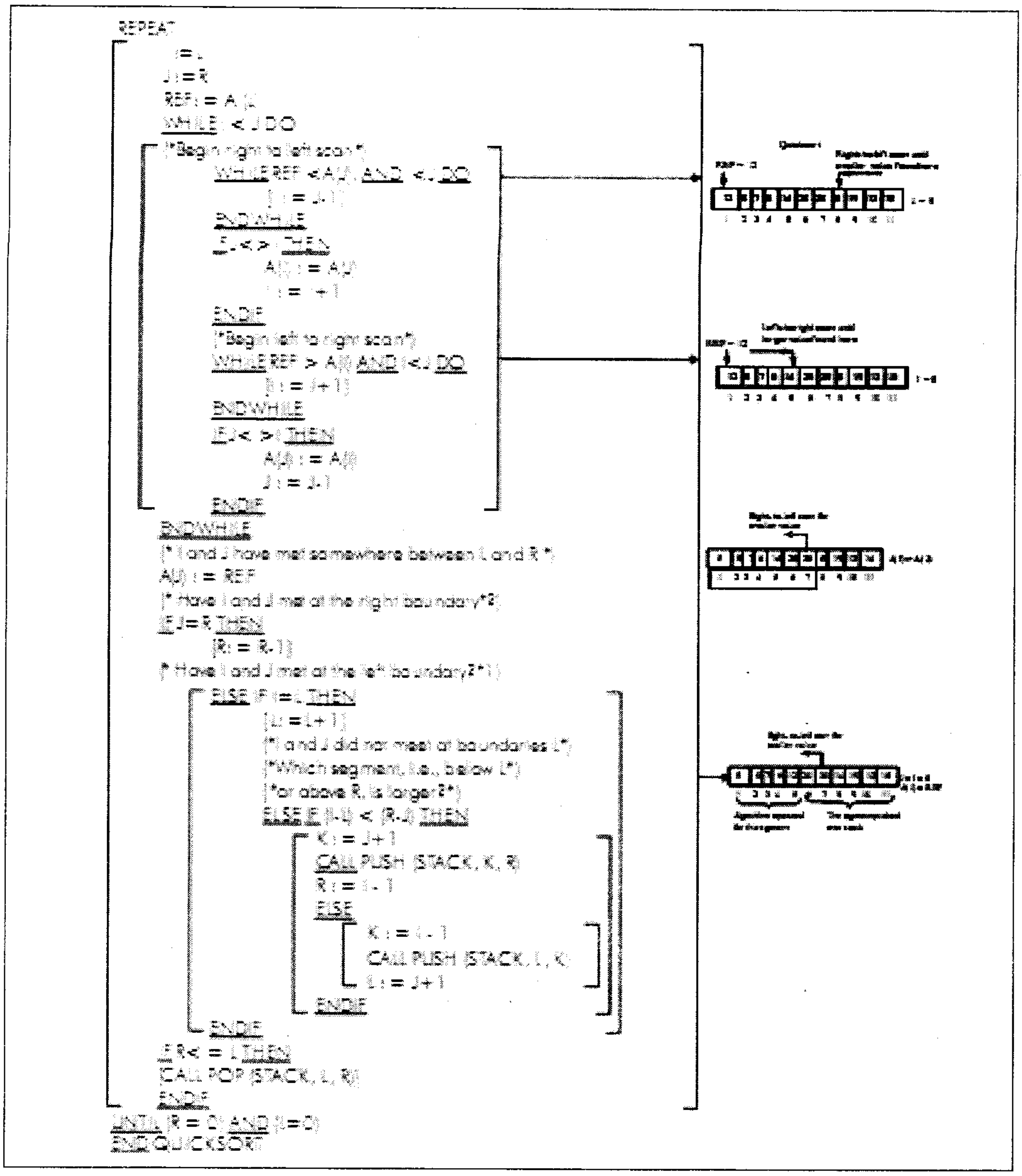
Figure 5.6 shows a quick sort applied to this array.

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
15	20	5	8	95	12	80	17	9	55
9	20	5	8	95	12	80	17	()	55
9	()	5	8	95	12	80	17	20	55
9	12	5	8	95	()	80	17	20	55
9	12	5	8	()	95	80	17	20	55
9	12	5	8	15	95	80	17	20	55

Figure 5.6: Quick Sort of an Array

The following steps are involved:

1. Remove the Ist data item, 15, mark its position and scan the array from right to left, comparing data item values with 15. When you find the Ist smaller value, remove it from its current position and put in position A(1). This is shown in line 2 of Figure 5.7.
2. Scan line 2 from left to right beginning with position A(2), comparing data item values with 15. When you find the Ist value greater than 15, extract it and store in the position marked by parentheses in line 2. This is shown in line 3 in the Figure 5.7.
3. Begin the right to left scan of line 3 with position A(8) looking for a value smaller than 15. When you find it, extract it and store it in the position marked by the parentheses in line 3 of Figure 5.7.



```
REPEAT
    I := L
    J := R
    REF := A(L)
    WHILE I < J DO
        (*Begin right to left scan*)
            WHILE REF < A(J) AND I < J DO
                (J := J-1)
            ENDWHILE
            IF J <> I THEN
                A(I) := A(J)
                I := I+1
            ENDIF
            (*Begin left to right scan*)
            WHILE REF > A(I) AND I < J DO
                (I := I+1)
            ENDWHILE
            IF J <> I THEN
                A(J) := A(I)
                J := J-1
            ENDIF
    ENDWHILE
    (* I and J have met somewhere between L and R *)
    A(J) := REF
    (* have I and J met at the right boundary*?)
    IF J = R THEN
        (R := R-1)
    (* Have I and J met at the left boundary?*))
        ELSE IF I = L THEN
            (L := L+1)
            (*I and J did not meet at boundaries L*)
            (*Which segment, i.e., below L*)
            (*or above R, is larger?*)
            ELSE IF (I-L) < (R-J) THEN
                K := J+1
                CALL PUSH (STACK, K, R)
                R := I-1
                ELSE
                    K := I-1
                    CALL PUSH (STACK, L, K)
                    L := J+1
                ENDIF
        ENDIF
    IF R <= L THEN
    CALL POP (STACK, L, R)
    ENDIF
UNTIL (R = 0) AND (L=0)
END QUICKSORT
```

Figure 5.7

4. Begin scanning line 4 from left to right at position A(3), find a value greater than 15, remove it, mark its position, and store it inside the parentheses in line 4. This is shown in line 5 of Figure 5.7.

5. Now, when you scan line 5 from right to left beginning at position A(7), you find no value smaller than 15. Moreover, you come to a parentheses position, position A(5). This is the location to put the Ist data item, 15, as shown in line 6 of Figure 5.7. At this stage 15 is in its correct place relative to final sorted array.

Analysis of Quick Sort

Assumption: File is of size n where n is a power of 2.

Say n = $2^{m,}$ so that m = $\log_2$n.

Average Case Behaviour

Proper position for the pivot always turns out to be exact middle of the subarray. There will be approximately n comparisons on the first pass after which file will split into two subfiles each of size n/2 approximately.

For each of these two files there will be approximately n/2 comparisons. So, after halving the subfiles m times, there are n files of size 1. Thus, the total number of comparisons fcr the entire sort is approximately:

n+2 * (n/2) + 4 * (n/4) + 8 * (n/8) + .. + n* (n/n)

or

n + n + n + n + .. n (m terms)

There are m terms because the file is divided m times. Thus, the total number of comparisons are:

O(n * m) or O(n log n) [as m = $\log_2$n]

Worst Case Behaviour

The worst case occurs when the first pivot fails to split the list. This happens when the original file is already sorted. If, for example, x[b] is in its correct position, the original file is split into subfiles of sizes O and n - 1.

If this process continues, a total of n – 1 subfiles are sorted, the first of size n; the second of size (n – 1) and so on. Total number of comparisons to sort the entire file are n + (n – 1) + (n – 2) + .. + (2) which is O(n^2)

Thus, the quick sort works best with completely unsorted files and worst for files that are completely sorted.

Check Your Progress

1. Define sorting. Name its various categories.
2. Fill in the blanks:
   (a) In a heap, the ——— node has the largest key value.
   (b) In merge sort, total passes are ——— and the total computing times is ————.

5.10 LET US SUM UP

- Sorting is an important operations associated with any data structure.
- Efficient and reliable data processing depends upon sorted data.
- The internal and external sorting methods have their relative efficiencies in different applications.
- In quick sort an element x is chosen from a specific position within the array such that each element in position 0 through that element is less than or equal to x and each of the elements in position greater than the position of x is greater than or equal to x.
- With a large size n and long keys, heap sort and merge sort can be used. With a large size n and short keys radix sort can be used.
- Heap is a binary tree with a condition that every node has a larger key value than its left and right child.
- While representing such a tree in array, if a node is placed at ith index then its left child will be at 2i and right child will be at 2i+1. Hence, any nodes parent will be at i/2.
- In merge sort we split the array into subarrays of some size and then sort each of them separately and then merge two sorted subarrays to get sorted list with each pass size of subarrays doubled.

5.11 KEYWORDS

Sorting: The operation of arranging data in some given order, such as increasing or decreasing with numerical data or alphabetically, with character data.

Quick Sort: A divide and conquer algorithm which works by creating two problems of half size, solving them recursively, then combining the solutions to the small problems to get a solution to the original problem.

Insertion Sort: A sorting technique that sorts a set of records by inserting records into an existing sorted file.

Merge Sort: Sorting method that uses merging of two ordered lists which can be combined to produce a single sorted list.

5.12 QUESTIONS FOR DISCUSSION

1. Why is there a need for sorting?
2. Write a short note on 'O' notation.
3. Sort the following list using the insertion sort method:

   6 2 8 3 1 9
4. Sort the following list using the shell sort method:

   36 28 39 42 19 10 9 54
5. What is a heap? Construct a heap with the following data:

   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

6. Explain sorting method through heap.
7. Write a 'C' code for construction of a heap.
8. Write a 'C' program for heap sort.
9. What is insertion sort? Explain with suitable example. Also explain its time complexity.
10. Write a 'C' function for insertion sort. For searching the smallest element in array use binary search. Explain your program's time complexity.
11. What is merging? Write a 'C' program to merge two sorted lists to get another sorted list. Explain its time complexity.
12. What is merge sort? Explain the method with a suitable example.
13. Write a 'C' program for merge sort, which uses the function Merge() to merge two sorted lists to get the third sorted list. Explain its time complexity.
14. Explain Quick Sort. Write a 'C' program for quick sort. Explain its time complexity.
15. Show that algorithm for quick sort takes O(n^2) time when the input file is already in sorted order.

Check Your Progress: Model Answer

1. Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data or alphabetically, with character data.
2. Sorting can be classified in two types:
   (i) Internal Sorting
   (ii) External Sorting
3. (a) root
   (b) [$\log_2$(N)], O(N*$\log_2$N)

5.13 SUGGESTED READING

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

UNIT IV

LESSON

6

GRAPH ALGORITHMS

CONTENTS

6.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Discuss graph definitions
- Define topological sort
- Discuss shortest path algorithms
- Discuss minimum spanning tree

6.1 INTRODUCTION

Graphs are natural models used to represent arbitrary relationship among data objects. We often need to represent such arbitrary relationship among the data objects while dealing with many problems in computer science, engineering, and many other disciplines. Therefore the study of graphs as one of the basic data structures is important.

This section presents the definition of a graph (both directed as well as undirected) and related terms. We will discuss various shortest path algorithms and minimum spanning tree.

6.2 DEFINITIONS

A graph is a structure made of two components, a set of vertices V, and the set of edges E. Therefore a graph is G=(V, E), where G is a graph. The graph may be directed or undirected. When the graph is directed every edge of a graph is an ordered pair of vertices connected by the edge, whereas when the graph is undirected every edge of a graph is an unordered pair of vertices connected by the edge. Given below in Figure 6.1 are the structures which are graphs.

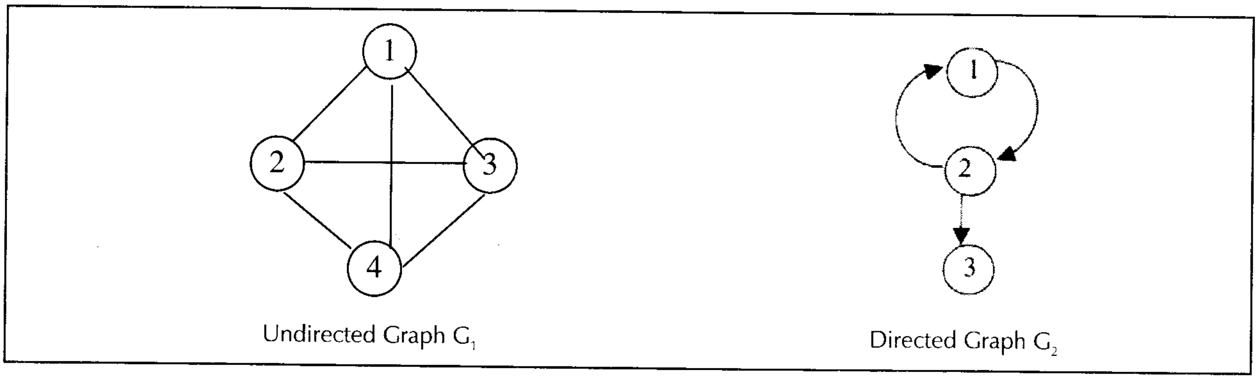


Figure 6.1

Incident edge: If (V_i, V_j) is an edge, then edge (V_i, V_j) is said to be incident on vertices v_i and v_j. For example, in the graph G_1 shown above in Figure 6.1 the edges incident on vertex 1 are (1,2), (1,4), and (1,3), whereas in G_2 the edges incident on vertex 1 are (1,2).

Degree of vertex: It is the number of edges incident on the vertex. For example, in graph G_1 shown above the degree of vertex 1 is 3, because 3 edges are incident on it. For a directed graph, we need to define indegree and outdegree.

Indegree of a vertex v_i is the number of edges incident on v_i, with v_i as the head. Outdegree of vertex v_i is the number of edges incident on v_i, with v_i as the tail. For a graph G_2 shown the indegree of the vertex 2 is 1, whereas the outdegree of the vertex 2 is 2.

Directed edge: A directed edge between the vertices v_i and v_j is an ordered pair, and denoted as $<v_i, v_j>$.

Undirected edge: An undirected edge between the vertices v_i and v_j is an unordered pair, and denoted as (v_i, v_j).

Path: A path between the vertices v_p and v_q is a sequence of vertices $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$ such that there exists a sequence of edges $(v_p, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v_q)$. In the case of a directed graph, a path between the vertices v_p and v_q is a sequence of vertices $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$ such that there exists a sequence of edges $<v_p, v_{i1}>, <v_{i1}, v_{i2}>, \ldots, <v_{in}, v_q>$. If there exists a path from vertex v_p to v_q in an undirected graph, then there always exists a path from v_q to v_p also. But in the case of a directed graph, if there exists a path from vertex v_p to v_q, then it does not necessarily imply that there exists a path from v_q to v_p also.

Simple path: A simple path is a path given by a sequence of vertices in which except the first and the last vertex all vertices are distinct. If the first and the last vertex is the same then the path will be a cycle.

Maximum number of edges: The maximum number of edges in an undirected graph with n vertices is n(n – 1)/2 whereas in case of a directed graph it is n(n – 1).

Subgraph

A subgraph of a graph G=(V,E) is a graph G' where

1. V(G') is a subset of V(G).
2. E (G') consists of edges (v_1,v_2) in E(G) such that both v_1 and v_2 are in V(G'). (Note: if G = (V, E) is a graph, then V(G) is set of vertices of G and E(G) is a set of edges of G.))

If E (G') consists of all edges (v_1,v_2) in E(G), such that both v_1 and v_2 are in V(G'), then G' is called an induced subgraph of G.

For example, the graph shown in Figure 6.2 is a subgraph of the graph G_2 shown in Figure 6.1.

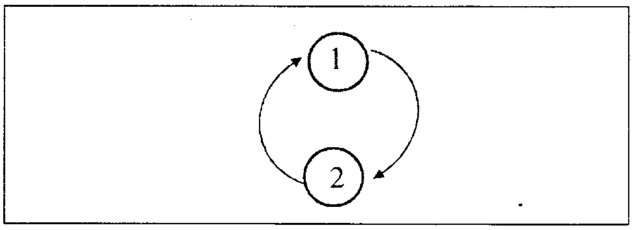


Figure 6.2: Subgraph

For the graph shown below in Figure 6.3, one of the induced subgraphs is shown in Figure 6.4.

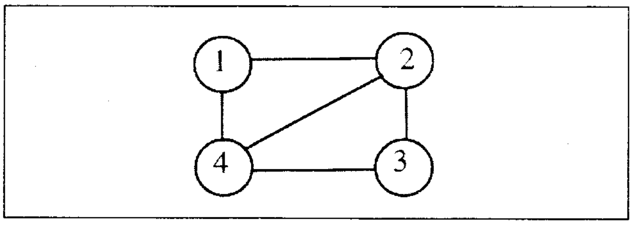


Figure 6.3: Graph G

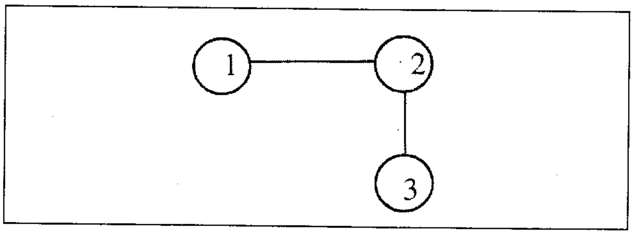


Figure 6.4: Induced Subgraph of Graph G of Figure 6.3

In an undirected graph G, the two vertices v_1 and v_2 are said to be connected, if there exist a path in G from v_1 to v_2.*(being undirected graph there exists a path from v2 to v1 also).*

Connected graph: A graph G is said to be connected if for every pair of distinct vertices (v_i,v_j) there is path from v_i to v_j. Given below in Figure 6.5 is a graph which is connected.

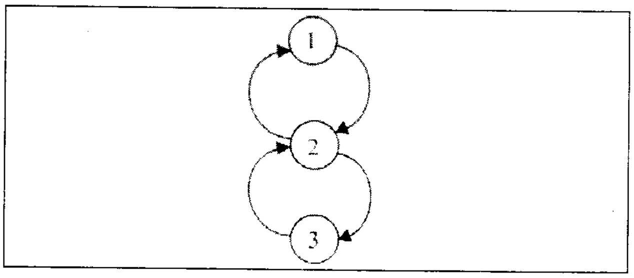


Figure 6.5: Connected Graph

Completely connected graph: A graph G is completely connected if for every pair of distinct vertices (v_i,v_j) there exists an edge. Given below in Figure 6.6 is a graph which is completely connected.

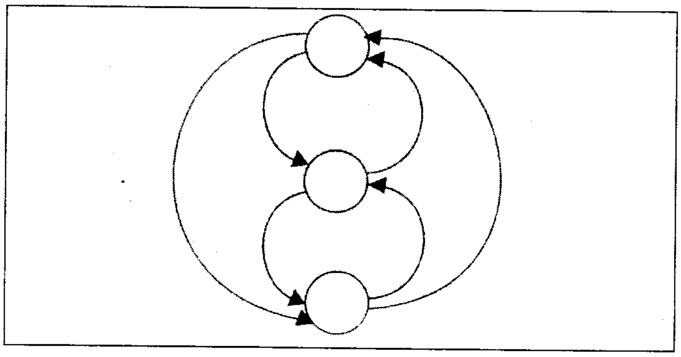


Figure 6.6

6.3 TOPOLOGICAL SORT

An application: You have a lay down of responsibilities. You are also told a set of preference relations; some tasks cannot be performed previous to others. How shall you plan the jobs exclusive of violating any prec restraint?

Job -> nodes; preference relations -> edges.

Evidently, if there is a cycle in the graph, no practicable plan.

When there is no cycle, *topological sorting* is a categorizing of vertices such that if there is a path from v_i to v_j, then v_i occurs prior to v_j in the plan.

```
Algorithm:
        Find a vertex v with zero in-degree      (must exist!)
        Print v, delete v, and its outgoing edges;
        Repeat;
Take O(V^2) time.
```

6.4 DIJKSTRA SHORTEST PATH ALGORITHM

E. W. Dijkstra developed an algorithm to determine the shorted 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.

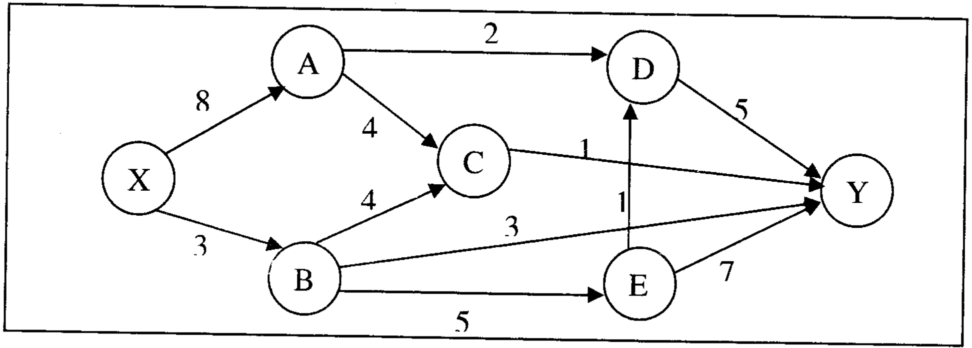
The algorithm finds the routes, by cost precedence. Let's assume that every cost is a positive number. The algorithm is equally applicable to a graph, a digraph, or even to a mixed graph with only some of its sides directed. If we consider a digraph, then every other case is fully covered as well since 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.

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**.
    i. Sort the vertices in **V** - **S** according to the current best estimate of their distance from the source.
    ii. Add **u**, the closest vertex in **V** - **S**, to **S**.
    iii. Re-compute the distances for the vertices in **V** - **S**

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}
2. Distances of all the nodes from
3. The nodes in the S:

   XA = 8 XB = 3 XC = ∞

   XD = ∞ XE = ∞ XY = ∞

4. Since, minimum distance from S to V – S is 3 (XB), S = {X, B} and E = {XB}
5. Distances of all the nodes from the nodes in the S:

   XA = 8 XC = ∞ XD = ∞ XE = ∞ XY = ∞

   XBA = ∞ XBC = 7 XBD = ∞ XBE = 8 XBY = 6

6. Since, minimum distance from S to V – S is 6 (XBY), S = {X, B, Y} and E = {XBY}.
7. Distances of all the nodes from the nodes in the S:

   XA = 8 XC = ∞ XD = ∞ XE = ∞

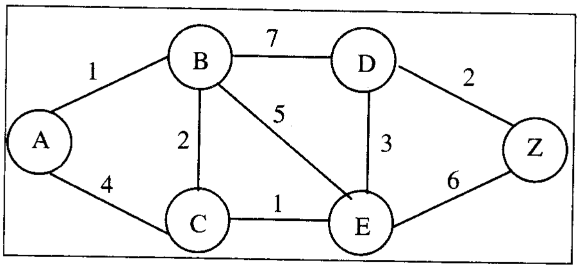
   XBA = ∞ XBC = 7 XBD = ∞ XBE = 8

   XBYA = ∞ XBYC = ∞ XBYD = ∞ XBYE = ∞

8. Continuing in similar manner, we find that the shortest path between nodes X and Y is XBY with cost value 6.

Network how Problems

Find the shortest path from A to Z for the given graph:



Solution:

Initially P = {A} and

T = {B,C,E,D,Z}

The lengths of different vertices (with respect to P) are:

L(B) = 1 , L(C) = 4, L(D) = L(E) = L(Z) = ¥

- So L(B) = 1 is the shortest value

  Now let P = {A,B} and T = {C,D,E,Z}

  so, L(C) = 3, L(D) = 8, L(E) = 6, L(Z) = ¥

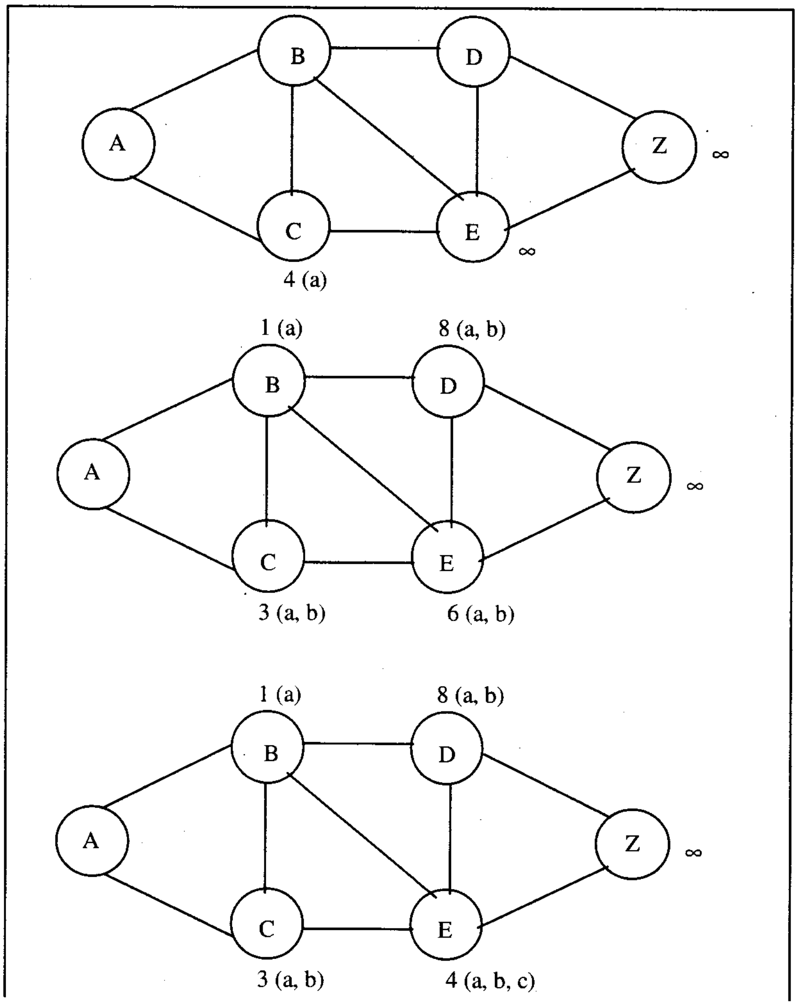
- So L(C) = 3, is the shortest value.

  Now P = {A,B, C} and T' = {D,E,Z}

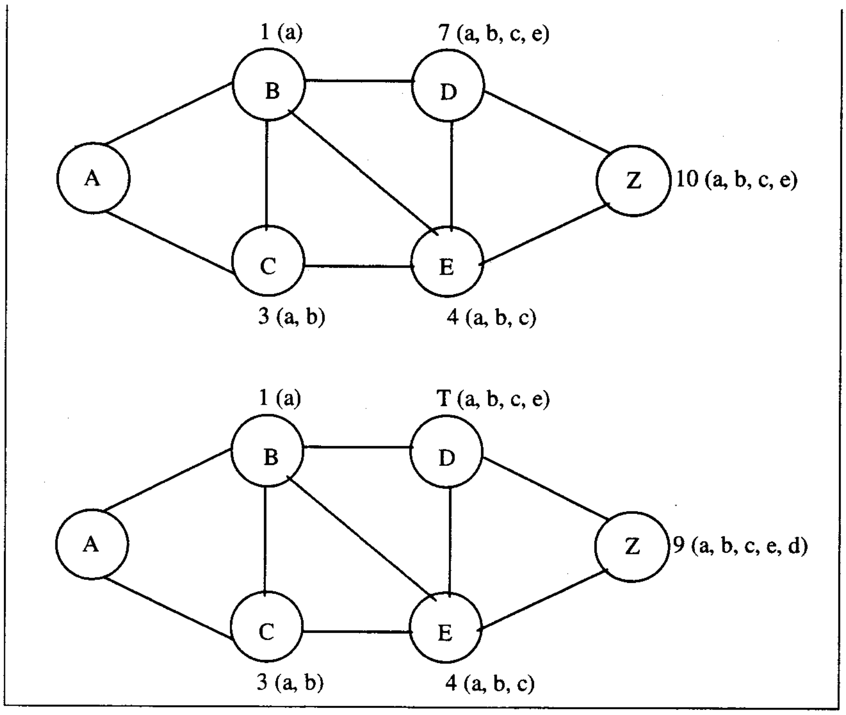
  L'(D) = Min{8,¥} = 8. L'(E) = Min{6,3 + 1} = 4

  L'(Z) = Min {¥,3 + ¥} = ¥

Successive observations are shown in following figures:



Contd....



Thus the shortest path from A to Z is (A, B, C, E, D, Z) of length 9.

6.5 WARSHALL ALGORITHM

Given the Adjacency Matrix A , this matrix produces the path matrix P.

```
1.  [Initialization]
        P¬A
2   [Perform a pass]
        Repeat thru step A for k=1(1)n
3   [Process Rows]
        Repeat step 4 for i=1(1) n.
4   [Across column]
        Repeat for j=1(1)n
        $P_{ij} \neg p_{ij} V(V_{ik} \wedge P_{kj})$
5.  [Exit]
```

6.6 MINIMAL ALGORITHM

Given the Adjacency Matrix B in which the zero elements are replaced by infinity or by some large number, the matrix produced by the following algorithm shows the minimum length of paths between the nodes. MIN is a function that selects the algebraic minimum of its two arguments.

```
[1] c¬B
[2] Repeat thru step 4 for k = 1(1) n
[3] Repeat thru step 4 for j = 1(1) n
    $c_{ij} = \text{MIN}\ (c_{ij}, c_{ik} + c_{kj})$
[5] exit
```

6.7 TRAVERSING A GRAPH

This section presents the methods of traversing a graph (directed as well as undirected). It also describes algorithms for traversing graphs.

6.7.1 Depth-first Traversal

A graph can be traversed either by using the depth first traversal or breadth first traversal. When a graph is traversed by visiting the nodes in the forward (deeper) direction as long as possible, the traversal is called depth first traversal. For example, for a graph shown in Figure 6.9, the depth first traversal starting at the vertex v_1 visits the node in the order shown in Figure 6.7 itself.

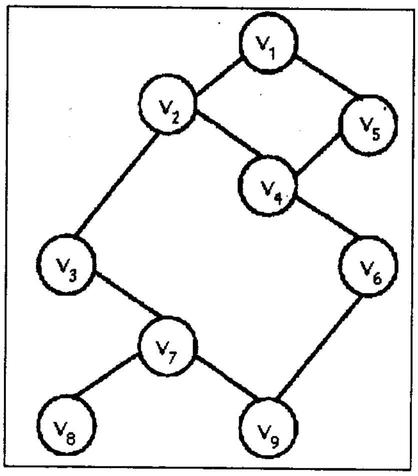


Figure 6.7: Graph g and its Depth First Traversals Starting at Vertex v_1

Some of the depth first traversal orders are:

(i) $v_1\ v_2\ v_3\ v_7\ v_8\ v_9\ v_6\ v_4\ v_5$

(ii) $v_1\ v_5\ v_4\ v_6\ v_9\ v_7\ v_8\ v_3\ v_2$

The procedure for depth first traversal of a graph is given below. The procedure makes use of a global array visited of n elements where n is the number of vertices of the graph, and the elements are boolean. If visited[i] = true then it means that i^{th} vertex is visited. Initially we set visited[i] = false, therefore:

```
For(i = 1; i < n; i++)
      visited[i] = false;
for(i = 1; i < n; i++)
```

```
        if(visited[i] == false) dfs(i);
void dfs(node x)
{
        visited[x] = true;
        for every adjacent y of x do
                dfs(y);
}
```

If the graph G to which the dfs is applied is represented by using adjacency lists then the vertices y adjacent to x can be determined by following the list of adjacent vertices for each vertex. Therefore the loop searching for adjacent vertices has the total cost of $d_1 + d_2 + ... + d_n$, where d_i is degree of vertex v_i because the number of nodes in the adjacency list of vertex v_i is d_i. If the graph G is having n vertices and e edges then the sum of the degree of each vertex, i.e., $(d_1 + d_2 + ... + d_n)$ is 2e. Therefore there are total 2e list nodes in the adjacency lists of G. (if G is directed graph then there are total e list nodes only). The algorithm examines each node in the adjacency lists at the most once. Hence the time required to complete the search is O(e) provided $n <= e$. Instead of using adjacency lists if adjacency matrix is used to represent a graph G, then the time required to determine all adjacent vertices of a vertex is O(n), and since most n vertices are visited the total time required is $O(n^2)$.

When this procedure is applied to the graph of Figure 6.7, then one of the orders in which the vertices gets visited is shown below:

v_1	false	true	true	true	true	true	true	true	true	true
v_2	false	false	true	true	true	true	true	true	true	true
v_3	false	false	false	true	true	true	true	true	true	true
v_4	false	false	false	false	false	false	false	false	true	true
v_5	false	false	false	false	false	false	false	false	false	true
v_6	false	false	false	false	false	false	false	true	true	true
v_7	false	false	false	false	true	true	true	true	true	true
v_8	false	false	false	false	false	true	true	true	true	true
v_9	false	false	false	false	false	false	true	true	true	true

6.7.2 Breadth-first Traversal

When a graph is traversed by visiting all the adjacent of a node/vertex first, the traversal is called breadth first traversal. For example, for a graph shown below one of the breadth first traversal starting at the vertex v_1 visits the node in the order shown below in Figure 6.8.

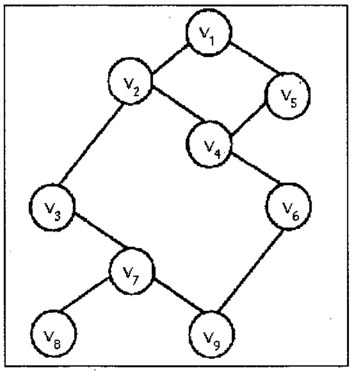


Figure 6.8: Bradth First Traversal of Graph G Starting at Vertex v_1

breadth first traversal order = v_1 v_2 v_5 v_3 v_4 v_7 v_6 v_8 v_9

The procedure for breadth first traversal of a graph is given below. The procedure makes use of a global array of n elements where n is number of vertices of the graph, and the elements are boolean. If visited[i] = true then it means that i^{th} vertex is visited. The procedure also makes use of a queue, and the procedures **addqueue** and **deletequeue** are assumed to be available for adding a vertex to the queue, and for deleting the vertex from the queue. Initially we set visited[i] := false, therefore:

```
For(i = 1; I < n; i++)
      visited[i] = false;


void bfs(node x)
{
      node y;
      addqueue(x);
      while(queue is not empty)
      {
      deletequeue(y );
      if(visited[y] == false)
            {
            visited[y] = true; .
            for every adjacent i of x do
            if (visited[i] == false)
                  addqueue(i);
            }
      }
}
```

If the graph G to which the bfs is applied is represented by using adjacency lists, then the vertices adjacent to x can be determined by following the list of adjacent vertices for each vertex. Therefore, the loop searching for adjacent vertices has the total cost of $d_1 + d_2 + ... + d_n$, where d_i is degree of vertex v_i because the number of nodes in the adjacency list of vertex v_i is d_i. If the graph G is having n vertices and e edges then the sum of the degree of each vertex, i.e $(d_1 + d_2 + ...+ d_n)$ is 2e. Therefore there are 2e list nodes in the adjacency lists of G. (if G is directed graph then there are e list nodes only). Each vertex gets added to queue exactly once, hence the loop while queue not empty is iterated at the most n times. Hence, the time required to complete the search is O(e) provided n< = e. Instead of using adjacency lists if adjacency matrix is used to represent a graph G, then the time required to determine all adjacent vertices of a vertex is O(n), and since every vertex gets added to queue exactly once the total time required is $O(n^2)$.

When this procedure is applied to the Figure 6.9 graph, then one of the orders in which the vertices gets visited is shown below:

v_1	false	true	true	true	true	true	true	true	true	true
v_2	false	false	true	true	true	true	true	true	true	true
v_3	false	false	false	false	true	true	true	true	true	true
v_4	false	false	false	false	false	true	true	true	true	
v_5	false	false	false	true	true	true	true	true	true	true
v_6	false	false	false	false	false	false	false	true	true	true
v_7	false	false	false	false	false	false	true	true	true	true
v_8	false	false	false	false	false	false	false	false	true	true
v_9	false	false	false	false	false	false	false	false	false	true

Figure 6.9

6.8 SPANNING TREES

This section presents the concept of spanning tree. It also presents the concept of weighted graph and minimum cost spanning tree for the weighted graph. It also discusses the properties of Minimum Spanning Tree (MST).

Depth-first Spanning Tree and Breadth-first Spanning Tree

If a graph G is connected, then the edges of G can be partitioned into two disjoint sets. One is a set of tree edges, which we denote as set T, and other is a set of back edges, which we denote as B. The tree edges are precisely those edges which are followed during the depth-first traversal or during the breadth first traversal of the graph G. If we consider only the tree edges, we get a subgraph of G containing all the vertices of G, and this sub-graph is a tree called spanning tree of the graph G. For example, consider the graph shown below in Figure 6.10.

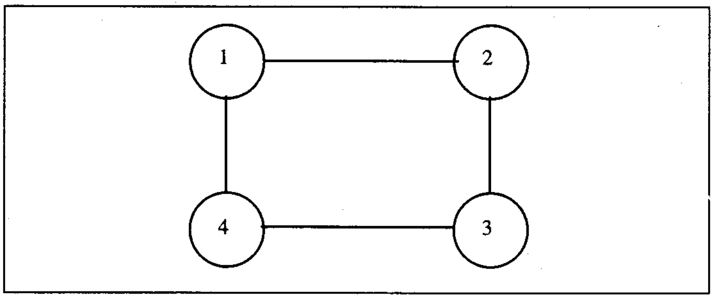


Figure 6.10: Graph G

One of the depth-first traversal orders for this tree is: 1-2-3-4; hence the tree edges are (1,2),(2,3) and (3,4). Therefore one of the spanning trees obtained using depth-first traversal of the graph of Figure 6.14 is shown in Figure 6.11.

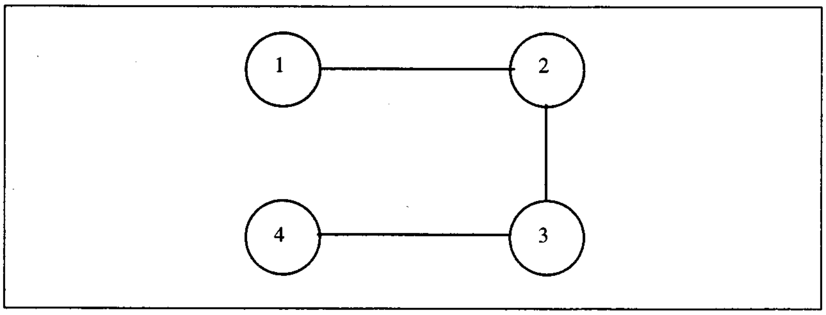


Figure 6.11: Depth-first Spanning Tree of the Graph of Figure 6.14

Similarly one of the breadth-first traversal orders for this tree is : 1-2-4-3; hence the tree edges are (1,2),(1,4) and (4,3). Therefore one of the spanning trees obtained using breadth first traversal of the graph is shown in Figure 6.12.

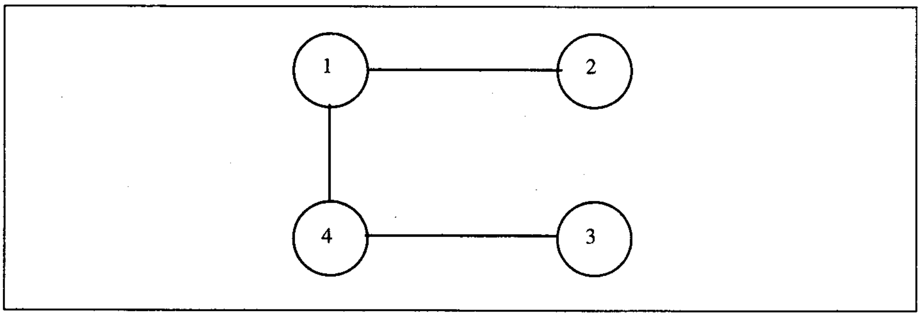


Figure 6.12: Breadth-first Spanning Tree of the Graph of Figure 6.14

The procedure for obtaining the depth first spanning tree is given below.

```
T = ϕ; /*initially set of tree nodes is empty*/
void dfst(node v)
{
      if(visited[v] == false)
      {
            visited[v] = true;
```

```
                for every adjacent i of v do
                {
                        T = T φ {(v,i)};
                        dfst(i);
                }
        }
}
```

If a graph G is not connected, then the tree edges, which are precisely those edges followed during the depth-first traversal of the graph G, constitutes the depth-first spanning forest. The depth-first spanning forest will be made of trees each of which is one of the connected components of graph G.

When a graph G is directed then the tree edges, which are precisely those edges followed during the depth-first traversal of the graph G, form a depth-first spanning forest for G. In addition to this, there are three other types of edges. These are called back edges, forward edges, and cross edges. An edge A → B is called a back edge if B is an ancestor of A in the spanning forest. A non-tree edge that goes from a vertex to a proper descendant is called a forward edge. An edge which goes from a vertex to another vertex that is neither an ancestor nor a descendant is called cross edge. An edge from a vertex to itself is a back edge.

For example, consider a directed graph G shown below in Figure 6.13.

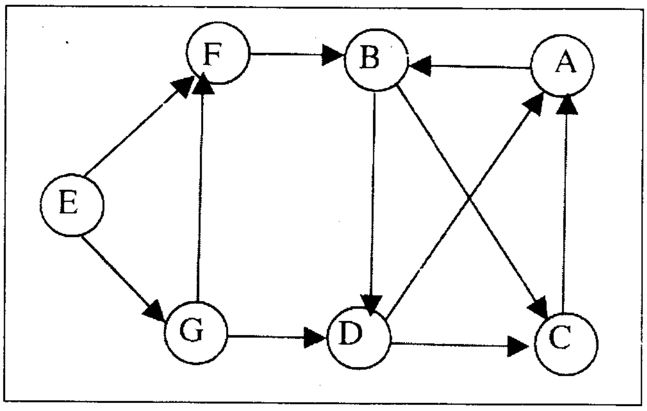


Figure 6.13: A Directed Graph G

The depth-first spanning forest for the graph G of Figure 6.13 is shown in Figure 6.14.

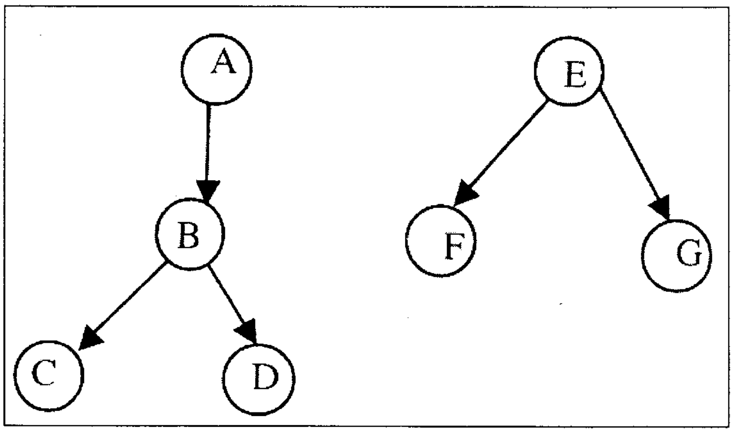


Figure 6.14: Depth-first Spanning Forest for the Graph G of Figure 7.17

Consider a graph show below in Figure 6.15.

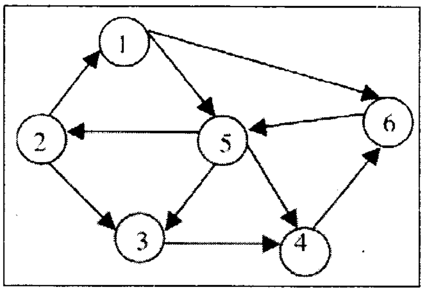


Figure 6.15: A Graph G

If we apply the procedure dfst to this graph one of the depth-first spanning trees that we get by starting with vertex 1 is shown below in Figure 6.16.

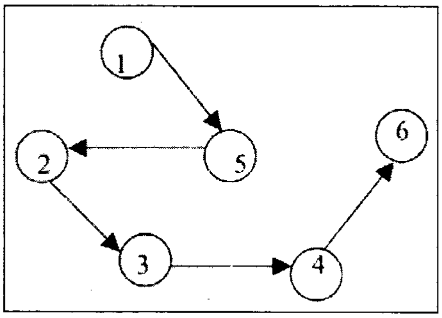


Figure 6.16: Depth-first Spanning Tree of the Graph G of Figure 6.19

6.9 MINIMUM-COST SPANNING TREE

When the edges of the graph have weights representing the cost in some suitable terms then we can obtain that spanning tree of a graph whose cost is minimum, in terms of the weights of the edges. For this, we start with the edge having the minimum-cost/weight add it to set T, and mark it visited. We next consider the edge with minimum-cost which is not yet visited, add it to T, and mark it visited. While adding an edge to the set T, we first check whether both the vertices of the edge are visited, if it is we do not add to the set T, because it will form a cycle. For example, consider the graph shown below in Figure 6.17.

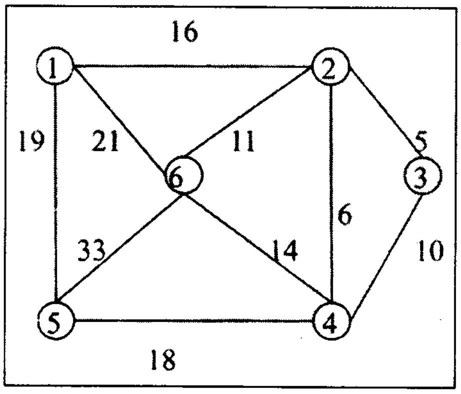


Figure 6.17: A Graph G

The minimum-cost spanning tree of the graph of Figure 6.17 is shown below in Figure 6.18.

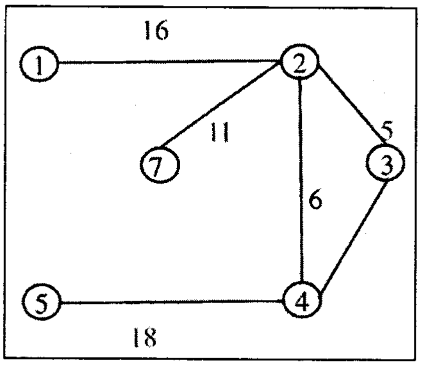


Figure 6.18: The Minimum-cost Spanning Tree of Graph G of Figure 6.17

6.9.1 MST Property

Let G = (V, E) be a connected graph with a cost function derided on the edges. Let U be some proper subset of the set of vertices V. If (u, v) is an edge the of lowest cost such that u is in U, and v is in V - U, then there is a minimum cost spanning tree that includes edge (u,v). Many of the methods of constructing a minimum cost spanning tree use the following properties:

Prim's Algorithm

Let G=(V, E) be a weighted graph, and suppose V={1,2,.. ..,n}. The prim's algorithm begins with a set U initialized to {1}, and at each stage finds the shortest edge (u, v) that connects u in U and v in V - U, and then adds v to U. It repeats this step until U = V.

```
void mcost(graph G, set of edges T)
{
      set of vertices U;
      vertex u, v;
      {
        T = ϕ
        U = {1}
        While U # V do
        {
                find the lowest cost edge (u,v)
                such that u is in U
                and v is in  V-U
                add (u,v) to T
                add v to U
        }
}
```

6.9.2 Application of Minimum-cost Spanning Tree

A property of a spanning tree of a graph G is that, a spanning tree is a minimal connected sub-graph of G (by minimal we mean the one with fewest number of edges). Therefore if nodes of G represent cities and the edges represent possible communication link connecting two cities, then the spanning trees of the graph G represent all feasible choices of the communication network. If each edge has weight representing cost measured in some suitable terms (like cost of construction or distance etc.), then the minimum-cost spanning tree of G is the selection of the required communication network.

Check Your Progress

1. What is minimum cost spanning tree?
2. What is depth-first traversal?

6.10 LET US SUM UP

- A graph consists of two non-empty subsets E(G) and V(G), where V(G) is a set of vertices and E(G) is a set of edges connecting those vertices.
- Graph is a superset of tree. Every tree is a graph but every graph is not necessarily a tree.
- A graph in which every edge is directed is called directed graph or digraph. A graph in which every edge is undirected is called an undirected graph.
- There are two methods to traverse a graph.
  (i) Depth-first search
  (ii) Breadth-first search
- Spanning tree is a tree obtained from a graph which covers all its vertices.

6.11 KEYWORDS

Digraph: A graph in which every edge is directed.

Undirected Graph: A graph in which every edge is undirected.

Null Graph: A graph containing only isolated nodes.

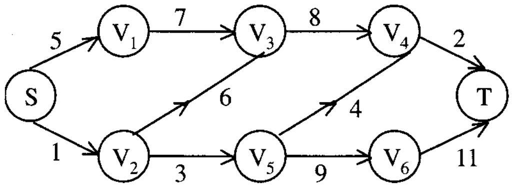
Spanning Tree: A tree obtained from a graph which covers all its vertices.

Minimum Spanning Tree: A tree from the set of spanning tree which has minimum weight.

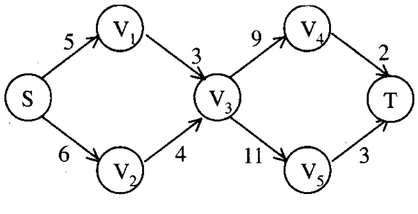
6.12 QUESTIONS FOR DISCUSSION

1. What is a graph? Compare graphs with trees.
2. Define these graphs:
   (i) Undirected graphs
   (ii) Directed graphs
3. Explain Warshall's minimal algorithm for finding the path matrix of a graph given its adjacency matrix.
4. What do you mean by traversal of any graph?
5. Write depth first search algorithm for the traversal of any graph. Write a "C" program for the same. Explain your algorithm's time complexity with the help of an example.
6. Explain breadth first search algorithm for the traversal of any graph with suitable examples. Define time complexity of the algorithm. Write a "C" program for the same.
7. Write Prim's algorithm for finding minimal spanning tree of any graph. Find the minimal spanning trees of the graph of previous questions by Prim's algorithm.
8. By considering the complete graph with n vertices, show that the number of spanning trees is at least 2^{n-1}-1.
9. Prove that when DFS and BFS are applied to a connected graph the edges of the graph form a tree.
10. What do you understand by shortest path from one node to another in a weighted graph. Write Dijkstra's algorithm to find the shortest path in a weighted graph. Find the shortest path from 3 to T using Dijkstra's algorithm in the following graphs:

(i)



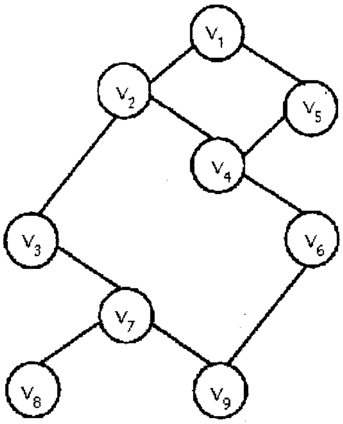
(ii)



11. Find the minimum distance between the nodes A and F in the following graph.

	A	B	C	D	E	F
A	0	(60)	0	(53)	0	(41)
B	(14)	0	0	(13)	(39)	(40)
C	(10)	(15)	0	0	(24)	0
D	0	(11)	(19)	0	(11)	0
E	(12)	0	(20)	0	0	(42)
F	0	(13)	(21)	(31)	0	0

12. Obtain a spanning tree for the following graph.



13. Obtain the minimum spanning tree for the following graph. The number in the parentheses are the cost of the corresponding edge.

	A	B	C	D	E	F
A	0	(60)	0	(53)	0	(41)
B	(14)	0	0	(13)	(39)	(40)
C	(10)	(15)	0	0	(24)	0
D	0	(11)	(19)	0	(11)	0
E	(12)	0	(20)	0	0	(42)
F	0	(13)	(21)	(31)	0	0

Check Your Progress: Model Answers

1. When the edges of the graph have weights representing the cost in some suitable terms then we can obtain that spanning tree of a graph whose cost is minimum, in terms of the weights of the edges.

2. When a graph is traversed by visiting the nodes in the forward (deeper) direction as long as possible, the traversal is called depth first traversal.

6.13 SUGGESTED READINGS

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

Birkhanser-Boston, *An Introduction to Data Structures and Algorithms*, Springer-New York

UNIT V

LESSON

7

ALGORITHM DESIGN TECHNIQUES

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Discuss greedy algorithms
- Describe the divide and conquer algorithms
- Explain the closest-points problem
- Define selection problem
- Analyse the theoretical improvements for arithmetic problems

7.1 INTRODUCTION

In this lesson, we will discuss about the *design* of algorithms. We will focus on some common types of algorithms used to solve problems. For many problems, it is pretty possible that at least one of these methods will work.

7.2 GREEDY ALGORITHMS

Firstly, we will discuss about *greedy algorithm*. Greedy algorithms perform in phases. In every phase, a verdict is made that emerges to be good, neglecting upcoming penalties. Usually, this shows that some *local optimum* is selected. The source of the name for this type of algorithms is "take what you can get now" policy. When the algorithm expires, we anticipate that the local optimum is identical to the *global optimum*. In this case, the algorithm is accurate; or else, the algorithm has created a suboptimal resolution. If the supreme best solution is not essential, then uncomplicated greedy algorithms are at times used to produce fairly accurate answers, instead of using the more intricate algorithms usually needed to produce a precise answer.

The most evident real-life case of greedy algorithms is the coin-changing problem. To formulate modification in U.S. currency, we frequently distribute the major quantity. Therefore, to provide seventeen dollars and sixty-one cents in change, we provide a ten-dollar bill, a five-dollar bill, two one-dollar bills, two quarters, one dime, and one penny. By doing this, we are assured to diminish the number of bills and coins. This algorithm does not function in all financial systems, but luckily, we can establish that it does function in the American financial system. Certainly, it functions even if two-dollar bills and fifty-cent pieces are permitted.

Another real-life example is traffic problems where building locally best possible choices does not forever work. For instance, for the duration of certain rush hour times in Miami, it is best to keep away from the prime lanes even if they seems to be vacant, as traffic will come to a languish a mile down the lane, and you will be trapped. Also more scandalous, it is healthier in some cases to make a momentary deviation in the direction opposite your target in order to evade all traffic holdups.

Now, we will discuss some applications that use greedy algorithms. The first application that will be discussed is a simple scheduling problem. Practically all scheduling problems are either *NP*-complete (or of alike complicated complexity) or are solved by a greedy algorithm. The second application that we will discuss is file compression and is one of the most primitive fallout in computer science. Finally, we will discuss an example of a greedy approximation algorithm.

7.2.1 A Simple Scheduling Problem

In simple scheduling problem we are provided with some jobs $j_1, j_2, \ldots, j_n$, all with given running times $t_1, t_2, \ldots, t_n$, respectively with a single processor. Now we need to know the finest way to schedule these jobs so as to reduce the average completion time.

Job	Time
J_1	16
J_2	8
J_3	3
J_4	14

Scheduling

J_1	J_2	J_3	J_4
16	20	28	40

Average completion time = (16+20+28+40)/4 = 26

J_3	J_2	J_4	J_1
3	12	24	40

Average completion time = (3+12+24+40)/4 = 19.75

Simple scheduling problem have some following properties:

- ***Greedy-choice property:*** If shortest job does not go initially the *y* jobs before it will finish 3 time units quicker, but j_3 will be delayed by time to finish all jobs previous to it.
- ***Optimal substructure:*** If shortest job is detached from optimal solution, left over solution for n-1 jobs is optimal.

  Optimality Proof

  Total cost of a schedule is

  N

  $\Sigma(N - k + 1)tik$

  k=1

  $t_1 + (t_1+t_2) + (t_1+t_2+t_3) \ldots (t_1+t_2+\ldots+t_n)$

  N

  $(N+1)\ \Sigma tik - \Sigma k^*tik$

  k=1

- First term is independent of ordering, as second term increases, total cost becomes smaller.

Assume that there is a job ordering such that x > y and tix < tiy. Swapping jobs (smaller first) increases second term decreasing total cost

Show: xtix + ytiy < ytix + xtiy

$xtix + ytiy = xtix + ytix + y(tiy - tix)$

$= ytix + xtix + y(tiy - tix)$

$< ytix + xtix + x(tiy - tix)$

$= ytix + xtix + xtiy - xtix = ytix + xtiy$

7.2.2 Huffman Codes

Now, we consider a second application of greedy algorithms, known as *file compression*.

The normal ASCII character set includes roughly 100 "printable" characters. To differentiate these characters, 7 bits are needed. Seven bits permit the demonstration of 128 characters, so the ASCII character set adds some other "nonprintable" characters. An eighth bit is added as a parity check.

Assume we have a file that encloses only the characters *a, e, i, s, t,* plus empty spaces and *newlines*. Assume further, that the file has ten *a*'s, fifteen *e*'s, twelve *i*'s, three *s*'s, four *t*'s, thirteen blanks, and one *newline*. As the table in Figure 7.1 shows, this file needs 174 bits to signify, since there are 58 characters and each character requires three bits.

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	3	39
newline	110	1	3
Total			174

Figure 7.1 Using a Standard Coding Scheme

In real case, files can be relatively bulky. Many of the very huge files are production of some program and there is typically a big difference between the most common and least common characters. For example, many huge data files have an enormously large quantity of digits, blanks, and *newlines*, but few *q*'s and *x*'s. We may be involved in minimizing the file size in the case where we are broadcasting it over a slow phone line. Also, as on virtually every machine disk space is valuable, one might wonder if it would be probable to offer a better code and decrease the total number of bits needed.

The answer is that this is achievable, and a simple policy attains 25 percent savings on usual huge files and as much as 50 to 60 percent savings on many huge data files. The common policy is to permit the code length to differ from character to character and to make sure that the often happening characters have short codes. Observe that if all the characters appear with the similar frequency, then there are not probable to be any savings.

Huffman's Algorithm

Compressing data is an imperative method for performing computing. Data sent over the Internet is required to be sent as densely as achievable. There are many dissimilar methods for condensing data, but one specific method makes use of a greedy algorithm—Huffman coding. This algorithm is named for the late David Huffman, an information philosopher and computer scientist who invented the practice in the 1950s. Data compressed using a Huffman code can accomplish savings of 20% to 90%. When data are compressed, the characters that build up the data are typically converted into some other demonstration to save space. A usual compression method is to convert each character to a binary character code, or bit string.

For instance, we can encode the character "a" as 000, the character "b" as 001, the character "c" as 010, and so on. This is known as a fixed-length code. The Huffman code algorithm takes a cord of characters, converts them to a variable-length binary string, and generates a binary tree for the intention of decoding the binary strings. The path to each left child is allocated the binary character 0 and each right child is allocated the binary character 1. The algorithm functions as follows:

Start with a string of characters you would like to compress. For each character in the string, compute its frequency of appearing in the string. Then arrange the characters into order from lowest frequency to highest frequency. Take the two characters with the minimum frequencies and make a node with each character (and its frequency) as children of the node. The parent node's data element consists of the sum of the frequencies of the two child nodes. Insert the node back into the list. Continue this process until every character is located into the tree. On the completion of this process, you have a complete binary tree that can be used to decode the Huffman code. Decoding comprises following a path of 0s and 1s until you get to a leaf node, which will enclose a character.

7.3 DIVIDE AND CONQUER

Divide and Conquer is another common technique used to design algorithms. Divide and conquer algorithms consist of two parts:

Divide: Smaller problems are resolved recursively.

Conquer: The key to the original problem is then produced from the solutions to the sub problems.

Conventionally, schedules in which the text consists of at least two recursive calls are known as divide and conquer algorithms, where as schedules whose text consists of only one recursive call are not. We usually persist that the sub problems be displaced (that is, basically nonoverlapping). Let us review some of the recursive algorithms that have been covered in this text.

We have already seen several divide and conquer algorithms. In lesson 3, we saw tree traversal strategies. In lesson 5, we saw the classic examples of divide and conquer, namely mergesort and quicksort, which have O (n log n) worst-case and average-case bounds, respectively.

Lesson 6 showed routines to recover the shortest path in Dijkstra's algorithm and other events to perform depth-first search in graphs. None of these algorithms are really divide and conquer algorithms, because only one recursive call is performed.

Now, we will see more cases of the divide and conquer pattern. Our first application is a problem in *computational geometry*. Specified n points in a plane, we will illustrate that the closest pair of points can be found in $O(n \log n)$ time. The rest of the discussion shows some awfully interesting, but mostly hypothetical, results. We offer an algorithm which solves the selection problem in $O(n)$ worst-case time. We also prove that 2 n-bit numbers can be multiplied in $o(n^2)$ operations and that two n x n matrices can be multiplied in $o(n^3)$ operations. Unluckily, yet these algorithms have improved worst-case bounds than the conventional algorithms, none are realistic barring very large inputs.

7.3.1 Running Time of Divide and Conquer Algorithms

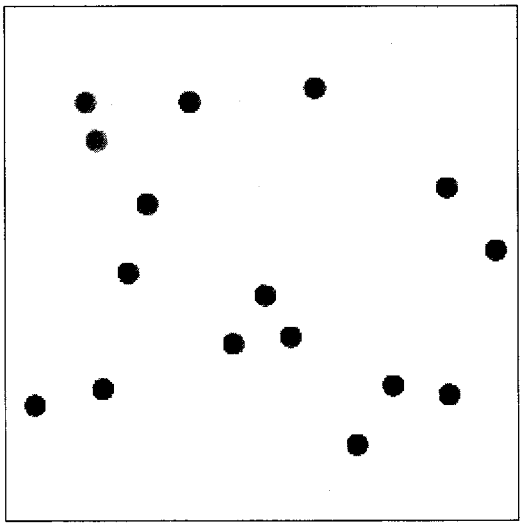
All the competent divide and conquer algorithms divide the problems into subproblems, each of which is some division of the original problem, and then execute some supplementary work to calculate the final answer. As an instance, we have seen that merge sort functions on two problems, each of which is half the size of the original, and then uses $O(n)$ supplementary work. This capitulate the running time equation (with appropriate initial conditions)

$T(n) = 2T(n/2) + O(n)$

7.3.2 Closest-points Problem

- Given n points in d-dimensions, find two whose shared distance is minimum.
- It is a primary problem in many applications as well as a key step in many algorithms.
- A naive algorithm takes O(dn2) time.
- Element individuality reduces to Closest Pair, so -(n log n) lower bound.

- We will build up a divide-and-conquer based O(n log n) algorithm; dimension assumed constant.



1-Dimension Problem

- 1D problem can be solved in O(n log n) by means of sorting.
- Sorting, though, does not simplify to higher dimensions. So, let's build up a divide-and-conquer for 1D.
- Divide the points S into two sets S1; S2 by some *x*-coordinate so that p < q for all p ∈S1 and q ∈S2.
- Recursively calculate closest pair (p1; p2) in S1 and (q1; q2) in S2.
- Let $ be the smallest division found until now:

  $ = min(|p2–p1|; |q2-*q*1|)

1D Divide & Conquer

- The closest pair is *{p*1*; p*2*}*, or *{q*1*; q*2*}*, or some *{p*3*; q*3*}* where *p*3 ∈ S1 and *q*3 ∈ *S*2.

  If m is the dividing coordinate, then p3; q3 must be within $ of m.
- In 1D, p3 must be the rightmost point of S1 and q3 the leftmost point of S2, but these ideas do not simplify to higher proportions.
- How many points of S1 can lie in the interval (m-$;m]?
- By definition of $, at most one. Same holds for S2.

1D Divide & Conquer

- Closest-Pair (S).
- If |S| = 1, output $ = infinity. If |S| = 2, output$ = |p2–p1|. Or else, perform the following steps:
  1. Let m = median(S).
  2. Divide S into S1; S2 at m.

3. \$1 = Closest-Pair(S1).
4. \$2 = Closest-Pair(S2).
5. \$12 is minimum distance across the cut.
6. Return \$ = min(\$1; \$2; \$12).

- Recurrence is T(n) = 2T(n=2) + O(n), which solves to T(n) = O(n log n).

2-D Closest Pair

- We partition S into S1; S2 by vertical line l defined by median x-coordinate in S.
- Recursively compute closest pair distances \$1 and \$2. Set \$= min(\$1; \$2).
- Now compute the closest pair with one point each in S1 and S2.
- In each candidate pair (p; q), where p ∈ S1 and q ∈ S2, the points p; q must both lie within \$of l.
- At this point, complications arise, which weren't present in 1D. It's entirely possible that all n=2 points of S1 (and S2) lie within \$ of l.
- Naively, this would require n2/4 calculations.
- We show that points in P1; P2 (\$ strip around) have a special structure, and solve the conquer step faster.

7.3.3 Selection Problem

In this problem, we are provided an unordered list of elements, and want to locate the kth largest element. An easy way of solving this problem is to initially arrange the list and then study the kth largest element. This takes time O(n log n). Yet, most probably locating only the kth largest element should be easier than arranging the whole list. For instance, we could preserve a list of the k largest elements and populate this list in time O(n log k). When k is a small constant, this takes only linear time. We will prove that we can execute selection in linear time for an subjective k using a divide and conquer method.

To get instinct of the solution of the problem, assume that we could locate the median of a list in linear time. We state that we can then use this as a sub procedure in a divide and conquer algorithm to locate the kth largest element. Specifically, we use the median to divide the list into two halves. Then we recursively locate the preferred constituent in one of the halves (the first half, if k _ n/2, and the second half otherwise). This algorithm takes time cn at the first level of recursion for some constant c, cn/2 at the next level (as we recurse in a list of size n/2), cn/4 at the third level, and so on. The total time taken is cn + cn/2 + cn/4 + · · · = 2cn = O(n).

Unluckily, however, locating the median doesn't appear to be much easier than locating the kth largest element. The main idea here is that for applying the recursion, we are not required a precise median – a near-median would do. Particularly assume we could locate an element at every step such that at least 3/10th of the elements in the list are minor than it and at least 3/10th of the elements are bigger than it, then we could still apply the same divide and conquer method as above. Assuming each divide step takes linear time, our running time would become at most cn + 7/10 cn + 49/100 cn + · · · = 3.33cn = O(n).

Lastly, it turns out that we can locate a near-median in linear time by again applying recursion.

Specifically, we divide the list into groups of 5 elements each, discover the median in each group in constant time (as each group is of constant size), and then discover the median of these medians recursively. The main point to observe is that the final step of locating the median of medians applies to a much smaller list - of size n/5, and so we still get a small enough running time.

This was just a coarse description and analysis of the algorithm. A more formal analysis defined below:

For straightforwardness of analysis, we suppose that all the list sizes we come across while running the algorithm are divisible by 5.

Algorithm for Selection

1. Divide the list into n/5 lists of 5 elements each.
2. Find the median in each sublist of 5 elements.
3. Recursively find the median of all the medians, call it m.
4. Partition the list into elements larger than m (call this sublist L1) and those no larger than m (call this sublist L2).
5. If k _ |L1|, return Selection(L1, k).
6. If k _ |L1| + 1, return Selection(L2,k - |L1|).

The accuracy of the algorithm is simple to quarrel and we will omit the disagreement. Let us analyse the running time. Observe that we make two recursive calls. The first is to a list of size n/5. The second is to either L1 or L2. We quarrel that these lists can be no bigger than 7n/10 in size. This is for the reason that there are n/10 medians at step 3 that are minor than m, and there are three elements in each of the sublists equivalent to these n/10 medians that are no bigger than the medians, and consequently no larger than m itself. As a result, L2 is of size at least 3n/10, and L1 is of size at most n−|L2| _ 7n/10. Similarly we can quarrel that L1 is of size at least 3n/10 and so L2 is of size at most 7n/10.

Thus we obtain the subsequent recurrence for the running time of the algorithm:

$$T(n) = cn + T(n/5) + T(7n/10)$$

where cn is the time taken to build the list of medians and to divide the list into L1 and L2 for a suitable constant c.

One way of solving this recurrence is to estimate that the running time is T(n) = c'n and then verify whether the equation is fulfilled for some worth of c'. Substituting this in the equation we get

$$c'n = cn + 9/10\, c'n$$

which entails c' = 10c.

7.3.4 Theoretical Improvements for Arithmetic Problems

Now, we will discuss a divide and conquer algorithm that multiplies two *n*-digit numbers. Our preceding model of calculation assumed that multiplication was performed in invariable time, since the numbers were small. For large numbers, this supposition is no longer applicable. If we gauge multiplication in view of the size of numbers being multiplied, then the normal multiplication algorithm takes quadratic time. The divide and conquer algorithm occurs in subquadratic time. We also represent the typical divide and conquer algorithm that multiplies two n by n matrices in sub cubic time.

- Multiplying Integers
- Matrix Multiplication

Multiplying Integers

Let us consider multiplying two n-digit numbers x and y. If precisely one of x and y is negative, then the solution is negative; or else it is positive.

If $x = 61{,}438{,}521$ and $y = 94{,}736{,}407$, $xy = 5{,}820{,}464{,}730{,}934{,}047$. Let us divide x and y into two halves, including the most important and least important digits, correspondingly. Then $xl = 6{,}143$, $xr = 8{,}521$, $yl = 9{,}473$, and $yr = 6{,}407$. We also have $x = xl104 + xr$ and $y = yl104 + yr$. It shows that

$xy = xlyl108 + (xlyr + xryl)104 + xryr$

Observe that this equation comprises of four multiplications, $xlyl$, $xlyr$, $xryl$, and $xryr$, which are each half the size of the original problem ($n/2$ digits). The multiplications by 108 and 104 amount to the placing of zeros. This and the following additions add only $O(n)$ supplementary work. If we execute these four multiplications recursively by means of this algorithm, discontinuing at an suitable base case, then we acquire the recurrence

$T(n) = 4T(n/2) + O(n)$

We know that $T(n) = O(n^2)$, so, unluckily, we have not enhanced the algorithm. To attain a subquadratic algorithm, we must use less than four recursive calls. The main inspection is that

$\mathrm{xlyr} + \mathrm{xryl} = (\mathrm{xl} - \mathrm{xr})(\mathrm{yr} - \mathrm{yl}) + \mathrm{xlyl} + \mathrm{xryr}$

Therefore, rather than using two multiplications to calculate the coefficient of 104, we can use one multiplication, plus the result of two multiplications that have by now been performed. It is simple to see that at the present the recurrence equation gratifies

$T(n) = 3T(n/2) + O(n)$,

and so we acquire $T(n) = O(n\log 23) = O(n1.59)$. To complete the algorithm, we must have a base case, which can be solved lacking recursion.

When both numbers are one-digit, we can do the multiplication by table lookup. If one number has zero digits, then we return zero. In practice, if we were to use this algorithm, we would choose the base case to be that which is most convenient for the machine.

Although this algorithm has better asymptotic performance than the standard quadratic algorithm, it is rarely used, because for small n the overhead is significant, and for larger n there are even better algorithms. These algorithms also make widespread use of divide and conquer.

Matrix Multiplication

A basic arithmetical problem is the multiplication of two matrices. Figure 7.2 gives a simple $O(n^3)$ algorithm to figure out $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are n by n matrices. The algorithm follows straightforwardly from the description of matrix multiplication. To calculate $C_{i,j}$, we compute the dot product of the ith row in **A** with the jth column in **B**. Typically arrays commences at index 0.

For a long time it was presumed that $O(n^3)$ was needed for matrix multiplication. Yet, in the late sixties Strassen showed how to break the $O(n^3)$ obstruction. The fundamental idea of Strassen's algorithm is to split each matrix into four quadrants, as shown in Figure 7.3. Then it is simple to show that

$C_{1,1} = A_{1,1} B_{1,1} + A_{1,2} B_{2,1}$

$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

/* Standard matrix multiplication. Arrays start at 0 */

void

matrix_multiply(matrix A, matrix B, matrix C, unsigned int n)

```
{
int i, j, k;
for( i=0; i<n; i++ )      /* Initialization */
for( j=0; j<n; j++ )
C[i][j] = 0.0;
for( i=0; i<n; i++ )
for( j=0; j<n; j++ )
for( k=0; k<n; k++ )
C[i][j] += A[i][k] * B[k][j];
}
```

Figure 7.2 Simple O(n³) matrix multiplication

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Figure 7.3 Decomposing AB = C into four quadrants

We know that $T(n) = O(n^3)$, thus we do not have an enhancement. As we have seen with integer multiplication, we must decrease the number of subproblems below 8. Strassen used a policy analogous to the integer multiplication divide and conquer algorithm and dispayed how to use only seven recursive calls by cautiously assembling the computations. The seven multiplications are

$M_1 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$

$M_2 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$

$M_3 = (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})$

$M_4 = (A_{1,1} + A_{1,2})B_{2,2}$

$M_5 = A_{1,1} (B_{1,2} - B_{2,2})$

$M_6 = A_{2,2}\,(B_{2,1} - B_{1,1})$

$M_7 = (A_{2,1} + A_{2,2})B_{1,1}$

After the multiplications are performed, the concluding solution can be acquired with eight more additions.

$C_{1,1} = M_1 + M_2 - M_4 + M_6$

$C_{1,2} = M_4 + M_5$

$C_{1,3} = M_6 + M_7$

$C_{1,4} = M_2 - M_3 + M_5 - M_7$

It is simple to confirm that this complicated ordering generates the preferred values. The running time now assures the recurrence

$T(n) = 7T(n/2) + O(n^2)$.

The solution of this recurrence is $T(n) = O(n\log 27) = O(n2.81)$.

Typically, there are particulars to consider, like the case when n is not a power of two, but these are essentially minor troubles. Strassen's algorithm is poorer than the simple algorithm until n is quite large. It does not simplify for the case where the matrices are light (contain many zero entries), and it does not effortlessly parallelize. When run with floating-point entries, it is less stable numerically than the typical algorithm. Therefore, it is has only restricted applicability. However, it symbolizes an imperative theoretical landmark and surely shows that in computer science, as in many other fields, even despite the fact that a problem appears to have an inherent difficulty, nothing is sure until verified.

Check Your Progress

1. Define greedy-choice property.
2. What is simple scheduling problem?

7.4 LET US SUM UP

This lesson illustrates some of the most common techniques found in algorithm design. When confronted with a problem, it is worthwhile to see if any of these methods apply. A proper choice of algorithm, combined with judicious use of data structures, can often lead quickly to efficient solutions.

7.5 KEYWORDS

Optimal Substructure: If shortest job is detached from optimal solution, left over solution for n-1 jobs is optimal.

Divide: Smaller problems are resolved recursively.

Conquer: The key to the original problem is then produced from the solutions to the sub problems.

7.6 QUESTIONS FOR DISCUSSION

1. What are greedy algorithms? Discuss some real life examples.
2. A file contains only colons, spaces, newline, commas, and digits in the following frequency: colon (100), space (605), newline (100), commas (705), 0 (431), 1 (242), 2 (176), 3 (59), 4 (185), 5 (250), 6 (174), 7 (199), 8 (205), 9 (217). Construct the Huffman code.
3. Complete the proof that Huffman's algorithm generates an optimal prefix code.
4. Write a program to implement file compression (and uncompression) using Huffman's algorithm.
5. Write a program to implement the closest-pair algorithm.

Check Your Progress: Model Answers

1. Greedy-choice property: If shortest job does not go initially the *y* jobs before it will finish 3 time units quicker, but j_3 will be delayed by time to finish all jobs previous to it.
2. In simple scheduling problem we are provided with some jobs $j_1, j_2, \ldots, j_n$, all with given running times $t_1, t_2, \ldots, t_n$, respectively with a single processor.

7.7 SUGGESTED READINGS

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

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