UNIT 1 ANALYSIS OF ALGORITHMS

Structure **Page Nos.**

1.0	Introduction	7
1.1	Objectives	7
1.2	Mathematical Background	8
1.3	Process of Analysis	12
1.4	Calculation of Storage Complexity	18
1.5	Calculation of Time Complexity	19
1.6	Summary	21
1.7	Solutions/Answers	22
1.8	Further Readings	22

1.0 INTRODUCTION

A common person's belief is that a computer can do anything. This is far from truth. In reality, computer can perform only certain predefined instructions. The formal representation of this model as a sequence of instructions is called an algorithm, and coded algorithm, in a specific computer language is called a program. Analysis of algorithms has been an area of research in computer science; evolution of very high speed computers has not diluted the need for the design of time-efficient algorithms.

Complexity theory in computer science is a part of theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are *time* (how many steps (time) does it take to solve a problem) and *space* (how much memory does it take to solve a problem). It may be noted that complexity theory differs from computability theory, which deals with whether a problem can be solved or not through algorithms, regardless of the resources required.

Analysis of Algorithms is a field of computer science whose overall goal is understand the complexity of algorithms. While an extremely large amount of research work is devoted to the worst-case evaluations, the focus in these pages is methods for average-case. One can easily grasp that the focus has shifted from computer to computer programming and then to creation of an algorithm. This is algorithm design, heart of problem solving.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the concept of algorithm;
- understand the mathematical foundation underlying the analysis of algorithm;
- to understand various asymptotic notations, such as Big O notation, theta notation and omega (big O, Θ, Ω) for analysis of algorithms;
- understand various notations for defining the complexity of algorithm;
- define the complexity of various well known algorithms, and
- learn the method to calculate time complexity of algorithm.

1.2 MATHEMATICAL BACKGROUND

To analyse an algorithm is to determine the amount of resources (such as time and storage) that are utilized by to execute. Most algorithms are designed to work with inputs of arbitrary length.

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

Definition of Algorithm

Algorithm should have the following five characteristic features:

1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Termination.

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input.

Complexity classes

All decision problems fall into sets of comparable complexity, called complexity classes.

The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time. This class corresponds to set of problems which can be effectively solved in the worst cases. We will consider algorithms belonging to this class for analysis of time complexity. Not all algorithms in these classes make practical sense as many of them have higher complexity. These are discussed later.

The complexity class NP is a set of decision problems that can be solved by a non-deterministic machine in polynomial time. This class contains many problems like Boolean satisfiability problem, Hamiltonian path problem and the Vertex cover problem.

What is Complexity?

Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as **asymptotic analysis.** It may be noted that we are dealing with complexity of an algorithm not that of a problem. For example, the simple problem could have high order of time complexity and vice-versa.

Asymptotic Analysis

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the "Big O", "Omega" and "Theta" notation for asymptotic performance.

The notations like "Little Oh" are similar in spirit to "Big Oh" ; but are rarely used in computer science for asymptotic analysis.

Tradeoff between space and time complexity

We may sometimes seek a tradeoff between space and time complexity. For example, we may have to choose a data structure that requires a lot of storage in order to reduce the computation time. Therefore, the programmer must make a judicious choice from an informed point of view. The programmer must have some verifiable basis based on which a data structure or algorithm can be selected Complexity analysis provides such a basis.

We will learn about various techniques to bind the complexity function. In fact, our aim is not to count the exact number of steps of a program or the exact amount of time required for executing an algorithm. In theoretical analysis of algorithms, it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input 'n'. Big O notation, omega notation Ω and theta notation Θ are used for this purpose. In order to measure the performance of an algorithm underlying the computer program, our approach would be based on a concept called asymptotic measure of complexity of algorithm. There are notations like big O, Θ, Ω for asymptotic measure of growth functions of algorithms. The most common being big-O notation. The asymptotic analysis of algorithms is often used because time taken to execute an algorithm varies with the input 'n' and other factors which may differ from computer to computer and from run to run. The essences of these asymptotic notations are to bind the growth function of time complexity with a function for sufficiently large input.

The Θ-Notation (Tight Bound)

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$, both inclusive. The *Figure 1.1* gives an idea about function *f(n)* and *g(n)* where $f(n) = \Theta(g(n))$. We will say that the function *g(n)* is asymptotically tight bound for *f(n)*.

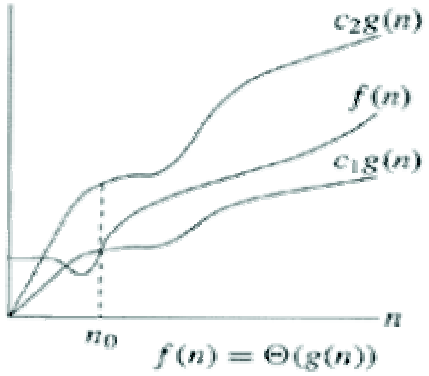


Figure 1.1 : Plot of $f(n) = \Theta(g(n))$

For example, let us show that the function $f(n) = \frac{1}{3} n^2 - 4n = \Theta(n^2)$.

Now, we have to find three positive constants, c_1, c_2 and n_o such that

$$c_1 n^2 \leq \frac{1}{3} n^2 - 4n \leq c_2 n^2 \quad \text{for all } n \geq n_o$$

$$\Rightarrow \quad c_1 \leq \frac{1}{3} - \frac{4}{n} \leq c_2$$

By choosing $n_o = 1$ and $c_2 \geq 1/3$ the right hand inequality holds true.

Similarly, by selecting $n_o = 13$ $c_1 \leq 1/39$, the right hand inequality holds true. So, for $c_1 = 1/39$, $c_2 = 1/3$ and $n_o \geq 13$, it follows that $1/3\, n^2 - 4n = \Theta(n^2)$.

Certainly, there are other choices for c_1, c_2 and n_o. Now we may show that the function $f(n) = 6n^3 \neq \Theta(n^2)$.

To prove this, let us assume that c_3 and n_o exist such that $6n^3 \leq c_3 n^2$ for $n \geq n_o$, But this fails for sufficiently large n. Therefore $6n^3 \neq \Theta(n^2)$.

The big O notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. *Figure 1.2* shows the plot of $f(n) = O(g(n))$ based on big O notation. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

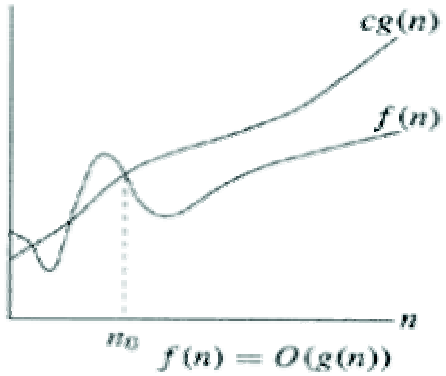


Figure 1.2: Plot of $f(n) = O(g(n))$

Mathematically for a given function $g(n)$, we denote a set of functions by $O(g(n))$ by the following notation:

$O(g(n)) = \{f(n) :$ There exists a positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

Clearly, we use O-notation to define the upper bound on a function by using a constant factor c.

We can see from the earlier definition of Θ that Θ is a tighter notation than big-O notation.

$f(n) = an + c$ *is* $O(n)$ is also $O(n^2)$, but $O(n)$ is asymptotically tight whereas $O(n^2)$ is notation.

Whereas in terms of Θ notation, the above function $f(n)$ is $\Theta(n)$. As big-O notation is upper bound of function, it is often used to describe the worst case running time of algorithms.

The Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$, if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. *Figure 1.3* depicts the plot of $f(n) = \Omega(g(n))$.

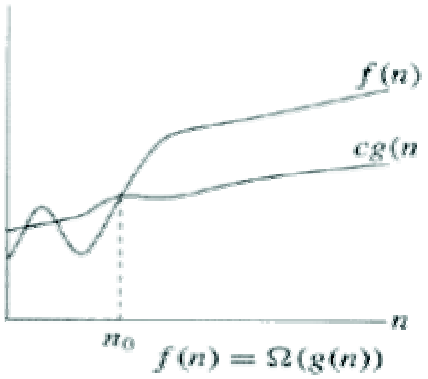


Figure 1.3: Plot of $f(n) = \Omega(g(n))$

Mathematically for a given function $g(n)$, we may define $\Omega(g(n))$ as the set of functions.

$\Omega(g(n)) = \{ f(n)$: there exists a constant c and $n_0 \geq 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$.

Since Ω notation describes lower bound, it is used to bound the best case running time of an algorithm.

Asymptotic notation

Let us define a few functions in terms of above asymptotic notation.

Example: $f(n) = 3n^3 + 2n^2 + 4n + 3$
$= 3n^3 + 2n^2 + O(n)$, as $4n + 3$ is of $O(n)$
$= 3n^3 + O(n^2)$, as $2n^2 + O(n)$ is $O(n^2)$
$= O(n^3)$

Example: $f(n) = n^2 + 3n + 4$ is $O(n^2)$, since $n^2 + 3n + 4 < 2n^2$ for all $n > 10$.
By definition of big-O, $3n + 4$ is also $O(n^2)$, too, but as a convention, we use the tighter bound to the function, i.e., $O(n)$.

Here are some rules about big-O notation:

1. $f(n) = O(f(n))$ for any function f. In other words, every function is bounded by itself.

2. $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$ for all $k \geq 0$ and for all $a_0, a_1, \ldots, a_k \in R$. In other words, every polynomial of degree k can be bounded by the function n^k. Smaller order terms can be ignored in big-O notation.

3. Basis of Logarithm can be ignored in big-O notation i.e. $\log_a n = O(\log_b n)$ for any bases a, b. We generally write $O(\log n)$ to denote a *logarithm* n to any base.

4. Any logarithmic function can be bounded by a polynomial i.e. $log_b\ n = O(n^c)$ for any b (base of logarithm) and any positive exponent $c > 0$.

5. Any polynomial function can be bounded by an exponential function i.e. $n^k = O\ (b^n)$.

6. Any exponential function can be bound by the factorial function. For example, $a^n = O(n!)$ for any base a.

☞ Check Your Progress 1

1) The function $9n+12$ and $1000n+400000$ are both $O(n)$. True/False

2) If a function $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n)+h(n) = O(g(n))$. True/False

3) If $f(n) = n^2 + 3n$ and $g(n) = 6000n + 34000$ then $O(f(n)) < O\ (g(n).)$ True/False

4) The asymptotic complexity of algorithms depends on hardware and other factors. True/False

5) Give simplified *big-O* notation for the following growth functions:

- $30n^2$
- $10n^3 + 6n^2$
- $5nlogn + 30n$
- $log\ n + 3n$
- $log\ n + 32$

……

……

……

……

1.3 PROCESS OF ANALYSIS

The objective analysis of an algorithm is to find its efficiency. Efficiency is dependent on the resources that are used by the algorithm. For example,

- CPU utilization (Time complexity)
- Memory utilization (Space complexity)
- Disk usage (I/O)
- Network usage (bandwidth).

There are two important attributes to analyse an algorithm. They are:

Performance: How much time/memory/disk/network bandwidth is actually used when a program is run. This depends on the algorithm, machine, compiler, etc.

Complexity: How do the resource requirements of a program or algorithm scale (the growth of resource requirements as a function of input). In other words, what happens to the performance of an algorithm, as the size of the problem being solved gets larger and larger? For example, the time and memory requirements of an algorithm which

computes the sum of 1000 numbers is larger than the algorithm which computes the sum of 2 numbers.

Time Complexity: The maximum time required by a *Turing machine* to execute on any input of length **n**.

Space Complexity: The amount of storage space required by an algorithm varies with the size of the problem being solved. The space complexity is normally expressed as an order of magnitude of the size of the problem, e.g., $O(n^2)$ means that if the size of the problem (n) doubles then the working storage (memory) requirement will become four times.

Determination of Time Complexity

The RAM Model

The random access model (RAM) of computation was devised by John von Neumann to study algorithms. Algorithms are studied in computer science because they are independent of machine and language.

We will do all our design and analysis of algorithms based on RAM model of computation:

- Each "simple" operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine.
- Each memory access takes exactly 1 step.

The complexity of algorithms using big-O notation can be defined in the following way for a problem of size n:

- Constant-time method is "order 1" : *O(1).* The time required is constant independent of the input size.
- Linear-time method is "order n": *O(n).* The time required is proportional to the input size. If the input size doubles, then, the time to run the algorithm also doubles.
- Quadratic-time method is "order N squared": $O(n^2)$. The time required is proportional to the square of the input size. If the input size doubles, then, the time required will increase by four times.

The process of analysis of algorithm (program) involves analyzing each step of the algorithm. It depends on the kinds of statements used in the program.

Consider the following example:

Example 1: Simple sequence of statements

```
Statement 1;
Statement 2;
  ...
  ...
Statement k;
```

The total time can be found out by adding the times for all statements:

Total time = time(statement 1) + time(statement 2) + ... + time(statement *k*).

It may be noted that time required by each statement will greatly vary depending on whether each statement is simple (involves only basic operations) or otherwise. Assuming that each of the above statements involve only basic operation, the time for each simple statement is constant and the total time is also constant: *O(1).*

Example 2: if-then-else statements

In this example, assume the statements are simple unless noted otherwise.

if-then-else statements

```
if (cond) {
   sequence of statements 1
}
else {
   sequence of statements 2
}
```

In this, if-else statement, either sequence 1 will execute, or sequence 2 will execute depending on the boolean condition. The worst-case time in this case is the slower of the two possibilities. For example, if sequence 1 is $O(N^2)$ and sequence 2 is *O(1),* then the worst-case time for the whole if-then-else statement would be $O(N^2)$.

Example 3: for loop

```
for (i = 0; i < n; i + +) {
sequence of statements
}
```

Here, the loop executes n times. So, the sequence of statements also executes *n* times. Since we assume the time complexity of the statements are *O(1),* the total time for the loop is *n* * *O(1),* which is *O(n).* Here, the number of statements does not matter as it will increase the running time by a constant factor and the overall complexity will be same *O(n).*

Example 4:nested for loop

```
for (i = 0; i < n; i + +) {
      for (j = 0; j < m; j + +) {
         sequence of statements
                        }
                    }
```

Here, we observe that, the outer loop executes *n* times. Every time the outer loop executes, the inner loop executes *m* times. As a result of this, statements in the inner loop execute a total of *n* * *m* times. Thus, the time complexity is *O(n * m).* If we modify the conditional variables, where the condition of the inner loop is $j < n$ instead of $j < m$ (i.e., the inner loop also executes *n* times), then the total complexity for the nested loop is $O(n^2)$.

Example 4: Now, consider a function that calculates partial sum of an integer ***n***.

```
int psum(int n)
    {
        int i, partial_sum;
```

```
    partial_sum = 0;                              /* Line 1 */
    for (i = 1; i <= n; i++) {                    /* Line 2 */
        partial_sum  = partial_sum  +  i*i;       /* Line 3 */
                                  }
    return partial_sum;                           /* Line 4 */
}
```

This function returns the sum from $i = 1$ to n of i squared, i.e. psum = $1^2 + 2^2 + 3^2 + \cdots\cdots + n^2$. As we have to determine the running time for each statement in this program, we have to count the number of statements that are executed in this procedure. The code at line 1 and line 4 are one statement each. The **for loop** on line 2 are actually $2n+2$ statements:

- i = 1; statement : simple assignment, hence one statement.
- i <= n; statement is executed once for each value of i from 1 to $n+1$ (till the condition becomes false). The statement is executed $n+1$ times.
- i++ is executed once for each execution of the body of the loop. This is executed n times.

Thus, the sum is $1+ (n+1) + n+1 = 2n+ 3$ times.

In terms of big-O notation defined above, this function is $O(n)$, because if we choose $c=3$, then we see that $cn > 2n+3$. As we have already noted earlier, big-O notation only provides a upper bound to the function, it is also $O(nlog(n))$ and $O(n^2)$, since $n^2 > nlog(n) > 2n+3$. However, we will choose the smallest function that describes the order of the function and it is $O(n)$.

By looking at the definition of Omega notation and Theta notation, it is also clear that it is of $\Theta(n)$, and therefore $\Omega(n)$ too. Because if we choose $c=1$, then we see that $cn < 2n+3$, hence $\Omega(n)$. Since $2n+3 = O(n)$, and $2n+3 = \Omega(n)$, it implies that $2n+3 = \Theta(n)$, too.

It is again reiterated here that smaller order terms and constants may be ignored while describing asymptotic notation. For example, if $f(n) = 4n+6$ instead of $f(n) = 2n +3$ in terms of big-O, Ω and Θ, this does not change the order of the function. The function $f(n) = 4n+6 = O(n)$ (by choosing c appropriately as 5); $4n+6 = \Omega(n)$ (by choosing $c = 1$), and therefore $4n+6 = \Theta(n)$. The essence of this analysis is that in these asymptotic notation, we can count a statement as one, and should not worry about their relative execution time which may depend on several hardware and other implementation factors, as long as it is of the order of 1, i.e. $O(1)$.

Exact analysis of insertion sort:

Let us consider the following pseudocode to analyse the exact runtime complexity of insertion sort.

Line	**Pseudocode**	**Cost factor**	**No. of iterations**
1	For j=2 to length [A] do	$c1$	$(n-1) + 1$
2	{ key = A[j]	$c2$	$(n-1)$
3	i = j – 1	$c3$	$(n-1)$
4	while (i > 0) and (A[i] > key) do	$c4$	$\sum_{j=2}^{n} T_j$
5	{ $A[i+1] = A[I]$	$c4$	$\sum_{j=2}^{n} T_j - 1$

6	*i = I –1 }*	*c5*	$\sum_{j=2}^{n} T_j - 1$
7	A[*I+1*] = key }	*c6*	*n–1*

}

T_j is the time taken to execute the statement during j^{th} iteration.

The statement at line 4 will execute T_j number of times.

The statements at lines 5 and 6 will execute $T_j - 1$ number of times (one step less) each

Line 7 will excute *(n–1)* times

So, total time is the sum of time taken for each line multiplied by their cost factor.

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} T_j + c5\sum_{j=2}^{n} T_j - 1 + c6\sum_{j=2}^{n} T_j - 1 + c7(n-1)$$

Three cases can emerge depending on the initial configuration of the input list. First, the case is where the list was already sorted, second case is the case wherein the list is sorted in reverse order and third case is the case where in the list is in random order (unsorted). The best case scenario will emerge when the list is already sorted.

Worst Case: Worst case running time is an upper bound for running time with any input. It guarantees that, irrespective of the type of input, the algorithm will not take any longer than the worst case time.

Best Case : It guarantees that under any cirumstances the running time of algorithms will at least take this much time.

Average case : This gives the average running time of algorithm. The running time for any given size of input will be the average number of operations over all problem instances for a given size.

Best Case : If the list is already sorted then *A[i]* <= key at line 4. So, rest of the lines in the inner loop will not execute. Then,

$T(n) = c1n + c2(n-1) + c3(n-1) + c4(n-1) = O(n)$, which indicates that the time complexity is linear.

Worst Case: This case arises when the list is sorted in reverse order. So, the boolean condition at line 4 will be true for execution of line 1.

So, step line 4 is executed $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ times

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4(n(n+1)/2 - 1) + c5(n(n-1)/2) + c6(n(n-1)/2) + c7(n-1)$$

$= O(n^2)$.

Average case : In most of the cases, the list will be in some random order. That is, it neither sorted in ascending or descending order and the time complexity will lie some where between the best and the worst case.

$$T(n)_{best} < T(n)_{Avg.} < T(n)_{worst}$$

Figure 1.4 depicts the best, average and worst case run time complexities of algorithms.

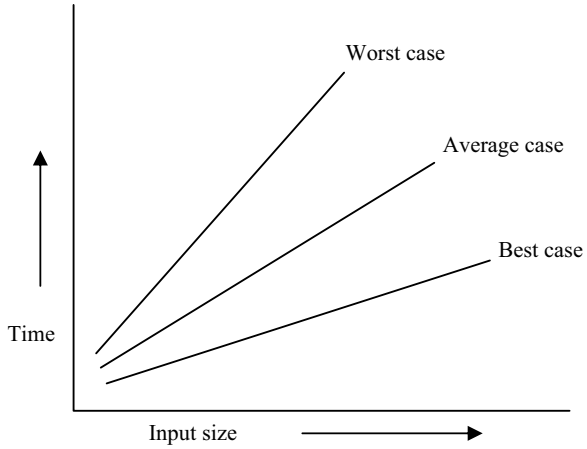


Figure 1.4 : Best, Average and Worst case scenarios

☞ Check Your Progress 2

1) The set of algorithms whose order is O (1) would run in the same time. True/False

2) Find the complexity of the following program in big O notation:

```
printMultiplicationTable(int max){
    for(int i = 1 ; i <= max ; i + +)
            {
            for(int j = 1 ; j <= max ; j + +)
                    cout << (i * j) << " " ;
                    cout << endl ;
            } //for
```

……

3) Consider the following program segment:

```
for (i = 1; i <= n; i *= 2)
  {
   j = 1;
  }
```

What is the running time of the above program segment in big O notation?

……

4) Prove that if $f(n) = n^2 + 2n + 5$ and $g(n) = n^2$ then $f(n) = O(g(n))$.

5) How many times does the following **for loop** will run

```
for (i=1; i<= n; i*2)
    k = k + 1;
end;
```

1.4 CALCULATION OF STORAGE COMPLEXITY

As memory is becoming more and more cheaper, the prominence of runtime complexity is increasing. However, it is very much important to analyse the amount of memory used by a program. If the running time of algorithms is not good then it will

take longer to execute. But, if it takes more memory (the space complexity is more) beyond the capacity of the machine then the program will not execute at all. It is therefore more critical than run time complexity. But, the matter of respite is that memory is reutilized during the course of program execution.

We will analyse this for recursive and iterative programs.

For an iterative program, it is usually just a matter of looking at the variable declarations and storage allocation calls, e.g., number of variables, length of an array etc.

The analysis of recursive program with respect to space complexity is more complicated as the space used at any time is the total space used by all recursive calls active at that time.

Each recursive call takes a constant amount of space and some space for local variables and function arguments, and also some space is allocated for remembering where each call should return to. General recursive calls use linear space. That is, for *n* recursive calls, the space complexity is *O(n).*

Consider the following example: *Binary Recursion (*A *binary-recursive* routine (potentially) calls itself twice).

1. If *n* equals 0 or 1, then return 1
2. Recursively calculate $f(n-1)$
3. Recursively calculate $f(n-2)$
4. Return the sum of the results from steps 2 and 3.

Time Complexity: O(exp *n*)
Space Complexity: O(exp *n*)

Example: Find the greatest common divisor (GCD) of two integers, *m* and *n*.
The algorithm for GCD may be defined as follows**:**

While *m* is greater than zero:
If *n* is greater than *m*, swap *m* and *n*.
Subtract *n* from *m*.
n is the *GCD*

Code in C

```
int gcd(int m, int n)
/* The precondition are : m>0 and n>0.  Let g = gcd(m,n). */
 { while( m > 0 )
    {
     if( n > m )
       { int t = m; m = n; n = t; } /* swap  m and n*/
      /* m >= n > 0 */
       m - = n;
    }
   return n;
 }
```

The space-complexity of the above algorithm is a constant. It just requires space for three integers *m, n* and *t*. So, the space complexity is *O(1)*.

The time complexity depends on the loop and on the condition whether $m>n$ or not. The real issue is, how many iterations take place? The answer depends on both m and n.

Best case: If $m = n$, then there is just one iteration. $O(1)$
Worst case : If $n = 1$,then there are m iterations; this is the worst-case
(also equivalently, if $m = 1$ there are n iterations) $O(n)$.

The *space complexity* of a computer program is the amount of memory required for its proper execution. The important concept behind space required is that unlike time, space can be reused during the execution of the program. As discussed, there is often a trade-off between the time and space required to run a program.

In formal definition, the space complexity is defined as follows:

Space complexity of a Turing Machine: The (worst case) maximum length of the tape required to process an input string of length n.

In complexity theory, the class *PSPACE* is the set of decision problems that can be solved by a Turing machine using a polynomial amount of memory, and unlimited time.

☞ Check Your Progress 3

1) Why space complexity is more critical than time complexity?

……

……

2) What is the space complexity of Euclid Algorithm?

……

……

1.5 CALCULATION OF TIME COMPLEXITY

Example 1: Consider the following of code :

$x = 4y + 3$
$z = z + 1$
$p = 1$
As we have seen, x, y, z and p are all scaler variables and the running time is constant irrespective of the value of x,y,z and p. Here, we emphasize that each line of code may take different time, to execute, but the bottom line is that they will take constant amount of time. Thus, we will describe run time of each line of code as $O(1)$.

Example 2: Binary search

Binary search in a sorted list is carried out by dividing the list into two parts based on the comparison of the key. As the search interval halves each time, the iteration takes place in the search. The search interval will look like following after each iteration
$N, N/2, N/4, N/8$, ... 8, 4, 2, 1
The number of iterations (number of elements in the series) is not so evident from the above series. But, if we take logs of each element of the series, then

$\log_2 N , \log_2 N -1, \log_2 N-2, \log_2 N-3$, ..., 3, 2, 1, 0

As the sequence decrements by 1 each time the total elements in the above series are $log_2 N + 1$. So, the number of iterations is $log_2 N + 1$ which is of the order of $O(log_2 N)$.

Example 3: Travelling Salesman problem
Given: n connected cities and distances between them

Find: tour of minimum length that visits every city.

Solutions: How many tours are possible?
$n*(n-1)...*1 = n!$

Because $n! > 2^{(n-1)}$
So $n! = \Omega(2^n)$ (lower bound)

As of now, there is no algorithm that finds a tour of minimum length as well as covers all the cities in polynomial time. However, there are numerous very good heuristic algorithms.

The complexity Ladder:

- $T(n) = O(1)$. This is called constant growth. $T(n)$ does not grow at all as a function of n, it is a constant. For example, array access has this characteristic. $A[i]$ takes the same time independent of the size of the array A.

- $T(n) = O(log_2 (n))$. This is called logarithmic growth. $T(n)$ grows proportional to the base 2 logarithm of n. Actually, the base of logarithm does not matter. For example, binary search has this characteristic.

- $T(n) = O(n)$. This is called linear growth. $T(n)$ grows linearly with n. For example, looping over all the elements in a one-dimensional array of n elements would be of the order of $O(n)$.

- $T(n) = O(n \log (n))$. This is called **nlogn** growth. $T(n)$ grows proportional to n times the base 2 logarithm of n. Time complexity of Merge Sort has this characteristic. In fact, no sorting algorithm that uses comparison between elements can be faster than n log n.

- $T(n) = O(n^k)$. This is called polynomial growth. $T(n)$ grows proportional to the k-th power of n. We rarely consider algorithms that run in time $O(n^k)$ where k is bigger than 2 , because such algorithms are very slow and not practical. For example, selection sort is an $O(n^2)$ algorithm.

- $T(n) = O(2^n)$ This is called exponential growth. $T(n)$ grows exponentially. Exponential growth is the most-danger growth pattern in computer science. Algorithms that grow this way are basically useless for anything except for very small input size.

Table 1.1 compares various algorithms in terms of their complexities.

Table 1.2 compares the typical running time of algorithms of different orders.

The growth patterns above have been listed in order of increasing size.

That is, $O(1) < O(log(n)) < O(n \log(n)) < O(n^2) < O(n^3), \ldots, O(2^n)$.

Notation	Name	Example
$O(1)$	Constant	Constant growth. Does

		not grow as a function of n. For example, accessing array for one element A[i]
O(log n*)*	Logarithmic	Binary search
*O(*n*)*	Linear	Looping over n elements, of an array of size n (normally).
*O(*n *log* n*)*	Sometimes called "linearithmic"	Merge sort
O(n^2*)*	Quadratic	Worst time case for insertion sort, matrix multiplication
O(n^c*)*	Polynomial, sometimes "geometric"	
O(c^n*)*	Exponential	
*O(*n!*)*	Factorial	

Table 1.1 : Comparison of various algorithms and their complexities

Array size	Logarithmic: $\log_2 N$	Linear: N	Quadratic: N^2	Exponential: 2^N
8	3	8	64	256
128	7	128	16,384	$3.4*10^{38}$
256	8	256	65,536	$1.15*10^{77}$
1000	10	1000	1 million	$1.07*10^{301}$
100,000	17	100,000	10 billion	.....

Table 1.2: Comparison of typical running time of algorithms of different orders

1.6 SUMMARY

Computational complexity of algorithms are generally referred to by space complexity (space required for running program) and time complexity (time required for running the program). In the field of computer of science, the concept of runtime complexity has been studied vigorously. Enough research is being carried out to find more efficient algorithms for existing problems. We studied various asymptotic notation, to describe the time complexity and space complexity of algorithms, namely the *big-O, Omega* and *Theta* notations. These asymptotic orders of time and space complexity describe how best or worst an algorithm is for a sufficiently large input.

We studied about the process of calculation of runtime complexity of various algorithms. The exact analysis of insertion sort was discussed to describe the best case, worst case and average case scenario.

1.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1) True

2) True
3) False
4) False
5) *O(n^2), O(n^3), O(n log n), O(log n),O(log n)*

Check Your Progress 2

1) True
2) *O(max*(2*max))=O(2*max*max) = O(2 * n * n) = O($2n^2$) = O(n^2)*
3) *O(log(n))*
5) *log n*

Check Your Progress 3

1) If the running time of algorithms is not good, then it will take longer to execute. But, if it takes more memory (the space complexity is more) beyond the capacity of the machine then the program will not execute.
2) *O(1)*.

1.8 FURTHER READINGS

1. *Fundamentals of Data Structures in C++;* E.Horowitz, Sahni and D.Mehta; Galgotia Publications.

2. *Data Structures and Program Design in C*; Kruse, C.L.Tonodo and B.Leung; Pearson Education.

Reference Websites

http://en.wikipedia.org/wiki/Big_O_notation
http://www.webopedia.com

UNIT 2 ARRAYS

Structure **Page Nos.**

2.0	Introduction	23
2.1	Objectives	24
2.2	Arrays and Pointers	24
2.3	Sparse Matrices	25
2.4	Polynomials	28
2.5	Representation of Arrays	30
	2.5.1 Row Major Representation	
	2.5.2 Column Major Representation	
2.6	Applications	31
2.7	Summary	32
2.8	Solutions/Answers	32
2.9	Further Readings	32

2.0 INTRODUCTION

This unit introduces a data structure called Arrays. The simplest form of array is a one-dimensional array that may be defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations. For example, an array may contain all integers or all characters or any other data type, but may not contain a mix of data types.

The general form for declaring a single dimensional array is:

data_type array_name[expression];

where data_type represents data type of the array. That is, integer, char, float etc. array_name is the name of array and expression which indicates the number of elements in the array.

For example, consider the following C declaration:

```
int   a[100];
```

It declares an array of 100 integers.

The amount of storage required to hold an array is directly related to its type and size. For a single dimension array, the total size in bytes required for the array is computed as shown below.

Memory required (in bytes) = size of (data type) X length of array

The first array index value is referred to as its lower bound and in C it is always 0 and the maximum index value is called its upper bound. The number of elements in the array, called its range is given by upper bound-lower bound.

We store values in the arrays during program execution. Let us now see the process of initializing an array while declaring it.

```
int a[4] = {34,60,93,2};
int b[] = {2,3,4,5};
float c[] = {-4,6,81,– 60};
```

We conclude the following facts from these examples:

(i) If the array is initialized at the time of declaration, then the dimension of the array is optional.
(ii) Till the array elements are not given any specific values, they contain garbage values.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- use Arrays as a proper data structure in programs;
- know the advantages and disadvantages of Arrays;
- use multidimensional arrays, and
- know the representation of Arrays in memory.

2.2 ARRAYS AND POINTERS

C compiler does not check the bounds of arrays. It is your job to do the necessary work for checking boundaries wherever needed.

One of the most common arrays is a string, which is simply an array of characters terminated by a null character. The value of the null character is zero. A string constant is a one-dimensional array of characters terminated by a null character(\0).

For example, consider the following:

char message[ ]= {'e', 'x', 'a', 'm', 'p', 'l','e','\0'};

Also, consider the following string which is stored in an array:

"sentence\n"

Figure 2.1 shows the way a character array is stored in memory. Each character in the array occupies one byte of memory and the last character is always '\0'. Note that '\0' and '0' are not the same. The elements of the character array are stored in contiguous memory locations.

s	**e**	**n**	**t**	**e**	**n**	**c**	**e**	**\n**	**\0**

Figure 2.1: String in Memory

C concedes a fact that the user would use strings very often and hence provides a short cut for initialization of strings.

For example, the string used above can also be initialized as

char name[ ] = "sentence\n";

Note that, in this declaration '\0' is not necessary. C inserts the null character automatically.

Multidimensional arrays are defined in the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets and so on.

The format of declaration of a multidimensional array in C is given below:

data_type array_name [expr 1] [expr 2] …. [expr n];

where data_type is the type of array such as int, char etc., array_name is the name of array and expr 1, expr 2, ….expr n are positive valued integer expressions.

The schematic of a two-dimensional array of size 3 × 5 is shown in *Figure 2.2.*

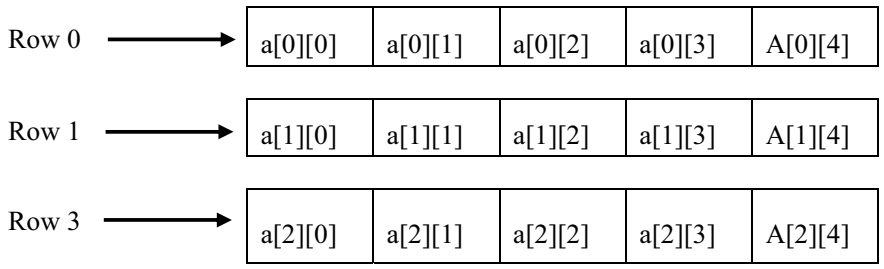


Figure 2.2: Schematic of a Two-Dimensional Array

In the case of a two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:

bytes = size of 1^{st} index × size of 2^{nd} index × size of (base type)

The pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array.

Consider the following array:

char p[10];

p and &p[0] are identical because the address of the first element of an array is the same as the address of the array. So, an array name without an index generates a pointer. Conversely a pointer can be indexed as if it were declared to be an array.

For example, consider the following program fragment:

```
int *x, a [10];
x = a;
x[5] = 100;
* (x+5) = 100;
```

Both assignment statements place the value 100 in the sixth element of a. Furthermore the (0,4) element of a two-dimensional array may be referenced in the following two ways: either by array indexing a[0][4], or by the pointer *((int *) a+4).

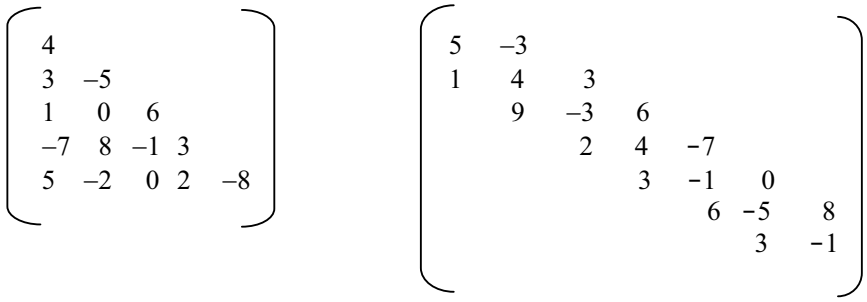
In general, for any two-dimensional array a[j][k] is equivalent to:

*((base type *)a + (j * rowlength)*k)

2.3 SPARSE MATRICES

Matrices with good number of zero entries are called sparse matrices.

Consider the following matrices of *Figure 2.3.*



(a) **(b)**

Figure 2.3: (a) Triangular Matrix (b) Tridiagonal Matrix

A triangular matrix is a square matrix in which all the elements either above or below the main diagonal are zero. Triangular matrices are sparse matrices. A tridiagonal matrix is a square matrix in which all the elements except for the main diagonal, diagonals on the immediate upper and lower side are zeroes. Tridiagonal matrices are also sparse matrices.

Let us consider a sparse matrix from storage point of view. Suppose that the entire sparse matrix is stored. Then, a considerable amount of memory which stores the matrix consists of zeroes. This is nothing but wastage of memory. In real life applications, such wastage may count to megabytes. So, an efficient method of storing sparse matrices has to be looked into.

Figure 2.4 shows a sparse matrix of order 7 × 6.

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	5	0	0
1	0	4	0	0	0	0
2	0	0	0	0	9	0
3	0	3	0	2	0	0
4	1	0	2	0	0	0
5	0	0	0	0	0	0
6	0	0	8	0	0	0

Figure 2.4: Representation of a sparse matrix of order 7 × 6

A common way of representing non zero elements of a sparse matrix is the 3-tuple form. The first row of sparse matrix always specifies the number of rows, number of columns and number of non zero elements in the matrix. The number 7 represents the total number of rows sparse matrix. Similarly, the number 6 represents the total number of columns in the matrix. The number 8 represents the total number of non zero elements in the matrix. Each non zero element is stored from the second row, with the 1st and 2nd elements of the row, indicating the row number and column number respectively in which the element is present in the original matrix. The 3rd element in this row stores the actual value of the non zero element. For example, the 3- tuple representation of the matrix of *Figure 2.4* is shown in *Figure 2.5*.

```
7,    7,    9
0,    3,    5
1,    1,    4
2,    4,    9
3,    1,    3
3,    3,    2
4,    0,    1
4,    2,    2
6,    2,    8
```

Figure 2.5: 3-tuple representation of Figure 2.4

The following program 1.1 accepts a matrix as input, which is sparse and prints the corresponding 3-tuple representations.

Program 1.1

/* The program accepts a matrix as input and prints the 3-tuple representation of it*/

```
#include<stdio.h>

void main()
{
        int a[5][5],rows,columns,i,j;

        printf("enter the order of the matrix. The order should be less than 5 × 5:\n");
        scanf("%d %d",&rows,&columns);
        printf("Enter the elements of the matrix:\n");

        for(i=0;i<rows;i++)
          for(j=0;j<columns;j++)

          { scanf("%d",&a[i][j]);
          }
           printf("The 3-tuple representation of the matrix is:\n");

        for(i=0;i<rows;i++)
                for(j=0;j<columns;j++)
                {
                        if (a[i][j]!=0)
                        {
                                printf("%d     %d       %d\n", (i+1),(j+1),a[i][j]);
                        }
                }
}
```

Output:

```
enter the order of the matrix. The order should be less than 5 × 5:
3 3
Enter the elements of the matrix:
1 2 3
0 1 0
0 0 4
The 3-tuple representation of the matrix is:
1    1       1
1    2       2
1    3       3
2    2       1
3    3       4
```

The program initially prompted for the order of the input matrix with a warning that the order should not be greater than 5 × 5. After accepting the order, it prompts for the elements of the matrix. After accepting the matrix, it checks each element of the matrix for a non zero. If the element is non zero, then it prints the row number and column number of that element along with its value.

☞ Check Your Progress 1

1) If the array is _____ at the time of declaration, then the dimension of the array is optional.

2) A sparse matrix is a matrix which is having good number of _____ elements.

3) At maximum, an array can be a two-dimensional array. True/False

2.4 POLYNOMIALS

Polynomials like $5x^4 + 2x^3 + 7x^2 + 10x - 8$ can be represented using arrays. Arithmetic operations like addition and multiplication of polynomials are common and most often, we need to write a program to implement these operations.

The simplest way to represent a polynomial of degree '*n*' is to store the coefficient of (*n+1*) terms of the polynomial in an array. To achieve this, each element of the array should consist of two values, namely, coefficient and exponent. While maintaining the polynomial, it is assumed that the exponent of each successive term is less than that of the previous term. Once we build an array to represent a polynomial, we can use such an array to perform common polynomial operations like addition and multiplication.

Program 1.2 accepts two polynomials as input and adds them.

Program 1.2

/* The program accepts two polynomials as input and prints the resultant polynomial due to the addition of input polynomials*/

```
#include<stdio.h>

void main()
{
        int poly1[6][2],poly2[6][2],term1,term2,match,proceed,i,j;

        printf("Enter the number of terms in the first polynomial. They should be less
than 6:\n");
        scanf("%d",&term1);
        printf("Enter the number of terms in the second polynomial. They should be
less than 6:\n");
        scanf("%d",&term2);
        printf("Enter the coefficient and exponent of each term of the first
polynomial:\n");
        for(i=0;i<term1;i++)
        {scanf("%d %d",&poly1[i][0],&poly1[i][1]);
        }

   printf("Enter the coefficient and exponent of each term of the second
polynomial:\n");
   for(i=0;i<term2;i++)
        {scanf("%d %d",&poly2[i][0],&poly2[i][1]);
```

```
        }
   printf("The resultant polynomial due to the addition of the input two
polynomials:\n");

        for(i=0;i<term1;i++)
        {
                match=0;
                for(j=0;j<term2;j++)

                { if (match==0)

                        if(poly1[i][1]==poly2[j][1])
                        { printf("%d    %d\n",(poly1[i][0]+poly2[j][0]), poly1[i][1]);
                          match=1;

                        }
                }
        }

for(i=0;i<term1;i++)
{  proceed=1;

                for(j=0;j<term2;j++)
                {  if(proceed==1)
                        if(poly1[i][1]!=poly2[j][1])
                         proceed=1;
                        else
                                proceed=0;

                        }
                if (proceed==1)
                        printf("%d %d\n",poly1[i][0],poly1[i][1]);

        }

for(i=0;i<term2;i++)
{  proceed=1;

                for(j=0;j<term1;j++)
                {  if(proceed==1)
                        if(poly2[i][1]!=poly1[j][1])
                         proceed=1;
                        else
                                proceed=0;

                        }
                if (proceed==1)
                        printf("%d %d",poly2[i][0],poly2[i][1]);
                }

}
```

Output:

```
Enter the number of terms in the first polynomial.They should be less than 6 : 5.
Enter the number of terms in the second polynomial.They should be less than 6 : 4.
Enter the coefficient and exponent of each term of the first polynomial:
1 2
2 4
3 6
```

1 8
5 7

Enter the coefficient and exponent of each term of the second polynomial:
5 2
6 9
3 6
5 7

The resultant polynomial due to the addition of the input two polynomials:
6 2
6 6
10 7
2 4
1 8
6 9

The program initially prompted for the number of terms of the two polynomials. Then, it prompted for the entry of the terms of the two polynomials one after another. Initially, it adds the coefficients of the corresponding terms of both the polynomials whose exponents are the same. Then, it prints the terms of the first polynomial who does not have corresponding terms in the second polynomial with the same exponent. Finally, it prints the terms of the second polynomial who does not have corresponding terms in the first polynomial.

2.5 REPRESENTATION OF ARRAYS

It is not uncommon to find a large number of programs which process the elements of an array in sequence. But, does it mean that the elements of an array are also stored in sequence in memory. The answer depends on the operating system under which the program is running. However, the elements of an array are stored in sequence to the extent possible. If they are being stored in sequence, then how are they sequenced. Is it that the elements are stored row wise or column wise? Again, it depends on the operating system. The former is called row major order and the later is called column major order.

2.5.1 Row Major Representation

The first method of representing a two-dimensional array in memory is the row major representation. Under this representation, the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set, and so forth.

The schematic of row major representation of an Array is shown in *Figure 2.6.*
Let us consider the following two-dimensional array:

a b c d
e f g h
i j k l

To make its equivalent row major representation, we perform the following process:

Move the elements of the second row starting from the first element to the memory location adjacent to the last element of the first row. When this step is applied to all the rows except for the first row, you have a single row of elements. This is the Row major representation.

By application of above mentioned process, we get {a, b, c, d, e, f, g, h, i, j, k, l }

Row 0	Row 1	Row 2	…..	Row i		

Figure 2.6: Schematic of a Row major representation of an Array

2.5.2 Column Major Representation

The second method of representing a two-dimensional array in memory is the column major representation. Under this representation, the first column of the array occupies the first set of the memory locations reserved for the array. The second column occupies the next set and so forth. The schematic of a column major representation is shown in *Figure 2.7.*

Consider the following two-dimensional array:

a b c d
e f g h
i j k l

To make its equivalent column major representation, we perform the following process:

Transpose the elements of the array. Then, the representation will be same as that of the row major representation.

By application of above mentioned process, we get {a, e, i, b, f, j, c, g, k, d, h, i}

Col 0	Col 1	Col 2	…..	Col i		

Figure 2.7: Schematic of a Column major representation of an Array

☞ Check Your Progress 2

1) An array can be stored either_____or _____.

2) In _____, the elements of array are stored row wise.

3) In _____, the elements of array are stored column wise.

2.6 APPLICATIONS

Arrays are simple, but reliable to use in more situations than you can count. Arrays are used in those problems when the number of items to be solved is fixed. They are easy to traverse, search and sort. It is very easy to manipulate an array rather than other subsequent data structures. Arrays are used in those situations where in the size of array can be established before hand. Also, they are used in situations where the insertions and deletions are minimal or not present. Insertion and deletion operations will lead to wastage of memory or will increase the time complexity of the program due to the reshuffling of elements.

2.7 SUMMARY

In this unit, we discussed the data structure **arrays** from the application point of view and representation point of view. Two applications namely representation of a sparse matrix in a 3-tuple form and addition of two polynomials are given in the form of programs. The format for declaration and utility of both single and two-dimensional arrays are covered. Finally, the most important issue of representation was discussed. As part of it, row major and column major orders are discussed.

2.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Initialized
2) Zero
3) False

Check Your Progress 2

1) Row wise, column wise
2) Row major representation
3) Column major representation

2.9 FURTHER READINGS

Reference Books

1. *Data Structures using C and C++*, Yedidyah Langsam, Moshe J.Augenstein, Aaron M Tanenbaum, Second Edition, PHI Publications.
2. *Data Structures, Seymour Lipscutz, Schaum's outline series*, McGraw Hill

Reference Websites

http://www.webopedia.com

UNIT 3 LISTS

Structure **Page Nos.**

3.0	Introduction	33
3.1	Objectives	33
3.2	Abstract Data Type-List	33
3.3	Array Implementation of Lists	34
3.4	Linked Lists-Implementation	38
3.5	Doubly Linked Lists-Implementation	44
3.6	Circularly Linked Lists-Implementation	46
3.7	Applications	54
3.8	Summary	56
3.9	Solutions/Answers	56
3.10	Further Readings	56

3.0 INTRODUCTION

In the previous unit, we have discussed arrays. Arrays are data structures of fixed size. Insertion and deletion involves reshuffling of array elements. Thus, array manipulation is time-consuming and inefficient. In this unit, we will see abstract data type-lists, array implementation of lists and linked list implementation, Doubly and Circular linked lists and their applications. In linked lists, items can be added or removed easily to the end or beginning or even in the middle.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- define and declare Lists;
- understand the terminology of Singly linked lists;
- understand the terminology of Doubly linked lists;
- understand the terminology of Circularly linked lists, and
- use the most appropriate list structure in real life situations.

3.2 ABSTRACT DATA TYPE-LIST

Abstract Data Type (**ADT**) is a useful tool for specifying the logical properties of data type. An ADT is a collection of values and a set of operations on those values. Mathematically speaking, "a **TYPE** is a set, and elements of set are **Values** of that type".

ADT List

A list of elements of type **T** is a finite **sequence** of elements of type **T** together with the operations of create, update, delete, testing for empty, testing for full, finding the size, traversing the elements.

In defining Abstract Data Type, we are not concerned with space or time efficiency as well as about implementation details. The elements of a list may be integers, characters, real numbers and combination of multiple data types.

Consider a real world problem, where we have a company and we want to store the details of employees. To store this, we need a data type which can store the type details containing names of employee, date of joining, etc. The list of employees may

increase depending on the recruitment and may decrease on retirements or termination of employees. To make it very simple and for understanding purposes, we are taking the name of employee field and ignoring the date of joining etc. The operations we have to perform on this list of employees are creation, insertion, deletion, visiting, etc. We define employee_list as

```
typedef struct
{
        char    name[20];
        ……….
        …….
}  emp_list;
```

Operations on emp_list can be defined as

```
Create_emplist (emp_list    * emp_list )
{
/* Here, we will be writing create function by taking help of 'C' programming
language. */
 }
```

The list has been created and **name** is a valid entry in **emplist**, and position **p** specifies the position in the list where name has to inserted

```
insert_emplist (emp_list    * emp_list ,    char      *name, int position    )
{
/* Here,  we will be writing insert function by taking help of 'C' programming
language. */
}
delete_emplist (emp_list    * emp_list,    char      *name)
{
/* Here, we will be writing delete function by taking help of 'C' programming
language. */
}
visit_emplist (emp_list    * emp_list )
{
/* Here, we will be writing visit function by taking help of 'C' programming
language. */
}
```

The list can be implemented in two ways: the contiguous (Array) implementation and the linked (pointer) implementation. In contiguous implementation, the entries in the list are stored next to each other within an array. The linked list implementation uses pointers and dynamic memory allocation. We will be discussing array and linked list implementation in our next section.

3.3 ARRAY IMPLEMENTATION OF LISTS

In the array implementation of lists, we will use array to hold the entries and a separate counter to keep track of the number of positions are occupied. A structure will be declared which consists of Array and counter.

```
typedef struct
{
        int count;
        int entry[100];
}list;
```

For simplicity, we have taken list entry as integer. Of course, we can also take list entry as structure of employee record or student record, etc.

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	

Insertion

In the array implementation of lists, elements are stored in continuous locations. To add an element to the list at the end, we can add it without any problem. But, suppose if we want to insert the element at the beginning or middle of the list, then we have to rewrite all the elements after the position where the element has to be inserted. We have to shift $(n)^{th}$ element to $(n+1)^{th}$ position, where 'n' is number of elements in the list. The $(n-1)^{th}$ element to $(n)^{th}$ position and this will continue until the (r) th element to $(r + 1)^{th}$ position, where 'r' is the position of insertion. For doing this, the **count** will be incremented.

From the above example, if we want to add element '35' after element '33'. We have to shift 77 to 8^{th} position, 66 to 7^{th} position, so on, 44 to 5^{th} position.

Before Insertion

Count	1	2	3	4	5	6	7	
	11	22	33	44	55	66	77	

Step 1

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	77

Step 2

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	66	77

Step 3

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	55	66	77

Step 4

Count	1	2	3	4	5	6	7	8
	11	22	33	44	44	55	66	77

Step 5

Count	1	2	3	4	5	6	7	8
	11	22	33	35	44	55	66	77

Program 3.1 will demonstrate the insertion of an element at desired position

```
/* Inserting an element into contiguous list (Linear Array) at specified position */
/* contiguous_list.C */
# include<stdio.h>
/* definition of linear list */
typedef struct
{
        int data[10];
        int count;
}list;
/*prototypes of functions */
void insert(list *, int, int);
void create(list *);
```

```
void traverse(list *);

/* Definition of the insert funtion */

void insert(list *start, int position, int element)
{
        int temp = start->count;
        while( temp >= position)
        {
                start->data[temp+1] = start->data[temp];
                temp --;
        }

        start->data[position] = element;
        start->count++ ;
}

/* definition of create function to  READ data values into the list */

void create(list *start)
{
        int i=0, test=1;
        while(test)
        {
                fflush(stdin);
                printf("\n input value value for: %d:(zero to come out) ", i);
                scanf("%d", &start->data[i]);

                if(start->data[i] == 0)
                        test=0;
                else
                        i++;
        }
        start->count=i;
}

/* OUTPUT FUNCTION TO PRINT ON THE CONSOLE */

void traverse(list *start)
{
        int i;
        for(i = 0; i< start->count; i++)
        {
                printf("\n Value at the position: %d: %d ", i, start->data[i]);
        }
}

/* main function */

void main( )
{
        int position, element;
        list l;
        create(&l);
        printf("\n Entered list as follows:\n");
        fflush(stdin);
        traverse(&l);
```

```
        fflush(stdin);
        printf("\n input the position where you want to add a new data item:");
        scanf("%d", &position);
        fflush(stdin);
        printf("\n input the value for the position:");
        scanf("%d", &element);
        insert(&l, position, element);
        traverse(&l);
}
```

Program 3.1: Insertion of an element into a linear array.

Deletion

To delete an element in the list at the end, we can delete it without any problem. But, suppose if we want to delete the element at the beginning or middle of the list, then, we have to rewrite all the elements after the position where the element that has to be deleted exists. We have to shift $(r+1)^{th}$ element to r^{th} position , where 'r' is position of deleted element in the list, the $(r + 2)^{th}$ element to $(r + 1)^{th}$ position, and this will continue until the $(n)^{th}$ element to $(n-1)^{th}$ position, where n is the number of elements in the list. And then the count is decremented.

From the above example, if we want to delete an element '44' from list. We have to shift 55 to 4^{th} position, 66 to 5^{th} position, 77 to 6^{th} position.

Before deletion

Count 1 2 3 4 5 6 7

11	22	33	44	55	66	77	

Step 1

Count 1 2 3 4 5 6 7

11	22	33	55	55	66	77	

Step 2

Count 1 2 3 4 5 6 7

11	22	33	55	66	66	77	

Step 3

Count 1 2 3 4 5 6

11	22	33	55	66	77		

Program 3.2 will demonstrate deletion of an element from the linear array

```
/* declaration  of delete_list function */
void delete_list(list *, int);

/* definition of delete_list  function*/
/* the position of the element is given by the user and the element is deleted from the
list*/
void delete_list(list *start, int position)
{
        int temp = position;
        printf("\n information which we have to delete: %d",l->data[position]);

        while( temp <= start->count-1)
```

```
        {
                start->data[temp] = start->data[temp+1];
                temp ++;
        }
        start->count = start->count - 1 ;
}

/* main  function */
void main()
{
        ……..
        ……

        printf("\n input the position of element you want to delete:");
        scanf("%d", &position);
        fflush(stdin);
        delete_list(&l, position);
        traverse(&l);
}
```

Program 3.2: Deletion of an element from the linear array

3.4 LINKED LISTS - IMPLEMENTATION

The Linked list is a chain of structures in which each structure consists of data as well as pointer, which stores the address (link) of the next logical structure in the list.

A linked list is a data structure used to maintain a dynamic series of data. Think of a linked list as a line of bogies of train where each bogie is connected on to the next bogie. If you know where the first bogie is, you can follow its link to the next one. By following links, you can find any bogie of the train. When you get to a bogie that isn't holding (linked) on to another bogie, you know you are at the end.

Linked lists work in the same way, except programmers usually refer to nodes instead of bogies. A single node is defined in the same way as any other user defined type or object, except that it also contains a pointer to a variable of the same type as itself.

We will be seeing how the linked list is stored in the memory of the computer. In the following *Figure 3.1*, we can see that **start** is a pointer which is pointing to the node which contains data as *madan* and the node *madan* is pointing to the node *mohan* and the last node *babu* is not pointing to any node. 1000,1050,1200 are memory addresses.

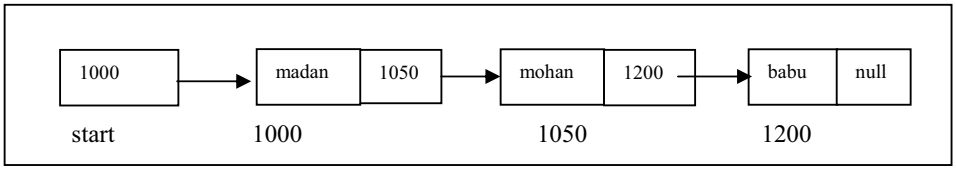


Figure 3.1: A Singly linked list

Consider the following definition:

```
typedef struct node
{
   int data;
   struct node *next;
} list;
```

Once you have a definition for a list node, you can create a list simply by declaring a pointer to the first element, called the "head". A pointer is generally used instead of a regular variable. List can be defined as

list *head;

It is as simple as that! You now have a linked list data structure. It isn't altogether useful at the moment. You can see if the list is empty. We will be seeing how to declare and define list-using pointers in the following program 3.3.

```
#include <stdio.h>

typedef struct node
{
   int data;
   struct node *next;
} list;

int main()
{
   list *head = NULL; /* initialize list head to NULL */
   if (head == NULL)
   {
      printf("The list is empty!\n");
   }
}
```

Program 3.3: Creation of a linked list

In the next example (Program 3.4), we shall look to the process of addition of new nodes to the list with the function create_list().

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list list;

void main()
{
        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
        printf("  \n  traversing the list \n");
        traverse(head);
        printf("\n number of elements in the list   %d \n", count(head));
}

void create(list *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
```

```
        if(start->data == -1111)
                start->next=NULL;
        else
        {
                start->next=(list*)malloc(sizeof(list));
                create(start->next);
        }
}

void traverse(list *start)
{
        if(start->next!=NULL)
        {
                printf("%d --> ", start->data);
                traverse(start->next);
        }
}

int count(list *start)

{
        if(start->next == NULL)
                return 0;
        else
                return (1+count(start->next));
}
```

Program 3.4: Insertion of elements into a Linked list

ALGORITHM (Insertion of element into a linked list)

Step 1	Begin
Step 2	if the list is empty or a new element comes before the start (head) element, then insert the new element as start element.
Step 3	else, if the new element comes after the last element, then insert the new element as the end element.
Step 4	else, insert the new element in the list by using the find function, find function returns the address of the found element to the insert_list function.
Step 5	End.

Figure 3.2 depicts the scenario of a linked list of two elements and a new element which has to be inserted between them. *Figure 3.3* depicts the scenario of a linked list after insertion of a new element into the linked list of *Figure 3.2*.

Before insertion

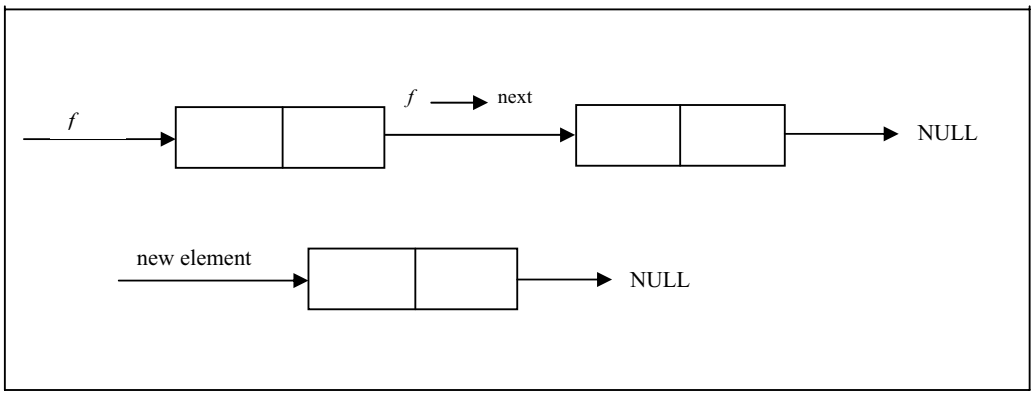


Figure 3.2: A linked list of two elements and an element that is to be inserted

After insertion

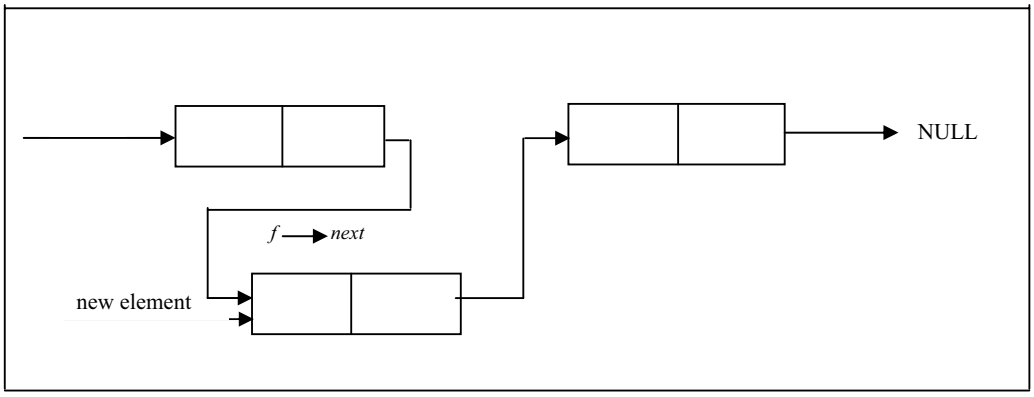


Figure 3.3: Insertion of a new element into linked list

Program 3.5 depicts the code for the insertion of an element into a linked list by searching for the position of insertion with the help of a **find** function.

INSERT FUNCTION

```
/*prototypes of insert and find functions */
list * insert_list(list *);
list * find(list *, int);
/*definition of insert function */
list * insert_list(list *start)
{
        list *n, *f;
        int key, element;
        printf("enter value of new element");
        scanf("%d", &element);
        printf("eneter value of key element");
        scanf("%d",&key);
        if(start->data ==key)
        {
                n=(list *)mallo(sizeof(list));
                n->data=element;
                n->next = start;
                start=n;
        }
        else
        {
                f = find(start, key);
```

```
                if(f == NULL)
                        printf("\n key is not found \n");
                else
                {
                        n=(list*)malloc(sizeof(list));
                        n->data=element;
                        n->next=f->next;
                        f->next=n;
                }
        }
        return(start);
}
/*definition of find function */
list * find(list *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}

void main()
{
        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
        printf("  \n  traversing the created list \n");
        traverse(head);
        printf("\n number of elements in the list   %d \n", count(head));
        head=insert_list(head);
        printf("  \n  traversing the list after insert \n");
        traverse(head);
}
```

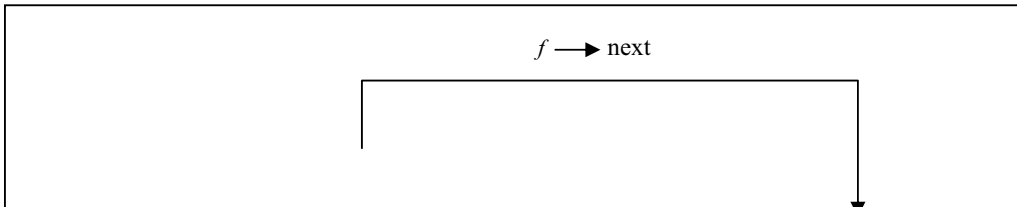
Program 3.5: Insertion of an element into a linked list at a specific position

ALGORITHM (Deletion of an element from the linked list)

Step 1 Begin
Step 2 if the list is empty, then element cannot be deleted
Step 3 else, if element to be deleted is first node, then make the start (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step 5 End

Figure 3.4 depicts the process of deletion of an element from a linked list.

After Deletion



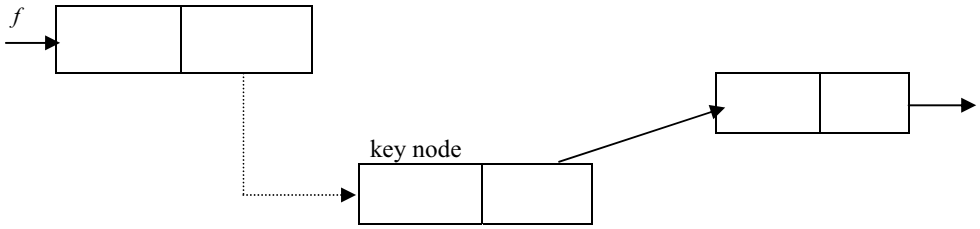


Figure 3.4: Deletion of an element from the linked list (Dotted line depicts the link prior to deletion)

Program 3.6 depicts the deletion of an element from the linked list. It includes a function which specifically searches for the element to be deleted.

DELETE_LIST FUNCTION

```
/*  prototype of delete_function */
list *delete_list(list *);
list *find(list *, int);

/*definition of delete_list */
list *delete_list(list *start)
{
        int key; list * f, * temp;
        printf("\n enter the value of element to be deleted \n");
        scanf("%d", &key);
        if(start->data == key)
        {
                temp=start->next;
                free(start);
                start=temp;
        }
        else
        {
                f = find(start,key);
                if(f==NULL)
                        printf("\n key not fund");
                else
                {
                        temp = f->next->next;
                        free(f->next);
                        f->next=temp;
                }
        }
        return(start);
}
void main()
{

        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
```

```
        printf(“  \n  traversing the created list \n”);
        traverse(head);
        printf(“\n number of elements in the list   %d \n”, count(head));
        head=insert(head);
        printf(“  \n  traversing the list after insert \n”);
        traverse(head);
        head=delete_list(head);
        printf(“  \n  traversing the list after delete_list \n”);
        traverse(head);
}
```

Program 3.6: Deletion of an element from the linked list by searching for element that is to be deleted

3.5 DOUBLY LINKED LISTS-IMPLEMENTATION

In a singly linked list, each element contains a pointer to the next element. We have seen this before. In single linked list, traversing is possible only in one direction. Sometimes, we have to traverse the list in both directions to improve performance of algorithms. To enable this, we require links in both the directions, that is, the element should have pointers to the right element as well as to its left element. This type of list is called **doubly linked list**.

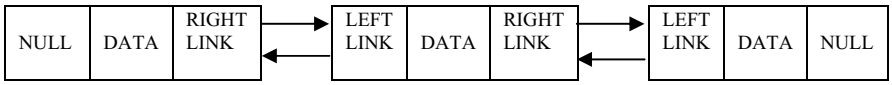


Figure 3.5: A Doubly Linked List

Doubly linked list **(***Figure 3.5***)** is defined as a collection of elements, each element consisting of three fields:

- pointer to left element,
- data field, and
- pointer to right element.

Left link of the leftmost element is set to NULL which means that there is no left element to that. And, right link of the rightmost element is set to NULL which means that there is no right element to that.

ALGORITHM (Creation)

Step 1 begin

Step 2 define a structure ELEMENT with fields
Data
Left pointer
Right pointer

Step 3 declare a pointer by name head and by using (malloc()) memory allocation function allocate space for one element and store the address in head pointer
Head = (ELEMENT *) malloc(sizeof(ELEMENT))

Step 4 read the value for head->data
head->left = NULL
head->right = (ELEMENT *) malloc(size of (ELEMENT))

Step 5 repeat step3 to create required number of elements

Step 6 end

Program 3.7 depicts the creation of a Doubly linked list.

```
/* CREATION OF A DOUBLY LINKED LIST */
/* DBLINK.C */

# include <stdio.h>
# include <malloc.h>

struct dl_list
{
        int data;
        struct dl_list *right;
        struct dl_list *left;
};
typedef struct dl_list dlist;

void dl_create (dlist *);
void traverse (dlist *);

/* Function creates a simple doubly linked list */

void dl_create(dlist *start)
{
        printf("\n Input the values of the element -1111 to come out :  ");
        scanf("%d", &start->data);
        if(start->data != -1111)
        {
                start->right = (dlist *) malloc(sizeof(dlist));
                start->right->left = start;
                start->right->right = NULL;
                dl_create(start->right);
        }
        else
        start->right = NULL;
}

/* Display the list */

void traverse (dlist *start)
{
        printf("\n traversing the list using right pointer\n");
        do {
                printf(" %d = ", start->data);
                start = start->right;
        } while (start->right);  /* Show value of last start only one time */


        printf("\n traversing the list using left pointer\n");
        start=start->left;
        do
        {
                printf(" %d =", start->data);
                start = start->left;
        }while(start->right);
}
```

```
void main()
{
        dlist *head;
        head = (dlist *) malloc(sizeof(dlist));
        head->left=NULL;
        head->right=NULL;
        dl_create(head);
        printf("\n Created doubly linked list is as follows");
        traverse(head);
}
```

Program 3.7: Creation of a Doubly Linked List

OUTPUT

```
Input the values of the element -1111 to come out :
1
Input the values of the element -1111 to come out :
2
Input the values of the element -1111 to come out :
3
Input the values of the element -1111 to come out :
-1111
Created doubly linked list is as follows
traversing the list using right pointer
1 = 2 = 3 =
traversing the list using left pointer
3 = 2 = 1 =
```

3.6 CIRCULARLY LINKED LISTS IMPLEMENTATION

A linked list in which the last element points to the first element is called CIRCULAR linked list. The chains do not indicate first or last element; last element does not contain the NULL pointer. The external pointer provides a reference to starting element.

The possible operations on a circular linked list are:

- Insertion,
- Deletion, and
- Traversing

Figure 3.6 depicts a Circular linked list.

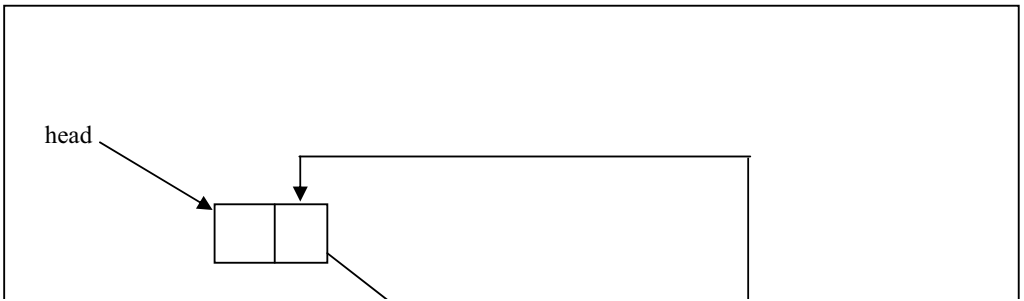


Figure 3.6 : A Circular Linked List

Program 3.8 depicts the creation of a Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list clist;

clist *head, *s;

void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
        head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
}

void create_clist(clist *start)
{
        printf("input the element  -1111 for coming out of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
        }
}

void traverse(clist *start)
{
```

```
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data, start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data, start-
>next);
}

int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.8: Creation of a Circular linked list

ALGORITHM (Insertion of an element into a Circular Linked List)

Step 1 Begin

Step 2 if the list is empty or new element comes before the start (head) element, then insert the new element as start element.

Step 3 else, if the new element comes after the last element, then insert the new element at the end element and adjust the pointer of last element to the start element.

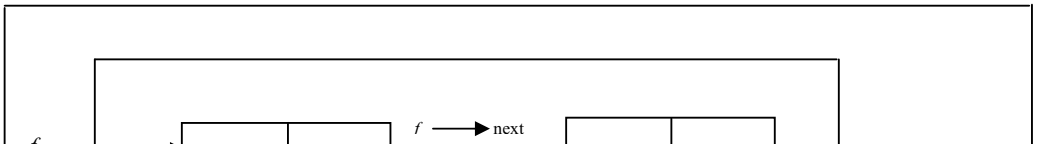
Step 4 else, insert the new element in the list by using the find function. find function returns the address of the found element to the insert_list function.

Step 5 End.

If new item is to be inserted after an existing element, then, call the find function recursively to trace the 'key' element. The new element is inserted before the 'key' element by using above algorithm.

Figure 3.7 depicts the Circular linked list with a new element that is to be inserted.

Figure 3.8 depicts a Circular linked list with the new element inserted between first and second nodes of *Figure 3.7*.



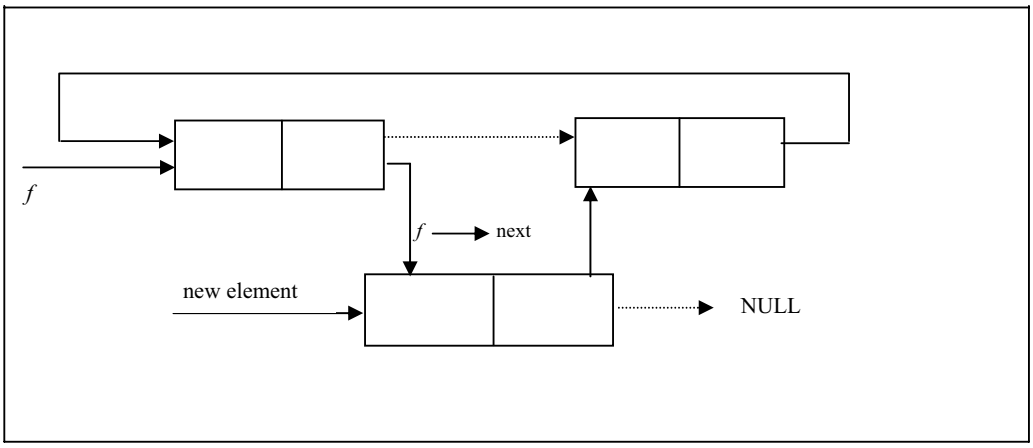


Figure 3.8: A Circular Linked List after insertion of the new element between first and second nodes (Dotted lines depict the links prior to insertion)

Program 3.9 depicts the code for insertion of a node into a Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0
struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list clist;
clist *head, *s;
/*  prototype of find and insert functions */
clist * find(clist *, int);
clist * insert_clist(clist *);
/*definition of insert_clist function */
clist * insert_clist(clist *start)    {
        clist *n, *n1;
        int key, x;
        printf("enter value of new element");
        scanf("%d", &x);
        printf("eneter value of key element");
        scanf("%d",&key);
        if(start->data ==key)
        {
```

```
                n=(clist *)malloc(sizeof(clist));
                n->data=x;
                n->next = start;
                start=n;
        }
        else
        {
                n1 = find(start, key);
                if(n1 == NULL)
                        printf("\n key is not found\n");
                else
                {
                        n=(clist*)malloc(sizeof(clist));
                        n->data=x;
                        n->next=n1->next;
                        n1->next=n;
                }
        }
        return(start);
}
/*definition of find function */
clist * find(clist *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}
void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
        head=insert_clist(head);
        printf("\n  traversing the clist after insert_clist and starting address is %u
\n",head);
        traverse(head);
}
void create_clist(clist *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
```

```
        }
}

void traverse(clist *start)
{
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data, start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data, start-
>next);
}
int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.9 Insertion of a node into a Circular Linked List

Figure 3.9 depicts a Circular linked list from which an element was deleted.

ALGORITHM (Deletion of an element from a Circular Linked List)

Step 1 Begin
Step 2 if the list is empty, then element cannot be deleted.
Step 3 else, if element to be deleted is first node, then make the start (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step 5 End.

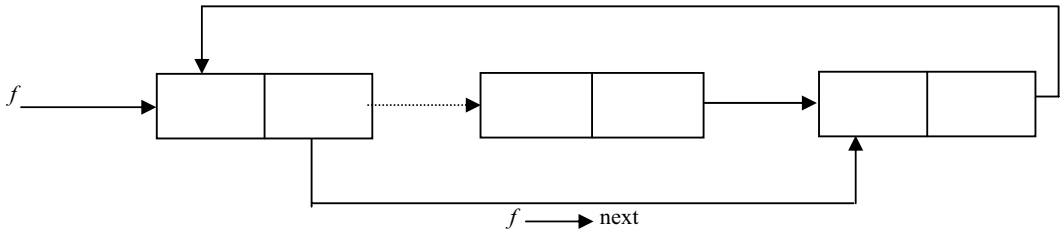


Figure 3.9 A Circular Linked List from which an element was deleted (Dotted line shows the linked that existed prior to deletion)

Program 3.10 depicts the code for the deletion of an element from the Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
        int data;
        struct linked_list *next;
};
```

```
typedef struct linked_list clist;
clist *head, *s;

/*  prototype of find and delete_function*/
clist * delete_clist(clist *);
clist * find(clist *, int);

/*definition of delete_clist       */
clist *delete_clist(clist *start)
{
        int key; clist * f, * temp;
        printf("\n enter the value of element to be deleted \n");
        scanf("%d", &key);
        if(start->data == key)
        {
                temp=start->next;
                free(start);
                start=temp;
        }
        else
        {
                f = find(start,key);
                if(f==NULL)
                        printf("\n key not fund");
                else
                {
                        temp = f->next->next;
                        free(f->next);
                        f->next=temp;
                }
        }
        return(start);
}
/*definition of find function */
clist * find(clist *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}

void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
        head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
        head=delete_clist(head);
```

```
        printf("  \n  traversing the clist after delete_clistand starting address is %u
        \n",head);
        traverse(head);
}
void create_clist(clist *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
        }
}

void traverse(clist *start)
{
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data,   start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data,    start-
>next);
}

int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.10: Deletion of an element from the circular linked list

3.7 APPLICATIONS

Lists are used to maintain POLYNOMIALS in the memory. For example, we have a function $f(x)= 7x^5 + 9x^4 - 6x^3 + 3x^2$. Figure 3.10 depicts the representation of a Polynomial using a singly linked list. 1000,1050,1200,1300 are memory addresses.

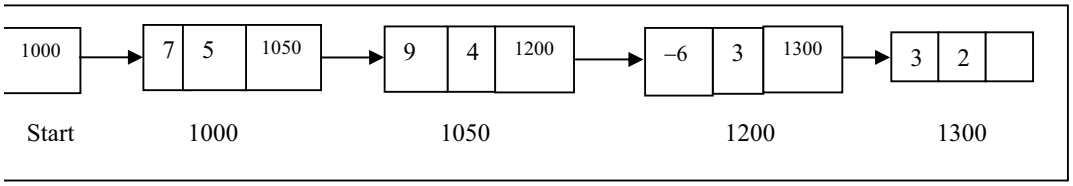


Figure 3.10: Representation of a Polynomial using a singly linked list

Polynomial contains two components, coefficient and an exponent, and 'x' is a formal parameter. The polynomial is a sum of terms, each of which consists of coefficient and an exponent. In computer, we implement the polynomial as list of structures consisting of coefficients and an exponents.

Program 3.11 accepts a Polynomial as input. It uses linked list to represent the Polynomial. It also prints the input polynomial along with the number of nodes in it.

```
/* Representation of Polynomial using Linked List */
# include <stdio.h>
# include <malloc.h>
struct link
{
        char sign;
        int coef;
        int expo;
        struct link *next;
};
typedef struct link poly;
void insertion(poly *);
void create_poly(poly *);
void display(poly *);
/* Function create a ploynomial list */
void create_poly(poly *start)
{
        char ch;
        static int i;
        printf("\n Input choice n for break: ");
        ch = getchar();
        if(ch != 'n')
        {
                printf("\n Input the sign: %d: ", i+1);
                scanf("%c", &start->sign);
                printf("\n Input the coefficient value: %d: ", i+1);
                scanf("%d", &start->coef);
                printf("\n Input the exponent value: %d: ", i+1);
                scanf("%d", &start->expo);
                fflush(stdin);
                i++;
                start->next = (poly *) malloc(sizeof(poly));
                create_poly(start->next);
        }
        else
        start->next=NULL;
}
/* Display the polynomial */
void display(poly *start)
{
        if(start->next != NULL)
        {
                printf(" %c", start->sign);
                printf(" %d", start->coef);
                printf("X^%d", start->expo);
                display(start->next);
        }
}
/* counting the number of nodes */
```

```
int count_poly(poly *start)
{
        if(start->next == NULL)
                return 0;
        else
                return(1+count_poly(start->next));
}
/* Function main */
void main()
{       poly *head = (poly *) malloc(sizeof(poly));
        create_poly(head);
        printf("\n Total nodes = %d \n", count_poly(head));
        display(head); }
```

Program 3.11: Representation of Polynomial using Linked list

☞ Check Your Progress

1) Write a function to print the memory location(s) which are used to store the data in a single linked list ?

……

……..……

.

2) Can we use doubly linked list as a circular linked list? If yes, Explain.

…….

……..……

3) Write the differences between Doubly linked list and Circular linked list.

..……

……..……..……...

4) Write a program to count the number of items stored in a single linked list.

..……

…….

5) Write a function to check the overflow condition of a list represented by an array.

……

……

3.8 SUMMARY

The advantage of Lists over Arrays is flexibility. Over flow is not a problem until the computer memory is exhausted. When the individual records are quite large, it may be difficult to determine the amount of contiguous storage that might be in need for the required arrays. With dynamic allocation, there is no need to attempt to allocate in advance. Changes in list, insertion and deletion can be made in the middle of the list, more quickly than in the contiguous lists.

The drawback of lists is that the links themselves take space which is in addition to the space that may be needed for data. One more drawback of lists is that they are not suited for random access. With lists, we need to traverse a long path to reach a desired node.

3.9 SOLUTIONS/ANSWERS

1)
```
void print_location(struct node *head)
{
        temp=head;
        while(temp->next !=NULL)
        {
           printf("%u", temp);
           temp=temp->next;
        }
        printf("%u", temp);
}
```

4)
```
void count_items(struct node *head)
{
        int count=0;
        temp=head;
        while(temp->next !=NULL)
        {
                count++;
        }
        count++;
        pintf("total items = %d", count);
}
```

5)
```
void Is_Overflow(int max_size, int last_element_position)
{
        if(last_element_position == max_size)
                printf("List Overflow");
        else
                printf("not Overflow");
}
```

3.10 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications

2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education

Reference Websites

http://www.webopedia.com
http://www.ieee.org

.

UNIT 4 STACKS

Structure **Page Nos.**

4.0	Introduction	5
4.1	Objectives	6
4.2	Abstract Data Type-Stack	7
4.3	Implementation of Stack	7
	4.3.1 Implementation of Stack Using Arrays	
	4.3.2 Implementation of Stack Using Linked Lists	
4.4	Algorithmic Implementation of Multiple Stacks	13
4.5	Applications	14
4.6	Summary	14
4.7	Solutions / Answers	15
4.8	Further Readings	15

4.0 INTRODUCTION

One of the most useful concepts in computer science is stack. In this unit, we shall examine this simple data structure and see why it plays such a prominent role in the area of programming. There are certain situations when we can insert or remove an item only at the beginning or the end of the list.

A stack is a linear structure in which items may be inserted or removed only at one end called the *top of the stack*. A stack may be seen in our daily life, for example, *Figure 4.1* depicts a stack of dishes. We can observe that any dish may

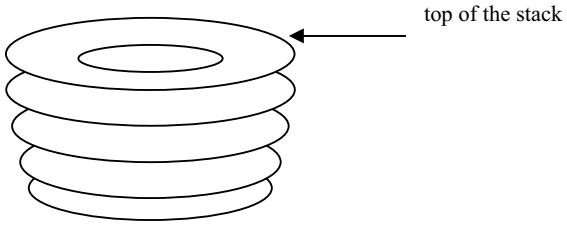


Figure 4.1: A stack of dishes

be added or removed only from the top of the stack. It concludes that the item added last will be the item removed first. Therefore, stacks are also called LIFO (Last In First Out) or FILO (First In Last Out) lists. We also call these lists as "piles" or "push-down list".

Generally, two operations are associated with the stacks named Push & Pop.

- *Push* is an operation used to insert an element at the top.
- *Pop* is an operation used to delete an element from the top

Example 4.1

Now we see the effects of push and pop operations on to an empty stack. *Figure 4.2(a)* shows (i) an empty stack; (ii) a list of the elements to be inserted on to stack;

and (iii) a variable top which helps us keep track of the location at which insertion or removal of the item would occur.

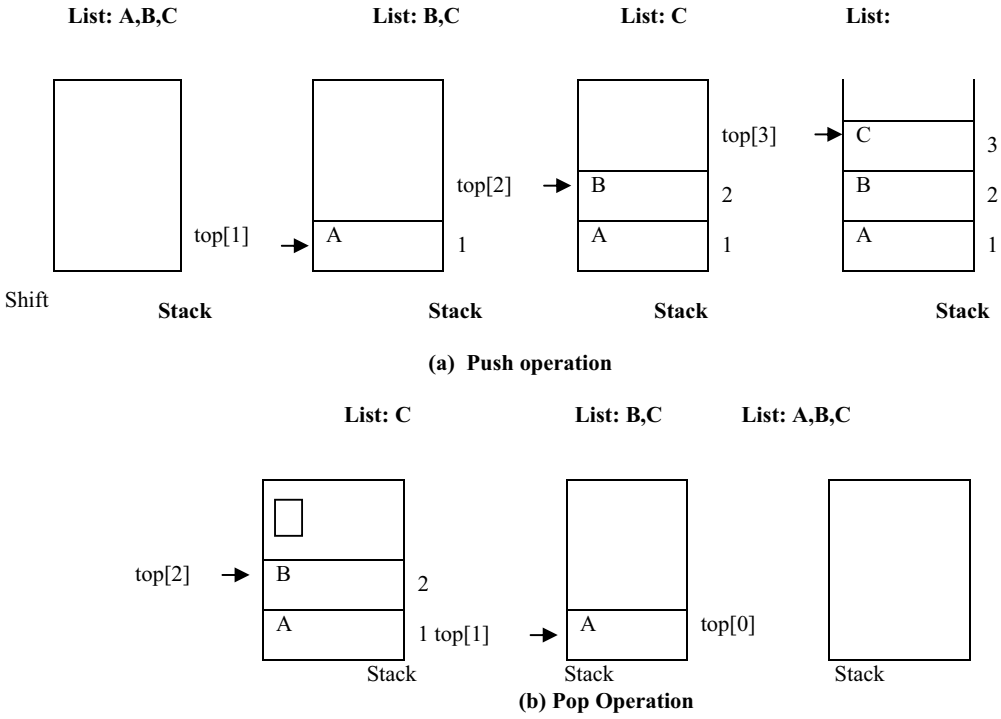


Figure 4.2: Demonstration of (a) Push operation, (b) Pop operation

Initially in *Figure 4.2(a)*, top contains 0, implies that the stack is empty. The list contains three elements, A, B &C. In *Figure 4.2(b),* we remove an element A from the list of elements, push it on to stack. The value of top becomes 1, pointing to the location of the stack at which A is stored.

Similarly, we remove the elements B & C from the list one by one and push them on to the stack. Accordingly, the value of the **top** is incremented. *Figure 4.2(a)* explains the pushing of B and C on to stack. The **top** now contains value 3 and pointing to the location of the last inserted element C.

On the other hand, *Figure 4.2(b)* explains the working of pop operation. Since, only the top element can be removed from the stack, in *Figure 4.2(b),* we remove the top element C (we have no other choice). C goes to the list of elements and the value of the top is decremented by 1. The top now contains value 2, pointing to B (the top element of the stack). Similarly, in *Figure 4.2(b),* we remove the elements B and A from the stack one by one and add them to the list of elements. The value of top is decremented accordingly.

There is no upper limit on the number of items that may be kept in a stack. However, if a stack contains a single item and the stack is popped, the resulting stack is called empty stack. The pop operation cannot be applied to such stacks as there is no element to pop, whereas the push operation can be applied to any stack.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the concept of stack;
- implement the stack using arrays;
- implement the stack using linked lists;
- implement multiple stacks, and
- give some applications of stack.

4.2 ABSTRACT DATA TYPE-STACK

Conceptually, the **stack** abstract data type mimics the information kept in a pile on a desk. Informally, we first consider materials on a desk, where we may keep separate stacks for bills that need paying, magazines that we plan to read, and notes we have taken. We can perform several operations that involve a stack:

- start a new stack;
- place new information on the top of a stack;
- take the top item off of the stack;
- read the item on the top; and
- determine whether a stack is empty. (There may be nothing at the spot where the stack should be).

When discussing these operations, it is conventional to call the addition of an item to the top of the stack as a **push operation** and the deletion of an item from the top as a **pop operation**. (These terms are derived from the working of a spring-loaded rack containing a stack of cafeteria trays. Such a rack is loaded by pushing the trays down on to the springs as each diner removes a tray, the lessened weight on the springs causes the stack to pop up slightly).

4.3 IMPLEMENTATION OF STACK

Before programming a problem solution that uses a stack, we must decide how to represent a stack using the data structures that exist in our programming language. Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Each approach has its advantages and disadvantages. A stack is generally implemented with two basic operations – push and pop. **Push** means to insert an item on to stack. The push algorithm is illustrated in *Figure 4.3(a)*. Here, **tos** is a pointer which denotes the position of top most item in the stack. Stack is represented by the array **arr** and **MAXSTACK** represents the maximum possible number of elements in the stack. The pop algorithm is illustrated in *Figure 4.3(b)*.

```
Step 1:  [Check for stack overflow]
                    if tos >=MAXSTACK
                    print “Stack overflow” and exit
Step 2:  [Increment the pointer value by one]
                    tos=tos+1
Step 3:  [Insert the item]
                    arr[tos]=value
Step 4:  Exit
```

Figure 4.3(a): Algorithm to push an item onto the stack

The pop operation removes the topmost item from the stack. After removal of top most value **tos** is decremented by 1.

```
Step 1:  [Check whether the stack is empty]
                    if tos = 0
                    print "Stack underflow" and exit

Step 2:  [Remove the top most item]
                    value=arr[tos]
                    tos=tos-1

Step 3:  [Return the item of the stack]
                    return(value)
```

Figure 4.3(b): Algorithm to pop an element from the stack

4.3.1 Implementation of Stack Using Arrays

A Stack contains an ordered list of elements and an array is also used to store ordered list of elements. Hence, it would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed. Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack. We can declare the array with a maximum size large enough to manage a stack. Program 4.1 implements a stack using an array.

```
#include<stdio.h>

int choice, stack[10], top, element;

void menu();
void push();
void pop();
void showelements();

void main()
{ choice=element=1;
  top=0;
  menu();
}

void menu()
{
        printf("Enter one of the following options:\n");
        printf("PUSH 1\n POP 2\n SHOW ELEMENTS 3\n  EXIT 4\n");
        scanf("%d", &choice);
        if (choice==1)
        {
         push(); menu();
        }
          if (choice==2)
        {
         pop();menu();
```

```
        }
          if (choice==3)
                                {
                                    showelements(); menu();
                                }
        }


void push()
{
        if (top<=9)
        {
                printf("Enter the element to be pushed to stack:\n");
                scanf("%d", &element);
                stack[top]=element;
                ++top;

        }
        else
        {
        printf("Stack is full\n");
        }
        return;
}

void pop()
{
        if (top>0)
        {
                --top;
                element = stack[top];
                printf("Popped element:%d\n", element);
         }
        else
        {
        printf("Stack is empty\n");
        }
        return;

}

void showelements()
{
        if (top<=0)
                printf("Stack is empty\n");

        else
                for(int i=0; i<top; ++i)
                        printf("%d\n", stack[i]);


}
```

Program 4.1: Implementation of stack using arrays

Explanation

The size of the stack was declared as 10. So, stack cannot hold more than 10 elements. The main operations that can be performed on a stack are push and pop. How ever, in a program, we need to provide two more options , namely, *showelements* and *exit*. *showelements* will display the elements of the stack. In case, the user is not interested to perform any operation on the stack and would like to get out of the program, then s/he will select *exit* option. It will log the user out of the program. *choice* is a variable which will enable the user to select the option from the push, pop, showelements and exit operations. *top* points to the index of the free location in the stack to where the next element can be pushed. *element* is the variable which accepts the integer that has to be pushed to the stack or will hold the top element of the stack that has to be popped from the stack. The array *stack* can hold at most 10 elements. *push* and *pop* will perform the operations of pushing the element to the stack and popping the element from the stack respectively.

4.3.2 Implementation of Stack Using Linked Lists

In the last subsection, we have implemented a stack using an array. When a stack is implemented using arrays, it suffers from the basic limitation of an array – that is, its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In the case of a linked stack, we shall push and pop nodes from one end of a linked list.
The stack, as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list. Program 4.2 implements a stack using linked lists.

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

/* Definition of the structure node */

typedef struct node

{

  int data;
  struct node *next;
 } ;

/* Definition of push function */

void  push(node **tos,int item)

{

   node *temp;
   temp=(node*)malloc(sizeof(node));      /* create a new node dynamically */
   if(temp==NULL)                         /* If sufficient amount of memory is */
    {                                     /* not available, the function malloc will */
     printf("\nError: Insufficient Memory Space");       /* return NULL to temp */
     getch();
     return;
    }

   else                                   /* otherwise*/

   {

       temp->data=item;   /* put the item in the data portion of  node*/


        temp->next=*tos;                  /*insert this node at the front of the stack */
       *tos=temp;                         /* managed by linked list*/
```

```
        }
}                                       /*end of function push*/


        /* Definition of pop function */
int pop(node **tos)
{
    node *temp;
    temp=*tos;
    int item;
    if(*tos==NULL)
        return(NULL);
   else
        {
           *tos=(*tos)->next;                   /* To pop an element from stack*/
            item=temp->data;                     /* remove the front node of the */
            free(temp);                          /* stack managed by L.L*/
            return (item);
         }
}  /*end of function pop*/

/* Definition of display function */
void display(node *tos)

{
 node *temp=tos;
 if(temp==NULL)                 /* Check whether the stack is empty*/
   {
     printf("\nStack is empty");
     return;
   }
     else
        {
          while(temp!=NULL)
          {
             printf("\n%d",temp->data);    /* display all the values of the stack*/
             temp=temp->next;              /* from the front node to the last node*/
          }
        }

  }                                          /*end of function display*/


/* Definition of main function */
void main()

{
  int item, ch;
  char choice='y';
  node *p=NULL;
  do
   {
        clrscr();
        printf("\t\t\t\t*****MENU*****");
```

```
        printf("\n\t\t\t1. To PUSH an element");
        printf("\n\t\t\t2. To POP an element");
        printf("\n\t\t\t3. To DISPLAY the elements of stack");
        printf("\n\t\t\t4. Exit");
        printf("\n\n\n\t\t\tEnter your choice:-");
        scanf("%d",&ch);
        switch(ch)
                {
                case 1:
                        printf("\n Enter an element which you want to push ");
                        scanf("%d",&item);
                        push(&p,item);
                        break;
                case 2:
                        item=pop(&p);
                        if(item!=NULL);
                        printf("\n Detected item is%d",item);
                        break;
                case 3:
                        printf("\nThe elements of stack are");
                        display(p);
                        break;
                case 4:
                        exit(0);

                }       /*switch closed */
                printf("\n\n\t\t Do you want to run it again y/n");
                scanf("%c",&choice);
    } while(choice=='y');

}
                                                /*end of function main*/
```

Program 4.2: Implementation of Stack using Linked Lists

Similarly, as we did in the implementation of stack using arrays, to know the working of this program, we executed it thrice and pushed 3 elements (10, 20, 30). Then we call the function display in the next run to see the elements in the stack.

Explanation

Initially, we defined a structure called *node*. Each node contains two portions, data and a pointer that keeps the address of the next node in the list. The *Push* function will insert a node at the front of the linked list, whereas *pop* function will delete the node from the front of the linked list. There is no need to declare the size of the stack in advance as we have done in the program where in we implemented the stack using arrays since we create nodes dynamically as well as delete them dynamically. The function *display* will print the elements of the stack.

☞ Check Your Progress 1

1) State True or False.

   (a) Stacks are sometimes called FIFO lists.

   (b) Stack allows Push and Pop from both ends.

   (c) TOS (top of the stack) gives the bottom most element in the stack.

2) Comment on the following.

   (a) Why is the linked list representation of the stack better than the array representation of the stack?

   (b) Discuss the underflow and overflow problem in stacks.

4.4 ALGORITHMIC IMPLEMENTATION OF MULTIPLE STACKS

So far, now we have been concerned only with the representation of a single stack. What happens when a data representation is needed for several stacks? Let us see an array X whose dimension is m. For convenience, we shall assume that the indexes of the array commence from 1 and end at m. If we have only 2 stacks to implement in the same array X, then the solution is simple.

Suppose A and B are two stacks. We can define an array stack A with n_1 elements and an array stack B with n_2 elements. Overflow may occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose, instead of that, we define a single array stack with $n = n_1 + n_2$ elements for stack A and B together. See the *Figure 4.4* below. Let the stack A "grow" to the right, and stack B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. It does not matter how many elements individually are there in each stack.

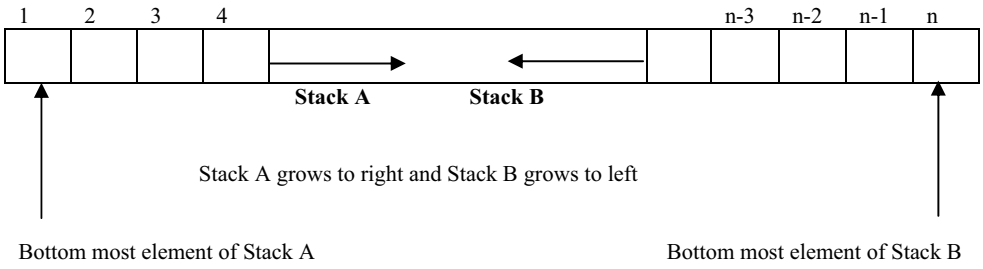
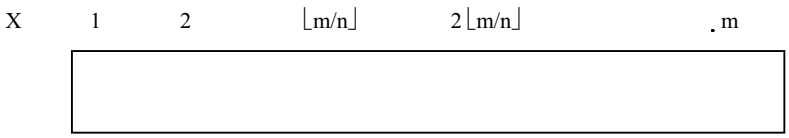


Figure 4.4: Implementation of multiple stacks using arrays

But, in the case of more than 2 stacks, we cannot represent these in the same way because a one-dimensional array has only two fixed points X(1) and X(m) and each stack requires a fixed point for its bottom most element. When more than two stacks, say **n**, are to be represented sequentially, we can initially divide the available memory X(1:m) into **n** segments. If the sizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks. If the sizes of the stacks are not known, then, X(1:m) may be divided into equal segments. For each stack i, we shall use BM (i) to represent a position one less than the position in X for the bottom most element of that stack. TM(i), $1 \leq i \leq n$ will point to the topmost element of stack i. We shall use the boundary condition BM (i) = TM (i) iff the i^{th} stack is empty (refer to *Figure 4.5*). If we grow the i^{th} stack in lower memory indexes than the $i+1^{st}$ stack, then, with roughly equal initial segments we have
BM (i) = TM (i) = $\lfloor m/n \rfloor (i - 1)$, $1 \leq i \leq n$, as the initial values of BM (i) and TM (i).



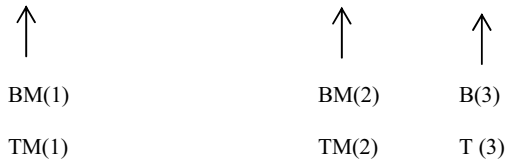


Figure 4.5: Initial configuration for n stacks in X(1:m)

All stacks are empty and memory is divided into roughly equal segments.

Figure 4.6 depicts an algorithm to add an element to the i^{th} stack. *Figure 4.7* depicts an algorithm to delete an element from the i^{th} stack.

```
    ADD(i,e)
Step1: if TM (i)=BM (i+1)
        Print "Stack is full" and exit
Step2: [Increment the pointer value by one]
            TM (i)← TM (i)+1
            X(TM (i))← e
Step3: Exit
```

Figure 4.6: Algorithm to add an element to i^{th} stack

//delete the topmost elements of stack i.

```
DELETE(i,e)
Step1: if TM (i)=BM (i)
        Print "Stack is empty" and exit
Step2: [remove the topmost item]
            e← X(TM (i))
            TM (i)←TM(i)-1
Step3: Exit
```

Figure 4.7: Algorithm to delete an element from i^{th} stack

4.5 APPLICATIONS

Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. Polish notations are evaluated by stacks. Conversions of different notations (Prefix, Postfix, Infix) into one another are performed using stacks. Stacks are widely used inside computer when recursive functions are called. The computer evaluates an arithmetic expression written in infix notation in two steps. First, it converts the infix expression to postfix expression and then it evaluates the postfix expression. In each step, stack is used to accomplish the task.

4.6 SUMMARY

In this unit, we have studied how the stacks are implemented using arrays and using liked list. Also, the advantages and disadvantages of using these two schemes were discussed. For example, when a stack is implemented using arrays, it suffers from the basic limitations of an array (fixed memory). To overcome this problem, stacks are implemented using linked lists. This unit also introduced learners to the concepts of multiple stacks. The problems associated with the implementation of multiple stacks are also covered.

☞ Check Your Progress 2

1) Multiple stacks can be implemented using _____.
2) _____ are evaluated by stacks.
3) Stack is used whenever a _____ function is called.

4.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1) (a) False (b) False
(c) False

Check Your Progress 2

1) Arrays or Pointers
2) Postfix expressions
3) Recursive

4.8 FURTHER READINGS

1. *Data Structures Using C and C++*, Yedidyah Langsam,Moshe J. Augenstein,Aaron M Tenenbaum, Second Edition,PHI publications.

2. *Data Structures,* Seymour Lipschutz, Schaum's Outline series, Mc GrawHill.

Reference Websites

http://www.cs.queensu.ca

UNIT 5 QUEUES

Structure **Page Nos.**

5.0 Introduction 16
5.1 Objectives 16
5.2 Abstract Data Type-Queue 16
5.3 Implementation of Queue 17
5.3.1 Array implementation of a queue
5.3.2 Linked List implementation of a queue
5.4 Implementation of Multiple Queues 21
5.5 Implementation of Circular Queues 22
5.5.1 Array Implementation of a circular queue
5.5.2 Linked List Implementation of a circular queue
5.6 Implementation of DEQUEUE 25
5.6.1 Array Implementation of a dequeue
5.6.2 Linked List Implementation of a dequeue
5.7 Summary 30
5.8 Solutions / Answers 30
5.9 Further Readings 30

5.0 INTRODUCTION

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service. In computer science, it is also called a FIFO (first in first out) list. In this chapter, we will study about various types of queues.

5.1 OBJECTIVES

After going through this unit, you should be able to

- define the queue as an abstract data type;
- understand the terminology of various types of queues such as simple queues, multiple queues, circular queues and dequeues, and
- get an idea about the implementation of different types of queues using arrays and linked lists.

5.2 ABSTRACT DATA TYPE-QUEUE

An important aspect of Abstract Data Types is that they describe the properties of a data structure without specifying the details of its implementation. The properties can be implemented independent of any implementation in any programming language.

Queue is a collection of elements, or items, for which the following operations are defined:

createQueue(Q) : creates an empty queue Q;
isEmpty(Q): is a boolean type predicate that returns ``true'' if Q exists and is empty, and returns ``false'' otherwise;
addQueue(Q,item) adds the given item to the queue Q; and
deleteQueue (Q, item) : delete an item from the queue Q;
next(Q) removes the least recently added item that remains in the queue Q, and returns it as the value of the function;

isEmpty (createQueue(Q)) : is always true, and
deleteQueue(createQueue(Q)) : error

The primitive isEmpty(Q) is required to know whether the queue is empty or not, because calling next on an empty queue should cause an error. Like stack, the situation may be such when the queue is "full" in the case of a finite queue. But we avoid defining this here as it would depend on the actual length of the Queue defined in a specific problem.

The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is the first to be served.

As pointed out earlier, Abstract Data Types describe the properties of a structure without specifying an implementation in any way. Thus, an algorithm which works with a "queue" data structure will work wherever it is implemented. Different implementations are usually of different efficiencies.

5.3 IMPLEMENTATION OF QUEUE

A physical analogy for a queue is a line at a booking counter. At a booking counter, customers go to the *rear* (end) of the line and customers are attended to various services from the *front* of the line. Unlike stack, customers are added at the rear end and deleted from the front end in a queue (FIFO).

An example of the queue in computer science is print jobs scheduled for printers. These jobs are maintained in a queue. The job fired for the printer first gets printed first. Same is the scenario for job scheduling in the CPU of computer.

Like a stack, a queue also (usually) holds data elements of the same type. We usually graphically display a queue horizontally. *Figure 5.1* depicts a queue of 5 characters.

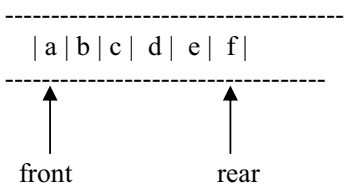


Figure 5.1: A queue of characters

The rule followed in a queue is that elements are added at the *rear* and come off of the *front* of the queue. After the addition of an element to the above queue, the position of rear pointer changes as shown below. Now the *rear* is pointing to the new element 'g' added at the rear of the queue(refer to *Figure 5.2*).

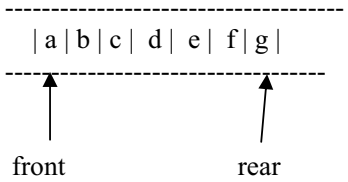


Figure 5.2: Queue of figure 5.1 after addition of new element

After the removal of element 'a' from the front, the queue changes to the following with the *front* pointer pointing to 'b' (refer to *Figure 5.3*).

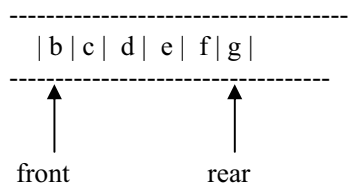


Figure 5.3: Queue of figure 5.2 after deletion of an element

Algorithm for addition of an element to the queue

Step 1: Create a new element to be added
Step 2: If the queue is empty, then go to step 3, else perform step 4
Step 3: Make the front and rear point this element
Step 4: Add the element at the end of the queue and shift the rear pointer to the newly added element.

Algorithm for deletion of an element from the queue

Step 1: Check for Queue empty condition. If empty, then go to step 2, else go to step 3
Step 2: Message "Queue Empty"
Step 3: Delete the element from the front of the queue. If it is the last element in the queue, then perform *step a* else *step b*
a) make front and rear point to null
b) shift the front pointer ahead to point to the next element in the queue

5.3 1 Array implementation of a queue

As the stack is a list of elements, the queue is also a list of elements. The stack and the queue differ only in the position where the elements can be added or deleted. Like other liner data structures, queues can also be implemented using arrays. Program 5.1 lists the implementation of a queue using arrays.

```
#include "stdio.h"
#define QUEUE_LENGTH 50
struct queue
{    int element[QUEUE_LENGTH];
     int front, rear, choice,x,y;
}

struct queue q;

main()
{
int choice,x;
printf ("enter 1 for add and 2 to remove element front the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)

  {

case 1 :
 printf ("Enter element to be added :");
```

```
scanf("%d",&x);
add(&q,x);
break;

case 2 :
delete();
break;

  }

}
add(y)
{
++q.rear;
if (q.rear < QUEUE_LENGTH)
   q.element[q.rear] = y;
else
 printf("Queue overflow")
}

delete()
{

if q.front > q.rear printf("Queue empty");
else{
  x = q.element[q.front];
 q.front++;
 }
retrun x;
}
```

Program 5.1: Array implementation of a Queue

5.3.2 Linked List Implementation of a queue

The basic element of a linked list is a "record" structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node (refer to *Figure 5.4*).

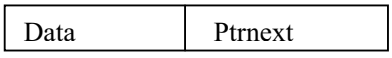


Figure 5.4: Structure of a node

The *data* component may contain data of any type. *Ptrnext* is a reference to the next element in the queue structure. *Figure 5.5* depicts the linked list representation of a queue.

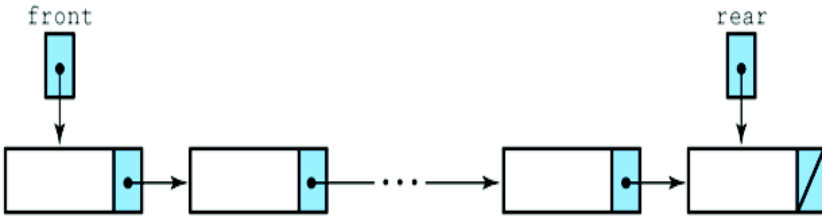


Figure 5.5: A linked list representation of a Queue

Program 5.2 gives the program segment for the addition of an element to the queue.

Program 5.3 gives the program segment for the deletion of an element from the queue.

```
add(int value)
{
struct queue *new;
new = (struct queue*)malloc(sizeof(queue));
new->value = value;
new->next = NULL;
if (front == NULL)
{
  queueptr = new;
  front = rear = queueptr
}
else
{
rear->next = new;
rear=new;
}
}
```

Program 5.2: Program segment for addition of an element to the queue

```
delete()
{
int delvalue = 0;
if (front == NULL) printf("Queue Empty");
{
  delvalue = front->value;
 if (front->next==NULL)
{
free(front);
queueptr=front=rear=NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr=front;

}
}
}
```

Program 5.3: Program segment for deletion of an element from the queue

☞ Check Your Progress 1

1) The queue is a data structure where addition takes place at _____ and deletion takes place at _____.

2) The queue is also known as _____ list.

3) Compare the array and linked list representations of a queue. Explain your answer.

5.4 IMPLEMENTATION OF MULTIPLE QUEUES

So far, we have seen the representation of a single queue, but many practical applications in computer science require several queues. Multiqueue is a data structure where multiple queues are maintained. This type of data structures are used for process scheduling. We may use one dimensional array or multidimensional array to represent a multiple queue.

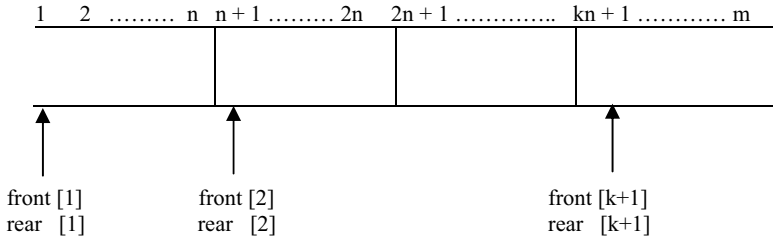


Figure 5.6: Multiple queues in an array

A multiqueue implementation using a single dimensional array with *m* elements is depicted in *Figure 5.6*. Each queue has *n* elements which are mapped to a liner array of *m* elements.

Array Implementation of a multiqueue

Program 5.4 gives the program segment using arrays for the addition of an element to a queue in the multiqueue.

```
addmq(i,x)  /* Add x to queue i */
{
int i,x;
++rear[i];
if ( rear[i] == front[i+1])
 printf("Queue is full");
else
{
rear[i] = rear[i]+1;
mqueue[rear[i]] = x;
}
}
```

Program 5.4: Program segment for the addition of an element to the queue

Program 5.5 gives the program segment for the deletion of an element from the queue.

```
delmq(i)   /* Delete an element from queue i */
{
int i,x;
if ( front[i] == rear[i])
 printf("Queue is empty");
{
x = mqueue[front[i]];
front[i] = front[i]-1 ;
return x;
}
}
```

Program 5.5: Program segment for the deletion of an element from the queue

5.5 IMPLEMENTATION OF CIRCULAR QUEUES

One of the major problems with the linear queue is the lack of proper utilisation of space. Suppose that the queue can store 100 elements and the entire queue is full. So, it means that the queue is holding 100 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full. In this way, space utilisation in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index *0* and ends at *n-1*.So, clearly , the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1(both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilisation of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element anti-clock wise. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. *Figure 5.7* depicts a circular queue.

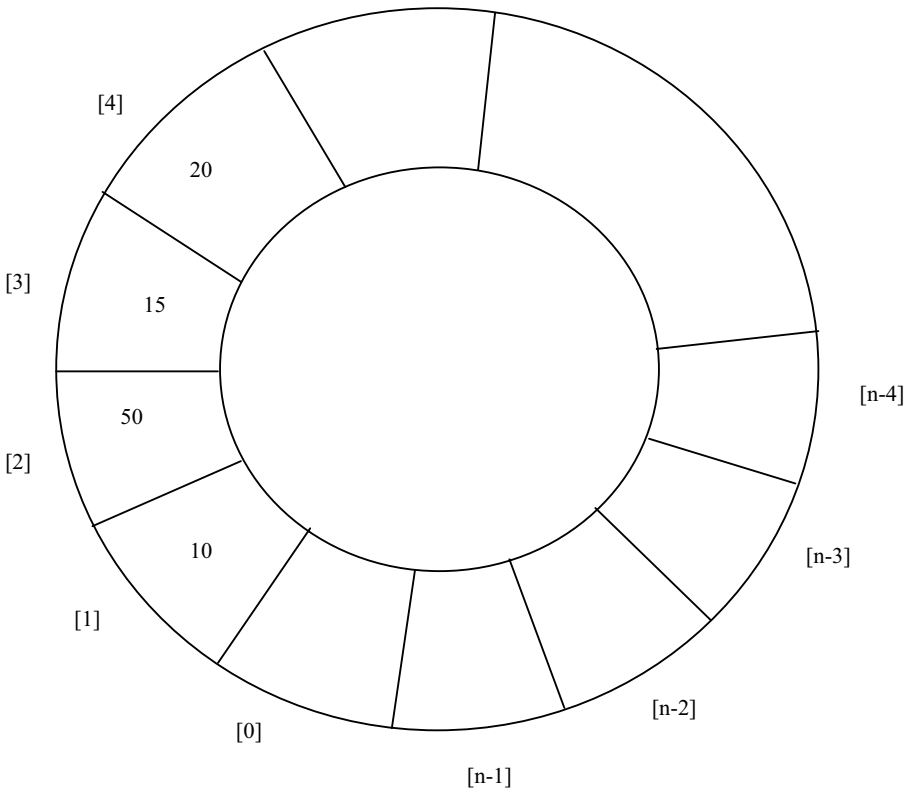


Figure 5.7 : A circular queue (Front = 0, Rear = 4)

Algorithm for Addition of an element to the circular queue:

Step-1: If "rear" of the queue is pointing to the last position then go to step-2 or else Step-3
Step-2: make the "rear" value as 0
Step-3: increment the "rear" value by one
Step-4: a. if the "front" points where "rear" is pointing and the queue holds a not NULL value for it, then its a "queue overflow" state, so quit; else go to step-b
b. add the new value for the queue position pointed by the "rear"

Algorithm for deletion of an element from the circular queue:

Step-1: If the queue is empty then say "queue is empty" and quit; else continue
Step-2: Delete the "front" element
Step-3: If the "front" is pointing to the last position of the queue then go to step-4 else go to step-5
Step-4: Make the "front" point to the first position in the queue and quit
Step-5: Increment the "front" position by one

5.5.1 Array implementation of a circular queue

A circular queue can be implemented using arrays or linked lists. Program 5.6 gives the array implementation of a circular queue.

```
#include "stdio.h"
void add(int);
void deleteelement(void);
int max=10;          /*the maximum limit for queue has been set*/
static int queue[10];
int front=0, rear=-1;  /*queue is initially empty*/
void main()
{
int choice,x;
printf ("enter 1 for addition and 2 to remove element front the queue and 3 for exit");
printf("Enter your choice");
scanf("%d",&choice);
switch (choice)
{
case 1 :
printf ("Enter the element to be added :");
scanf("%d",&x);
add(x);
break;
case 2 :
deleteelement();
break;
 }
}
void add(int y)
{
 if(rear == max-1)
   rear = 0;
   else
   rear = rear + 1;
   if( front == rear && queue[front] != NULL)
   printf("Queue Overflow");
   else
```

```
   queue[rear] = y;
}

void deleteelement()
{
int deleted_front = 0;
 if (front == NULL)
     printf("Error - Queue empty");
 else
    {
     deleted_front = queue[front];
     queue[front] = NULL;
      if (front == max-1)
      front = 0;
     else
     front = front + 1;
  }
}
```

Program 5.6: Array implementation of a Circular queue

5.5.2 Linked list implementation of a circular queue

Link list representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers. Program 5.7 gives the linked list representation of a circular queue.

```
#include “stdio.h”
 struct cq
{     int value;
      int *next;
 };

typedef struct  cq *cqptr
cqptr p, *front, *rear;

main()
{
int choice,x;

/*  Initialise the circular queue */
cqptr =  front = rear =  NULL;

printf (“Enter 1 for addition and 2 to delete element from the queue”)
printf(“Enter your choice”)
scanf(“%d”,&choice);
switch (choice)

{

case 1 :
              printf (“Enter the element to be added :”);
              scanf(“%d”,&x);
              add(&q,x);
              break;

case 2 :
              delete();
```

```
            break;

  }
}

/************ Add element ******************/

add(int value)
{
   struct cq *new;
   new = (struct cq*)malloc(sizeof(queue));
   new->value = value
   new->next = NULL;

   if (front == NULL)
   {
    cqptr = new;
    front = rear = queueptr;
  }
  else
      {
     rear->next = new;
     rear=new;
   }
}

/* *************** delete element ***********/
delete()
{
int delvalue = 0;
if (front == NULL)
{ printf("Queue is empty");
  delvalue = front->value;
  if (front->next==NULL)
 {
   free(front);
   queueptr = front = rear = NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr = front;

}
}
}
```

Program 5.7 : Linked list implementation of a Circular queue

5.6 IMPLEMENTATION OF DEQUEUE

Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends. Like a linear queue and a circular queue, a dequeue can also be implemented using arrays or linked lists.

5.6.1 Array implementation of a dequeue

If a Dequeue is implemented using arrays, then it will suffer with the same problems that a linear queue had suffered. Program 5.8 gives the array implementation of a Dequeue.

```
#include “stdio.h”
#define QUEUE_LENGTH 10;
int dq[QUEUE_LENGTH];
int front, rear, choice,x,y;
main()
{
  int choice,x;
  front = rear = -1; /* initialize the front and rear to null i.e empty queue */

 printf (“enter 1 for addition and  2 to remove element from the front of the queue");
 printf (“enter 3 for addition and  4 to remove element from the rear of the queue");
 printf(“Enter your choice”);
 scanf(“%d”,&choice);
 switch (choice)
 {
  case 1:
            printf (“Enter element to be added :”);
            scanf(“%d”,&x);
            add_front(x);
            break;
  case 2:
           delete_front();
           break;
  case 3:
           printf (“Enter the element to be added :”);
           scanf(“%d ”,&x);
           add_rear(x);
           break;
  case 4:
           delete_rear();
           break;
 }
}
/*************** Add at the front **************/
add_front(int y)
{
if (front == 0)
{
    printf(“Element can not be added at the front“);
    return;
else
    {
     front = front - 1;
     dq[front] = y;
      if (front == -1 ) front = 0;
    }
 }
/*************** Delete from the front **************/
delete_front()
{
 if front == -1
 printf(“Queue empty”);
```

```
 else
        return dq[front];
        if (front = = rear)
        front = rear = -1
        else
             front = front + 1;
}
/**************** Add at the rear ***************/
add_rear(int y)
if (front == QUEUE_LENGTH -1 )
{
 printf("Element can not be added at the rear ")
 return;
else
{
  rear  = rear + 1;
  dq[rear] = y;
  if (rear = = -1 )
  rear = 0;
}
}

/**************** Delete at the rear ***************/
delete_rear()
{
  if rear == -1
  printf("deletion is not possible from rear");
  else
 {
        if (front = = rear)
        front = rear = -1
        else
             { rear = rear – 1;
               return dq[rear];
             }
}
}
```

Program 5.8: Array implementation of a Dequeue

5.6.2 Linked list implementation of a dequeue

Double ended queues are implemented with doubly linked lists.

A doubly link list can traverse in both the directions as it has two pointers namely left and right. The right pointer points to the next node on the right where as the left pointer points to the previous node on the left. Program 5.9 gives the linked list implementation of a Dequeue.

```
# include "stdio.h"
#define NULL 0
struct dq {
              int info;
              int *left;
              int *right;
             };
typedef struct dq *dqptr;
dqptr p, tp;
```

```
dqptr head;
dqptr tail;
main()
{
        int choice, I, x;
       dqptr n;
       dqptr getnode();
        printf("\n Enter 1: Start  2 : Add at Front 3 : Add at Rear 4: Delete at Front 5:
Delete at Back");
while (1)
{
        printf("\n 1: Start  2 : Add at Front 3 : Add at Back 4: Delete at Front 5: Delete
at Back 6 : exit");
        scanf("%d", &choice);
        switch (choice)
        {
         case 1:
                    create_list();
                    break;
         case 2:
                    eq_front();
                    break;
         case 3:
                   eq_back();
                   break;
        case 4:
                  dq_front();
                  break;
        case 5:
                  dq_back();
                  break;
        case 6 :
                  exit(6);
   }
 }
}

create_list()
{
  int I, x;
  dqptr t;
  p = getnode();
  tp = p;
  p->left = getnode();
  p->info = 10;
  p_right = getnode();
  return;
}

dqptr getnode()
{
   p = (dqptr) malloc(sizeof(struct dq));
   return p;
}

dq_empty(dq q)
{
  return q->head = = NULL;
```

```
}
eq_front(dq q, void *info)
{
  if (dq_empty(q))
     q->head = q->tail = dcons(info, NULL, NULL);
else
{
    q-> head -> left =dcons(info, NULL, NULL);
    q->head -> left ->right = q->head;
    q ->head  = q->head ->left;
}
}

eq_back(dq q, void *info)
{
  if (dq_empty(q))
     q->head = q->tail = dcons(info, NULL, NULL)
  else
{
  q-> tail -> right =dcons(info, NULL, NULL);
  q->tail -> right -> left = q->tail;
  q ->tail  = q->tail ->right;
}
}
  dq_front(dq q)
{
   if dq is not empty
{
dq tp = q-> head;
void *info = tp -> info;
q ->head = q->head-> right;
free(tp);
if (q->head = = NULL)
   q -> tail = NULL;
else
 q -> head -> left = NULL;
return info;
}
}

dq_back(dq q)
{
   if (q!=NULL)
   {
    dq tp = q-> tail;
    *info = tp -> info;
    q ->tail = q->tail-> left;
    free(tp);
    if (q->tail = = NULL)
                  q -> head = NULL;
  else
                 q -> tail -> right = NULL;
    return info;
   }
}
```

Program 5.9 : Linked list implementation of a Dequeue

☞ Check Your Progress 2

1) _____ allows elements to be added and deleted at the front as well as at the rear.
2) It is not possible to implement multiple queues in an Array. (True/False)
3) The index of a circular queue starts at _____.

5.7 SUMMARY

In this unit, we discussed the data structure *Queue***.** It had two ends. One is front from where the elements can be deleted and the other if rear to where the elements can be added. A queue can be implemented using Arrays or Linked lists. Each representation is having it's own advantages and disadvantages. The problems with arrays are that they are limited in space. Hence, the queue is having a limited capacity. If queues are implemented using linked lists, then this problem is solved. Now, there is no limit on the capacity of the queue. The only overhead is the memory occupied by the pointers.

There are a number of variants of the queues. Normally, queues mean circular queues. Apart from linear queues, we also discussed circular queues in this unit. A special type of queue called Dequeue was also discussed in this unit. Dequeues permit elements to be added or deleted at either of the rear or front. We also discussed the array and linked list implementations of Dequeue.

5.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1. rear , front
2. First in First out (FIFO) list

Check Your Progress 2

1. Dequeue
2. False
3. 0

5.9 FURTHER READINGS

Reference Books

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein , PHI publications
2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications

Reference Websites

http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/queues.html
http://www.cs.toronto.edu/~wayne/libwayne/libwayne.html

UNIT 6 TREES

Structure **Page Nos.**

6.0	Introduction	31
6.1	Objectives	31
6.2	Abstract Data Type-Tree	31
6.3	Implementation of Tree	34
6.4	Tree Traversals	35
6.5	Binary Trees	37
6.6	Implementation of a Binary Tree	38
6.7	Binary Tree Traversals	40
	6.7.1 Recursive Implementation of Binary Tree Traversals	
	6.7.2 Non-Recursive Implementation of Binary Tree Traversals	
6.8	Applications	43
6.9	Summary	45
6.10	Solutions/Answers	45
6.11	Further Readings	46

6.0 INTRODUCTION

Have you ever thought how does the operating system manage our files? Why do we have a hierarchical file system? How do files get saved and deleted under hierarchical directories? Well, we have answers to all these questions in this section through a hierarchical data structure called Trees! Although most general form of a tree can be defined as an **acyclic graph**, we will consider in this section only rooted tree as general tree does not have a parent-child relationship.

Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion. Consider a Tree representing your family structure. Let us say that we start with your grand parent; then come to your parent and finally, you and your brothers and sisters. In this unit, we will go through the basic tree structures first (general trees), and then go into the specific and more popular tree called binary-trees.

6.1 OBJECTIVES

After going through this unit, you should be able

- to define a tree as abstract data type (ADT);
- learn the different properties of a Tree and a Binary tree;
- to implement the Tree and Binary tree, and
- give some applications of Tree.

6.2 ABSTRACT DATA TYPE-TREE

Definition: A set of data values and associated operations that are precisely specified independent of any particular implementation.

Since the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the

operations, relationship to other abstract data types, whether a programming language implements the particular data type, etc.

Consider the following abstract data type:

Structure Tree

type Tree = nil | fork (Element , Tree , Tree)

Operations:

null : Tree -> Boolean
leaf : Tree -> Boolean
fork : (Element , Tree , Tree) -> Tree
left : Tree -> Tree // It depicts the properties of tree that left of a tree is also a tree.
right: Tree -> Tree
contents: Tree -> Element
height (nil) = 0 |
height (fork(e,T,T')) = 1+max(height(T), height(T'))
weight (nil) = 0 |
weight (fork(e,T,T')) = 1+weight(T)+weight(T')

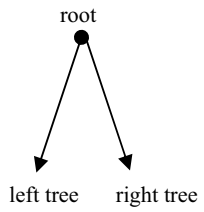


Figure 6.1: A binary tree

Rules:

null(nil) = true // nil is an empty tree

null(fork(e, T, T'))= false // e : element , T and T are two sub tree

leaf(fork(e, nil, nil)) = true
leaf(fork(e, T, T')) = false if not null(T) or not null(T')
leaf(nil) = error

left(fork(e, T, T')) = T
left(nil) = error

right(fork(e, T, T')) = T'
right(nil) = error

contents(fork(e, T, T')) = e
contents(nil) = error

Look at the definition of Tree (ADT). A way to think of a *binary tree* is that it is either empty (nil) or contains an element and two sub trees which are themselves binary trees (Refer to *Figure 6.1*). Fork operation joins two sub tree with a parent node and

produces another Binary tree. It may be noted that a tree consisting of a single leaf is defined to be of height 1.

Definition : A tree is a connected, acyclic graph (Refer to *Figure 6.2*).

It is so connected that any node in the graph can be reached from any other node by exactly one path.

It does not contain any cycles (circuits, or closed paths), which would imply the existence of more than one path between two nodes. This is the most general kind of tree, and may be converted into the more familiar form by designating a node as the root. We can represent a tree as a construction consisting of nodes, and edges which represent a relationship between two nodes. In *Figure 6.3*, we will consider most common tree called **rooted tree**. A rooted tress has a single root node which has no parents.

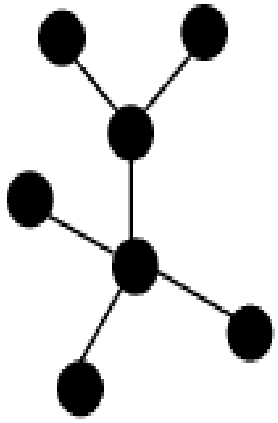


Figure 6.2 : Tree as a connected acyclic graph

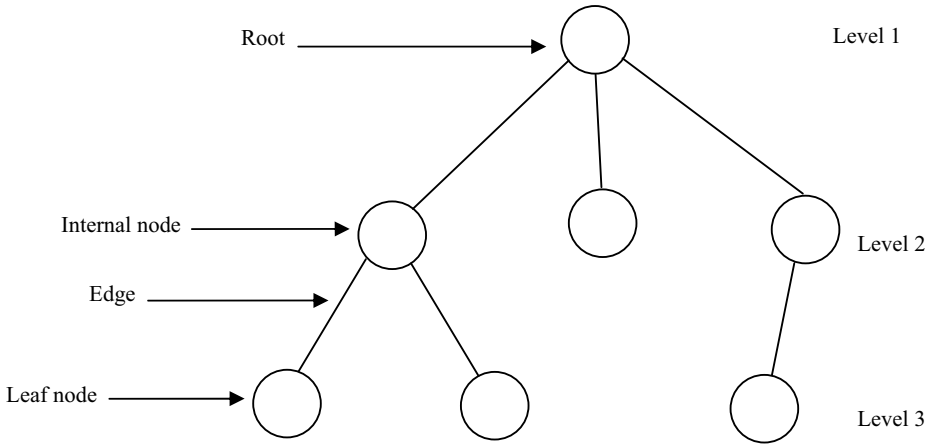


Figure 6.3 : A rooted tree

In a more formal way, we can define a tree T as a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in (T – { r }) are partitioned into n > 0 disjoint subsets T_1, T_2, ..., T_k each of which is a tree, and whose roots r1 , r2 , ..., rk , respectively, are children of r. The general tree is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is a Family Tree.
A tree is an instance of a more general category called graph.

- A tree consists of nodes connected by edges.
- A root is a node without parent.
- Leaves are nodes with no children.
- The root is at level 1. The child nodes of root are at level 2. The child nodes of nodes at level 2 are at level 3 and so on.
- The depth (height) of a Binary tree is equal to the number of levels in it.
- Branching factor defines the maximum number of children to any node. So, a branching factor of 2 means a binary tree.

- Breadth defines the number of nodes at a level.
- The depth of a node M in a tree is the length of the path from the root of the tree to M.
- A node in a Binary tree has at most 2 children.

The following are the properties of a Tree.

Full Tree : A tree with all the leaves at the same level, and all the non-leaves having the same degree

- Level h of a full tree has d^{h-1} nodes.
- The first h levels of a full tree have $1 + d + d^2 + d^3 + d^4 + \ldots\ldots + d^{h-1} = (d^h - 1)/(d - 1)$ nodes where d is the degree of nodes.
- The number of edges = the number of nodes – 1 (Why? Because, an edge represents the relationship between a child and a parent, and every node has a parent except the root.
- A tree of height *h* and degree *d* has at most $d^h - 1$ elements.

Complete Trees

A complete tree is a k-ary position tree in which all levels are filled from left to right. There are a number of specialized trees.

They are binary trees, binary search trees, AVL-trees, red-black trees, 2-3 trees.

Data structure- Tree

Tree is a dynamic data structures. Trees can expand and contract as the program executes and are implemented through pointers. A tree deallocates memory when an element is deleted.

Non-linear data structures: Linear data structures have properties of ordering relationship (can the elements/nodes of tree be sorted?). There is no first node or last node. There is no ordering relationship among elements of tree.

Items of a tree can be partially ordered into a hierarchy via parent-child relationship. Root node is at the top of the hierarchy and leafs are at the bottom layer of the hierarchy. Hence, trees can be termed as hierarchical data structures.

6.3 IMPLEMENTATION OF TREE

The most common way to add nodes to a general tree is to first find the desired parent of the node you want to insert, then add the node to the parent's child list. The most common implementations insert the nodes one at a time, but since each node can be considered a tree on its own, other implementations build up an entire sub-tree before adding it to a larger tree**.** As the nodes are added and deleted dynamically from a tree, tree are often implemented by link lists. However, it is simpler to write algorithms for a data representation where the numbers of nodes are fixed. *Figure* 6.4 depicts the structure of the node of a general k-ary tree.

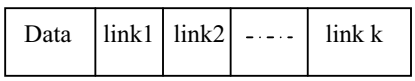


Figure 6.4 : Node structure of a general k-ary tree

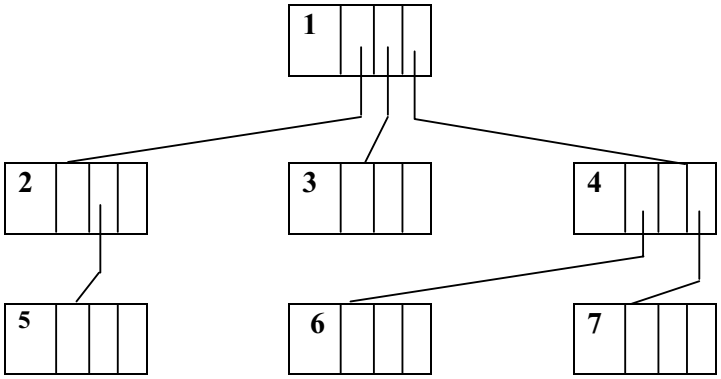


Figure 6.5: A linked list representation of tree (3-ary tree)

Figure 6.5 depicts a tree with one data element and three pointers. The number of pointers required to implement a general tree depend of the maximum degree of nodes in the tree.

6.4 TREE TRAVERSALS

There are three types of tree traversals, namely, Preorder, Postorder and Inorder.

Preorder traversal: Each node is visited before its children are visited; the root is visited first.

Algorithm for pre order traversal:

1. visit root node
2. traverse left sub-tree in preorder
3. traverse right sub-tree in preorder

Example of pre order traversal: Reading of a book, as we do not read next chapter unless we complete all sections of previous chapter and all it's sections (refer to *Figure 6.6*).

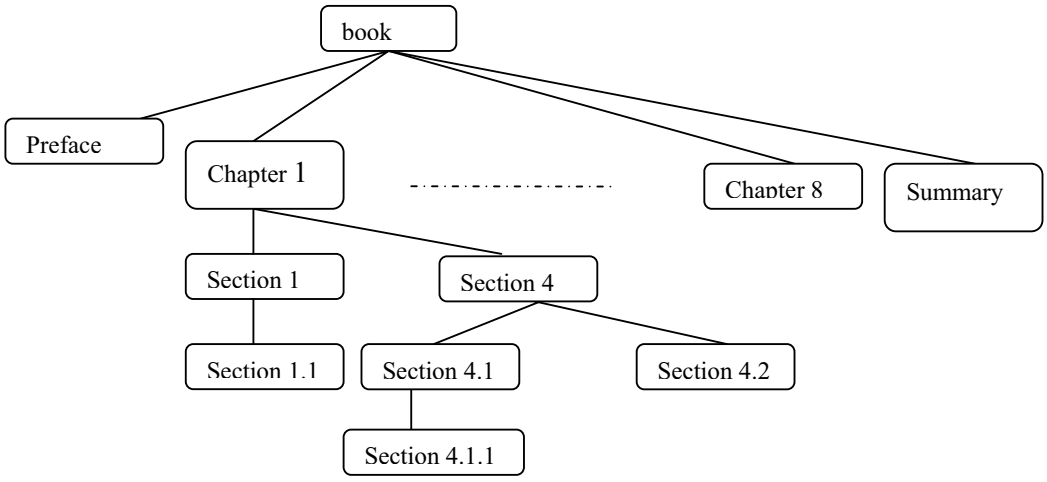


Figure 6.6 : Reading a book : A preorder tree traversal

As each node is traversed only once, the time complexity of preorder traversal is T(n) = O(n), where n is number of nodes in the tree.

Postorder traversal: The children of a node are visited before the node itself; the root is visited last. Every node is visited after its descendents are visited.

Algorithm for postorder traversal:

1. traverse left sub-tree in post order
2. traverse right sub-tree in post order
3. visit root node.

Finding the space occupied by files and directories in a file system requires a postorder traversal as the space occupied by directory requires calculation of space required by all files in the directory (children in tree structure) (refer to *Figure 6.7*)

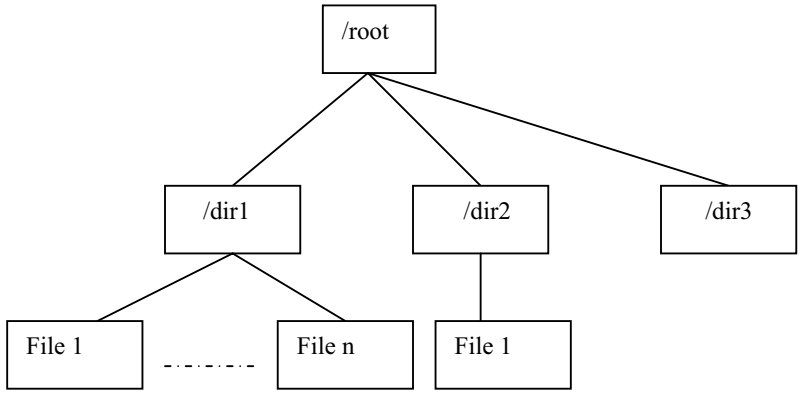


Figure 6.7 : Calculation of space occupied by a file system : A post order traversal

As each node is traversed only once, the time complexity of post order traversal is T(n) = O(n), where n is number of nodes in the tree.

Inorder traversal: The left sub tree is visited, then the node and then right sub-tree.

Algorithm for inorder traversal:

1. traverse left sub-tree
2. visit node
3. traverse right sub-tree

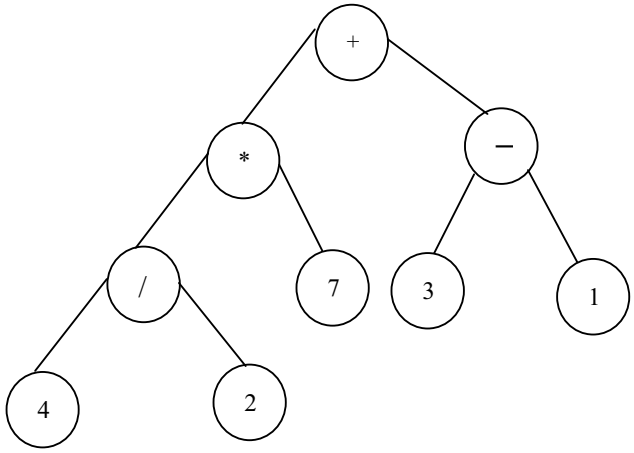


Figure 6.8 : An expression tree : An inorder traversal

Inorder traversal can be best described by an expression tree, where the operators are at parent node and operands are at leaf nodes.

Let us consider the above expression tree (refer to *Figure 6.8*). The preorder, postorder and inorder traversal are given below:

preorder Traversal : + * / 4 2 7 - 3 1
postorder traversal : 4 2 / 7 * 3 1 - +
inorder traversal : - ((((4 / 2) * 7) + (3 - 1))

There is another tree traversal (of course, not very common) is called level order, where all the nodes of the same level are travelled first starting from the root (refer to *Figure 6.9*).

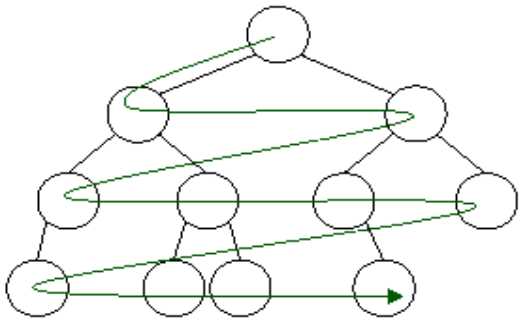


Figure 6.9: Tree Traversal: Level Order

☞ Check Your Progress 1

1) If a tree has 45 edges, how many vertices does it have?
2) Suppose a full 4-ary tree has 100 leaves. How many internal vertices does it have?
3) Suppose a full 3-ary tree has 100 internal vertices. How many leaves does it have?
4) Prove that if T is a full m-ary tree with v vertices, then T has ((m-1)v+1)/m leaves.

6.5 BINARY TREES

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

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

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

Properties of a binary tree

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

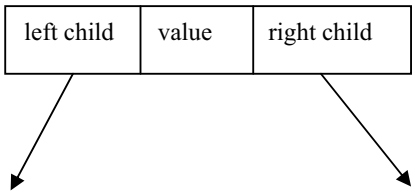
Full Binary Trees: A binary tree of height h which had $2^h - 1$ elements is called a Full Binary Tree.

Complete Binary Trees: A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full. If the bottom level is incomplete, then it has all nodes to the left side. That is the tree has been filled in the level order from left to right.

6.6 IMPLEMENTATION OF A BINARY TREE

Like general tree, binary trees are implemented through linked lists. A typical node in a Binary tree has a structure as follows (refer to *Figure 6.10*):

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



The 'left child'and 'right child' are pointers to another tree-node. The "leaf node" (not shown) here will have NULL values for these pointers.

Figure 6.10 : Node structure of a binary tree

The binary tree creation follows a very simple principle. For the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree, then move towards the left side of that element or else to its right. If there is no sub tree on the left, then make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly, the same has to done for the case when your new element is greater than the current element in the tree but this time with the right child. Though this logic is followed for the creation of a Binary tree, this logic is often suitable to search for a key value in the binary tree.

Algorithm for the implementation of a Binary tree:

Step-1: If value of new element < current element, then go to step-2 or else step -3
Step-2: If the current element does not have a left sub-tree, then make your new

element the left child of the current element; else make the existing left child as your current element and go to step-1

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

Program 6.1 depicts the segment of code for the creation of a binary tree.

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

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

Program 6.1 : Binary tree creation

Array-based representation of a Binary Tree

Consider a complete binary tree T having n nodes where each node contains an item (value). Label the nodes of the complete binary tree T from top to bottom and from left to right 0, 1, ..., n-1. Associate with T the array A where the i^{th} entry of A is the item in the node labelled i of T, i = 0, 1, ..., n-1. Figure 6.11 depicts the array representation of a Binary tree of Figure 6.16.

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

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

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

Figure 6.11 : Array Representation of a Binary Tree

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

6.7 BINARY TREE TRAVERSALS

We have already discussed about three tree traversal methods in the previous section on general tree. The same three different ways to do the traversal – preorder, inorder and postorder are applicable to binary tree also.

Let us discuss the inorder binary tree traversal for following binary tree (refer to *Figure 6.12*):

We start from the root i.e. * We are supposed to visit its left sub-tree then visit the node itself and its right sub-tree. Here, root has a left sub-tree rooted at +. So, we move to + and check for its left sub-tree (we are suppose repaeat this for every node). Again, + has a left sub-tree rooted at 4. So, we have to check for 4's left sub-tree now, but 4 doesn't have any left sub-tree and thus we will visit node 4 first (print in our case) and check for its right sub-tree. As 4 doesn't have any right sub-tree, we'll go back and visit node +; and check for the right sub-tree of +. It has a right sub-tree rooted at 5 and so we move to 5. Well, 5 doesn't have any left or right sub-tree. So, we just visit 5 (print 5)and track back to +. As we have already visited + so we track back to * . As we are yet to visit the node itself and so we visit * before checking for the right sub-tree of *, which is 3. As 3 does not have any left or right sub-trees, we visit 3.
So, the inorder traversal results in 4 + 5 * 3

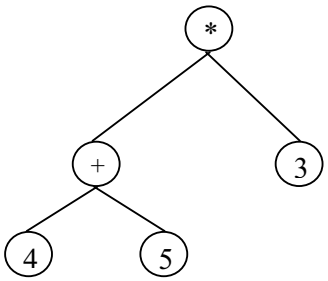


Figure 6.12 : A binary tree

Algorithm: Inorder
Step-1: For the current node, check whether it has a left child. If it has, then go to step-2 or else go to step-3
Step-2: Repeat step-1 for this left child
Step-3: Visit (i.e. printing the node in our case) the current node
Step-4: For the current node check whether it has a right child. If it has, then go to step-5
Step-5: Repeat step-1 for this right child

The preoreder and postorder traversals are similar to that of a general binary tree. The general thing we have seen in all these tree traversals is that the traversal mechanism is inherently recursive in nature.

6.7.1 Recursive Implementation of Binary Tree Traversals

There are three classic ways of recursively traversing a binary tree. In each of these, the left and right sub-trees are visited recursively and the distinguishing feature is **when the element in the root is visited or processed**.

Program 6.2, Program 6.3 and Program 6.4 depict the inorder, preorder and postorder traversals of a Binary tree.

```
struct NODE
{
```

```
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

inorder(struct NODE *curr)
{
 if(curr->left != NULL) inorder(curr->left);
 printf("%d", curr->value);

 if(curr->right != NULL) inorder(curr->right);

}
```

Program 6.2 : Inorder traversal of a binary tree

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

preorder(struct NODE *curr)
{
 printf("%d", curr->value);
 if(curr->left != NULL) preorder(curr->left);
 if(curr->right != NULL) preorder(curr->right);

}
```

Program 6.3 : Preorder traversal of a binary tree

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

postorder(struct NODE *curr)
{
  if(curr->left != NULL) postorder(curr->left);
 if(curr->right != NULL) postorder(curr->right);

printf("%d", curr->value);

}
```

Program 6.4 : Postorder traversal of a binary tree

In a preorder traversal, the root is visited first (pre) and then the left and right sub-trees are traversed. In a postorder traversal, the left sub-tree is visited first, followed by right sub-tree which is then followed by root. In an inorder traversal, the left sub-tree is visited first, followed by root, followed by right sub-tree.

6.7.2 Non-recursive implementation of binary tree traversals

As we have seen, as the traversal mechanisms were inherently recursive, the implementation was also simple through a recursive procedure. However, in the case of a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

Algorithm : Non-recursive preorder binary tree traversal

```
Stack S
push root onto S
repeat until S is empty
{
   v = pop S
   if v is not NULL
   visit v
   push v's right child onto S
   push v's left child onto S
 }
```

Program 6.5 depicts the program segment for the implementation of non-recursive preorder traversal.

```
/* preorder traversal of a binary tree, implemented using a stack */
void preorder(binary_tree_type *tree)
{
 stack_type *stack;
 stack = create_stack();
 push(tree, stack);          /* push the first element of the tree to the stack */
 while (!empty(stack))
   {
   tree = pop(stack);
   visit(tree);
   push(tree->right, stack);  /* push right child to the stack */
   push(tree->left, stack);     /* push left child to the stack */
   }
 }
```

Program 6.5: Non-recursive implementation of preorder traversal

In the worst case, for preorder traversal, the stack will grow to size n/2, where n is number of nodes in the tree. Another method of traversing binary tree non-recursively which does not use stack requires pointers to the parent node (called threaded binary tree).

A threaded binary tree is a binary tree in which every node that does not have a right child has a THREAD (a third link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a tree and use of stack, which makes use of a lot of memory and time.

A node structure of threaded binary is :

The node structure for a threaded binary tree varies a bit and its like this –

```
struct NODE
{
```

```
 struct NODE *leftchild;
 int node_value;
 struct NODE *rightchild;
 struct NODE *thread;   /* third pointer to it's inorder successor */
}
```

6.8 APPLICATIONS

Trees are used enormously in computer programming. These can be used for improving database search times (binary search trees, 2-3 trees, AVL trees, red-black trees), Game programming (minimax trees, decision trees, pathfinding trees), 3D graphics programming (quadtrees, octrees), Arithmetic Scripting languages (arithmetic precedence trees), Data compression (Huffman trees), and file systems (B-trees, sparse indexed trees, tries). Figure 6.13 depicts a tic-tac-toe game tree showing various stages of game.

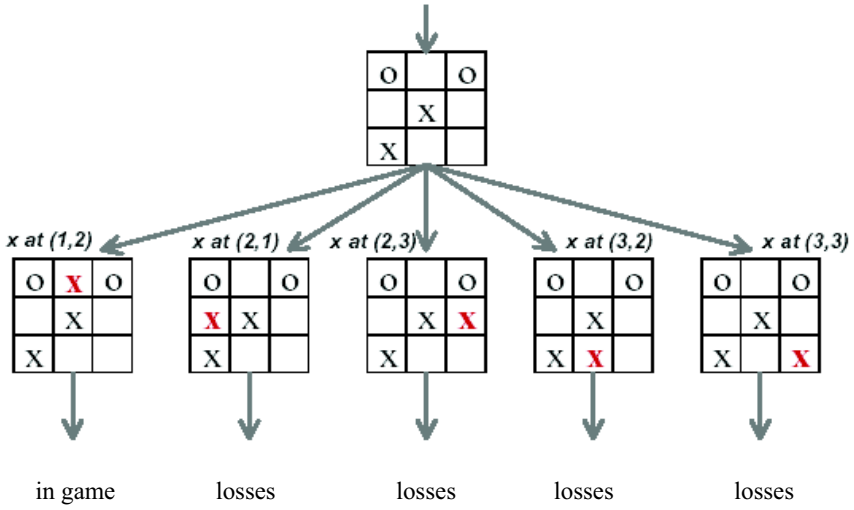


Figure 6.13 : A tic-tac-toe game tree showing various stages of game

In all of the above scenario except the first one, the player (playing with X) ultimately looses in subsequent moves.

The General tree (also known as Linked Trees) is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is in Family Tree programs. In game programming, many games use these types of trees for decision-making processes as shown above for tic-tac-toe. A computer program might need to make a decision based on an event that happened.

But this is just a simple tree for demonstration. A more complex AI decision tree would definitely have a lot more options. The interesting thing about using a tree for decision-making is that the options are cut down for every level of the tree as we go down, greatly simplifying the subsequent moves and improving the speed at which the AI program makes a decision.

The big problem with tree based level progressions, however, is that sometimes the tree can get too large and complex as the number of moves (level in a tree) increases. Imagine a game offering just two choices for every move to the next level at the end of each level in a ten level game. This would require a tree of 1023 nodes to be created.

Binary trees are used for searching keys. Such trees are called Binary Search trees(refer to *Figure 6.14*).

A Binary Search Tree (BST) is a binary tree with the following properties:

1. The key of a node is always greater than the keys of the nodes in its left sub-tree
2. The key of a node is always smaller than the keys of the nodes in its right sub-tree

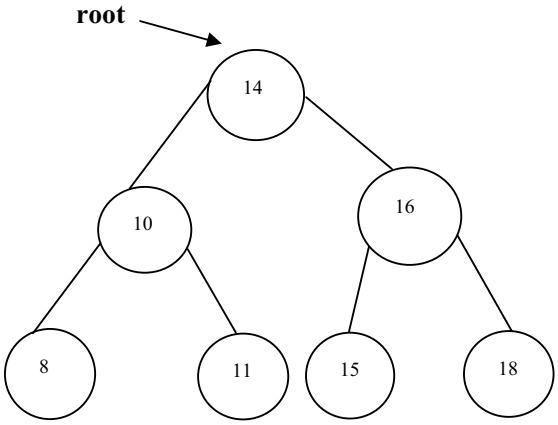


Figure 6.14 : A binary search tree (BST)

It may be seen that when nodes of a BST are traversed by inorder traversal, the keys appear in sorted order:

```
inorder(root)
{
inorder(root.left)
print(root.key)
inorder(root.right)
}
```

Binary Trees are also used for evaluating expressions.
A binary tree can be used to represent and evaluate arithmetic expressions.

1. If a node is a leaf, then the element in it specifies the value.
2. If it is not a leaf, then evaluate the children and combine them according to the operation specified by the element.

Figure 6.15 depicts a tree which is used to evaluate expressions.

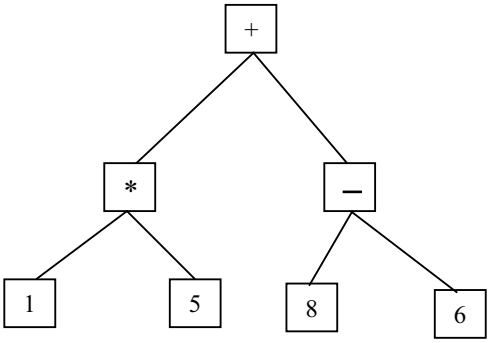


Figure 6.15 : Expression tree for 1 * 5 + 8 - 6

☞ **Check Your Progress 2**

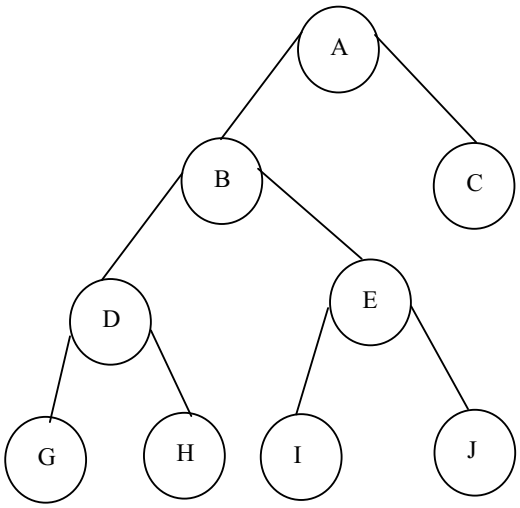


Figure 6.16 : A binary tree

1) With reference to *Figure 6.16*, find
   a) the leave nodes in the binary tree
   b) sibling of J
   c) Parent node of G
   d) depth of the binary tree
   e) level of node J
2) Give preorder, post order, inorder and level order traversal of above binary tree
3) Give array representation of the binary tree of *Figure 6.12*
4) Show that in a binary tree of N nodes, there are N+1 children with both the links as null (leaf node).

6.9 SUMMARY

Tree is one of the most widely used data structure employed for representing various problems. We studied tree as a special case of an acyclic graph. However, rooted trees are most prominent of all trees. We discussed definition and properties of general trees with their applications. Various tree traversal methods are also discussed.

Binary tree are the special case of trees which have at most two children. Binary trees are mostly implemented using link lists. Various tree traversal mechanisms include inorder, preorder and post order. These tree traversals can be implemented using recursive procedures and non-recursive procedures. Binary trees have wider applications in two way decision making problems which use yes/no, true/false etc.

6.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1) If a tree has e edges and n vertices, then e=n – 1. Hence, if a tree has 45 edges, then it has 46 vertices.

2) A full 4-ary tree with 100 leaves has i=(100 – 1)/(4 – 1)=33 internal vertices.

3) A full 3-ary tree with 100 internal vertices has l = (3 – 1)*100+ 1=201 leaves

Check Your Progress 2

1) Answers
   a. G,H,I and J
   b. I
   c. D
   d. 4
   e. 4
2) Preorder : ABDCHEIJC Postorder : GHDIJEBCA Inorder : GDHBIEFAC level-order: ABCDEGHIJ
3) Array representation of the tree in *Figure 6.12*

Index of Node	Item	Left child	Right child
0	*	1	2
1	+	3	4
2	3	-1	-1
3	4	-1	-1
4	5	-1	-1
5	?	?	?

6.11 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.

Reference websites

http://www.csee.umbc.edu
http://www.cse.ucsc.edu
http://www.webopedia.com

UNIT 7 ADVANCED TREES

Structure **Page Nos.**

7.0 Introduction 5
7.1 Objectives 5
7.2 Binary Search Trees 5
7.2.1 Traversing a Binary Search Tree
7.2.2 Insertion of a node into a Binary Search Tree
7.2.3 Deletion of a node from a Binary Search Tree
7.3 AVL Trees 9
7.3.1 Insertion of a node into an AVL tree
7.3.2 Deletion of a node from an AVL tree
7.3.3 AVL tree rotations
7.3.4 Applications of AVL trees
7.4 B-Trees 14
7.4.1 Operations on B-trees
7.4.2 Applications of B-trees
7.5 Summary 18
7.6 Solutions/Answers 18
7.7 Further Readings 19

7.0 INTRODUCTION

Linked list representations have great advantages of flexibility over the contiguous representation of data structures. But, they have few disadvantages also. Data structures organised as trees have a wide range of advantages in various applications and it is best suited for the problems related to information retrieval.
These data structures allow the searching, insertion and deletion of node in the ordered list to be achieved in the minimum amount of time.

The data structures that we discuss primarily in this unit are Binary Search Trees, AVL trees and B-Trees. We cover only fundamentals of these data structures in this unit. Some of these trees are special cases of other trees and Trees are having a large number of applications in real life.

7.1 OBJECTIVES

After going through this unit, you should be able to

- know the fundamentals of Binary Search trees;
- perform different operations on the Binary Search Trees;
- understand the concept of AVL trees;
- understand the concept of B-trees, and
- perform various operations on B-trees.

7.2 BINARY SEARCH TREES

A Binary Search Tree is a binary tree that is either empty or a node containing a key value, left child and right child.

By analysing the above definition, we note that BST comes in two variants namely empty BST and non-empty BST.

The empty BST has no further structure, while the non-empty BST has three components.

The non-empty BST satisfies the following conditions:

a) The key in the left child of a node (if exists) is less than the key in its parent node.
b) The key in the right child of a node (if exists) is greater than the key in its parent node.
c) The left and right subtrees of the root are again binary search trees.

The following are some of the operations that can be performed on Binary search trees:

- Creation of an empty tree
- Traversing the BST
- Counting internal nodes (non-leaf nodes)
- Counting external nodes (leaf nodes)
- Counting total number of nodes
- Finding the height of tree
- Insertion of a new node
- Searching for an element
- Finding smallest element
- Finding largest element
- Deletion of a node.

7.2.1 Traversing a Binary Search Tree

Binary Search Tree allows three types of traversals through its nodes. They are as follow:

1. Pre Order Traversal
2. In Order Traversal
3. Post Order Traversal

In Pre Order Traversal, we perform the following three operations:

1. Visit the node
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

In Order Traversal,we perform the following three operations:

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder.

In Post Order Traversal, we perform the following three operations:

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

Consider the BST of *Figure 7.1*

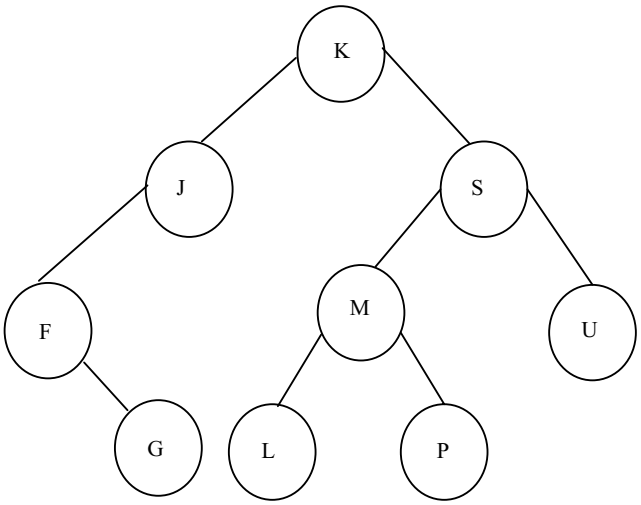


Figure 7.1: A Binary Search Tree(BST)

The following are the results of traversing the BSTof *Figure 7.1:*

Preorder : K J F G S M L P U
Inorder : F G J K L M P S U
Postorder: G F J L P M U S K

7.2.2 Insertion of a node into a Binary Search Tree

A binary search tree is constructed by the repeated insertion of new nodes into a binary tree structure.

Insertion must maintain the order of the tree. The value to the left of a given node must be less than that node and value to the right must be greater.

In inserting a new node, the following two tasks are performed :

- Tree is searched to determine where the node is to be inserted.
- On completion of search, the node is inserted into the tree

Example: Consider the BST of *Figure 7.2* After insertion of a new node consisting of value 5, the BST of Figure 7.3 results.

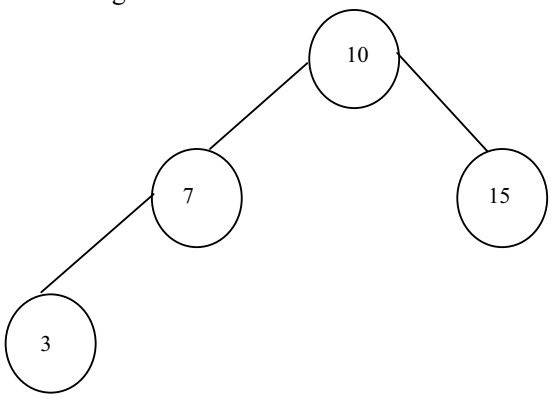


Figure 7.2: A non-empty

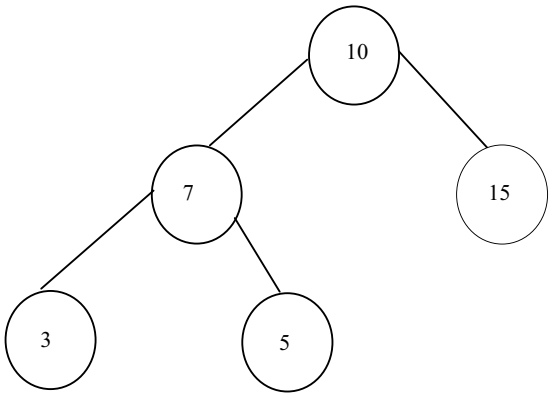


Figure 7.3: Figure 7.2 after insertion of 5

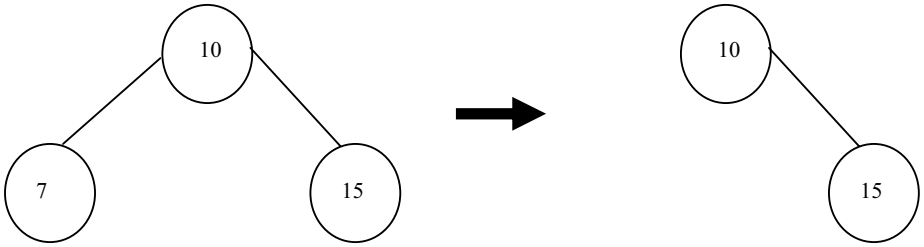
7.2.3 Deletion of a node from a Binary Search Tree

The algorithm to delete a node with key from a binary search tree is not simple where as many cases needs to be considered.

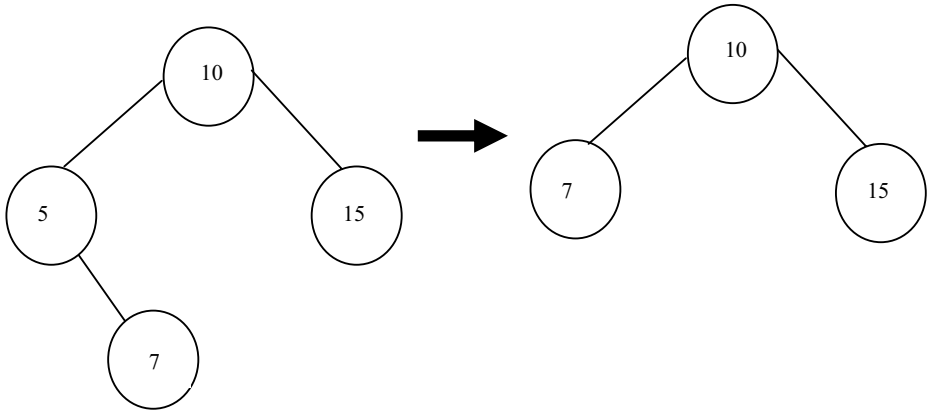
- If the node to be deleted has no sons, then it may be deleted without further adjustment to the tree.
- If the node to be deleted has only one subtree, then its only son can be moved up to take its place.
- The node *p* to be deleted has two subtrees, then its inorder successor *s* must take its place. The inorder successor cannot have a left subtree. Thus, the right son of *s* can be moved up to take the place of *s*.

Example: Consider the following cases in which node 5 needs to be deleted.

1. The node to be deleted has no children.



2. The node has one child



3. The node to be deleted has two children. This case is complex. The order of the binary tree must be kept intact.

☞ **Check Your Progress 1**

1) What are the different ways of traversing a Binary Search Tree?

……

……

2) What are the major features of a Binary Search Tree?

……

……

7.3 AVL TREES

An AVL tree is a binary search tree which has the following properties:

- The sub-tree of every node differs in height by at most one.
- Every sub tree is an AVL tree.

Figure 7.4 depicts an AVL tree.

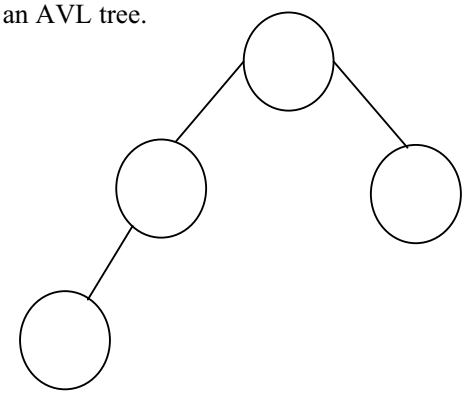


Figure 7.4 : Balance requirement for an AVL tree: the left and right subtree differ by at most one in height

AVL stands for the names of G.M. Adelson – Velskii and E.M. Landis, two Russian mathematicians, who came up with this method of keeping the tree balanced.

An AVL tree is a binary search tree which has the balance property and in addition to its key, each node stores an extra piece of information: the current balance of its subtree. The three possibilities are:

- Left – HIGH (balance factor -1)
  The left child has a height that is greater than the right child by 1.

- BALANCED (balance factor 0)
  Both children have the same height

- RIGHT – HIGH (balance factor +1)
  The right child has a height that is greater by 1.

An AVL tree which remains balanced guarantees O(log n) search time, even in the worst case. Here, n is the number of nodes. The AVL data structure achieves this property by placing restrictions on the difference in heights between the sub-trees of a given node and rebalancing the tree even if it violates these restrictions.

7.3.1 Insertion of a node into an AVL tree

Nodes are initially inserted into an AVL tree in the same manner as an ordinary binary search tree.

However, the insertion algorithm for an AVL tree travels back along the path it took to find the point of insertion and checks the balance at each node on the path.

If a node is found that is unbalanced (if it has a balance factor of either -2 or +2) then rotation is performed, based on the inserted nodes position relative to the node being examined (the unbalanced node).

7.3.2 Deletion of a node from an AVL tree

The deletion algorithm for AVL trees is a little more complex as there are several extra steps involved in the deletion of a node. If the node is not a leaf node, then it has at least one child. Then the node must be swapped with either its in-order successor or predecessor. Once the node has been swapped, we can delete it.

If a deletion node was originally a leaf node, then it can simply be removed.

As done in insertion, we traverse back up the path to the root node, checking the balance of all nodes along the path. If unbalanced, then the respective node is found and an appropriate rotation is performed to balance that node.

7.3.3 AVL tree rotations

AVL trees and the nodes it contains must meet strict balance requirements to maintain O(log n) search time. These balance restrictions are maintained using various rotation functions.

The four possible rotations that can be performed on an unbalanced AVL tree are given below. The before and after status of an AVL tree requiring the rotation are shown (refer to *Figures 7.5, 7.6, 7.7 and 7.8*).

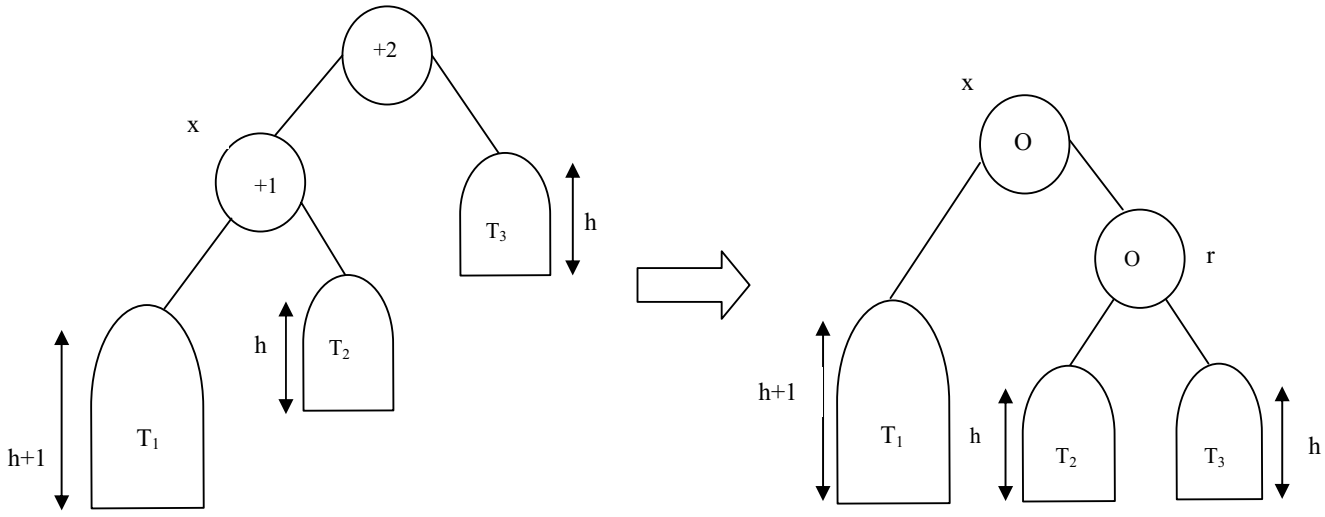


Figure 7.5: LL Rotation

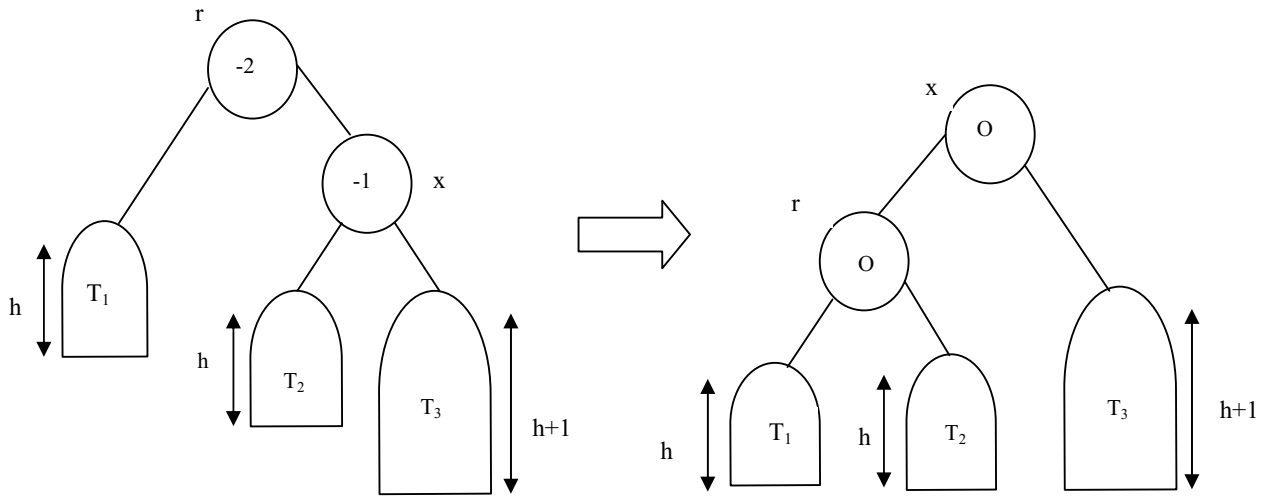


Figure 7.6: RR Rotation

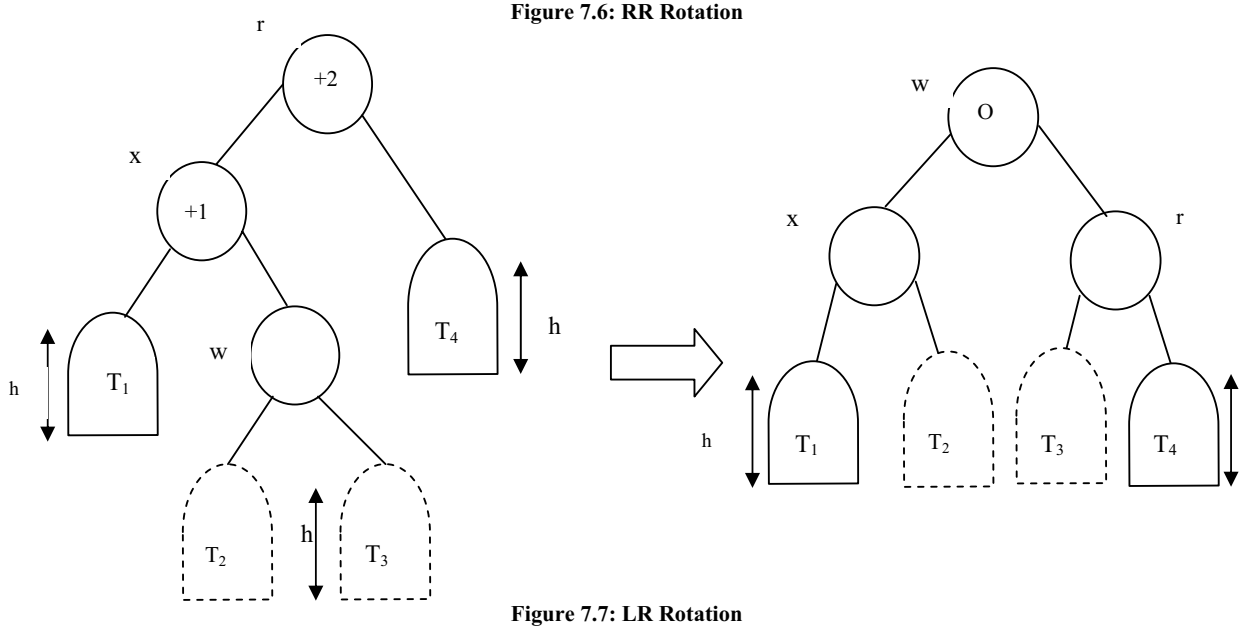


Figure 7.7: LR Rotation

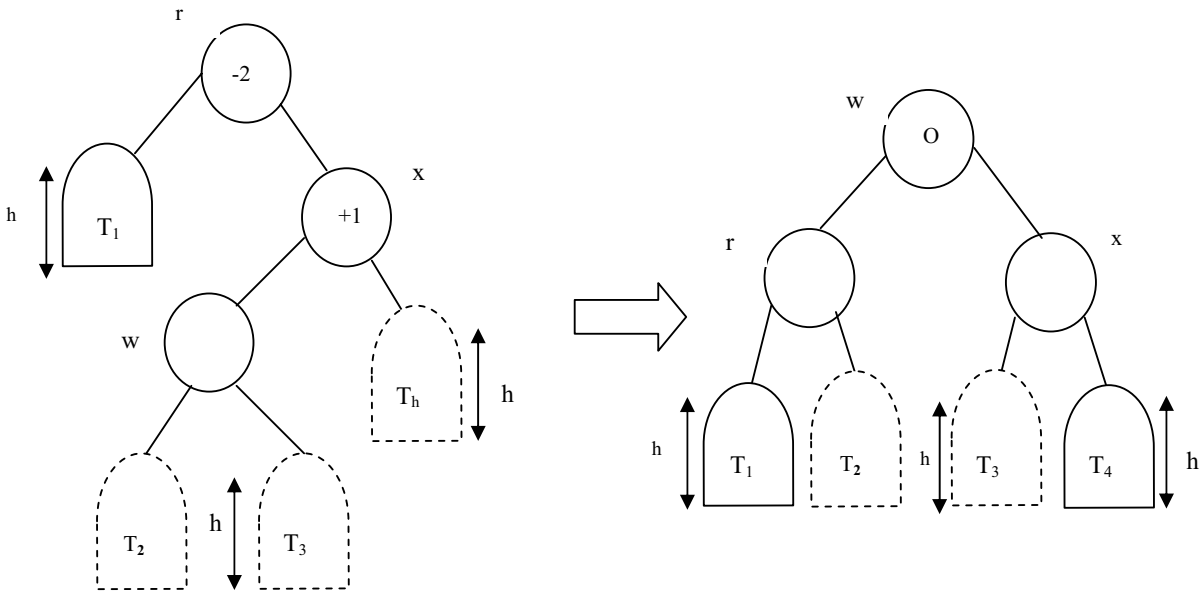


Figure 7.8: RL Rotation

Example: (Single rotation in AVL tree, when a new node is inserted into the AVL tree (LL Rotation)) (refer to *Figure 7.9*).

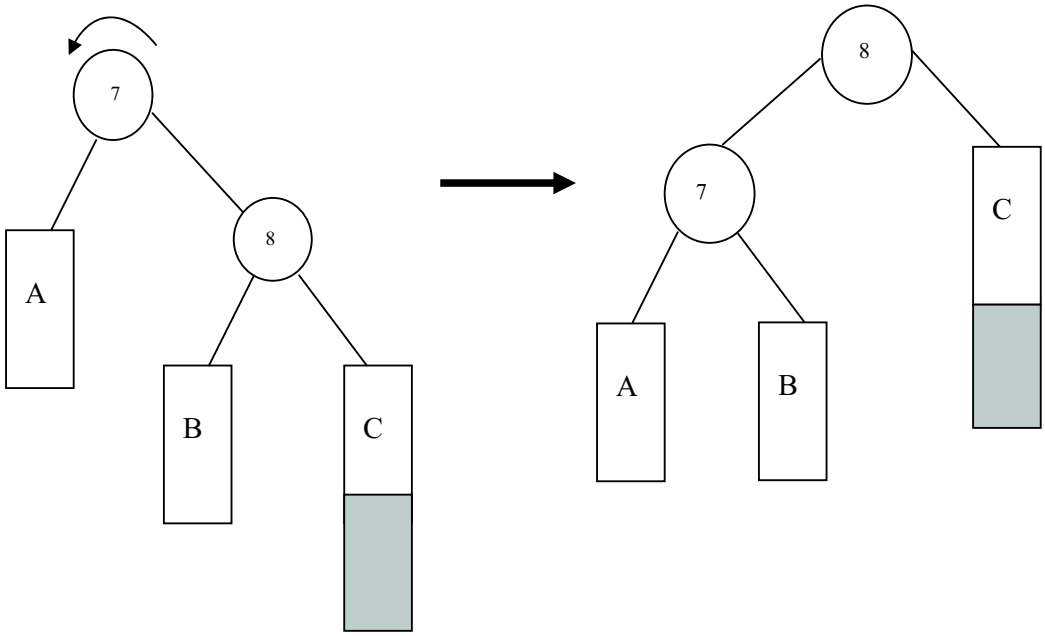


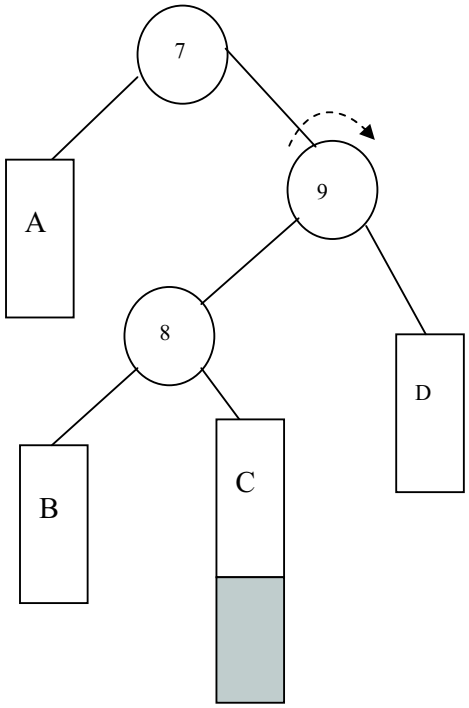
Figure 7.9: LL Rotation

The rectangles marked A, B and C are trees of equal height. The shaded rectangle stands for a new insertion in the tree C. Before the insertion, the tree was balanced, for the right child was taller then the left child by one.

The balance was broken when we inserted a node into the right child of 7, since the difference in height became 7.

To fix the balance we make 8 the new root, make c the right child move the old root (7) down to the left together with its left subtree A and finally move subtree B across and make it the new right child of 7.

Example: (Double left rotation when a new node is inserted into the AVL tree (RL rotation)) (refer to *Figure 7.10 (a),(b),(c)*).



(a)

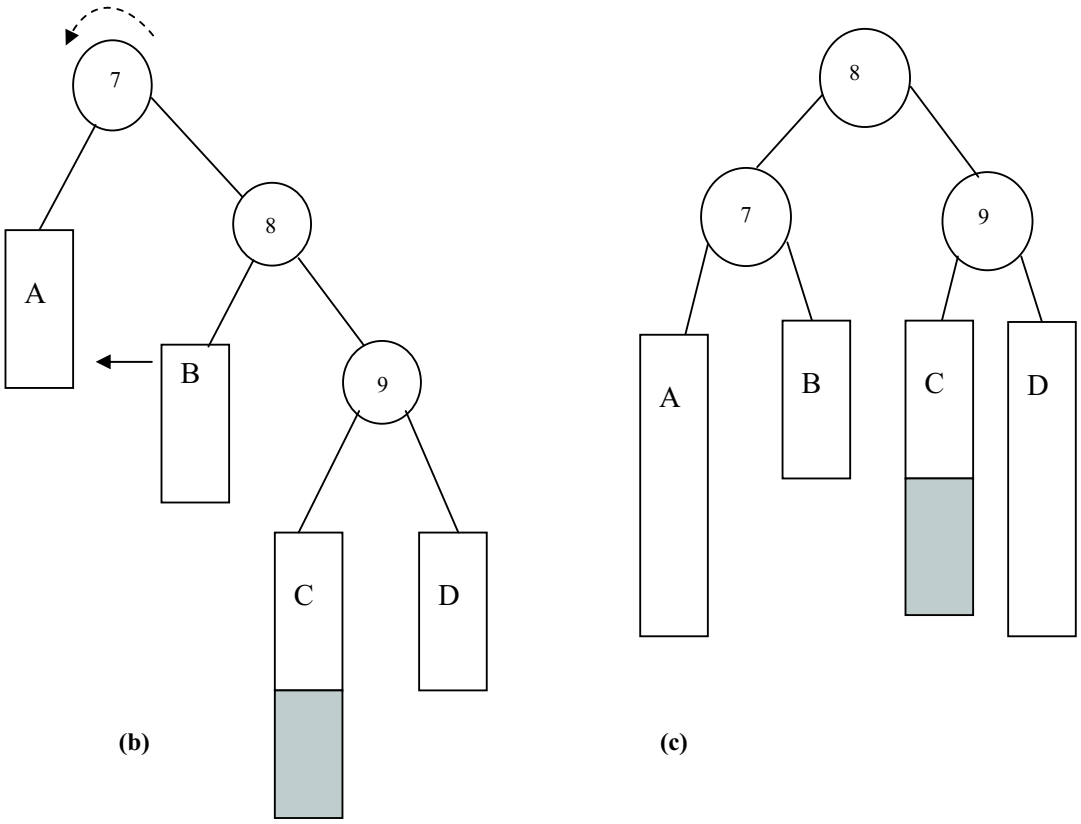


Figure 7.10: Double left rotation when a new node is inserted into the AVL tree

A node was inserted into the subtree C, making the tree off balance by 2 at the root. We first make a right rotation around the node 9, placing the C subtree into the left child of 9.

Then a left rotation around the root brings node 9 (together with its children) up a level and subtree A is pushed down a level (together with node 7). As a result we get correct AVL tree equal balance.

An AVL tree can be represented by the following structure:

```
struct avl  {
        struct node *left;
        int info;
        int bf;
        struct node *right;
};
```

bf is the balance factor, info is the value in the node.

7.3.4 Applications of AVL Trees

AVL trees are applied in the following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearly sorted

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

Consider the following:

1. A classic problem in computer science is how to store information dynamically so as to allow for quick look up. This searching problem arises often in dictionaries, telephone directory, symbol tables for compilers and while storing business records etc. The records are stored in a balanced binary tree, based on the keys (alphabetical or numerical) order. The balanced nature of the tree limits its height to O (log n), where *n* is the number of inserted records.

2. AVL trees are very fast on searches and replacements. But, have a moderately high cost for addition and deletion. If application does a lot more searches and replacements than it does addition and deletions, the balanced (AVL) binary tree is a good choice for a data structure.

3. AVL tree also has applications in file systems.

☞ Check Your Progress 2

1) Define the structure of an AVL tree.

……

……

7.4 B – TREES

B-trees are special m–ary balanced trees used in databases because their structure allows records to be inserted, deleted and retrieved with guaranteed worst case performance.

A B-Tree is a specialised multiway tree. In a B-Tree each node may contain a large number of keys. The number of subtrees of each node may also be large. A B-Tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that height of the tree is relatively small.

This means that only a small number of nodes must be read from disk to retrieve an item.

A B-Tree of order m is multiway search tree of order m such that

- All leaves are on the bottom level
- All internal nodes (except root node) have atleast m/2 (non empty) children
- The root node can have as few as 2 children if it is an internal node and can have no children if the root node is a leaf node
- Each leaf node must contain atleast (m/2) – 1 keys.

The following is the structure for a B-tree :

```
struct btree

{       int count;              // number of keys stored in the current node
        item_type key[3];               // array to hold 3 keys
        long branch [4];        // array of fake pointers (records numbers)
};
```

Figure 7.11 depicts a B-tree of order 5.

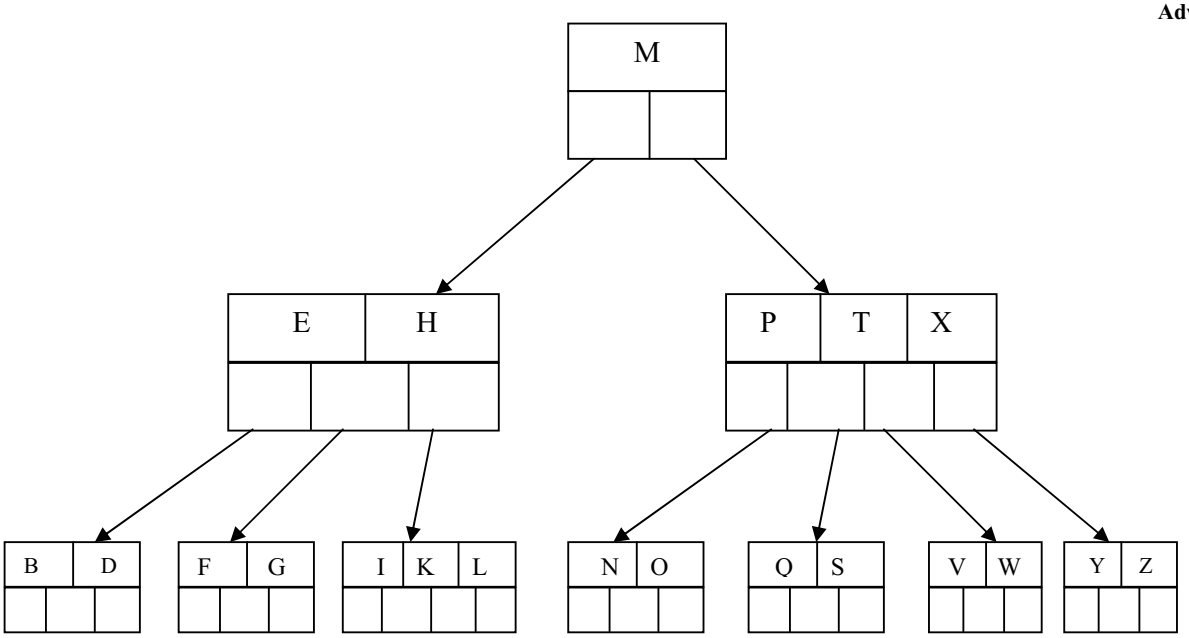


Figure 7.11: A B-tree of order 5

7.4.1 Operations on B-Trees

The following are various operations that can be performed on B-Trees:

- Search
- Create
- Insert

B-Tree strives to minimize disk access and the nodes are usually stored on disk

All the nodes are assumed to be stored in secondary storage rather than primary storage. All references to a given node are preceded by a read operation. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with write operation.

The following is the algorithm for searching a B-tree:

B-Tree Search (x, k)

```
i < - 1
while i < = n [x] and k > key_i[x]
        do i ← i + 1
if i < = n [x] and k = key_1 [x]
        then return (x, i)
if leaf [x]
        then return NIL
else Disk – Read (c_i[x])
        return B – Tree Search (C_i[x], k)
```

The search operation is similar to binary tree. Instead of choosing between a left and right child as in binary tree, a B-tree search must make an n-way choice.

The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to desired value, the child pointer to the immediate left to that value is followed.

The exact running time of search operation depends upon the height of the tree.

The following is the algorithm for the creation of a B-tree:

B-Tree Create (T)

```
x ← Allocate-Node ( )
Leaf [x] ← True
n [x] ← 0
Disk-write (x)
root [T] ← x
```

The above mentioned algorithm creates an empty B-tree by allocating a new root that has no keys and is a leaf node.

The following is the algorithm for insertion into a B-tree:

B-Tree Insert (T,K)

```
r ← root (T)
if n[r] = 2t – 1
        then S ← Allocate-Node ( )
                root[T] ← S
                leaf [S] ← FALSE
                n[S] ← 0
                C1 ← r
                B–Tree-Split-Child (s, I, r)
                B–Tree-Insert-Non full (s, k)
                  else
                B – Tree-Insert-Non full (r, k)
```

To perform an insertion on B-tree, the appropriate node for the key must be located. Next, the key must be inserted into the node.

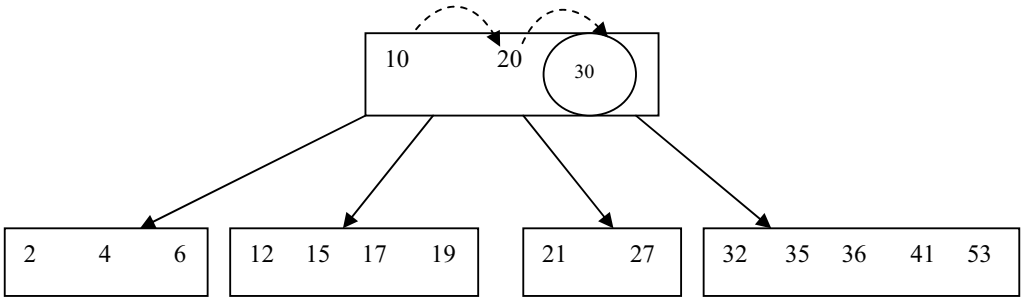
If the node is not full prior to the insertion, then no special action is required.

If node is full, then the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full. Else, another split operation is required.

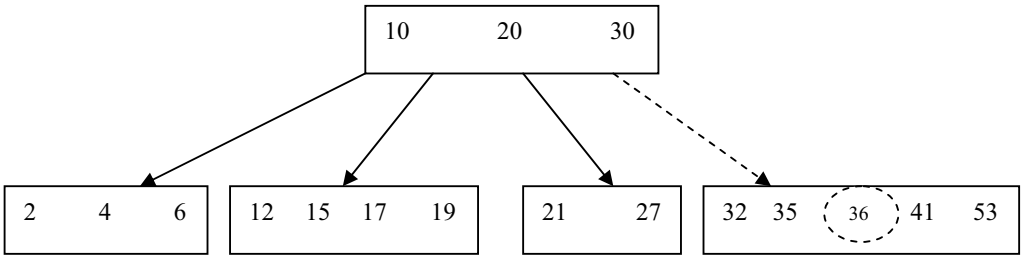
This process may repeat all the way up to the root and may require splitting the root node.

Example: Insertion of a key 33 into a B-Tree (w/split) (refer to *Figure 7.12)*

Step 1: Search first node for key nearest to 33. Key 30 was found.



Step 2: Node pointed by key 30, is searched for inserting 33. Node is split and 36 is shifted upwards.



Step 3: Key 33 is inserted between 32 and 35.

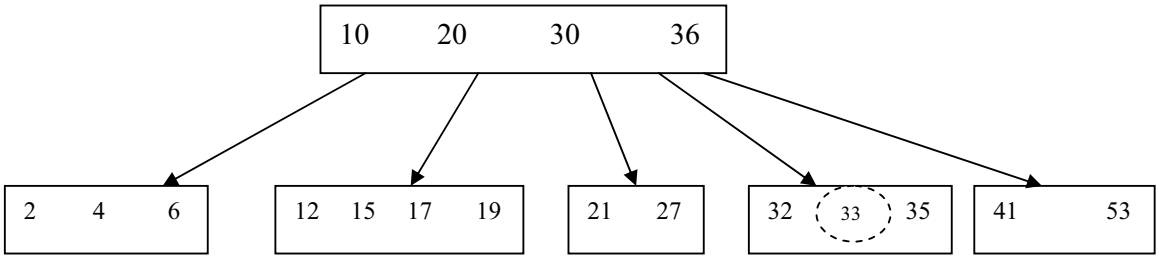
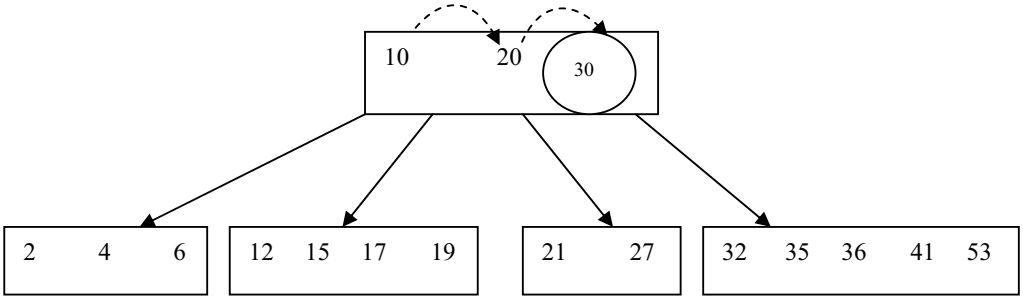


Figure 7.12 : A B-tree

Deletion of a key from B-tree is possible, but care must be taken to ensure that the properties of b-tree are maintained if the deletion reduces the number of keys in a node below the minimum degree of tree, this violation must be connected by combining several nodes and possibly reducing the height if the tree. If the key has children, the children must be rearranged.

Example (Searching of a B – Tree for key 21(refer to Figure 7.13))

Step 1: Search for key 21 in first node. 21 is between 20 and 30.



Step2 : Searching is conducted on the nodes connected by 30.

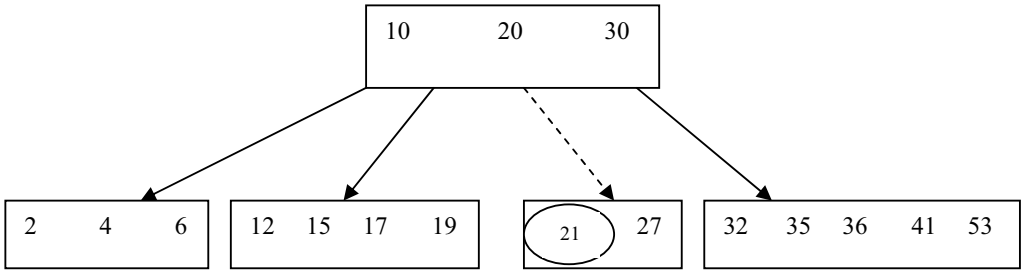


Figure 7.13 : A B-tree

7.4.2 Applications of B-trees

A database is a collection of data organised in a fashion that facilitates updation, retrieval and management of the data. Searching an unindexed database containing n keys will have a worst case running time of O (n). If the same data is indexed with a b-tree, then the same search operation will run in O(log n) time. Indexing large amounts of data can significantly improve search performance.

☞ Check Your Progress 3

1) Create a B – Tree of order 5 for the following:
CNGAHEKQMSWLTZDPRXYS
……....
……

2) Define a multiway tree of order m.
……
……

7.5 SUMMARY

In this unit, we discussed Binary Search Trees, AVL trees and B-trees.

The striking feature of Binary Search Trees is that all the elements of the left subtree of the root will be less than those of the right subtree. The same rule is applicable for all the subtrees in a BST. An AVL tree is a Height balanced tree. The heights of left and right subtrees of root of an AVL tree differ by 1. The same rule is applicable for all the subtrees of the AVL tree. A B-tree is a m-ary binary tree. There can be multiple elements in each node of a B-tree. B-trees are used extensively to insert , delete and retrieve records from the databases.

7.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1) preorder, postorder and inorder
2) The major feature of a Binary Search Tree is that all the elements whose values is less than the root reside in the nodes of left subtree of the root and all the elements whose values are larger than the root reside in the nodes of right subtree of the root. The same rule is applicable to all the left and right subtrees of a BST.

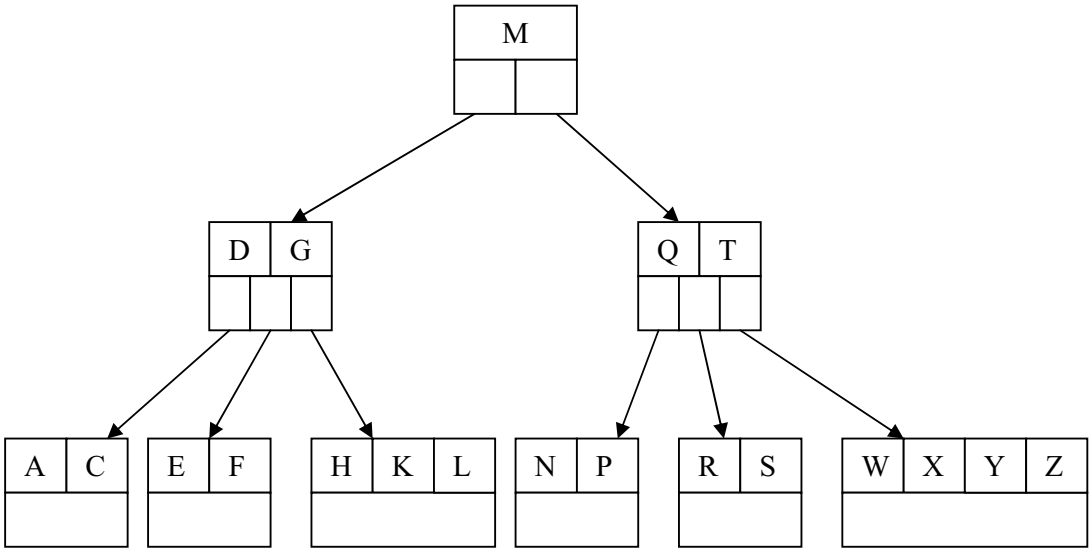
Check Your Progress 2

1) The following is the structure of an AVL tree:

```
struct avl  {
        struct node *left;
        int info;
        int bf;
        struct node *right;
};
```

Check Your Progress 3

1)



2) A multiway tree of order n is an ordered tree where each node has at most m children. For each node, if k is the actual no. of children in the node, then k-1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is multiway search tree of order m.

7.7 FURTHER READINGS

1. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J. Augenstein and Aaron M. Tanenbaum, PHI Publications.

2. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

Reference Websites

http:// www.cs.umbc.edu
http://www.fredosaurus.com

UNIT 8 GRAPHS

Structure **Page Nos.**

8.0 Introduction 20
8.1 Objectives 20
8.2 Definitions 20
8.3 Shortest Path Algorithms 23
8.3.1 Dijkstra's Algorithm
8.3.2 Graphs with Negative Edge costs
8.3.3 Acyclic Graphs
8.3.4 All Pairs Shortest Paths Algorithm
8.4 Minimum cost Spanning Trees 30
8.4.1 Kruskal's Algorithm
8.4.2 Prims's Algorithm
8.4.3 Applications
8.5 Breadth First Search 34
8.6 Depth First Search 34
8.7 Finding Strongly Connected Components 36
8.8 Summary 38
8.9 Solutions/Answers 39
8.10 Further Readings 39

8.0 INTRODUCTION

In this unit, we will discuss a data structure called Graph. In fact, graph is a general tree with no parent-child relationship. Graphs have many applications in computer science and other fields of science. In general, graphs represent a relatively less restrictive relationship between the data items. We shall discuss about both undirected graphs and directed graphs. The unit also includes information on different algorithms which are based on graphs.

8.1 OBJECTIVES

After going through this unit, you should be able to

- know about graphs and related terminologies;
- know about directed and undirected graphs along with their representations;
- know different shortest path algorithms;
- construct minimum cost spanning trees;
- apply depth first search and breadth first search algorithms, and
- finding strongly connected components of a graph.

8.2 DEFINITIONS

A graph **G** may be defined as a finite set **V** of vertices and a set **E** of edges (pair of connected vertices). The notation used is as follows:

Graph G = (V, E)
Consider the graph of *Figure 8.1.*

The set of vertices for the graph is V = {1, 2, 3, 4, 5}.
The set of edges for the graph is E = {(1,2), (1,5), (1,3), (5,4), (4,3), (2,3) }.

The elements of E are always a pair of elements.

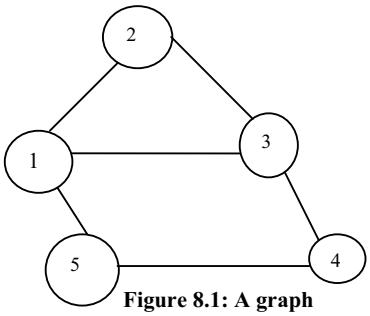


Figure 8.1: A graph

It may be noted that unlike nodes of a tree, graph has a very limited relationship between the nodes (vertices). There is no direct relationship between the vertices 1 and 4 although they are connected through 3.

Directed graph and Undirected graph: If every edge (a,b) in a graph is marked by a direction from a to b, then we call it a Directed graph (digraph). On the other hand, if directions are not marked on the edges, then the graph is called an Undirected graph.

In a Directed graph, the edges (1,5) and (5,1) represent two different edges whereas in an Undirected graph, (1,5) and (5,1) represent the same edge. Graphs are used in various types of modeling. For example, graphs can be used to represent connecting roads between cities.

Graph terminologies :

Adjacent vertices: Two vertices a and b are said to be adjacent if there is an edge connecting a and b. For example, in *Figure 8.1,* vertices 5 and 4 are adjacent.

Path: A path is defined as a sequence of distinct vertices, in which each vertex is adjacent to the next. For example, the path from 1 to 4 can be defined as a sequence of adjacent vertices (1,5), (5,4).

A *path*, **p**, of length, **k**, through a graph is a sequence of connected vertices:

$$\mathbf{p = <v_0,v_1,...,v_k>}$$

Cycle : A graph contains *cycles* if there is a path of non-zero length through the graph, $p = <v_0,v_1,...,v_k>$ such that $v_0 = v_k$.

Edge weight : It is the cost associated with edge.

Loop: It is an edge of the form (*v*,*v*).

Path length : It is the number of edges on the path.

Simple path : It is the set of all distinct vertices on a path (except possibly first and last).

Spanning Trees: A *spanning tree* of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.

There are different representations of a graph. They are:

- Adjacency list representation
- Adjacency matrix representation

Adjacency list representation

An Adjacency list representation of a Graph G = {V, E} consists of an array of adjacency lists denoted by *adj of V* list. For each vertex uϵV, adj[u] consists of all vertices adjacent to u in the graph G.

Consider the graph of *Figure 8.2.*

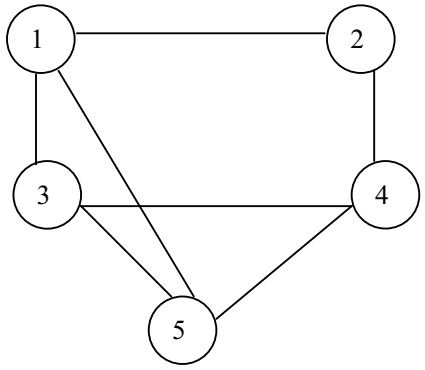


Figure 8.2: A Graph

The following is the adjacency list representation of graph of *Figure 8.2:*

adj [1] = {2, 3, 5}
adj [2] = {1, 4}
adj [3] = {1, 4, 5}
adj [4] = {2, 3, 5}
adj [5] = {1, 3, 4}

An adjacency matrix representation of a Graph G=(V, E) is a matrix A(a_{ij}) such that

a_{ij} = { 1 if edge (i, j) belongs to E
0 otherwise

The adjacency matrix for the graph of *Figure 8.2* is given below:

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	1
2	1	0	0	1	1
3	1	0	0	1	1
4	0	1	1	0	1
5	1	0	1	1	0

Observe that the matrix is symmetric along the main diagonal. If we define the adjacency matrix as A and the transpose as A^T , then for an undirected graph G as above, $A = A^T$.

Graph connectivity :

A connected graph is a graph in which path exists between every pair of vertices.

A strongly connected graph is a directed graph in which every pair of distinct vertices are connected with each other.

A weakly connected graph is a directed graph whose underlying graph is connected, but not strongly connected.

A complete graph is a graph in which there exists edge between every pair of vertices.

☞ Check Your Progress 1

1) A graph with no cycle is called _____ graph.
2) Adjacency matrix of an undirected graph is _____ on main diagonal.
3) Represent the following graphs(*Figure 8.3* and *Figure 8.4*) by adjacency matrix:

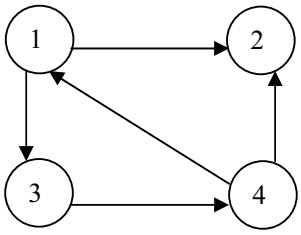


Figure 8.3: A Directed Graph

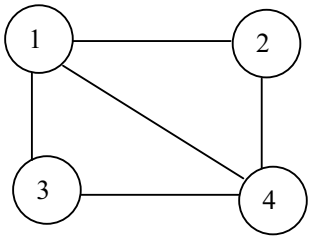


Figure 8.4: A Graph

8.3 SHORTEST PATH ALGORITHMS

A driver takes shortest possible route to reach destination. The problem that we will discuss here is similar to this kind of finding shortest route in a graph. The graphs are weighted directed graphs. The weight could be time, cost, losses other than distance designated by numerical values.

Single source shortest path problem : To find a shortest path from a single source to every vertex of the Graph.

Consider a Graph G = (V, E). We wish to find out the shortest path from a single source vertex sϵV, to every vertex vϵV. The single source shortest path algorithm (Dijkstra's Algorithm) is based on assumption that no edges have negative weights.

The procedure followed to find shortest path are based on a concept called relaxation. This method repeatedly decreases the upper bound of actual shortest path of each vertex from the source till it equals the shortest-path weight. Please note that shortest path between two vertices contains other shortest path within it.

8.3.1 Dijkstra's Algorithm

Djikstra's algorithm (named after its discover, Dutch computer scientist E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination with non-negative weight edge.

It turns out that one can find the shortest paths from a given source to *all* vertices (points) in a graph in the same time. Hence, this problem is sometimes called the *single-source shortest paths* problem. Dijkstra's algorithm is a greedy algorithm, which finds shortest path between all pairs of vertices in the graph. Before describing the algorithms formally, let us study the method through an example.

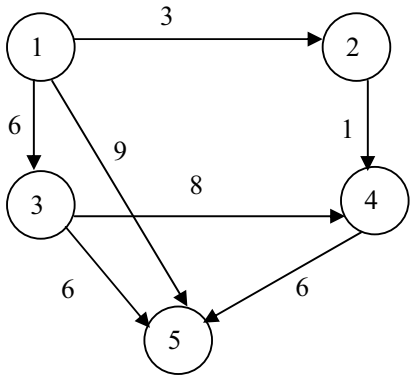


Figure 8.5: A Directed Graph with no negative edge(s)

Dijkstra's algorithm keeps two sets of vertices:

S is the set of vertices whose shortest paths from the source have already been determined

Q = V-S is the set of remaining vertices .

The other data structures needed are:

d array of best estimates of shortest path to each vertex from the source

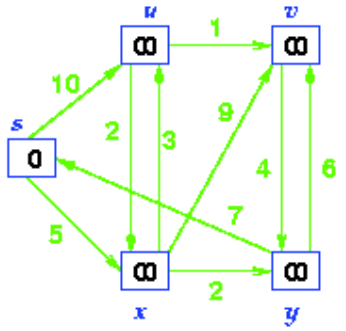
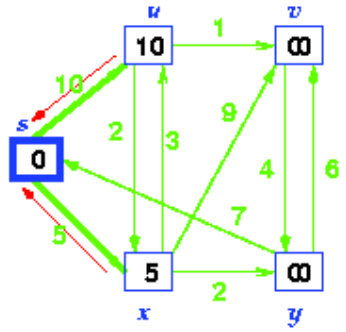
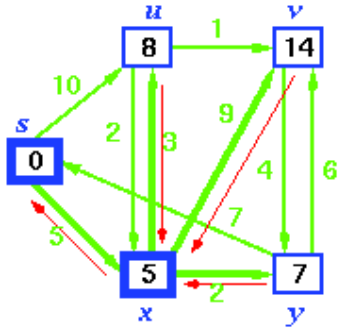
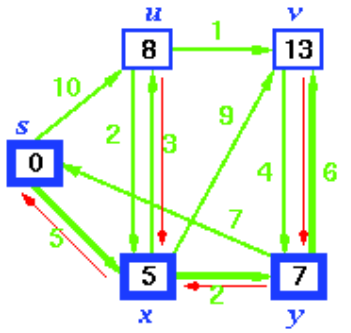
pi an array of predecessors for each vertex. *predecessor* is an array of vertices to which shortest path has already been determined.

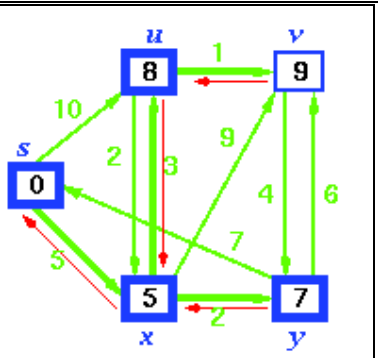
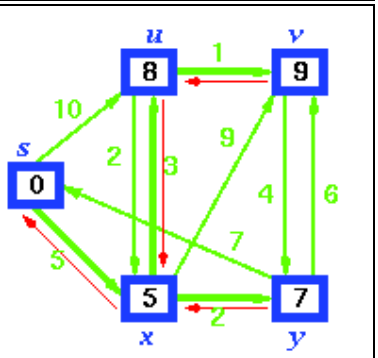
The basic operation of Dijkstra's algorithm is edge relaxation. If there is an edge from u to v, then the shortest known path from s to u can be extended to a path from s to v by adding edge (u,v) at the end. This path will have length d[u]+w(u,v). If this is less than d[v], we can replace the current value of d[v] with the new value.

The predecessor list is an array of indices, one for each vertex of a graph. Each vertex entry contains the index of its predecessor in a path through the graph.

Operation of Algorithm

The following sequence of diagrams illustrate the operation of Dijkstra's Algorithm. The bold vertices indicate the vertex to which shortest path has been determined.

	Initialize the graph, all the vertices have infinite costs except the source vertex which has zero cost
	From all the adjacent vertices, choose the closest vertex to the source **s**. As we initialized d[s] to 0, it's **s**. (shown in bold circle) Add it to **S** Relax all vertices adjacent to **s**, i.e u and x Update vertices u and x by 10 and 5 as the distance from s.
	Choose the nearest vertex, **x**. Relax all vertices adjacent to **x** Update predecessors for **u, v** and **y**. Predecessor of x = s Predecessor of v = x ,s Predecessor of y = x ,s add x to S
	Now **y** is the closest vertex. Add it to **S**. Relax **v** and adjust its predecessor.

	**u** is now closest, add it to S and adjust its adjacent vertex, **v**.
	Finally, add **v to S**. The predecessor list now defines the shortest path from each node to **s**.

Dijkstra's algorithm

```
 * Initialise d and pi*
for each vertex v in V( g )
      g.d[v] := infinity
      g.pi[v] := nil
      g.d[s] := 0;
* Set S to empty *
S := { 0 }
Q := V(g)
* While (V-S) is not null*
while not Empty(Q)
```

1. Sort the vertices in **V-S** according to the current best estimate of their distance from the source
   u := Extract-Min (Q);

2. Add vertex **u**, the closest vertex in **V-S**, to **S**,
   AddNode(S, u);

3. Relax all the vertices still in **V-S** connected to **u**
   relax(Node u, Node v, double w[][])
   if d[v] > d[u] + w[u]v] then
   d[v] := d[u] + w[u][v]
   pi[v] := u

In summary, this algorithm starts by assigning a weight of infinity to all vertices, and then selecting a source and assigning a weight of zero to it. Vertices are added to the set for which shortest paths are known. When a vertex is selected, the weights of its adjacent vertices are relaxed. Once all vertices are relaxed, their predecessor's vertices

are updated (pi). The cycle of selection, weight relaxation and predecessor update is repeated until the shortest path to all vertices has been found.

Complexity of Algorithm

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is $\Theta(n^2)$.

8.3.2 Graphs with Negative Edge costs

We have seen that the above Dijkstra's single source shortest-path algorithm works for graphs with non-negative edges (like road networks). The following two scenarios can emerge out of negative cost edges in a graph:

- Negative edge with non- negative weight cycle reachable from the source.
- Negative edge with non-negative weight cycle reachable from source.

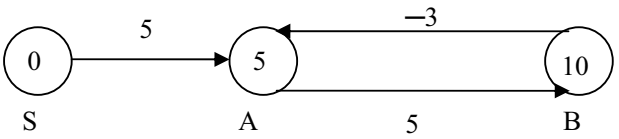


Figure 8.6 : A Graph with negative edge and non-negative weight cycle

The net weight of the cycle is 2(non-negative)(refer to *Figure 8.6*).

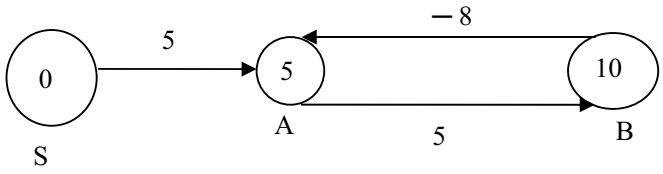


Figure 8.7: A graph with negative edge and negative weight cycle

The net weight of the cycle is −3(negative) (refer to *Figure 8.7*). The shortest path from A to B is not well defined as the shortest path to this vertex are infinite, i.e., by traveling each cycle we can decrease the cost of the shortest path by 3, like (S, A, B) is path (S, A, B, A, B) is a path with less cost and so on.

Dijkstra's Algorithm works only for directed graphs with non-negative weights (cost).

8.3.3 Acyclic Graphs

A path in a directed graph is said to form a cycle is there exists a path (A,B,C,…..P) such that A = P. A graph is called acyclic if there is no cycle in the graph.

8.3.4 All Pairs Shortest Paths Algorithm

In the last section, we discussed about shortest path algorithm which starts with a single source and finds shortest path to all vertices in the graph. In this section, we shall discuss the problem of finding shortest path between all pairs of vertices in a graph. This problem is helpful in finding distance between all pairs of cities in a road atlas. All pairs shortest paths problem is mother of all shortest paths problems.

In this algorithm, we will represent the graph by adjacency matrix.

The weight of an edge C_{ij} in an adjacency matrix representation of a directed graph is represented as follows

$$C_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of the directed edge from i to j i.e (i,j)} & \text{if } i \neq j \text{ and (i j) belongs to E} \\ \infty & \text{if } i \neq j \text{ and (i, j) does not belong to E} \end{cases}$$

Given a directed graph G = (V, E), where each edge (v, w) has a non-negative cost C(v , w), for all pairs of vertices (v, w) to find the lowest cost path from v to w.

The All pairs shortest paths problem can be considered as a generalisation of single-source-shortest-path problem, by using Dijkstra's algorithm by varying the source node among all the nodes in the graph. If negative edge(s) is allowed, then we can't use Dijkstra's algorithm.

In this section we shall use a recursive solution to all pair shortest paths problem known as Floyd-Warshall algorithm, which runs in $O(n^3)$ time.

This algorithm is based on the following principle. For graph G let V = {1, 2, 3,…,n}.Let us consider a sub set of the vertices {1, 2, 3, …..,k. For any pair of vertices that belong to V, consider all paths from i to j whose intermediate vertices are from {1, 2, 3, ….k}. This algorithm will exploit the relationship between path p and shortest path from i to j whose intermediate vertices are from {1, 2, 3, ….k-1} with the following two possibilities:

1. If k is not an intermediate vertex in the path p, then all the intermediate vertices of the path p are in {1, 2, 3, ….,k-1}. Thus, shortest path from i to j with intermediate vertices in {1, 2, 3, ….,k-1} is also the shortest path from i to j with vertices in {1, 2, 3, …, k}.

2. If k is an intermediate vertex of the path p, we break down the path p into path p1 from vertex i to k and path p2 from vertex k to j. So, path p1 is the shortest path from i to k with intermediate vertices in {1, 2, 3, ….,k-1}.

During iteration process we find the shortest path from i to j using only vertices (1, 2, 3, …, k-1} and in the next step, we find the cost of using the k^{th} vertex as an intermediate step. If this results in lower cost, then we store it.

After n iterations (all possible iterations), we find the lowest cost path from i to j using all vertices (if necessary).

Note the following:

Initialize the matrix

C[i][j] = ∞ if (i, j) does not belong to E for graph G = (V, E)

Initially, D[i][j] = C[i][j]

We also define a path matrix P where P[i][j] holds intermediate vertex k on the least cost path from i to j that leads to the shortest path from i to j .

Algorithm (All Pairs Shortest Paths)

N = number of rows of the graph
D[i[j] = C[i][j]
For k from 1 to n
  Do for i = 1 to n
    Do for j = 1 to n
      D[i[j]= minimum($d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$))
    Enddo
  Enddo
Enddo

where $d_{ij}^{(k-1)}$ = minimum path from i to j using k-1 intermediate vertices
where $d_{ik}^{(k-1)}$ = minimum path from j to k using k-1 intermediate vertices
where $d_{kj}^{(k-1)}$ = minimum path from k to j using k-1 intermediate vertices

Program 8.1 gives the program segment for the All pairs shortest paths algorithm.

```
AllPairsShortestPaths(int N, Matrix C, Matrix P, Matrix D)
{
        int i, j, k

        if i = j then C[i][j] = 0
        for ( i = 0; i < N; i++)
        {
                for (j = 0; j < N; j++)
                {
                  D[i][j] = C[i][j];
                P[i][j] = -1;
                }
          D[i][j] = 0;
        }

        for (k=0; k<N; k++)
        {
                for (i=0; i<N; i++)
                {
                        for (j=0; J<N; J++)
                        {
                          if (D[i][k] + D[k][j]  < D[i][j])
                          {
                            D[i][j] = D[i][k] + D[k][j];
                            P[i][j] = k;
                          }
                        }
        }
         }
}

/********** End *************/
```

Program 8.1 : Program segment for All pairs shortest paths algorithm

From the above algorithm, it is evident that it has $O(N^3)$ time complexity.

Shortest path algorithms had numerous applications in the areas of Operations Research, Computer Science, Electrical Engineering and other related areas.

☞ Check Your Progress 2

1) _____ is the basis of Dijkstra's algorithm

2) What is the complexity of All pairs shortest paths algorithm?

……..

8.4 MINIMUM COST SPANNING TREES

A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree (with no cycle). A graph may have many spanning trees.

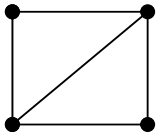


Figure 8.8: A Graph

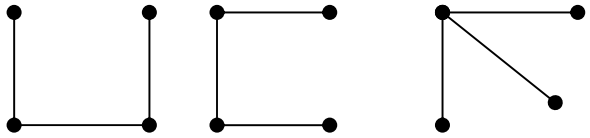


Figure 8.9 : Spanning trees of the Graph of Figure 8.9

Consider the graph of *Figure 8.8*. It's spanning trees are shown in *Figure 8.9*.
Now, if the graph is a weighted graph (length associated with each edge). The weight of the tree is just the sum of weights of its edges. Obviously, different spanning trees have different weights or lengths. Our objective is to find the minimum length (weight) spanning tree.

Suppose, we have a group of islands that we wish to link with bridges so that it is possible to travel from one island to any other in the group. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the minimum cost spanning tree.

8.4.1 Kruskal's Algorithm

Krushkal's algorithm uses the concept of *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it links two trees together. If it forms a cycle, it would simply mean that it links two nodes that were already connected. So, we reject it.

The steps in Kruskal's Algorithm are as follows:

1. The forest is constructed from the graph G - with each node as a separate tree in the forest.
2. The edges are placed in a priority queue.
3. Do until we have added **n**-1 edges to the graph,
    1. Extract the cheapest edge from the queue.
    2. If it forms a cycle, then a link already exists between the concerned nodes. Hence reject it.
    3. Else add it to the forest. Adding it to the forest will join two trees together.

The forest of trees is a partition of the original set of nodes. Initially all the trees have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

Let us see the sequence of operations to find the Minimum Cost Spanning Tree(MST) in a graph using Kruskal's algorithm. Consider the graph of *Figure 8.10., Figure 8.11* shows the construction of MST of graph of *Figure 8.10.*

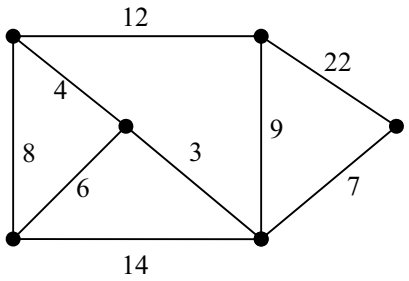
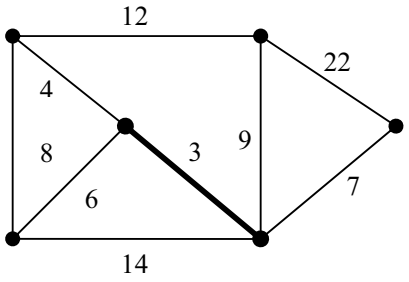


Figure 8.10 : A Graph



Step 1

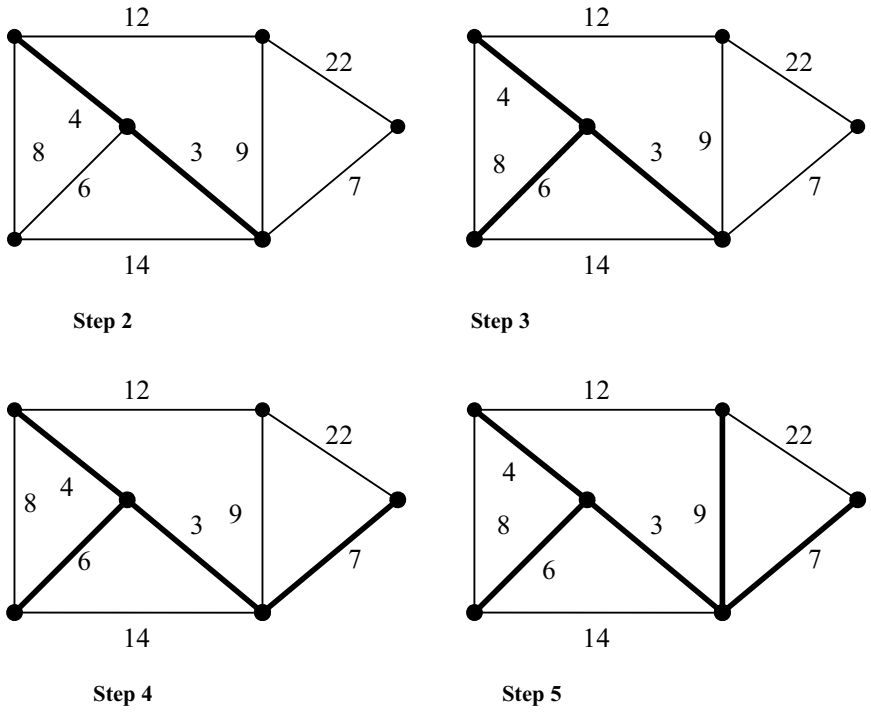


Figure 8.11 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Kruskal's algorithm

The following are various steps in the construction of MST for the graph of *Figure 8.10* using Kruskal's algorithm.

Step 1 : The lowest cost edge is selected from the graph which is not in MST (initially MST is empty). The lowest cost edge is 3 which is added to the MST (shown in bold edges)

Step 2: The next lowest cost edge which is not in MST is added (edge with cost 4).

Step 3 : The next lowest cost edge which is not in MST is added (edge with cost 6).

Step 4 : The next lowest cost edge which is not in MST is added (edge with cost 7).

Step 5 : The next lowest cost edge which is not in MST is 8 but will form a cycle. So, it is discarded . The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

8.4.2 Prim's Algorithm

Prim's algorithm uses the concept of sets. Instead of processing the graph by sorted order of edges, this algorithm processes the edges in the graph randomly by building up disjoint sets.

It uses two disjoint sets A and $\bar{A}$. Prim's algorithm works by iterating through the nodes and then finding the shortest edge from the set A to that of set A (i.e. out side A), followed by the addition of the node to the new graph. When all the nodes are processed, we have a minimum cost spanning tree.

Rather than building a sub-graph by adding one edge at a time, Prim's algorithm builds a tree one vertex at a time.

The steps in Prim's algorithm are as follows:

Let G be the graph with n vertices for which minimum cost spanning tree is to be generated.

Let T be the minimum spanning tree.
Let T be a single vertex x.
while (T has fewer than n vertices)
{
  find the smallest edge connecting T to G-T
  add it to T
}

Consider the graph of *Figure 8.10. Figure 8.12* shows the various steps involved in the construction of Minimum Cost Spanning Tree of graph of *Figure 8.10.*

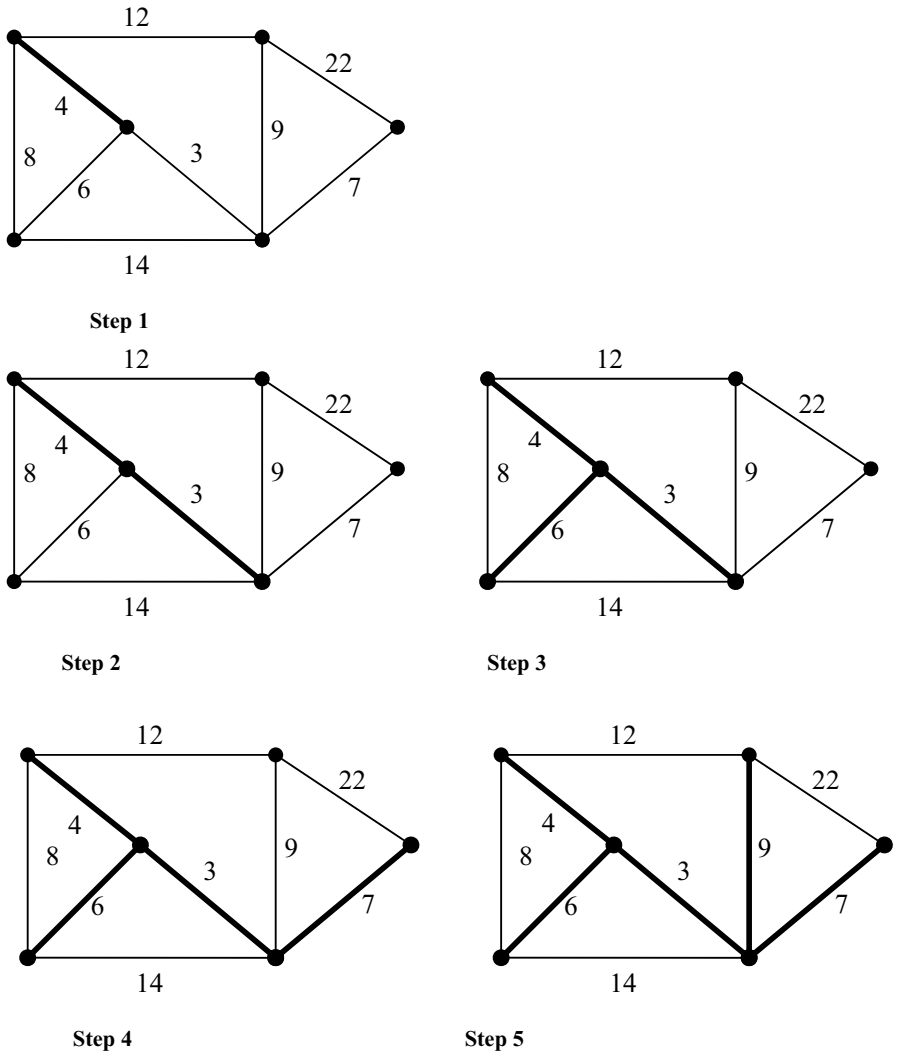


Figure 8.12 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Prim's algorithm

The following are various steps in the construction of MST for the graph of *Figure 8.10* using Prim's algorithm.

Step 1 : We start with a single vertex (node). Now the set A contains this single node and set A contains rest of the nodes. Add the edge with the lowest cost from A to A. The edge with cost 4 is added.

Step 2: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 3) is selected and added to MST.

Step 3: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 6) is selected and added to MST.

Step 4: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 73) is selected and added to MST.

Step 5: The next lowest cost edge to the set not in MST is 8 but forms a cycle. So, it is discarded. The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

Comparison of Kruskal's algorithm and Prim's algorithm

	Kruskal's algorithm	**Prim's algorithm**
Principle	Based on generic minimum cost spanning tree algorithms	A special case of generic minimum cost spanning tree algorithm. Operates like Dijkstra's algorithm for finding shortest path in a graph.
Operation	Operates on a single set of edges in the graph	Operates on two disjoint sets of edges in the graph
Running time	O(E log E) where E is the number of edges in the graph	O(E log V), which is asymptotically same as Kruskal's algorithm

For the above comparison, it may be observed that for dense graphs having more number of edges for a given number of vertices, Prim's algorithm is more efficient.

8.4.3 Applications

The minimum cost spanning tree has wide applications in different fields. It represents many complicated real world problems like:

1. Minimum distance for travelling all cities at most one (travelling salesman problem).
2. In electronic circuit design, to connect n pins by using n-1 wires, using least wire.
3. Spanning tree also finds their application in obtaining independent set of circuit equations for an electrical network.

8.5 BREADTH FIRST SEARCH (BFS)

When BFS is applied, the vertices of the graph are divided into two categories. The vertices, which are visited as part of the search and those vertices, which are not visited as part of the search. The strategy adopted in breadth first search is to start search at a vertex(source). Once you started at source, the number of vertices that are visited as part of the search is 1 and all the remaining vertices need to be visited. Then, search the vertices which are adjacent to the visited vertex from left to order. In this way, all the vertices of the graph are searched.

Consider the digraph of *Figure 8.13*. Suppose that the search started from S. Now, the vertices (from left to right) adjacent to S which are not visited as part of the search are B, C, A. Hence, B,C and A are visited after S as part of the BFS. Then, F is the unvisited vertex adjacent to B. Hence, the visit to B, C and A is followed by F. The unvisited vertex adjacent of C is D. So, the visit to F is followed by D. There are no

unvisited vertices adjacent to A. Finally, the unvisited vertex E adjacent to D is visited.

Hence, the sequence of vertices visited as part of BFS is S, B, C, A, F, D and E.

8.6 DEPTH FIRST SEARCH (DFS)

The strategy adopted in depth first search is to search deeper whenever possible. This algorithm repeatedly searches deeper by visiting unvisited vertices and whenever an unvisited vertex is not found, it backtracks to previous vertex to find out whether there are still unvisited vertices.

As seen, the search defined above is inherently recursive. We can find a very simple recursive procedure to visit the vertices in a depth first search. The DFS is more or less similar to pre-order tree traversal. The process can be described as below:

Start from any vertex (source) in the graph and mark it visited. Find vertex that is adjacent to the source and not previously visited using adjacency matrix and mark it visited. Repeat this process for all vertices that is not visited, if a vertex is found visited in this process, then return to the previous step and start the same procedure from there.

If returning back to source is not possible, then DFS from the originally selected source is complete and start DFS using any unvisited vertex.

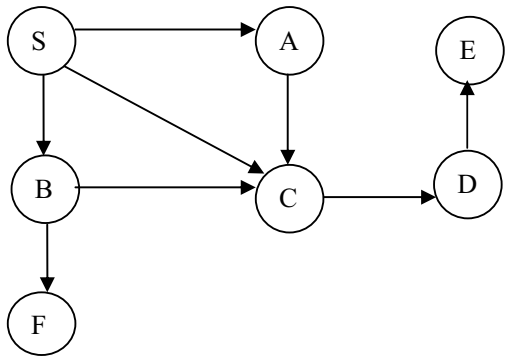


Figure 8.13 : A Digraph

Consider the digraph of *Figure 8.13.* Start with S and mark it visited. Then visit the next vertex A, then C and then D and at last E. Now there are no adjacent vertices of E to be visited next. So, now, backtrack to previous vertex D as it also has no unvisited vertex. Now backtrack to C, then A, at last to S. Now S has an unvisited vertex B. Start DFS with B as a root node and then visit F. Now all the nodes of the graph are visited.

Figure 8.14 shows a DFS tree with a sequence of visits. The first number indicates the time at which the vertex is visited first and the second number indicates the time at which the vertex is visited during back tracking.

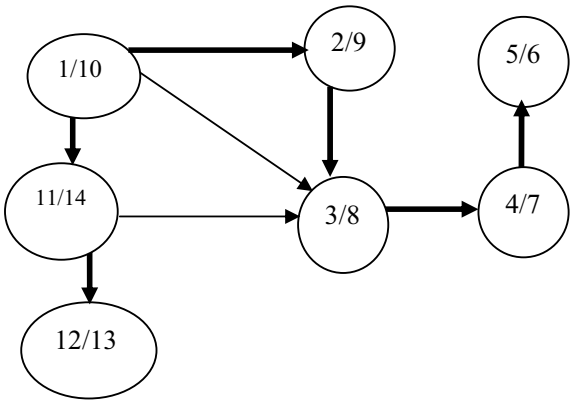


Figure 8.14 : DFS tree of digraph of Figure 8.13

The DFS forest is shown with shaded arrow in *Figure 8.14*.

Algorithm for DFS

Step 1: Select a vertex in the graph and make it the source vertex and mark it visited.

Step 2: Find a vertex that is adjacent to the souce vertex and start a new search if it is not already visited.

Step 3: Repeat step 2 using a new source vertex. When all adjacent vertices are visited, return to previous source vertex and continue search from there.

If n is the number of vertices in the graph and the graph is represented by an adjacency matrix, then the total time taken to perform DFS is $O(n^2)$. If G is represented by an adjacency list and the number of edges of G are e, then the time taken to perform DFS is O(e).

8.7 FINDING STRONGLY CONNECTED COMPONENTS

A beautiful application of DFS is finding a strongly connected component of a graph.

Definition: For graph G = (V, E) , where V is the set of vertices and E is the set of edges, we define a strongly connected components as follows:

U is a sub set of V such that u, v belongs to U such that, there is a path from u to v and v to u. That is, all pairs of vertices are reachable from each other.

In this section we will use another concept called transpose of a graph. Given a directed graph G a transpose of G is defined as G^T. G^T is defined as a graph with the same number of vertices and edges with only the direction of the edges being reversed. G^T is obtained by transposing the adjacency matrix of the directed graph G.

The algorithm for finding these strongly connected components uses the transpose of G, G^T.

G = (V, E), G^T = (V, E^T), where E^T = { (u, v): (v, u) belongs to E }

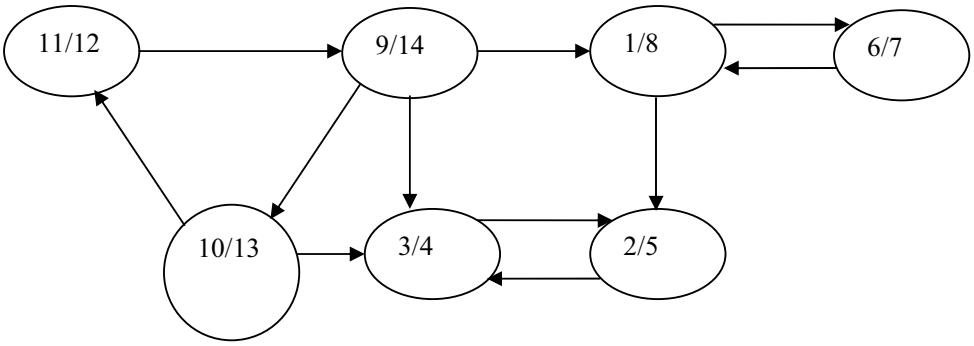


Figure 8.15: A Digraph

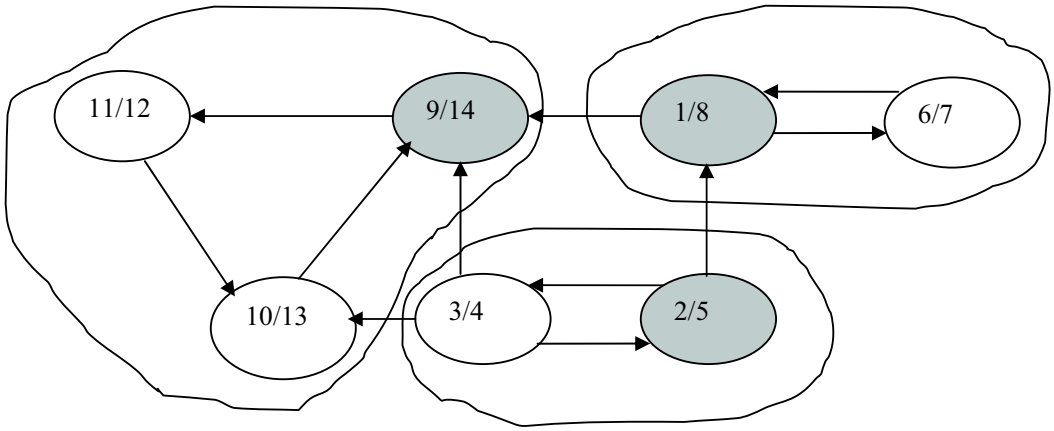


Figure 8.16: Transpose and strongly connected components of digraph of Figure 8.15

Figure 8.15 shows a directed graph with sequence in DFS (first number of the vertex shows the discovery time and second number shows the finishing time of the vertex during DFS. *Figure 8.16* shows the transpose of the graph in *Figure 8.15* whose edges are reversed. The strongly connected components are shown in zig-zag circle in *Figure 8.16*.

To find strongly connected component we start with a vertex with the highest finishing time and start DFS in the graph G^T and then in decreasing order of finishing time. DFS with vertex with finishing time 14 as root finds a strongly connected component. Similarly, vertices with finishing times 8 and then 5, when selected as source vertices also lead to strongly connected components.

Algorithm for finding strongly connected components of a Graph:

Strongly Connected Components (G)

where d[u] = discovery time of the vertex u during DFS , f[u] = finishing time of a vertex u during DFS, G^T = Transpose of the adjacency matrix

Step 1: Use DFS(G) to compute f[u] $\forall u \in V$
Step 2: Compute G^T
Step 3: Execute DFS in G^T

Step 4: Output the vertices of each tree in the depth-first forest of Step 3 as a separate strongly connected component.

☞ Check Your Progress 3

1) Which graph traversal uses a queue to hold vertices that are to be processed next ?

……

……..

2) Which graph traversal is recursive by nature?

……..

……

3) For a dense graph, Prim's algorithm is faster than Kruskal's algorithm
True/False

4) Which graph traversal technique is used to find strongly connected component of a graph?

……

……

8.8 SUMMARY

Graphs are data structures that consist of a set of vertices and a set of edges that connect the vertices. A graph where the edges are directed is called directed graph. Otherwise, it is called an undirected graph. Graphs are represented by adjacency lists and adjacency matrices. Graphs can be used to represent a road network where the edges are weighted as the distance between the cities. Finding the minimum distance between single source and all other vertices is called single source shortest path problem. Dijkstra's algorithm is used to find shortest path from a single source to every other vertex in a directed graph. Finding shortest path between every pair of vertices is called all pairs shortest paths problem.

A spanning tree of a graph is a tree consisting of only those edges of the graph that connects all vertices of the graph with minimum cost. Kruskal's and Prim's algorithms find minimum cost spanning tree in a graph. Visiting all nodes in a graph systematically in some manner is called traversal. Two most common methods are depth-first and breadth-first searches.

8.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) an acyclic
2) symmetric
3) The adjacency matrix of the directed graph and undirected graph are as follows:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

(Refer to Figure 8.3)

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

(Refer to Figure 8.3)

Check Your Progress 2

1) Node relaxation
2) $O(N^3)$

Check Your Progress 3

1) BFS
2) DFS
3) True
4) DFS

8.10 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.
3. *Data Structures and Algorithms* by Alfred V.Aho; Addison Wesley.

Reference Websites

http://www.onesmartclick.com/engineering/data-structure.html
http://msdn.microsoft.com/vcsharp/programming/datastructures/
http://en.wikipedia.org/wiki/Graph_theory

UNIT 9 SEARCHING

Structure **Page Nos.**

9.0	Introduction	40
9.1	Objectives	40
9.2	Linear Search	41
9.3	Binary Search	44
9.4	Applications	47
9.5	Summary	48
9.6	Solutions / Answers	48
9.7	Further Readings	48

9.0 INTRODUCTION

Searching is the process of looking for something: Finding one piece of data that has been stored within a whole group of data. It is often the most time-consuming part of many computer programs. There are a variety of methods, or algorithms, used to search for a data item, depending on how much data there is to look through, what kind of data it is, what type of structure the data is stored in, and even where the data is stored - inside computer memory or on some external medium.

Till now, we have studied a variety of data structures, their types, their use and so on. In this unit, we will concentrate on some techniques to ***search*** a particular data or piece of information from a large amount of data. There are basically two types of searching techniques, ***Linear or Sequential Search and Binary Search.***

Searching is very common task in day-to-day life, where we are involved some or other time, in searching either for some needful at home or office or market, or searching a word in dictionary. In this unit, we see that if the things are organised in some manner, then search becomes efficient and fast.

All the above facts apply to our computer programs also. Suppose we have a telephone directory stored in the memory in an array which contains Name and Numbers. Now, what happens if we have to find a number? The answer is search that number in the array according to name (given). If the names were organised in some order, searching would have been fast.

So, basically a search algorithm is an algorithm which accepts an argument 'a' and tries to find the corresponding data where the match of 'a' occurs in a file or in a table.

9.1 OBJECTIVES

After going through this unit, you should be able to:

- know the basic concepts of searching;
- know the process of performing the Linear Search;
- know the process of performing the Binary Search and
- know the applications of searching.

9.2 LINEAR SEARCH

Linear search is not the most efficient way to search for an item in a collection of items. However, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren't many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach.

Before studying Linear Search, let us define some terms related to search.

A ***file*** is a collection of records and a record is in turn a collection of fields. A field, which is used to differentiate among various records, is known as a ***'key'***.

For example, the telephone directory that we discussed in previous section can be considered as a file, where each record contains two fields: name of the person and phone number of the person.

Now, it depends on the application whose field will be the 'key'. It can be the name of person (usual case) and it can also be phone number. We will locate any particular record by matching the input argument 'a' with the key value.

The simplest of all the searching techniques is *Linear or Sequential Search.* As the name suggests, all the records in a file are searched sequentially, one by one, for the matching of key value, until a match occurs.

The Linear Search is applicable to a table which it should be organised in an array. Let us assume that a file contains 'n' records and a record has 'a' fields but only one key. The values of key are organised in an array say 'm'. As the file has 'n' records, the size of array will be 'n' and value at position R(i) will be the key of record at position i. Also, let us assume that *'el'* is the value for which search has to be made or it is the search argument.

Now, let us write a simple algorithm for Linear Search.

Algorithm

Here, m represents the unordered array of elements
n represents number of elements in the array and
el represents the value to be searched in the list

```
Sep 1: [Initialize]
       k=0
       flag=1

Step 2: Repeat step 3 for k=0,1,2…..n-1

Step 3: if (m[k]=el )
        then
                flag=0
               print "Search is successful" and element is found at location (k+1)
               stop
        endif

Step 4: if  (flag=1) then
               print "Search is unsuccessful"
```

endif

Step 5: stop

Program 9.1 gives the program for Linear Search.

```
/*Program for Linear Search*/
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Global Variables*/
int search;
int flag;
/*Function Declarations*/
int input (int *, int, int);
void linear_search (int *, int, int);
void display (int *, int);
/*Functions */
void linear_search(int m[ ], int n, int el)
{
        int k;
        flag = 1;
        for(k=0; k<n; k++)
        {
                if(m[k]==el
                {
                        printf("\n Search is Successful\n");
                        printf("\n Element : %i Found at location : %i", element, k+1);
                        flag = 0;
                }
          }
        if(flag==1)
                printf("\n Search is unsuccessful");
}
void display(int m[ ], int n)
{
         int i;
         for(i=0; i< 20; i++)
         {
                  printf("%d", m[i];
          }
}
int input(int m[ ], int n, int el)
{
         int i;
         n = 20;
         el = 30;
         printf("Number of elements in the list : %d", n);
         for(i=0;i<20;i++)
                {
                  m[i]=rand( )%100;
                }
         printf("\n Element to be searched :%d", el);
         search = el;
         return n;
        }
        /* Main Function*/
```

```
void main( )
{
        int n, el, m[200];
        number = input(m, n,el);
        el = search;
        printf("\n Entered list as follows: \n");
        display(m, n);
        linear_search(m, n, el);
        printf("\n In the following list\n");
        display(m, n);
}
```

Program 9.1: Linear Search

Program 9.1 examines each of the key values in the array 'm', one by one and stops when a match occurs or the total array is searched.

Example:

A *telephone directory* with n = 10 records and Name field as key. Let us assume that the names are stored in array 'm' i.e. m(0) to m(9) and the search has to be made for name "Radha Sharma", i.e. element = "Radha Sharma".

Telephone Directory

Name	***Phone No.***
Nitin Kumar	25161234
Preeti Jain	22752345
Sandeep Singh	23405678
Sapna Chowdhary	22361111
Hitesh Somal	24782202
R.S.Singh	26254444
Radha Sharma	26150880
S.N.Singh	25513653
Arvind Chittora	26252794
Anil Rawat	26257149

The above algorithm will search for element = "Radha Sharma" and will stop at 6th index of array and the required phone number is "26150880", which is stored at position 7 i.e. 6+1.

Efficiency of Linear Search

How many number of comparisons are there in this search in searching for a given element?

The number of comparisons depends upon where the record with the argument key appears in the array. If record is at the first place, number of comparisons is '1', if record is at last position 'n' comparisons are made.

If it is equally likely for that the record can appear at any position in the array, then, a successful search will take (n+1)/2 comparisons and an unsuccessful search will take 'n' comparisons.

In any case, the order of the above algorithm is ***O(n)***.

☞ Check Your Progress 1

1) Linear search uses an exhaustive method of checking each element in the array against a key value. When a match is found, the search halts. Will sorting the array before using the linear search have any effect on its order of efficiency?

……

2) In a best case situation, the element was found with the fewest number of comparisons. Where, in the list, would the key element be located?

……

9.3 BINARY SEARCH

An unsorted array is searched by *linear search* that scans the array elements one by one until the desired element is found.

The reason for sorting an array is that we search the array "quickly". Now, if the array is sorted, we can employ *binary search*, which brilliantly halves the size of the search space each time it examines one array element.

An array-based binary search selects the middle element in the array and compares its value to that of the key value. Because, the array is sorted, if the key value is less than the middle value then the key must be in the first half of the array. Likewise, if the value of the key item is greater than that of the middle value in the array, then it is known that the key lies in the second half of the array. In either case, we can, in effect, "throw out" one half of the search space or array with only one comparison.

Now, knowing that the key must be in one half of the array or the other, the binary search examines the mid value of the half in which the key must reside. The algorithm thus narrows the search area by half at each step until it has either found the key data or the search fails.

As the name suggests, binary means two, so it divides an array into *two* halves for searching. This search is applicable only to an ***ordered table*** (in either ascending or in descending order).

Let us write an algorithm for Binary Search and then we will discuss it. The array consists of elements stored in ascending order.

Algorithm

Step 1: Declare an array 'k' of size 'n' i.e. k(n) is an array which stores all the keys of a file containing 'n' records

Step 2: i←0

Step 3: low←0, high←n-1

Step 4: while (low <= high)do
mid = (low + high)/2
if (key=k[mid]) then
write "record is at position", mid+1 //as the array starts from the 0^{th} position
else
if(key < k[mid]) then
high = mid - 1

```
                    else
                        low = mid + 1
                    endif
                endif
            endwhile
```

Step 5: Write “Sorry, key value not found”

Step 6: Stop

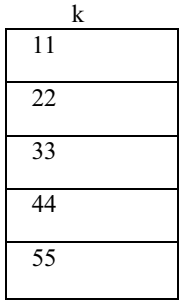
Program 9.2 gives the program for Binary Search.

```
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Functions*/
void binary_search(int array[ ], int value, int size)
{
        int found=0;
        int high=size-1, low=0, mid;
        mid = (high+low)/2;
        printf(“\n\n Looking for %d\n”, value);
        while((!found)&&(high>=low))
        {
            printf(“Low %d Mid%d High%d\n”, low, mid, high);
            if(value==array[mid] )
            {printf(“Key value found at position %d”,mid+1);
             found=1;
            }
            else
            {if (value<array[mid])
                high = mid-1;
            else
                low = mid+1;
            mid = (high+low)/2;
            }
        }
        if (found==1
        printf(“Search successful”);
        else
        printf(“Key value not found”);
}
/*Main Function*/
void main(void)
{
        int array[100], i;
        /*Inputting Values to Array*/
        for(i=0;i<100;i++)
          { printf(“Enter the name:”);
           scanf(“%d”, array[i]);
          }
        printf(“Result of search %d\n”, binary_searchy(array,33,100));
        printf(“Result of search %d\n”, binary_searchy(array, 75,100));
        printf(“Result of search %d\n”, binary_searchy(array,1,100));
}
```

Program 9.2 : Binary Search

Example:

Let us consider a file of 5 records, i.e., n = 5
And k is a sorted array of the keys of those 5 records.



Let key = 55, low = 0, high = 4

Iteration 1: mid = (0+4)/2 = 2
k(mid) = k (2) = 33
Now key > k (mid)
So low = mid + 1 = 3
Iteration 2: low = 3, high = 4 (low <= high)
Mid = 3+4 / 2 = 3.5 ~ 3 (integer value)
Here key > k (mid)
So low = 3+1 = 4
Iteration 3: low = 4, high = 4 (low<= high)
Mid = (4+4)/2 = 4
Here key = k(mid)

So, the record is at mid+1 position, i.e., 5

Efficiency of Binary Search

Each comparison in the binary search reduces the number of possible candidates where the key value can be found by a factor of 2 as the array is divided in two halves in each iteration. Thus, the maximum number of key comparisons are approximately log n. So, the order of binary search is ***O (log n).***

Comparative Study of Linear and Binary Search

Binary search is lots faster than linear search. Here are some comparisons:

NUMBER OF ARRAY ELEMENTS EXAMINED

array size	linear search (avg. case)	binary search (worst case)
8	4	4
128	64	8
256	128	9
1000	500	11
100,000	50,000	18

A ***binary search*** on an array is $O(\log_2 n)$ because at each test, you can "throw out" one half of the search space or array whereas a ***linear search*** on an array is $O(n)$.

It is noteworthy that, for very small arrays a ***linear search*** can prove faster than a ***binary search***. However, as the size of the array to be searched increases, the binary

search is the clear winner in terms of number of comparisons and therefore overall speed.

Still, the binary search has some drawbacks. First, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched, it can throw off the entire process. When presented with a set of unsorted data, the efficient programmer must decide whether to sort the data and apply a binary search or simply apply the less-efficient linear search. Is the cost of sorting the data is worth the increase in search speed gained with the binary search? If you are searching only once, then it is probably to better do a linear search in most cases.

☞ Check Your Progress 2

1) State True or False
   a. The order of linear search in worst case is O (n/2) True/False
   b. Linear search is more efficient than Binary search. True/False
   c. For Binary search, the array has to be sorted in ascending order only. True/False
2) Write the Binary search algorithm where the array is sorted in descending order.

9.4 APPLICATIONS

The searching techniques are applicable to a number of places in today's world, may it be Internet, search engines, on line enquiry, text pattern matching, finding a record from database, etc.

The most important application of searching is to track a particular record from a large file, efficiently and faster.

Let us discuss some of the *applications of Searching* in the world of computers.

1. Spell Checker

This application is generally used in ***Word Processors***. It is based on a program for checking spelling, which it checks and searches sequentially. That is, it uses the concept of *Linear Search*. The program looks up a word in a list of words from a dictionary. Any word that is found in the list is assumed to be spelled correctly. Any word that isn't found is assumed to be spelled wrong.

2. Search Engines

Search engines use software robots to survey the Web and build their databases. Web documents are retrieved and indexed using keywords. When you enter a query at a search engine website, your input is checked against the search engine's keyword indices. The best matches are then returned to you as hits. For checking, it uses any of the Search algorithms.

Search Engines use software programs known as robots, spiders or crawlers. A robot is a piece of software that automatically follows hyperlinks from one document to the next around the Web. When a robot discovers a new site, it sends information back to its main site to be indexed. Because Web documents are one of the least static forms of publishing (i.e., they change a lot), robots also update previously catalogued sites. How quickly and comprehensively they carry out these tasks vary from one search engine to the next.

3. String Pattern matching

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit, search and transport documents over the Internet, and to display documents on printers and computer screens. Web 'surfing' and Web searching are becoming significant and important computer applications, and many of the key computations in all of this document processing involves character strings and string pattern matching. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing. This is accomplished using ***trie*** data structure, which is a tree-based structure that allows for faster searching in a collection of strings.

9.5 SUMMARY

Searching is the process of looking for something. Searching a list consisting of 100000 elements is not the same as searching a list consisting of 10 elements. We discussed two searching techniques in this unit namely Linear Search and Binary Search. Linear Search will directly search for the key value in the given list. Binary search will directly search for the key value in the given sorted list. So, the major difference is the way the given list is presented. Binary search is efficient in most of the cases. Though, it had the overhead that the list should be sorted before search can start, it is very well compensated through the time (which is very less when compared to linear search) it takes to search. There are a large number of applications of Searching out of whom a few were discussed in this unit.

9.6 SOLUTIONS / ANSWERS

Check Your Progress 1

1) No
2) It will be located at the beginning of the list

Check Your Progress 2

1) (a) F
   (b) F
   (c) F

9.7 FURTHER READINGS

Reference Books

1. *Fundamentals of Data Structures in C++* by E. Horowitz, Sahai and D. Mehta, Galgotia Publications.

2. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J. Augenstein and Aaron M. Tanenbaum, PHI Publications.

3. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

Reference Websites

http:// www.cs.umbc.edu
http://www.fredosaurus.com

UNIT 10 SORTING

Structure **Page Nos.**

10.0 Introduction 5
10.1 Objectives 5
10.2 Internal Sorting 6
10.2.1 Insertion Sort
10.2.2 Bubble Sort
10.2.3 Quick Sort
10.2.4 2-way Merge Sort
10.2.5 Heap Sort
10.3 Sorting on Several Keys 13
10.4 Summary 13
10.5 Solutions/Answers 14
10.6 Further Readings 14

10.0 INTRODUCTION

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods. Sorting algorithms can be characterised in the following two ways:

1. Simple algorithms which require the order of n^2 (written as $O(n^2)$)comparisons to sort n items.
2. Sophisticated algorithms that require the $O(n\log_2 n)$ comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order already present in the data.

There are two basic categories of sorting methods: **Internal Sorting and External Sorting.** Internal sorting is applied when the entire collection of data to be sorted is small enough so that the sorting can take place within the main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices. Read and write access times are a major concern in determining sorting performances of such methods.

In this unit, we will study some methods of internal sorting. The next unit will discuss methods of external sorting.

10.1 OBJECTIVES

After going through this unit, you should be able to:

- list the names of some sorting methods;
- discuss the performance of several sorting methods, and
- describe sorting methods on several keys.

10.2 INTERNAL SORTING

In internal sorting, all the data to be sorted is available in the high speed main memory of the computer. We will study the following methods of internal sorting:

1. Insertion sort
2. Bubble sort
3. Quick sort
4. Two-way Merge sort
5. Heap sort

10.2.1 Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example (refer to *Figure 10.1*) before presenting the formal algorithm.

Example : Sort the following list using the insertion sort method:

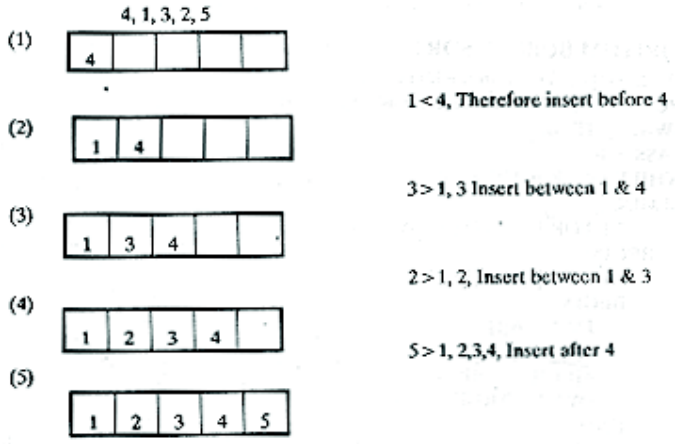


Figure 10.1: Insertion sort

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

10.2.2 Bubble Sort

In this sorting algorithm, multiple swappings take place in one pass. Smaller elements move or 'bubble' up to the top of the list, hence the name given to the algorithm.

In this method, adjacent members of the list to be sorted are compared.If the item on top is greater than the item immediately below it, then they are swapped. This process is carried on till the list is sorted.

The detailed algorithm follows:

Algorithm: BUBBLE SORT

1. Begin

```
2.    Read the n elements
3.    for i=1 to n
      for j=n downto i+1
          if a[j] <=  a[j-1]
                swap(a[j],a[j-1])
4.    End // of Bubble Sort
```

Total number of comparisons in Bubble sort :

$= (N-1) +(N-2) \ldots + 2 + 1$

$= (N-1)*N / 2 = O(N^2)$

This inefficiency is due to the fact that an item moves only to the next position in each pass.

10.2.3 Quick Sort

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases. The basis of quick sort is the *divide and conquer* strategy i.e. Divide the problem [list to be sorted] into sub-problems [sub-lists], until solved sub problems [sorted sub-lists] are found. This is implemented as follows:

Choose one item A[I] from the list A[ ].

Rearrange the list so that this item is in the proper position, i.e., all preceding items have a lesser value and all succeeding items have a greater value than this item.

1. Place A[0], A[1] .. A[I-1] in sublist 1

2. A[I]

3. Place A[I + 1], A[I + 2] ... A[N] in sublist 2

Repeat steps 1 & 2 for sublist1 & sublist2 till A[ ] is a sorted list.

As can be seen, this algorithm has a recursive structure.

The *divide'* procedure is of utmost importance in this algorithm. This is usually implemented as follows:

1. Choose A[I] as the dividing element.

2. From the left end of the list (A[O] onwards) scan till an item A[R] is found whose value is greater than A[I].

3. From the right end of list [A[N] backwards] scan till an item A[L] is found whose value is less than A[1].

4. Swap A[R] & A[L].

5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.

6. At this point, sublist1 & sublist are ready.

7. Now do the same for each of sublist1 & sublist2.

Program 10.1 gives the program segment for Quick sort. It uses recursion.

```
Quicksort(A,m,n)
int A[ ],m,n
{
        int i, j, k;
        if m<n
        {
                i=m;
                j=n+1;
                k=A[m];
                do
                  do
                     ++i;
                          while (A[i] < k);
                  do
                     --j;
                          while (A[j] > k);
                  if (i < j)
                  {
                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                  }
                     while (i<j);
 temp = A[m];
 A[m] = A[j];
 A[j] = temp;
 Quicksort(A,m,j-1);
 Quicksort(A,j+1,n);
}
```

Program 10.1: Quick Sort

The Quick sort algorithm uses the $O(N \log_2 N)$ comparisons on average. The performance can be improved by keeping in mind the following points.

1. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.

2. Use a better dividing element in the implementations.

It is also possible to write the non-recursive Quick sort algorithm.

10.2.4 2-Way Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea in this is to divide the list into a number of sublists, sort each of these sublists and merge them to get a single sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get n/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called *Concatenate sort*. *Figure 10.2* depicts 2-way merge sort.

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the $0(n \log_2 n)$.

The disadvantage of using mergesort is that it requires two arrays of the same size and space for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

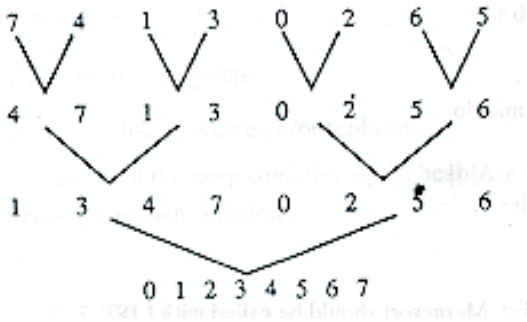


Figure 10.2: 2-way .merge sort

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the $0(n \log_2 n)$.

The disadvantage of using mergesort is that it requires two arrays of the same size and space for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

10.2.5 Heap Sort

We will begin by defining a new structure called *Heap. Figure 10.3* illustrates a Binary tree.

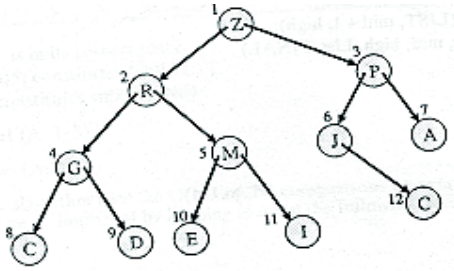


Figure 10.3: A Binary Tree

A complete binary tree is said to satisfy the 'heap condition' if the key of each node is greater than or equal to the key 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 values of the nodes are then assigned to array positions whose index is given by the number of the node. For the example tree, the corresponding array is depicted in *Figure 10.4*.

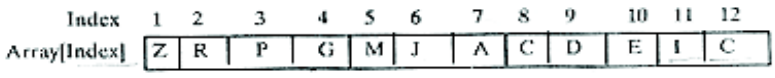


Figure 10.4: Array for the binary tree of figure 10.3

The relationships of a node can also be determined from this 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 $\lfloor J/2 \rfloor$.

Consider the node M. It is at position 5. Its parent node is, therefore, at position

$5/2\rfloor = 2$ i.e. the parent is R. Its children are at positions 2×5 & $(2 \times 5) + 1$, i.e. 10 + 11 respectively i.e. E & I are its children.

A *Heap* is a complete binary tree, in which each node satisfies the heap condition, represented as an array.

We will now study the operations possible on a heap and see how these can be combined to generate a sorting algorithm.

The operations on a heap work in 2 steps.

1. The required node is inserted/deleted/or replaced.
2. The above operation may cause violation of the heap condition so the heap is traversed and modified to rectify any such violations.

Examples: Consider the insertion of a 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. So, the heap condition is violated.
3. Move R upto position 6 and move J down to position 13.
4. R > P. Therefore, the heap condition is still violated.
5. Swap R and P.
4. The heap condition is now satisfied by all nodes to get the heap of *Figure 10.5*.

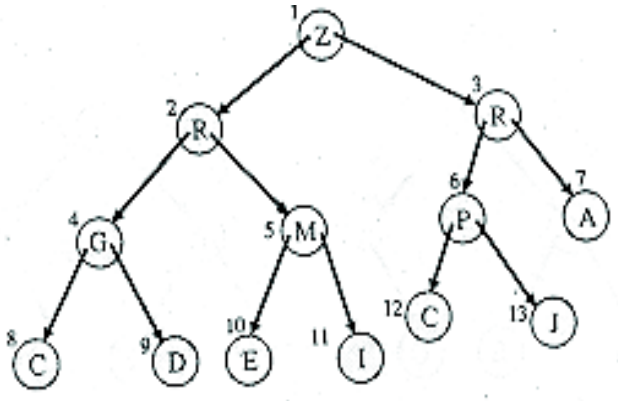


Figure 10.5: A Heap

This algorithm is guaranteed to sort n elements in $(n \log_2 n)$ time.

We will first see two methods of heap construction and then removal in order from the heap to sort the list.

1. Top down heap construction

   - Insert items into an initially empty heap, satisfying the heap condition at all steps.

2. Bottom up heap construction

   - Build a heap with the items in the order presented.

   - From the right most node modify to satisfy the heap condition.

We will exemplify this with an example.

Example: Build a heap of the following using top down approach for heap construction.

PROFESSIONAL

Figure 10.6 shows different steps of the top down construction of the heap.

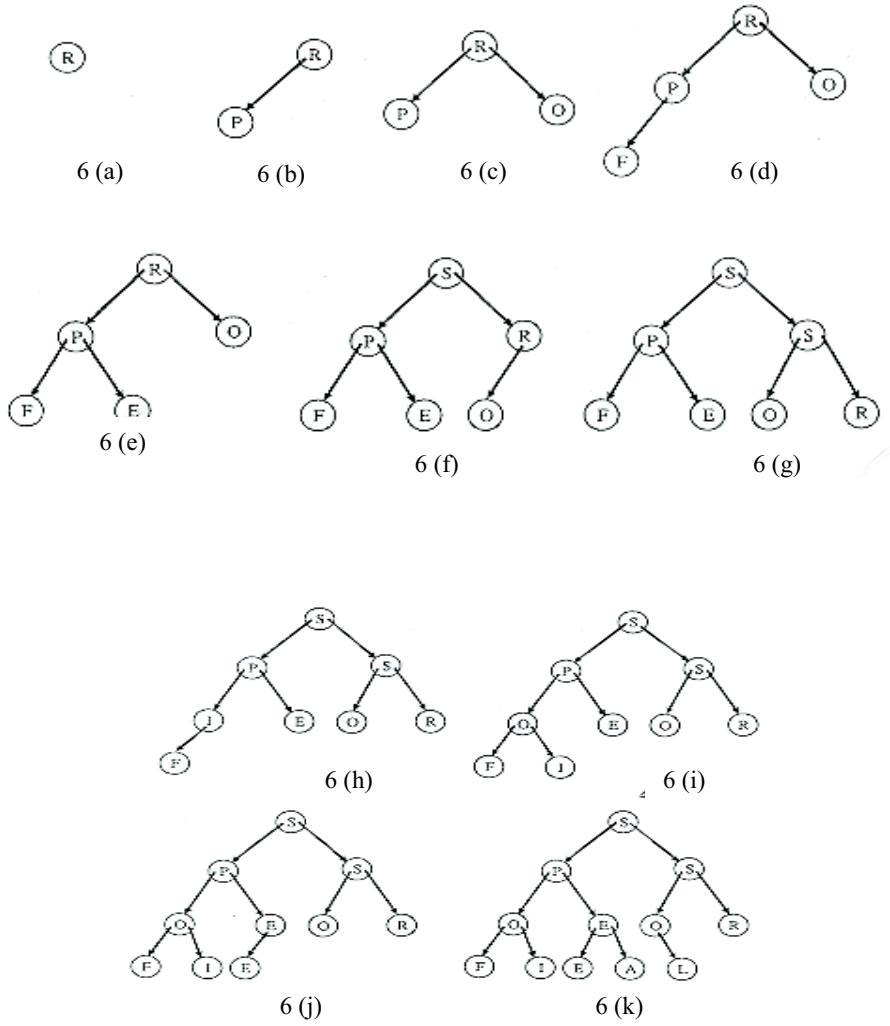


Figure 10.6: Heap Sort (Top down Construction)

Example: The input file is (2,3,81,64,4,25,36,16,9, 49). When the file is interpreted as a binary tree, it results in *Figure 10.7*. *Figure 10.8* depicts the heap.

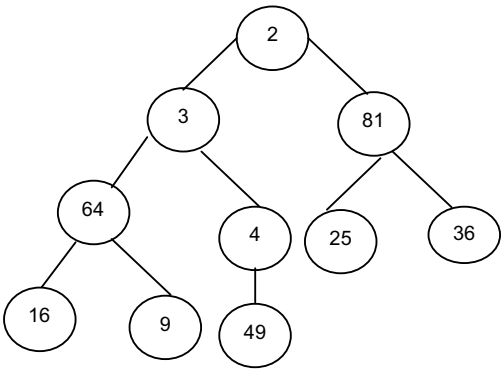


Figure 10.7: A Binary tree

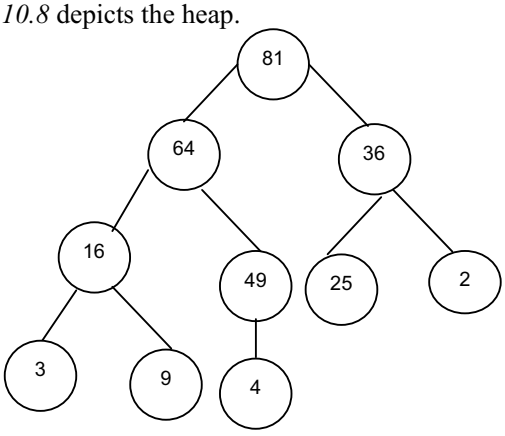
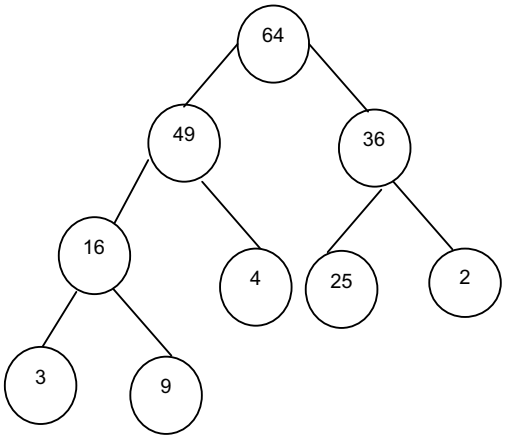
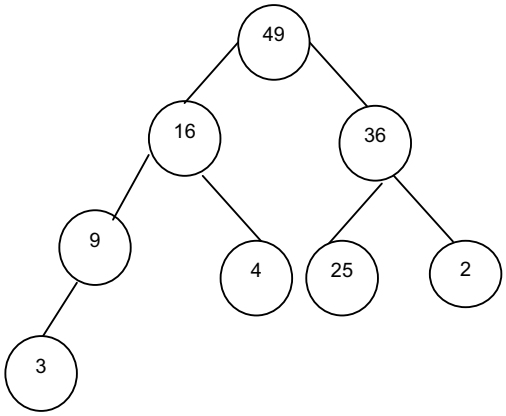


Figure 10.8: Heap of figure 10.7

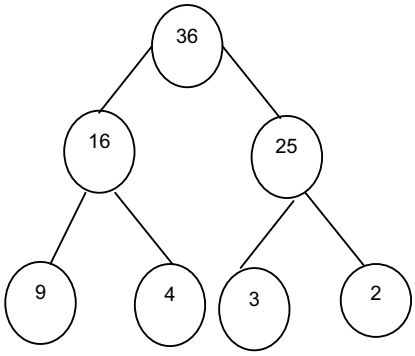
Figure 10.9 illustrates various steps of the heap of *Figure 10.8* as the sorting takes place.



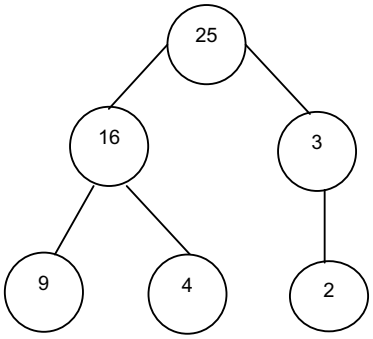
Sorted: 81
Heap size: 9



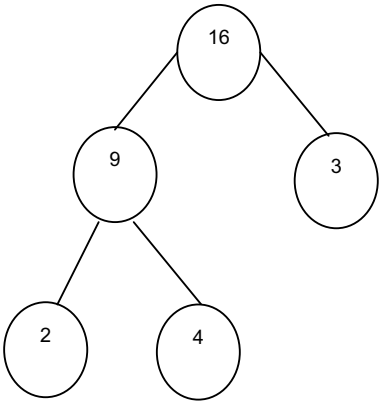
Sorted: 81,64
Heap size: 8



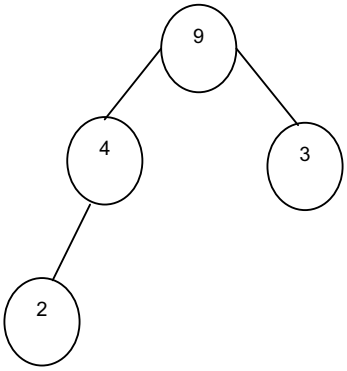
Sorted: 81,64,49
Heap size:7



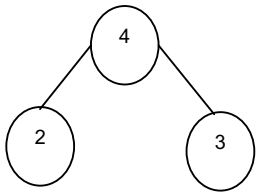
Sorted:81,64,49,36
Heap size:6



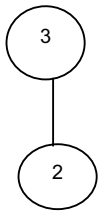
Sorted: 81, 64, 49, 36, 25
Size: 5



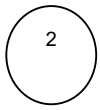
Sorted:81, 64, 49, 36, 25, 16
Size:4



Sorted: 81, 64, 49, 36, 25, 16, 9
Size: 3



Sorted:81, 64, 49, 36, 25, 16, 9, 4
Size: 2



Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3
Size : 1

Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3, 2
Result

Figure 10.9 : Various steps of figure 10.8 for a sorted file

10.3 SORTING ON SEVERAL KEYS

So far, we have been considering sorting based on single keys. But, in real life applications, we may want to sort the data on several keys. The simplest example is that of sorting a deck of cards. The first key for sorting is the suit-clubs, spades, diamonds and hearts. Then, within each suit, sorting the cards in ascending order from Ace, twos to king. This is thus a case of sorting on 2 keys.

Now, this can be done in 2 ways.

1
- Sort the 52 cards into 4 piles according to the suit.
- Sort each of the 4 piles according to face value of the cards.

2
- Sort the 52 cards into 13 piles according to face value.
- Stack these piles in order and then sort into 4 piles based on suit.

The first method is called the MSD (Most Significant Digit) sort and the second method is called the LSD (Least Significant Digit) sort. Digit stands for a key. Though they are called sorting methods, MSD and LSD sorts only decide the *order* of sorting. The actual sorting could be done by any of the sorting methods discussed in this unit.

☞ Check Your Progress 1

1) The complexity of Bubble sort is _____
2) Quick sort algorithm uses the programming technique of _____
3) Write a program in ‘C’ language for 2-way merge sort.
4) The complexity of Heap sort is _____

10.4 SUMMARY

Sorting is an important application activity. Many sorting algorithms are available. But, each is efficient for a particular situation or a particular kind of data. The choice of a sorting algorithm is crucial to the performance of the application.

In this unit we have studied many sorting algorithms used in internal sorting. This is not a conclusive list and the student is advised to read the suggested books for

exposure to additional sorting methods and for detailed discussions of the methods introduced here.

The following are the *three* most important efficiency criteria:

- use of storage space
- use of computer time
- programming effort.

10.5 SOLUTIONS/ANSWERS

1) $O(N^2)$ where N is the number of elements in the list to be sorted.
2) Divide and Conquer.
3) O(NlogN) where N is the number of elements to be sorted.

10.6 FURTHER READINGS

Reference Books

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein, PHI publications.

2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications.

Reference Websites

http://www.it.jcu.edu.au/Subjects/ cp2001/1998/LectureNotes/Sorting/

http://oopweb.com/Algorithms/Files/Algorithms.html

UNIT 11 ADVANCED DATA STRUCTURES

Structure **Page Nos.**

11.0	Introduction	15
11.1	Objectives	15
11.2	Splay Trees	15
	11.2.1 Splaying steps	
	11.2.2 Splaying Algorithm	
11.3	Red-Black trees	20
	11.3.1 Properties of a Red-Black tree	
	11.3.2 Insertion into a Red- Black tree	
	11.3.3 Deletion from a Red-Black tree	
11.4	AA-Trees	26
11.5	Summary	29
11.6	Solutions/Answers	29
11.7	Further Readings	30

11.0 INTRODUCTION

In this unit, the following four advanced data structures have been practically emphasized. These may be considered as alternative to a height balanced tree, i.e., AVL tree.

- Splay tree
- Red_Black tree
- AA_tree
- Treap

The key factors which have been discussed in this unit about the above mentioned data structures involve complexity of code in terms of Big oh notation, cost involved in searching a node, the process of deletion of a node and the cost involved in inserting a node.

11.1 OBJECTIVES

After going through this unit, you should be able to:

- know about Splay trees;
- know about Red-Black tree;
- know about AA-trees, and
- know about Treaps.

11.2 SPLAY TREES

Addition of new records in a Binary tree structure always occurs as leaf nodes, which are further away from the root node making their access slower. If this new record is to be accessed very frequently, then we cannot afford to spend much time in reaching it but would require it to be positioned close to the root node. This would call for readjustment or rebuilding of the tree to attain the desired shape. But, this process of rebuilding the tree every time as the preferences for the records change is tedious and time consuming. There must be some measure so that the tree adjusts itself automatically as the frequency of accessing the records changes. Such a self-adjusting tree is the Splay tree.

Splay trees are self-adjusting binary search trees in which every access for insertion or retrieval of a node, lifts that node all the way up to become the root, pushing the other nodes out of the way to make room for this new root of the modified tree. Hence, the frequently accessed nodes will frequently be lifted up and remain around the root position; while the most infrequently accessed nodes would move farther and farther away from the root.

This process of readjusting may at times create a highly imbalanced splay tree, wherein a single access may be extremely expensive. But over a long sequence of accesses, these expensive cases may be averaged out by the less expensive ones to produce excellent results over a long sequence of operations. The analytical tool used for this purpose is the Amortized algorithm analysis. This will be discussed in detail in the following sections.

11.2.1 Splaying Steps

Readjusting for tree modification calls for rotations in the binary search tree. Single rotations are possible in the left or right direction for moving a node to the root position. The task would be achieved this way, but the performance of the tree amortized over many accesses may not be good.

Instead, the key idea of splaying is to move the accessed node two levels up the tree at each step. Basic terminologies in this context are:

Zig: Movement of one step down the path to the left to fetch a node up.
Zag: Movement of one step down the path to the right to fetch a node up.

With these two basic steps, the possible splay rotations are:

Zig-Zig: Movement of two steps down to the left.
Zag-Zag: Movement of two steps down to the right.
Zig-Zag: Movement of one step left and then right.
Zag-Zig: Movement of one step right and then left.

Figure 11.1 depicts the splay rotations.

Zig:

Zig-Zig:

Figure 11.1: Splay rotations

Splaying may be top-down or bottom-up. In bottom-up splaying, splaying begins at the accessed node, moving up the chain to the root. While in top-down splaying, splaying begins from the top while searching for the node to access. In the next section, we would be discussing the top-down splaying procedure:

As top-down splaying proceeds, the tree is split into three parts:

a) **Central SubTree**: This is initially the complete tree and may contain the target node. Search proceeds by comparison of the target value with the root and ends with the root of the central tree being the node containing the target if present or null node if the target is not present.

b) **Left SubTree**: This is initially empty and is created as the central subtree is splayed. It consists of nodes with values less than the target being searched.

c) **Right SubTree**: This is also initially empty and is created similar to left subtree. It consists of nodes with values more than the target node.

Figure 11.2 depicts the splaying procedure with an example, attempting to splay at 20.

Initially,

The first step is Zig-Zag:

The next step is Zig-Zig:

The next step is the terminal zig:

Finally, reassembling the three trees, we get:

Figure 11.2: Splaying procedure

11.2.2 Splaying Algorithm

Insertion and deletion of a target key requires splaying of the tree. In case of insertion, the tree is splayed to find the target. If, target key is found, then we have a duplicate and the original value is maintained. But, if it is not found, then the target is inserted as the root.

In case of deletion, the target is searched by splaying the tree. Then, it is deleted from the root position and the remaining trees reassembled, if found.

Hence, splaying is used both for insertion and deletion. In the former case, to find the proper position for the target element and avoiding duplicity and in the latter case to bring the desired node to root position.

Splaying procedure

For splaying, three trees are maintained, the central, left and right subtrees. Initially, the central subtree is the complete tree and left and right subtrees are empty. The target key is compared to the root of the central subtree where the following two conditions are possible:

a) Target > Root: If target is greater than the root, then the search will be more to the right and in the process, the root and its left subtree are shifted to the left tree.

b) Target < Root: If the target is less than the root, then the search is shifted to the left, moving the root and its right subtree to right tree.

We repeat the comparison process till either of the following conditions are satisfied:

a) Target is found: In this, insertion would create a duplicate node. Hence, original node is maintained. Deletion would lead to removing the root node.

b) Target is not found and we reach a null node: In this case, target is inserted in the null node position.

Now, the tree is reassembled. For the target node, which is the new root of our tree, the largest node is the left subtree and is connected to its left child and the smallest node in the right subtree is connected as its right child.

Amortized Algorithm Analysis

In the amortized analysis, the time required to perform a set of operations is the average of all operations performed. Amortized analysis considers a long sequence of operations instead of just one and then gives a worst-case estimate. There are three different methods by which the amortized cost can be calculated and can be differentiated from the actual cost. The three methods, namely, are:

- Aggregate analysis: It finds the average cost of each operation. That is, T(n)/n. The amortized cost is same for all operations.
- Accounting method: The amortized cost is different for all operations and charges a credit as prepaid credit on some operations.
- Potential method: It also has different amortized cost for each operation and charges a credit as the potential energy to other operations.

There are different operations such as stack operations (push, pop, multipop) and an increment which can be considered as examples to examine the above three methods.

Every operation on a splay tree and all splay tree operations take O(log n) amortized time.

☞ Check Your Progress 1

1) Consider the following tree. Splay at node 2.

11.3 RED-BLACK TREES

A Red-Black Tree (RBT) is a type of Binary Search tree with one extra bit of storage per node, i.e. its color which can either be red or black. Now the nodes can have any of the color (red, black) from root to a leaf node. These trees are such that they guarantee O(log n) time in the worst case for searching.

Each node of a red black tree contains the field color, key, left, right and p (parent). If a child or a parent node does not exist, then the pointer field of that node contains NULL value.

11.3.1 Properties of a Red-Black Tree

Any binary search tree should contain following properties to be called as a red-black tree.

1. Each node of a tree should be either red or black.

2. The root node is always black.

3. If a node is red, then its children should be black.

4. For every node, all the paths from a node to its leaves contain the same number of black nodes.

We define the number of black nodes on any path from but not including a node x down to a leaf, the black height of the node is denoted by bh (x).

Figure 11.3 depicts a Red-Black Tree.

Figure 11.3: A Red-Black tree

Red-black trees contain two main operations, namely INSERT and DELETE. When the tree is modified, the result may violate red-black properties. To restore the tree properties, we must change the color of the nodes as well as the pointer structure. We can change the pointer structure by using a technique called rotation which preserves inorder key ordering. There are two kinds of rotations: left rotation and right rotation (refer to *Figures 11.4 and 11.5*).

Figure 11.4: Left rotation **Figure 11.5: Right rotation**

When we do a left rotation on a node y, we assume that its right child x is non null. The left rotation makes x as the new root of the subtree with y as x's left child and x's left child as y's right child.

The same procedure is repeated vice versa for the right rotation.

11.3.2 Insertion into a Red-Black Tree

The insertion procedure in a red-black tree is similar to a binary search tree i.e., the insertion proceeds in a similar manner but after insertion of nodes x into the tree T, we color it red. In order to guarantee that the red-black properties are preserved, we then fix up the updated tree by changing the color of the nodes and performing rotations. Let us write the pseudo code for insertion.

The following are the two procedures followed for insertion into a Red-Black Tree:

Procedure 1: This is used to insert an element in a given Red-Black Tree. It involves the method of insertion used in binary search tree.

Procedure 2: Whenever the node is inserted in a tree, it is made red, and after insertion, there may be chances of loosing Red-Black Properties in a tree, and, so, some cases are to be considered inorder to retain those properties.

During the insertion procedure, the inserted node is always red. After inserting a node, it is necessary to notify that which of the red-black properties are violated. Let us now look at the execution of fix up. Let Z be the node which is to be inserted and is colored red. At the start of each iteration of the loop,

1. Node Z is red
2. If P(Z) is the root, then P(Z) is black
3. Now if any of the properties i.e. property 2 is violated if Z is the root and is red OR when property 4 is violated if both Z and P(Z) are red, then we consider 3 cases in the fix up algorithm. Let us now discuss those cases.

Case 1(Z's uncle y is red): This is executed when both parent of Z (P(Z)) and uncle of Z, i.e. y are red in color. So, we can maintain one of the property of Red-Black tree by making both P(Z) and y black and making point of P(Z) to be red, thereby maintaining one more property. Now, this while loop is repeated again until color of y is black.

Case 2 (Z's uncle is black and Z is the right child): So, make parent of Z to be Z itself and apply left rotation to newly obtained Z.

Case 3 (Z's uncle is black and Z is the left child): This case executes by making parent of Z as black and P(P(Z)) as red and then performing right rotation to it i.e., to (P(Z)).

The above 3 cases are also considered conversely when the parent of Z is to the right of its own parent. All the different cases can be seen through an example.Consider a red-black tree drawn below with a node z (17 inserted in it) (refer to *Figure 11.6*).

Figure 11.6: A Red-Black Tree after insertion of node 17

Before the execution of any case, we should first check the position of P(Z) i.e. if it is towards left of its parent, then the above cases will be executed but, if it is towards the right of its parent, then the above 3 cases are considered conversely.

Now, it is seen that Z is towards the left of its parent (refer to *Figure 11.7*). So, the above cases will be executed and another node called y is assigned which is the uncle of Z and now cases to be executed are as follows:

Figure 11.7: Z is to the left of it's parent

Case 1: Property 4 is violated as both z and parent(z) are red (refer to *Figure 11.8*).

Figure 11.8: Both Z and P(Z) are red

Now, let us check to see which case is executed.

Case 2: The application of this case results in *Figure 11.9*.

Figure 11.9: Result of application of case-2

Case 3: The application of this case results in *Figure 11.10*.

Figure 11.10: Result of application of case-3

Finally, it resulted in a perfect Red-Black Tree (*Figure 11.10*).

11.3.3 Deletion from a Red-Black Tree

Deletion in a RBT uses two main procedures, namely,

Procedure 1: This is used to delete an element in a given Red-Black Tree. It involves the method of deletion used in binary search tree.

Procedure 2: Whenever the node is deleted from a tree, and after deletion, there may be chances of losing Red-Black Properties in a tree and so, some cases are to be considered in order to retain those properties.

This procedure is called only when the successor of the node to be deleted is Black, but if y is red, the red- black properties still hold and for the following reasons:

- No red nodes have been made adjacent
- No black heights in the tree have changed
- y could not have been the root

Now, the node (say x) which takes the position of the deleted node (say z) will be called in procedure 2. Now, this procedure starts with a loop to make the extra black up to the tree until

- X points to a black node
- Rotations to be performed and recoloring to be done
- X is a pointer to the root in which the extra black can be easily removed

This while loop will be executed until x becomes root and its color is red. Here, a new node (say w) is taken which is the sibling of x.

There are four cases which we will be considering separately as follows:

Case 1: If color of w's sibling of x is red
Since W must have black children, we can change the colors of w and p (x) and then left rotate p (x) and the new value of w to be the right node of parent of x. Now, the conditions are satisfied and we switch over to case 2, 3 and 4.

Case 2: If the color of w is black and both its children are also black.

Since w is black, we make w to be red leaving x with only one black and assign parent (x) to be the new value of x. Now, the condition will be again checked, i.e. x = left (p(x)).

Figure 11.11: Application of case-1

Figure 11.12: Application of case-2

Figure 11.13: Application of case-3

Figure 11.14: Application of case-4

Case 3: If the color of w is black, but its left child is red and w's right child is black.

After entering case-3, we change the color of left child of w to black and that of w to be red and then perform right rotation on w without violating any of the black properties. The new sibling w of x is now a black node with a red right child and thus case 4 is obtained.

Case 4: When w is black and w's right child is red.

This can be done by making some color changes and performing a left rotation on p(x). We can remove the extra black on x, making it single black. Setting x to be the root causes the while loop to terminate.

Note: In the above *Figures 11.11, 11.12, 11.13* and *11.14*, α, α', β, β', γ, ε are assumed to be either red or black depending upon the situation.

11.4 AA-Trees

Red-Black trees have introduced a new property in the binary search tree, i.e., an extra property of color (red, black). But, as these trees grow, in their operations like insertion, deletion, it becomes difficult to retain all the properties, especially in case of deletion. Thus, a new type of binary search tree can be described which has no property of having a color, but has a new property introduced based on the color which is the information for the new. This information of the level of a node is stored in a small integer (may be 8 bits). Now, AA-trees are defined in terms of level of each node instead of storing a color bit with each node. A red-black tree used to have various conditions to be satisfied regarding its color and AA-trees have also been designed in such a way that it should satisfy certain conditions regarding its new property, i.e., level.

The level of a node will be as follows:

1. Same of its parent, if the node is red.
2. One if the node is a leaf.
3. Level will be one less than the level of its parent, if the node is black.

Any red-black tree can be converted into an AA-tree by translating its color structure to levels such that left child is always one level lower than its parent and right child is always same or at one level lower than its parent. When the right child is at same level to its parent, then a horizontal link is established between them. Thus, we conclude that it is necessary that horizontal links are always at the right side and that there may not be two consecutive links. Taking into consideration of all the above properties, we show a AA-tree as follows (refer to *Figure 11.15*).

Figure 11.15: AA-tree

After having a look at the AA-tree above, we now look at different operations that can be performed at such trees.

The following are various operations on a AA-tree:

1. Searching: Searching is done by using an algorithm that is similar to the search algorithm of a binary search tree.
2. Insertion: The insertion procedure always start from the bottom level. But, while performing this function, either of the two problems can occur:

   (a) Two consecutive horizontal links (right side)
   (b) Left horizontal link.

While studying the properties of AA-tree, we said that conditions (a) and (b) should not be satisfied. Thus, in order to remove conditions (a) and (b), we use two new functions namely skew( ) and split( ) based on the rotations of the node, so that all the properties of AA-trees are retained.

The condition that (a) two consecutive horizontal links in an AA-tree can be removed by a left rotation by split( ) whereas the condition (b) can be removed by right rotations through function show( ). Either of these functions can remove these condition, but can also arise the other condition. Let us demonstrate it with an example. Suppose, in the AA-tree of *Figure 11.15*, we have to insert node 50.

According to the condition, the node 50 will be inserted at the bottom level in such a way that it satisfies Binary Search tree property also (refer to *Figure 11.16*).

Figure 11.16: After inserting node 50

Figure 11.17: Split at node 39(left rotation)

Now, we should be aware as to how this left rotation is performed. Remember, that rotation is introduced in Red-black tree and these rotations (left and right) are the same as we performed in a Red-Black tree. Now, again split ( ) has removed its condition but has created skew conditions (refer to *Figure 11.17*). So, skew ( ) function will now be called again and again until a complete AA-tree with a no false condition is obtained.

Figure 11.18: Skew at 55 (right rotation)

Figure 11.19: Split at 45

A skew problem arises because node 90 is two-level lower than its parent 75 and so in order to avoid this, we call skew / split function again.

Figure 11.20: The Final AA-tree

Thus, introducing horizontal left links, in order to avoid left horizontal links and making them right horizontal links, we make 3 calls to skew and then 2 calls to split to remove consecutive horizontal links (refer to *Figures 11.18, 11.19* and *11.20*).

A Treap is another type of Binary Search tree and has one property different from other types of trees. Each node in the tree stores an item, a left and right pointer and a priority that is randomly assigned when the node is created. While assigning the priority, it is necessary that the heap order priority should be maintained: node's priority should be at least as large as its parent's. A treap is both binary search tree with respect to node elements and a heap with respect to node priorities.

☞ Check Your Progress 2

1) Explain the properties of red-black trees along with an example.

……

……

11.5 SUMMARY

This is a unit of which focused on the emerging data structures. Splay trees, Red-Black trees, AA-trees and Treaps are introduced. The learner should explore the possibilities of applying these concepts in real life.

Splay trees are binary search trees which are self adjusting. *Self adjusting* basically means that whenever a splay tree is accessed for insertion or deletion of a node, then that node pushes all the remaining nodes to become root. So, we can conclude that any node which is accessed frequently will be at the top levels of the Splay tree.

A Red-Black tree is a type of binary search tree in which each node is either red or black. Apart from that, the root is always black. If a node is red, then its children should be black. For every node, all the paths from a node to its leaves contain the same number of black nodes.

AA-trees are defined in terms of level of each node instead of storing a color bit with each node. AA-trees have also been designed in such a way that it should satisfy certain conditions regarding its new property i.e. level.

The priorities of nodes of a Treap should satisfy the heap order. Hence, the priority of any node must be as large as it's parent's. Treap is the simplest of all the trees.

11.6 SOLUTIONS/ANSWERS

Check Your Progress 1

Ans. 1

Check Your Progress 2

1) Any Binary search tree should contain following properties to be called as a red-black tree.

1. Each node of a tree should be either red or black.
2. The root node is always black.
3. If a node is red then its children should be black.
4. For every node, all the paths from a node to its leaves contain the same number of black nodes.

Example of a red-black tree:

11.7 FURTHER READINGS

Reference Books

1. *Data Structures and Algorithm Analysis in C by Mark Allen Weiss, Pearson Education*

Reference Websites

http://www.link.cs.cmu.edu/splay/

http://www.cs.buap.mx/~titab/files/AATrees.pdf

UNIT 12 FILE STRUCTURES

Structure **Page Nos.**

12.0 Introduction 31
12.1 Objectives 31
12.2 Terminology 32
12.3 File Organisation 32
12.4 Sequential Files 33
12.4.1 Structure
12.4.2 Operations
12.4.3 Disadvantages
12.4.4 Areas of use
12.5 Direct File Organisation 35
12.6 Indexed Sequential File Organisation 35
12.7 Summary 37
12.8 Solutions/Answers 37
12.9 Further readings 37

12.0 INTRODUCTION

The structures of files change from operating system to operating system. In this unit, we shall discuss the fundamentals of file structures along with the generic file organisations.

A file may be defined as a collection of records. Though, a text file doesn't conform to this definition at the first glance, it is also a collection of records and records are words in the file.

Consider a file consisting of information about students. We may name such a file as *Student* file. The typical records of such file are shown in *Figure 12.1*.

Enum	Name	Address	State	Country	Programme
012786345	John	D-51, Nebsarai, Maidan Garhi	Delhi	India	BCA
98387123	Suresh	E-345, Banjara Hills	Hyderabad	India	MCA

Figure 12.1: Typical records of a *Student* file

A file should always be stored in such a way that the basic operations on it can be performed easily. In other words, queries should be able to be executed with out much hassle. We focus, in this unit, on the ways of storing the files on external storage devices. Selection of a particular way of storing the file on the device depends on factors such as retrieval records, the way the queries can be put on the file, the total number of keys in each record etc.

12.1 OBJECTIVES

After going through this unit, you should be able to

- learn the terminology of file structures;
- learn the underlying concepts of Sequential files, and
- know the Indexed sequential file organisation.

12.2 TERMINOLOGY

The following are the definitions of some important terms:

1) **Field:** It is an elementary data item characterised by its size, length and type. For example,

   Name : a character type of size 10
   Age : a numeric type

2) **Record:** It is a collection of related fields that can be treated as a unit from an application point of view.

   For example:

A university could use a student record with the fields, namely, University enrolment no. and names of courses.

3) **File:** Data is organised for storage in files. A file is a collection of similar, related records. It has an identifying name.

For example, "STUDENT" could be a file consisting of student records for all the students in a university.

4) **Index:** An index file corresponds to a data file. It's records contain a key field and a pointer to that record of the data file which has the same value of the key field.

The data stored in files is accessed by software which can be divided into the following two categories:

i) **User Programs:** These are usually written by a programmer to manipulate retrieved data in the manner required by the application.

ii) **File Operations:** These deal with the physical movement of data, in and out of files. User programs effectively use file operations through appropriate programming language syntax The File Management System manages the independent files and acts as the software interface between the user programs and the file operations.

   File operations can be categorised as

   - CREATION of the file
   - INSERTION of records into the file
   - UPDATION of previously inserted records
   - RETRIEVAL of previously inserted records
   - DELETION of records
   - DELETION of the file.

12.3 FILE ORGANISATION

File organisation may be defined as a method of storing records in file. Also, the subsequent implications on the way these records can be accessed. The following are the factors involved in selecting a particular file organisation:

- Ease of retrieval
- Convenience of updates
- Economy of storage
- Reliability
- Security
- Integrity.

Different file organisations accord different weightages to the above factors. The choice must be made depending upon the individual needs of the particular application in question.

We now introduce in brief, the various commonly encountered file organisations.

- **Sequential Files**

  Data records are stored in some specific sequence e.g., order of arrival value of key field etc. Records of a sequential file cannot be accessed at random i.e., to access the n^{th} record, one must traverse the preceding (n–1) records. Sequential files will be dealt with at length in the next section.

- **Relative Files**

  Each data record has a fixed place in a relative file. Each record must have associated with it in integer key value that will help identify this slot. This key, therefore, will be used for insertion and retrieval of the record. Random as well as sequential access is possible. Relative files can exist only on random access devices like disks.

- **Direct Files**

  These are similar to relative files, except that the key value need not be an integer. The user can specify keys which make sense to his application.

- **Indexed Sequential Files**

  An index is added to the sequential file to provide random access. An overflow area needs to be maintained to permit insertion in sequence.

- **Indexed Flies**

  In this file organisation, no sequence is imposed on the storage of records in the data file, therefore, no overflow area is needed. The index, however, is maintained in strict sequence. Multiple indexes are allowed on a file to improve access.

12.4 SEQUENTIAL FILES

In this section, we shall discuss about Sequential file organisation. Sequential files have data records stored in a specific sequence. A sequentially organised file may be stored on either a serial-access or a direct-access storage medium.

12.4.1 Structure

To provide the ‘sequence’ required, a ‘key’ must be defined for the data records. Usually a field whose values can uniquely identify data records is selected as the key. If a single field cannot fulfil this criterion, then a combination of fields can serve as the key.

12.4.2 Operations

1. **Insertion:** Records must be inserted at the place dictated by the sequence of the keys. As is obvious, direct insertions into the main data file would lead to

frequent rebuilding of the file. This problem could be mitigated by reserving' overflow areas' in the file for insertions. But, this leads to wastage of space.

The common method is to use transaction logging. This works as follows:

- It collects records for insertion in a transaction file in their order of their arrival.
- When population of the transactions file has ceased, sort the transaction file in the order of the key of the primary data file
- Merge the two files on the basis of the key to get a new copy of the primary sequential file.

Such insertions are usually done in a batch mode when the activity/program which populates the transaction file have ceased. The structure of the transaction files records will be identical to that of the primary file.

2. **Deletion:** Deletion is the reverse process of insertion. The space occupied by the record should be freed for use. Usually deletion (like-insertion) is not done immediately. The concerned record is written to a transaction file. At the time of merging, the corresponding data record will be dropped from the primary data file.

3. **Updation:** Updation is a combination of insertion and deletions. The record with the new values is inserted and the earlier version deleted. This is also done using transaction files.

4. **Retrieval:** User programs will often retrieve data for viewing prior to making decisions. Therefore, it is vital that this data reflects the latest state of the data, if the merging activity has not yet taken place.

Retrieval is usually done for a particular value of the key field. Before return in to the user, the data record should be merged with the transaction record (if any) for that key value.

The other two operations 'creation' and 'deletion' of files are achieved by simple programming language statements.

12.4.3 Disadvantages

Following are some of the disadvantages of sequential file organisation:

- Updates are not easily accommodated.
- By definition, random access is not possible.
- All records must be structurally identical. If a new field has to be added, then every record must be rewritten to provide space for the new field.

12.4.4 Areas of Use

Sequential files are most frequently used in commercial batch oriented data processing where there is the concept of a master file to which details are added periodically. Example is payroll applications.

Check Your Progress

1) Describe the record structure to be used for the lending section of a library.

……

……

……

.....

12.5 DIRECT FILE ORGANISATION

It offers an effective way to organise data when there is a need to access individual records directly.

To access a record directly (or random access) a relationship is used to translate the key value into a physical address. This is called the mapping function R.
R(key value) = Address

Direct files are stored on DASD (Direct Access Storage Device).

A calculation is performed on the key value to get an address. This address calculation technique is often termed as hashing. The calculation applied is called a hash function.

Here, we discus a very commonly used hash function called Division - Remainder.

Division-Remainder Hashing

According to this method, key value is divided by an appropriate number, generally a prime number, and the division of remainder is used as the address for the record.

The choice of appropriate divisor may not be so simple. If it is known that the file is to contain n records, then we must, assuming that only one record can be stored at a given address, have divisor n.

Also we may have a very large key space as compared to the address space. Key space refers to all the possible key values. The address space possibly may not match actually for the key values in the file. Hence, calculated address may not be unique. It is called Collision, i.e.,

R(K1) = R(K2), where K1 = K2.

Two unequal keys have been calculated to have the same address. The keys are called synonyms.

There are various approaches to handle the problem of collisions. One of these is to hash to buckets. A bucket is a space that can accommodate multiple records. The student is advised to read some text on bucket Addressing and related topics.

12.6 INDEXED SEQUENTIAL FILE ORGANISATION

When there is need to access records sequentially by some key value and also to access records directly by the same key value, the collection of records may be organised in an effective manner called Indexed Sequential Organisation.

You must be familiar with search process for a word in a language dictionary. The data in the dictionary is stored in sequential manner. However an index is provided in terms of thumb tabs. To search for a word we do not search sequentially. We access the index. That is, locate an approximate location for the word and then proceed to find the word sequentially.

To implement the concept of indexed sequential file organisations, we consider an approach in which the index part and data part reside on a separate file. The index file has a tree structure and data file has a sequential structure. Since the data file is sequenced, it is not necessary for the index to have an entry for each record. Consider the sequential file with a two-level index.

Level 1 of the index holds an entry for each three-record section of the main file. The Level 2 indexes Level 1 in the same way.

When the new records are inserted in the data file, the sequence of records need to be preserved and also the index is accordingly updated.

Two approaches used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the contents of the static index may change, but the structure does not change. In case of dynamic indexing approach, insertions and deletions in the main data file may lead to changes in the index structure. Recall the change in height of B-Tree as records are inserted and deleted.

Both dynamic and static indexing techniques are useful depending on the type of application.

A directory is a component of file. Consider a file which doesn't have any keys for its records. When a query is executed on such a file, the time consumed to execute the query is more when compared to another file which is having keys, because, there may be arising necessity where in the file has to be sorted on the field(s) on which the query is based. So, for each query, the file has to be sorted on the field on which the query is based which is cumbersome. In the case of files which have keys, different versions of the files which result due to sorting on the keys are stored in the directory of that file. Such files are called index files and the number of index files vary from file to file. For example, consider the file of *Figure 12.1*. If we designate Enrolment Number (Enum) and Name as keys, then we may have two index files based on the each key. Of course, we can have more than two index files, to deal with queries which use both the keys. Different software store index files in a different manner so that the operations on the records can be performed as soon as possible after the query is submitted.

One of the prominent indexing techniques is Cylinder-Surface indexing.

Since, there exists a primary key for each of the files, there will be an index file based on the primary key. Cylinder-Surface Indexing is useful for such index file. In this type of indexing, the records of the file are stored one after another in such a way that the primary keys of the records are in increasing order. The index file will have two fields. They are cylinder index and corresponding surface indexes. There will be multiple cylinders and there are multiple surfaces corresponding to each cylinder. Suppose that the file needs m cylinders, then cylinder index will have m entries. Each cylinder will be having one entry which corresponds to the largest key value in that cylinder. Assume that the disk has n surfaces which can be used. Then, each surface index has n entries. The k-th entry in surface index for cylinder l^{th} cylinder if the value of the largest key on the l^{th} track of the k^{th} surface. Hence, m.n indicates the total number of surface index entries.

Suppose that the need arises to search for a record whose key value is B. Then, the first step is to load the cylinder index of the file into memory. Usually, each cylinder index occupies only one track as the number of cylinders are only few. The cylinder which holds the desired record is found by searching the cylinder index. Usually, the search takes O(log m) time. After the search of cylinder index, the corresponding cylinder is determined. Based on the cylinder, the corresponding surface index is retrieved to look the record for which the search has started. Whenever a search is initiated for the surface index, usually sequential search is used. Of course, it depends on the number of surfaces. But, usually, the number of surfaces are less. After finding the cylinder to be accessed and after finding the surface to be accessed, the corresponding track is loaded into memory and that track is searched for the needed record.

12.7 SUMMARY

This unit dealt with the methods of physically storing data in the files. The terms fields, records and files were defined. The organisation types were introduced.

The various file organisation techniques were discussed. Sequential File Organisation finds use in application areas where batch processing is more common. Sequential files are simple to use and can be stored on inexpensive media. They are suitable for applications that require direct access to only particular records of the collection. They do not provide adequate support for interactive applications.

In Direct file organisation, there exists a predictable relationship between the key used and to identify a particular record on secondary storage. A direct file must be stored on a direct access device. Direct files are used extensively in application areas where interactive processing is used.

An Indexed Sequential file supports both sequential access by key value and direct access to a particular record, given its key value. It is implemented by building an index on top of a sequential data file that resides on a direct access storage device.

12.8 SOLUTIONS/ANSWERS

Check Your Progress

1) The following record structure could take care of the general requirements of a lending library.

   Member No., Member Name, Book Classification, i.e., Book Name, Author, Issue Date, Due Date.

12.9 FURTHER READINGS

Reference Books

1. *Fundamentals of Data Structures in C++* by E. Horowitz, Sahani and D. Mehta, Galgotia Publications.

2. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J. Augenstein and Aaron M. Tanenbaum, PHI Publications.

3. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

Reference Websites

http:// www.cs.umbc.edu

http://www.fredosaurus.com