UNIT 1 ANALYSIS OF ALGORITHMS

Structure **Page Nos.**

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)*.

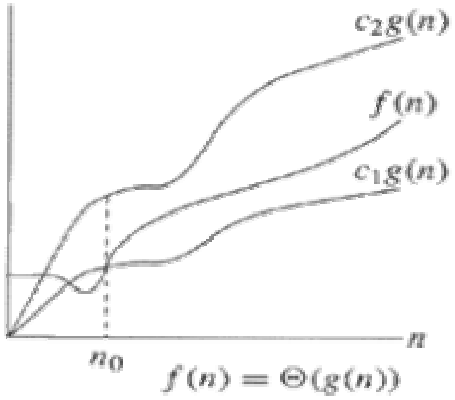


Figure 1.1 : Plot of $f(n) = \Theta(g(n))$

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

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

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

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

Similarly, by selecting $n_o = 13$ $c_1 \le 1/39$, the right hand inequality holds true. So, for $c_1 = 1/39$, $c_2 = 1/3$ and $n_o \ge 13$, 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 $f(n) = 6n^3 \ne \Theta(n^2)$.

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

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 $cg(n)$.

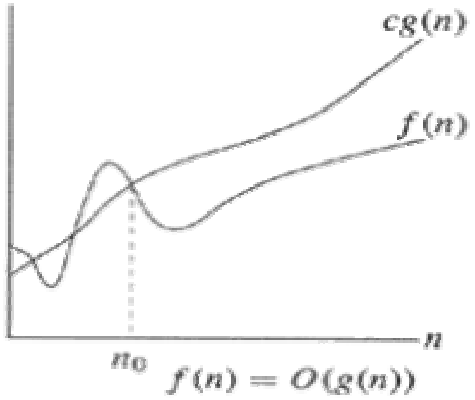


Figure 1.2: Plot of $f(n) = O(g(n))$

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 $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

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 $f(n) = \Omega(g(n))$.

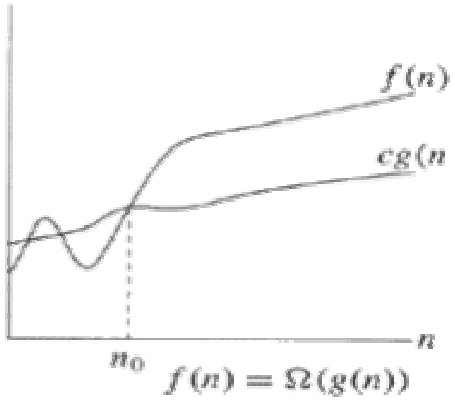


Figure 1.3: Plot of $f(n) = \Omega(g(n))$

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.

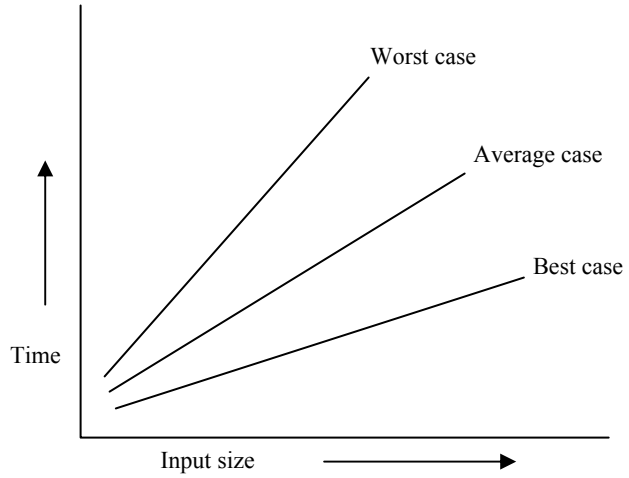


Figure 1.4 : Best, Average and Worst case scenarios

☞ 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

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.*

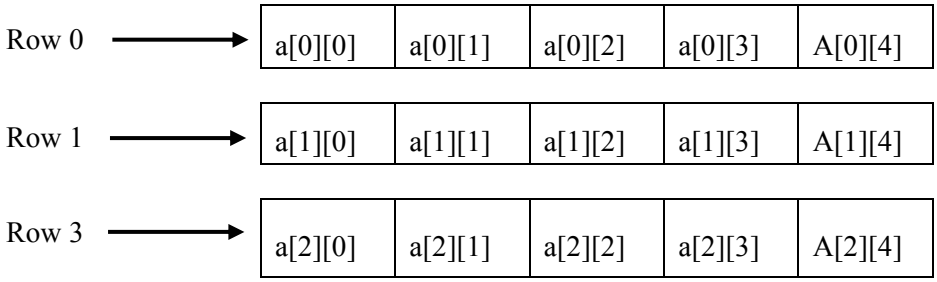


Figure 2.2: Schematic of a Two-Dimensional Array

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 1st index × size of 2nd 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).

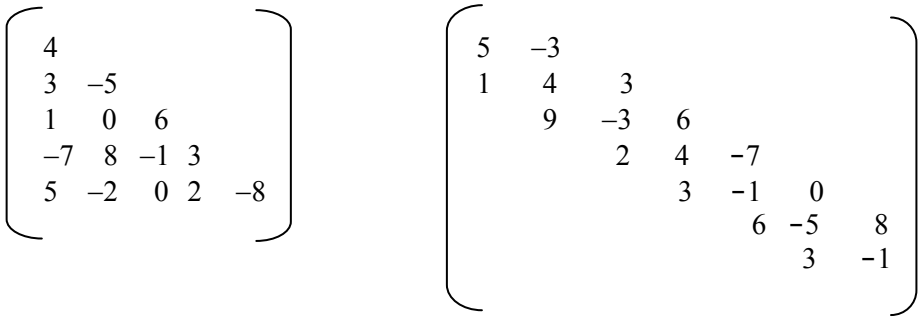
In general, for any two-dimensional array a[j][k] is equivalent to:

*((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

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.

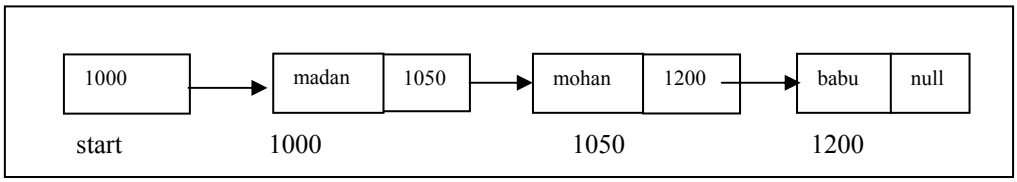


Figure 3.1: A Singly linked list

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

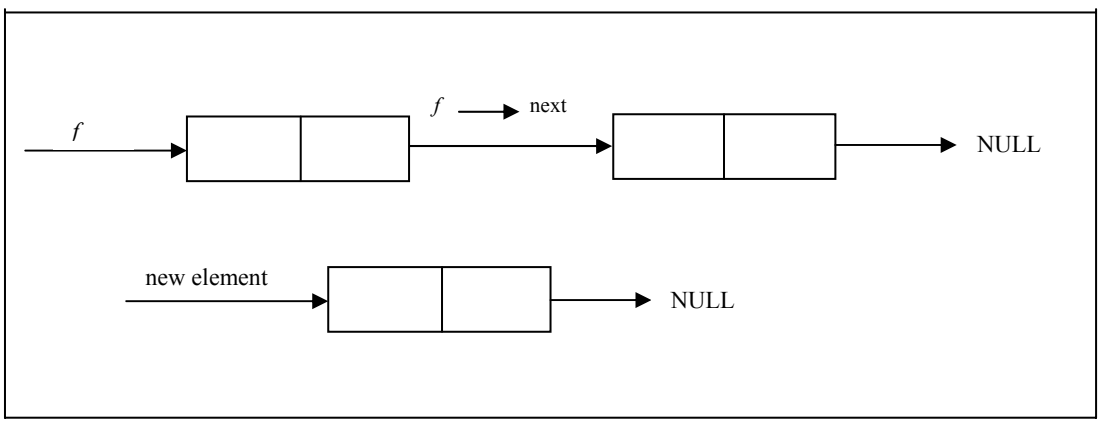


Figure 3.2: A linked list of two elements and an element that is to be inserted

After insertion

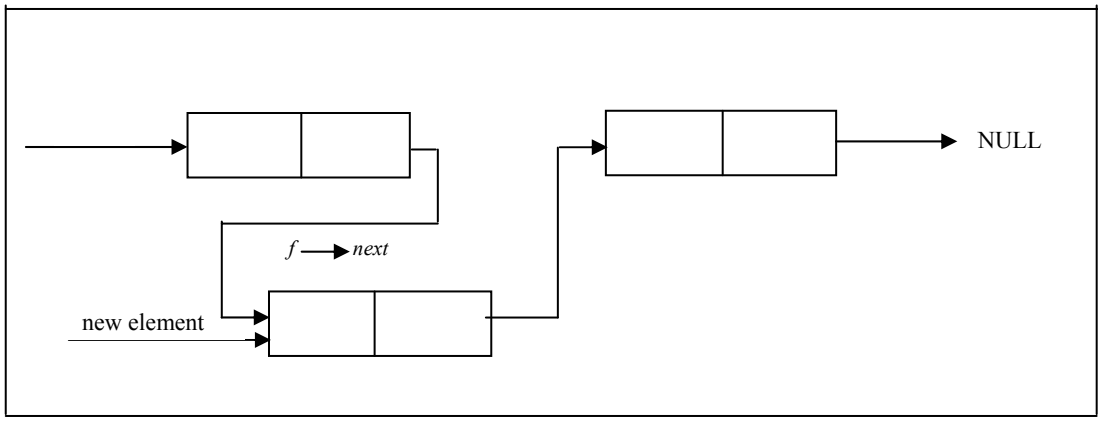


Figure 3.3: Insertion of a new element into linked list

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);
}
```

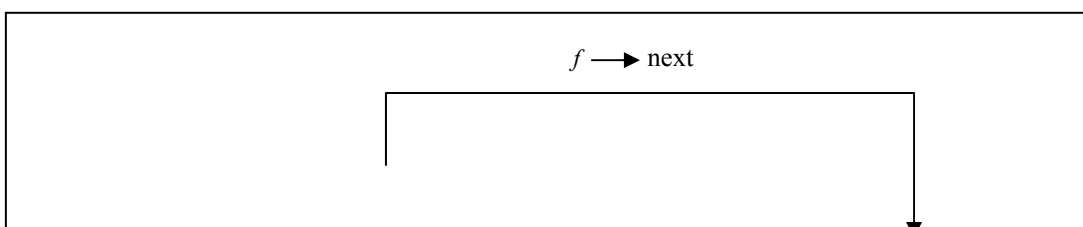
Program 3.5: Insertion of an element into a linked list at a specific position

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



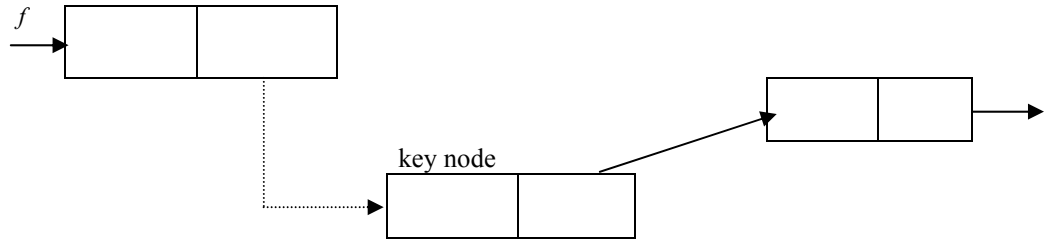


Figure 3.4: Deletion of an element from the linked list (Dotted line depicts the link prior to deletion)

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**.

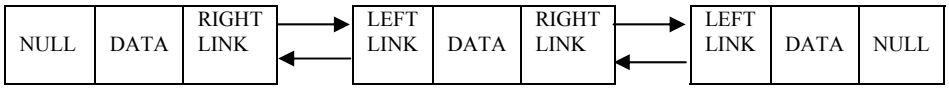


Figure 3.5: A Doubly Linked List

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.

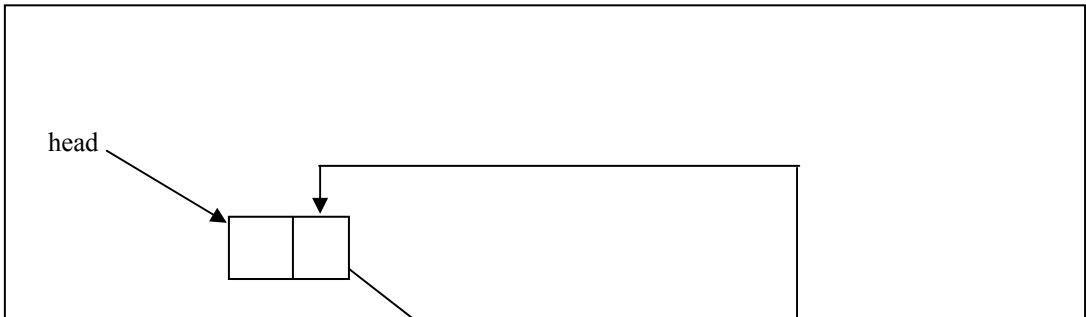


Figure 3.6 : A Circular Linked List

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.

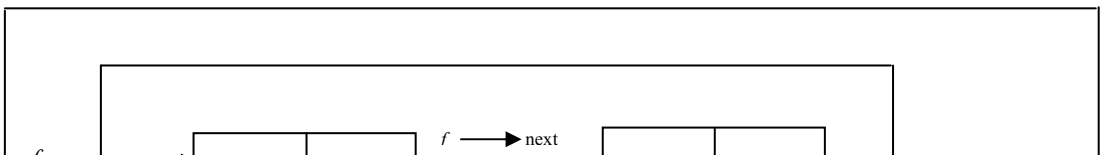
Step 4 else, insert the new element in the list by using the find function. find function returns the address of the found element to the insert_list function.

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.*



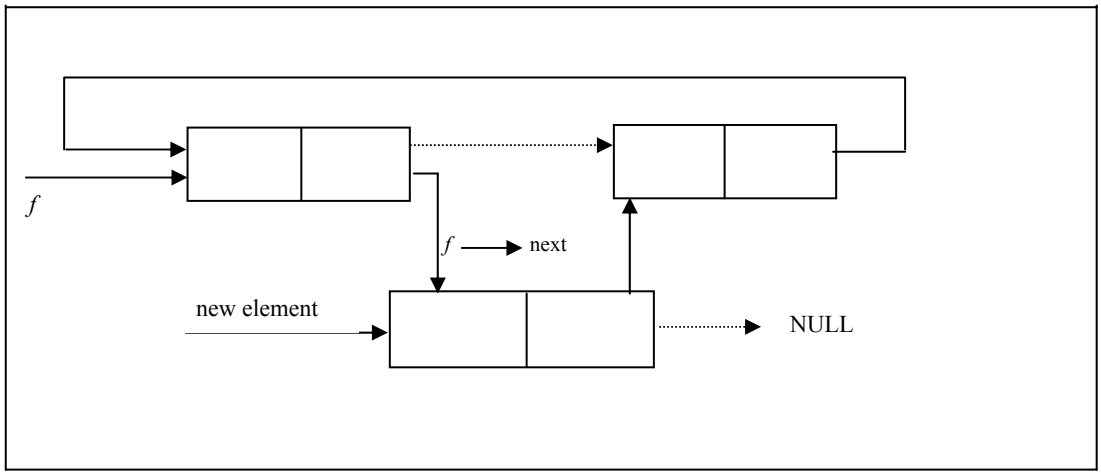


Figure 3.8: A Circular Linked List after insertion of the new element between first and second nodes (Dotted lines depict the links prior to insertion)

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.

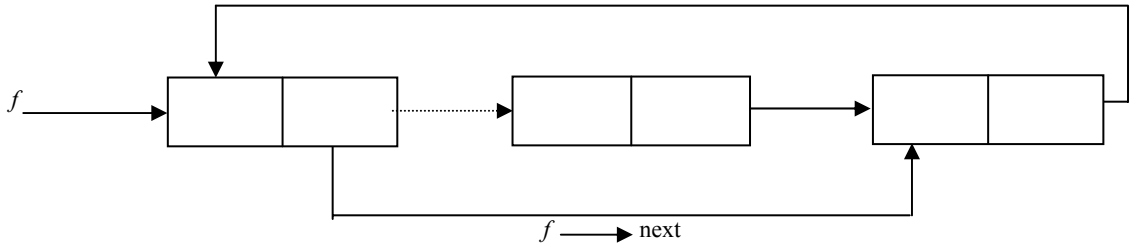


Figure 3.9 A Circular Linked List from which an element was deleted (Dotted line shows the linked that existed prior to deletion)

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.

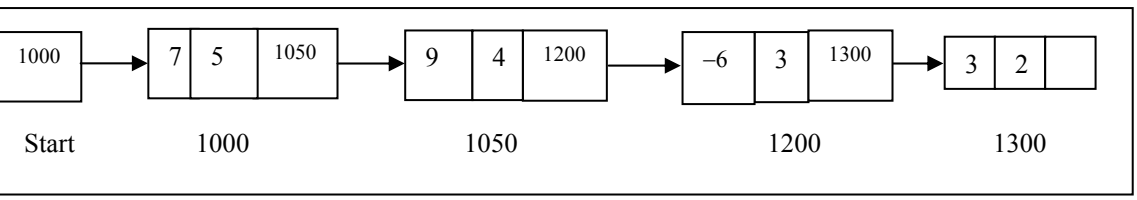


Figure 3.10: Representation of a Polynomial using a singly linked list

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

.