DATA STRUCTURE

II B. Tech I semester (JNTUH-R13)

INFORMATION TECHNOLOGY

Data Structures Through C

UNIT – I

Basic concepts of Algorithm

Preliminaries of Algorithm:

An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

The algorithm word originated from the Arabic word "Algorism" which is linked to the name of the Arabic mathematician AI Khwarizmi. He is considered to be the first algorithm designer for adding numbers.

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

An algorithm is endowed with the following properties:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.
5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

Practical Algorithm Design Issues:

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is considerable desirable.

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In

performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analyzing Algorithms:

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) with some standard functions. The most common computing times are

$O(1), O(\log_2 n), O(n), O(n \log_2 n), O(n^2), O(n^3), O(2^n)$

Example:

Program Segment A

```
--------------------------
      x =x + 2;
--------------------------
```

Program Segment B

```
--------------------------
for k =1 to n do
        x =x + 2;
end;
--------------------------
```

Program Segment C

```
--------------------------
for j =1 to n do
  for x = 1 to n do
        x =x + 2;
   end
end;
--------------------------
```

Total Frequency Count of Program Segment A

Program Statements	Frequency Count
-------------------------- x =x + 2; --------------------------	1
Total Frequency Count	1

Total Frequency Count of Program Segment B

Program Statements	Frequency Count
-------------------------- for k =1 to n do	(n+1)

x =x + 2;
end;
---------- 	
	n
	n

Total Frequency Count	3n+1

Total Frequency Count of Program Segment C

Program Statements	Frequency Count

---------- 	
for j =1 to n do
for x = 1 to n do
x =x + 2;
end
end;
---------- 	
	
	(n+1)
	n(n+1)
	n^2
	n^2
	n

Total Frequency Count	$3n^2+3n+1$

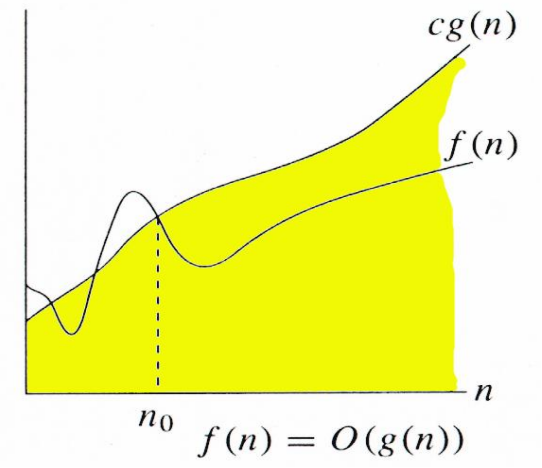
The total frequency counts of the program segments A, B and C given by 1, (3n+1) and $(3n^2+3n+1)$ respectively are expressed as O(1), O(n) and $O(n^2)$. These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations, which is nothing but the amount of memory they require for their execution.

Asymptotic Notations:

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here g(n) is the upper bound of the function f(n).

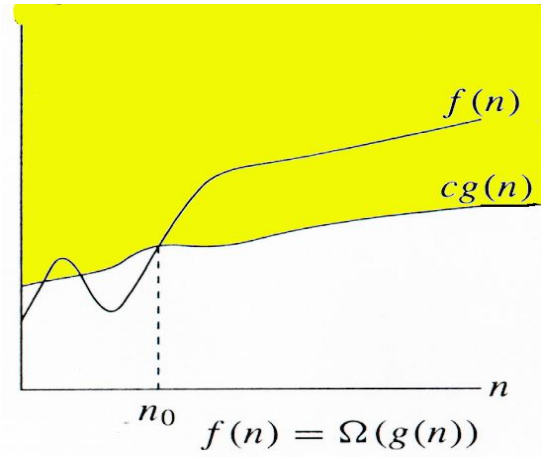
f(n)	g(n)	
$16n^3 + 45n^2 + 12n$	n^3	$f(n) = O(n^3)$
34n – 40	n	f(n) = O(n)
50	1	f(n) = O(1)



Omega(Ω): Definition: f(n) = Ω(g(n)) (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number c such that $|f(n)| \geq c\ |g(n)|$ for all $n \geq n_0$. Here g(n) is the lower bound of the

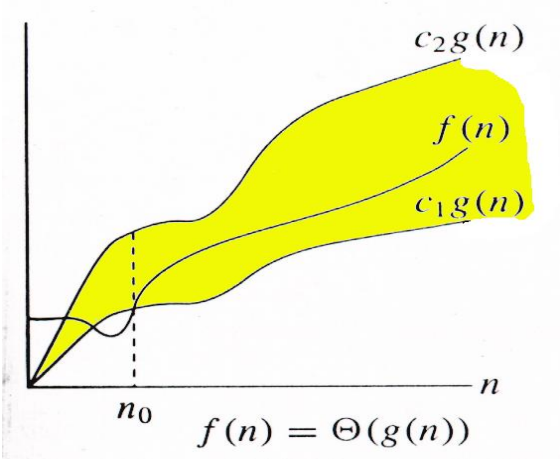
function f(n).

f(n)	g(n)	
$16n^3+8n^2+2$	n^3	$f(n) = \Omega(n^3)$
24n + 9	n	$f(n) = \Omega(n)$



Theta(Θ): Definition: f(n) = Θ(g(n)) (read as f of n is theta of g of n), if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that $c_1 |g(n)| \le |f(n)| \le c_2 |g(n)|$ for all $n \ge n_0$. The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, $n \ge n_0$.

f(n)	g(n)	
$16n^3 + 30n^2 - 90$	n^2	$f(n) = \Theta(n^2)$
$7.2^n + 30n$	2^n	$f(n) = \Theta(2^n)$



Little oh(o): Definition: f(n) = O(g(n)) (read as f of n is little oh of g of n), if f(n) = O(g(n)) and $f(n) \ne \Omega(g(n))$.

f(n)	g(n)	
18n + 9	n^2	$f(n) = o(n^2)$ since $f(n) = O(n^2)$ and $f(n) \ne \Omega(n^2)$ however $f(n) \ne O(n)$.

Relations Between O, Ω, Θ:

Theorem : For any two functions g(n) and f(n),

$f(n) = \Theta(g(n))$ iff

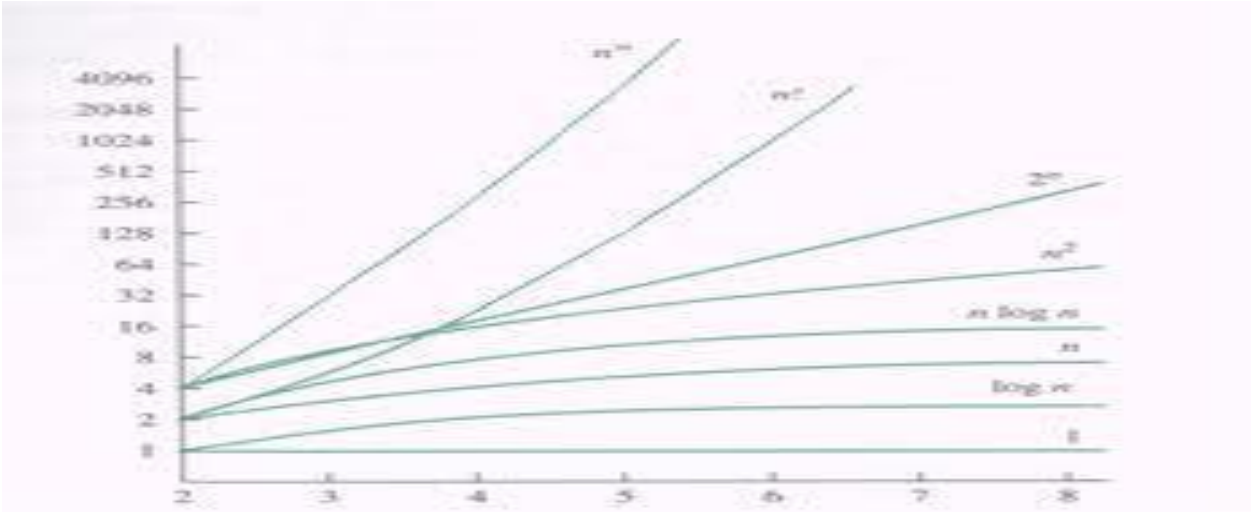
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Time Complexity:

Complexity	Notation	Description
Constant	O(1)	Constant number of operations, not depending on the input data size.

Logarithmic	O(logn)	Number of operations proportional of log(n) where n is the size of the input data.
Linear	O(n)	Number of operations proportional to the input data size.
Quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data.
Cubic	$O(n^3)$	Number of operations proportional to the cube of the size of the input data.
Exponential	$O(2^n)$	Exponential number of operations, fast growing.
	$O(k^n)$	
	O(n!)	

Time Complexities of various Algorithms:



Numerical Comparision of Different Algorithms:

S.No.	log_2n	n	$nlog_2n$	n^2	n^3	2^n
1.	0	1	1	1	1	2
2.	1	2	2	4	8	4
3.	2	4	8	16	64	16
4.	3	8	24	64	512	256
5.	4	16	64	256	4096	65536

Reasons for analyzing algorithms:

1. To predict the resources that the algorithm requires
    - Computational Time(CPU consumption).
    - Memory Space(RAM consumption).
    - Communication bandwidth consumption.
2. To predict the running time of an algorithm
    - Total number of primitive operations executed.

Recursion Definition:

1. Recursion is a technique that solves a problem by solving a smaller problem of the same type.
2. A recursive function is a function invoking itself, either directly or indirectly.
3. Recursion can be used as an alternative to iteration.
4. Recursion is an important and powerful tool in problem solving and programming.
5. Recursion is a programming technique that naturally implements the divide and conquer problem solving methodology.

Four criteria of a Recursive Solution:

1. A recursive function calls itself.
2. Each recursive call solves an identical, but smaller problem.
3. A test for the **base case** enables the recursive calls to stop.
    - There must be a case of the problem(known as **base case** or **stopping case**) that is handled differently from the other cases.
    - In the **base case**, the recursive calls stop and the problem is solved directly.
4. Eventually, one of the smaller problems must be the base case.

Example: To define n! recursively, n! must be defined in terms of the factorial of a smaller number.

n! = n * (n-1)!

Base case: 0! =1

n! = 1 if n=0

n! = n * (n-1)! if n >0

Designing Recursive Algorithms:

The Design Methodology:

1. The statement that solves the problem is known as the **base case**.
2. Every recursive algorithm must have a **base case**. The rest of the algorithm is known as the **general case**.
3. The **general case** contains the logic needed to reduce the size of the problem.
   Example: in factorial example, the base case is fact(0) and the general case is n * fact(n-1).

Rules for designing a Recursive Algorithm:

1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion:

1. Recursion works best when the algorithm uses a data structure that naturally supports recursion. E.g. Trees.
2. In other cases, certain algorithms are naturally suited to recursion. E.g. binary search, towers of hanoi.
3. On the other hand, not all looping algorithms can or should be implemented with reursion.
4. Recursive solutions may involve extensive overhead (both time and memory) because they use calls. Each call takes time to execute. A recursive algorithm therefore generally runs more slowly

than its nonrecursive implementation.

Recusive Examples:

GCD Design: Given two integers a and b, the greatest common divisor is recursively found using the formula

gcd(a,b) = a if b=0
b if a=0 } Base case
gcd(b, a mod b) } General case

Fibonacci Design: To start a fibonacci series, we need to know the first two numbers.

Fibonacci(n) = 0 if n=0
1 if n=1 } Base case
Fibonacci(n-1) + fibonacci(n-2) } General case

Tracing a Recursive Function:

1. A stack is used to keep track of function calls.
2. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement.
3. For each function call, an activation call is created on the stack.

Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Reursion uses stack whereas iteration does not use stack.
3. Recursion uses more memory than iteration as its concept is based on stacks.
4. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
5. Recursion makes code smaller and iteration makes code longer.
6. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
7. While using recursion multiple activation records are created on stack for each call where as in iteration everything is done in one activation record.
8. Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
9. Recursion uses selection structure whereas iteration uses repetetion structure.

Types of Recursion:

Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called **direct recursion** and the later is called **indirect recursion**. Thus there are two types of recursion:

- Direct Recursion
- Indirect Recursion

Recursion may be further categorized as:

- Linear Recursion
- Binary Recursion

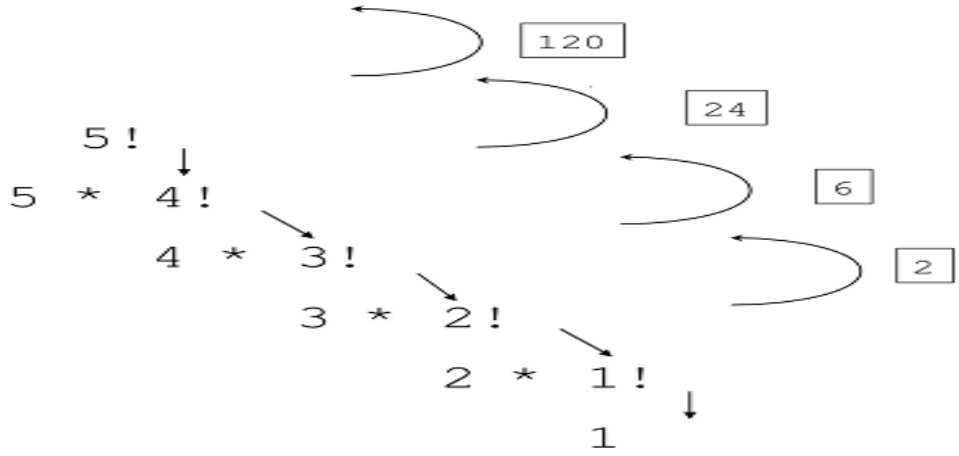
- Multiple Recursion

Linear Recursion:

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.

Example1: Finding the factorial of a number.

```
fact(int f)
{
   if (f == 1) return 1;
   return (f * fact(f - 1)); //called in function only once
}
int main()
{
   int fact;
   fact = fact(5);
   printf("Factorial is %d", fact);
   return 0;
}
```



Binary Recursion:

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function fib provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule:

```
fib(n) = 0  if n is 0,
       = 1  if n is 1,
       = fib(n-1) + fib(n-2) otherwise
```

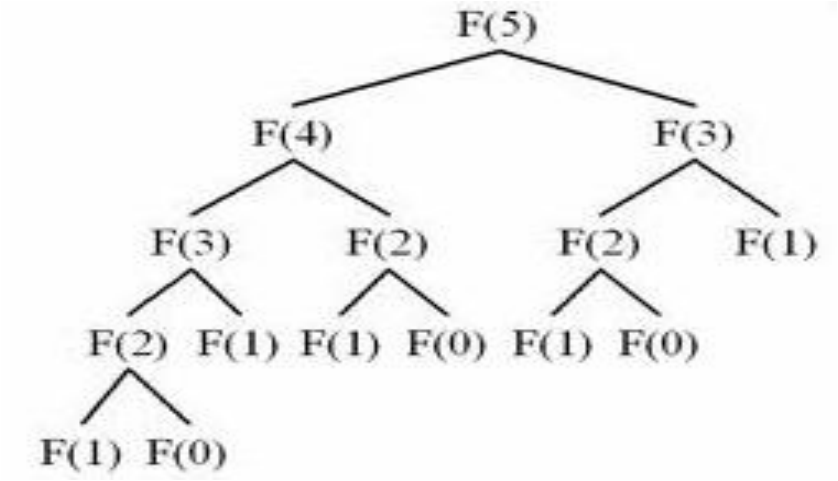
For example, the first seven Fibonacci numbers are

```
Fib(0) = 0
Fib(1) = 1
Fib(2) = Fib(1) + Fib(0) = 1
Fib(3) = Fib(2) + Fib(1) = 2
Fib(4) = Fib(3) + Fib(2) = 3
Fib(5) = Fib(4) + Fib(3) = 5
```

Fib(6) = Fib(5) + Fib(4) = 8

This leads to the following implementation in C:

```
int  fib(int n)
{
  if (n == 0)
        return 0;
  if (n == 1)
        return 1;
  return  fib(n - 1) + fib(n - 2);
}
```



Tail Recursion:

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

```
int gcd(int m, int n)
{
        int r;
        if (m < n) return gcd(n,m);
        r = m%n;
        if (r == 0) return(n);
        else return(gcd(n,r));
}
```

Example: Convert the following tail-recursive function into an iterative function:

```
int pow(int a, int b)
{
        if (b==1) return a;
        else return a * pow(a, b-1);
}
int pow(int a, int b)
{
        int i, total=1;
        for(i=0; i<b; i++) total *= a;
        return total;
}
```

Recursive algorithms for Factorial, GCD, Fibonacci Series and Towers of Hanoi:

Factorial(n)

Input: integer n ≥ 0

Output: n!

1. If n = 0 then return (1)
2. else return prod(n, factorial(n − 1))

GCD(m, n)
Input: integers m > 0, n ≥ 0
Output: gcd (m, n)

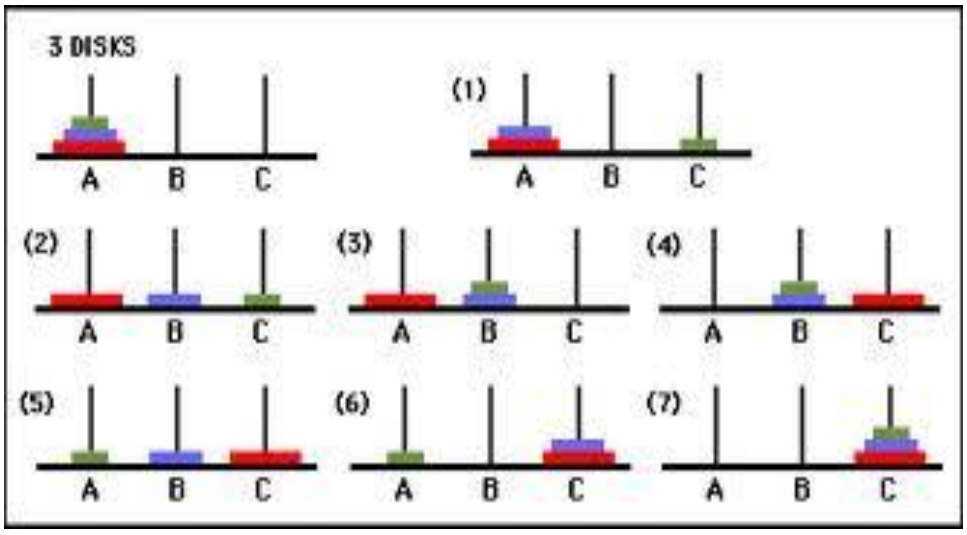
1. If n = 0 then return (m)
2. else return gcd(n,m mod n)

Time-Complexity: O(ln n)
Fibonacci(n)
Input: integer n ≥ 0
Output: Fibonacci Series: 1 1 2 3 5 8 13……..

1. if n=1 or n=2
2. then Fibonacci(n)=1
3. else Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Towers of Hanoi
Input: The aim of the tower of Hanoi problem is to move the initial n different sized disks from needle A to needle C using a temporary needle B. The rule is that no larger disk is to be placed above the smaller disk in any of the needle while moving or at any time, and only the top of the disk is to be moved at a time from any needle to any needle.
Output:

1. If n=1, move the single disk from A to C and return,
2. If n>1, move the top n-1 disks from A to B using C as temporary.
3. Move the remaining disk from A to C.
4. Move the n-1 disk disks from B to C, using A as temporary.



Linear Search:

1. Read search element.
2. Call function linear search function by passing N value, array and search element.
3. If a[i] ==k, return i value, else return -1, returned value is stored in pos.
4. If pos ==-1 print element not found, else print pos+1 value.

Source Code: (Recursive)

```
#include<stdio.h>
#include<conio.h>
void linear_search(int n,int a[20],int i,int k)
{
        if(i>=n)
        {
                printf("%d is not found",k);
                return;
```

```
        }
        if(a[i]==k)
        {
                printf("%d is found at %d",k,i+1);
                return;
        }
        else
                linear_search(n,a,i+1,k);
}
void main()
{
        int i,a[20],n,k;
        clrscr();
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter elements:");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter search element:");
        scanf("%d",&k);
        linear_search(n,a,0,k);
        getch();
}
```

Input & Output:

```
Enter no of elements:3
Enter elements:1   2    3
Enter search element:6
6 is not found

Enter no of elements:5
Enter elements:1  2   3   4   5
Enter search element:3
3 is found at position 3
```

Time Complexity of Linear Search:

If input array is not sorted, then the solution is to use a sequential search.
Unsuccessful search: O(N)
Successful Search: Worst case: O(N)
Average case: O(N/2)

Binary Search:

1. Read search data.
2. Call binary_search function with values N, array, and data.
3. If low is less than high, making mid value as mean of low and high.
4. If a [mid] ==data, make flag=1 and break, else if data is less than a[mid] make high=mid-1,else low=mid+1.
5. If flag ==1, print data found at mid+1, else not found.

Source Code: (Recursive)

```
#include<stdio.h>
#include<conio.h>
void binary_search(int a[20],int data,int low,int high)
{
        int mid;
        if(low<=high)
        {
                mid=(low+high)/2;
                if(a[mid]==data)
                        printf("Data found at %d",mid+1);
                else
```

```
                if(a[mid]>data)
                        binary_search(a,data,low,mid-1);
                else
                        binary_search(a,data,mid+1,high);
        }
}
void main()
{
        int i,a[20],n,data;
        clrscr();
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter elements:");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter search element:");
        scanf("%d",&data);
        binary_search(a,data,0,n-1);
        getch();
}
```

Input & Output:

```
Enter no of elements:3
Enter elements:1   2   3
Enter search element:25
Not found

Enter no of elements:3
Enter elements:1  2   3
Enter search element:3
Data found at 3
```

Time Complexity of Binary Search:

Time Complexity for binary search is $O(\log_2 N)$

Fibonacci Search:

Source Code: (Recursive)

```
#include<stdio.h>
#include<conio.h>
void fib_search(int a[],int n,int search,int pos,int begin,int end)
{
        int fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144};
        if(end<=0)
        {
                printf("\nNot found");
                return;//data not found
        }
        else
        {
                pos=begin+fib[--end];
                if(a[pos]==search && pos<n)
                {
                        printf("\n Found at %d",pos);
                        return;//data found
                }
                if((pos>=n)||(search<a[pos]))
                        fib_search(a,n,search,pos,begin,end);
                else
                {
                        begin=pos+1;
```

```
                        end--;
                        fib_search(a,n,search,pos,begin,end);
                }
        }
}
void main()
{
        int n,i,a[20],search,pos=0,begin=0,k=0,end;
        int fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144};
        clrscr();
        printf("Enter the n:");
        scanf("%d",&n);
        printf("Enter elements to array:");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter the search element:");
        scanf("%d",&search);
        while(fib[k]<n)
        {
                k++;
        }
        end=k;
        printf("Max.no of passes : %d",end);
        fib_search(a,n,search,pos,begin,end);
        getch();
}
```

Input & Output:

```
Enter the n:5
Enter elements to array:1  2   3   6   59
Enter the search element:56
Max no of passes required is : 5
Search element not found.....
```

Time Complexity of Fibonacci Search:

Time complexity for Fibonacci search is $O(\log_2 N)$

LINKED LIST

Introduction to Linked List:

A **linked list** is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**.Each **node** is divided into two parts:

1. The first part contains the **information** of the element and
2. The second part contains the address of the next node (**link /next pointer field**) in the list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Linked lists using dynamic variables:

1. In array implementation of the linked lists a fixed set of nodes represented by an array is established at the beginning of the execution
2. A pointer to a node is represented by the relative position of the node within the array.
3. In array implementation, it is not possible to determine the number of nodes required for the linked list. Therefore;
    a. Less number of nodes can be allocated which means that the program will have overflow problem.
    b. More number of nodes can be allocated which means that some amount of the memory storage will be wasted.
4. The solution to this problem is to allow nodes that are ***dynamic***, rather than static.
5. When a node is required storage is reserved /allocated for it and when a node is no longer needed, the memory storage is released /freed.

Allocating and freeing dynamic variables:

1. C library function **malloc()** is used for dynamically allocating a space to a pointer. Note that the malloc() is a library function in <stdlib.h> header file.
2. The following lines allocate an integer space from the memory pointed by the pointer p.
   int *p;
   p = (int *) malloc(sizeof(int));

Note that sizeof() is another library function that returns the number of bytes required for the operand. In this example, 4 bytes for the int.

Advantages of linked lists:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.

2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	**LINKED LIST**
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.

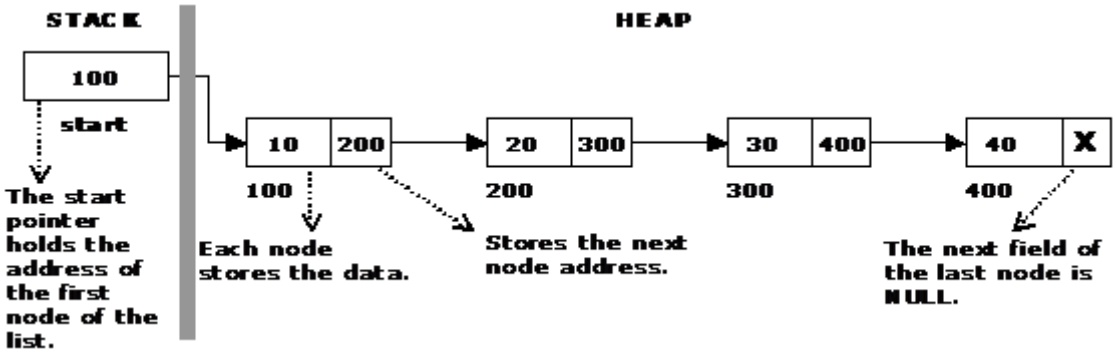
3. Linked lists are to implement stack, queue, trees and graphs.

4. Implement the symbol table in compiler construction.

Single Linked List:

The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.



The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Single Linked List

Implementation of Single Linked List:

1. Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
2. Initialize the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Advantages of singly linked list:

1. Dynamic data structure.
2. We can perform deletion and insertion anywhere in the list.
3. We can merge two lists easily.

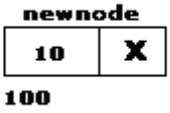
Disadvantages of singly linked list:

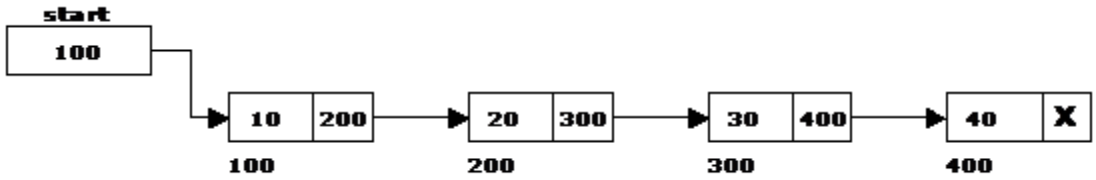
1. Backward traversing is not possible in singly linked list.
2. Insertion is easy but deletion take some additional time, because disadvantage of backward traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.

```
node* getnode()
{
     node* newnode;
     newnode = (node *) malloc(sizeof(node));
     printf("\n Enter data: ");
     scanf("%d", &newnode -> data);
     newnode -> next = NULL;
     return newnode;
}
```





Insertion of a Node:

The new node can then be inserted at three different places namely:

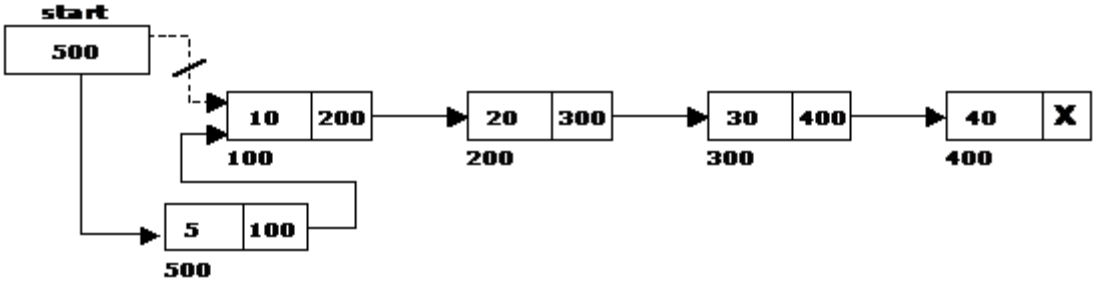
1. Inserting a node at the beginning.
2. Inserting a node at the end.
3. Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty then *start = newnode.*
3. If the list is not empty, follow the steps given below:

   newnode -> next = start;
   start = newnode;



Inserting a node at the end:

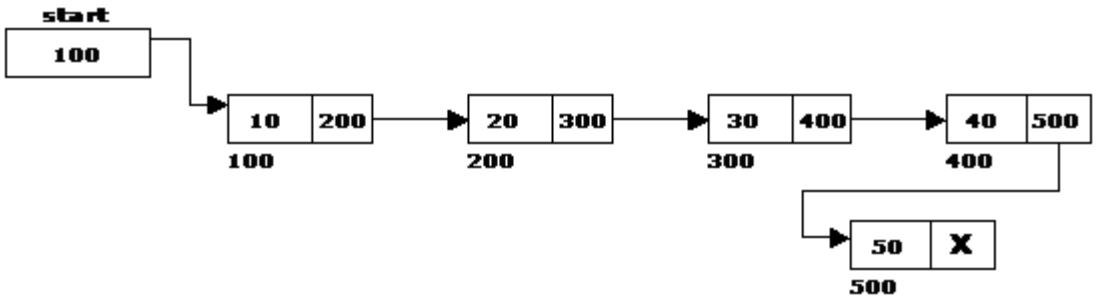
1. The following steps are followed to insert a new node at the end of the list:
2. Get the new node using getnode()

   newnode = getnode();

3. If the list is empty then *start = newnode*.
4. If the list is not empty follow the steps given below:

   temp = start;
   while(temp -> next != NULL)
   1. temp = temp -> next;

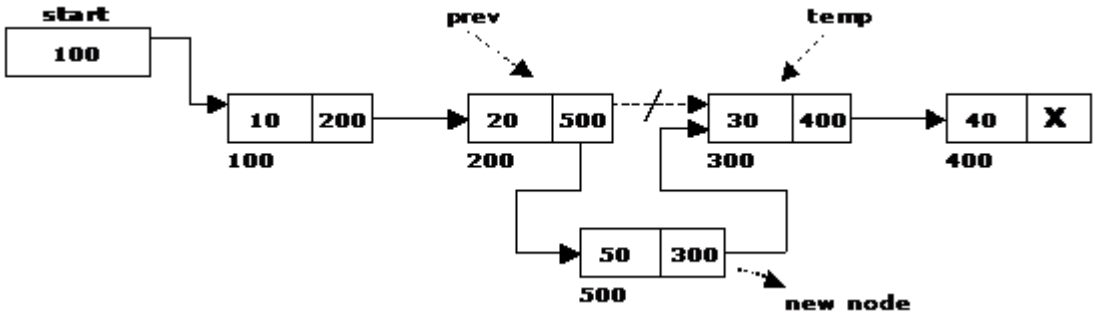
   temp -> next = newnode;



Inserting a node at intermediate position:

1. The following steps are followed, to insert a new node in an intermediate position in the list:
2. Get the new node using getnode().
   newnode = getnode();
3. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
4. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
5. After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```



Deletion of a node:

A node can be deleted from the list from three different places namely.

1. Deleting a node at the beginning.
2. Deleting a node at the end.
3. Deleting a node at intermediate position.

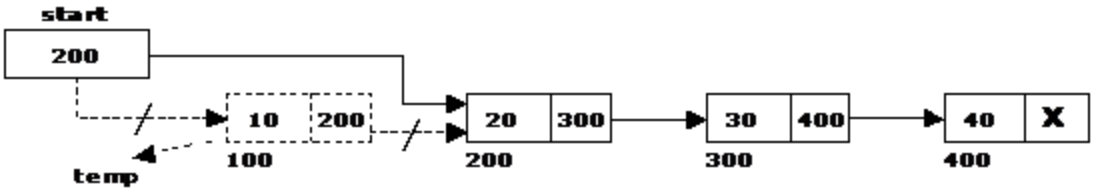
Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.

2. If the list is not empty, follow the steps given below:

   i. temp = start;
   ii. start = start -> next;
   iii. free(temp);



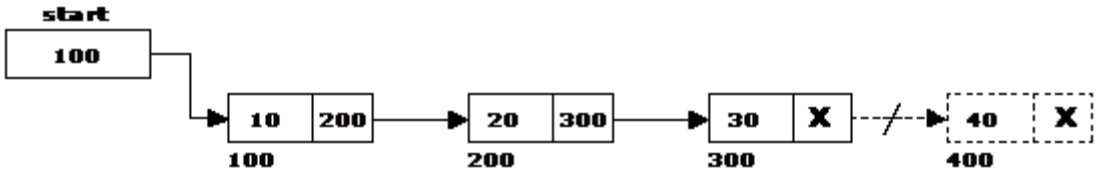
Deleting a node at the end:

1. The following steps are followed to delete a node at the end of the list:

2. If list is empty then display 'Empty List' message.

3. If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
    1. prev = temp;
    2. temp = temp -> next;
}

prev -> next = NULL;
free(temp);
```



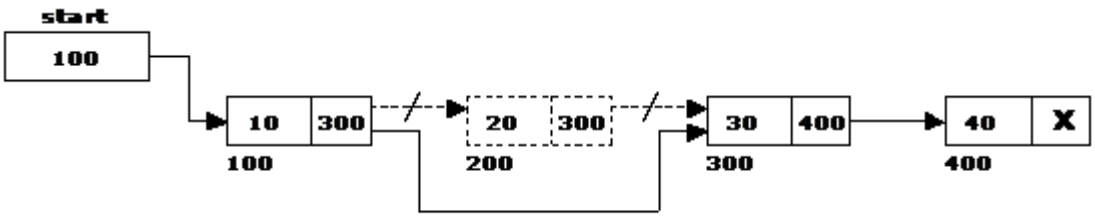
Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

1. If list is empty then display 'Empty List' message

2. If the list is not empty, follow the steps given below.

```
if(pos > 1 && pos < nodectr)
{
        temp = prev = start;
        ctr = 1;
        while(ctr < pos)

                prev = temp;
                temp = temp -> next;
                ctr++;
        }
        prev -> next = temp -> next;
        free(temp);
        printf("\n node deleted..");
}
```



Traversal and displaying a list (Left to Right):

Traversing a list involves the following steps:

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

Counting the Number of Nodes:

```
int countnode(node *st)
{
        if(st == NULL)
                return 0;
        else
                return(1 + countnode(st -> next));
}
```

Source Code:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct linklist

{

        int data;

        struct linklist *next;

};

typedef struct linklist node;

node *start=NULL;

int menu()

{

        int ch;

        printf("\n\t *****IMPLIMENTATION OF SINGLE LINKED LIST*****");

        printf("\n\t ---------------------------------------\n");

        printf("\n\t 1.Create list");

        printf("\n\t 2.Traverse the list(left to right)");

        printf("\n\t 3.Traverse the list(right to left)");

        printf("\n\t 4.Number of nodes");

        printf("\n\t 5.Insertion at Begining");

        printf("\n\t 6.Insertion at End");

        printf("\n\t 7.Insertion at Middle");

        printf("\n\t 8.Deletion at Beginning");

        printf("\n\t 9.Deletion at End");

        printf("\n\t 10.Deletion at Middle");
```

```
        printf("\nEnter your choice:");
        scanf("%d",&ch);
        return ch;
}
node* getnode()
{
        node *newnode;
        newnode=(node*)malloc(sizeof(node));
        printf("Enter data:\n");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        return newnode;
}
int countnode(node*start)
{
        if(start==NULL)
                return 0;
        else
                return 1+countnode(start->next);
}

void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i=0;i<n;i++)
        {
                newnode=getnode();
                if(start==NULL)
```

```
                {
                        start=newnode;
                }
                else
                {
                        temp=start;
                        while(temp->next!=NULL)
                                temp=temp->next;
                        temp->next=newnode;
                }
        }
}
void traverse()
{
        node *temp;
        temp=start;
        printf("The contents of the list(left to right)\n");
        if(start==NULL)
                {
                        printf("\n Empty list");
                        return;
                }
        else
        {
                while(temp!=NULL)
                {
                        printf("%d\t",temp->data);
                        temp=temp->next;
                }
        }
```

```
        printf("X");
}
void rev_traverse(node *start)
{
        if(start==NULL)
                {
                        return;
                }
        else
        {
                rev_traverse(start->next);
                printf("%d\t",start->data);
        }
}
void insert_at_beg()
{
        node *newnode;
        newnode=getnode();
        if(start==NULL)
        {
                start=newnode;
        }
        newnode->next=start;
        start=newnode;
}
void insert_at_end()
{
        node *newnode,*temp;
        newnode=getnode();
        if(start==NULL)
```

```
        {
                start=newnode;
        }
        else
        {
                temp=start;
                while(temp->next!=NULL)
                        temp=temp->next;
                temp->next=newnode;
        }

}
void insert_at_mid()
{
        node *newnode,*pre,*temp;
        int pos,ctr=1,nodectr;
        printf("Enter position:");
        scanf("%d",&pos);
        nodectr=countnode(start);
        if(pos>1 && pos<nodectr)
        {

                newnode=getnode();
                temp=pre=start;
                while(ctr<pos)
                {
                        pre=temp;
                        temp=temp->next;
                        ctr++;
                }
```

```
                pre->next=newnode;
                newnode->next=temp;


        }
        else
        {

                printf("\nNot a middle position");
        }
}
void del_at_beg()
{
        node *temp;
        if(start==NULL)
        {
                printf("List is empty");
                return;
        }
        else
        {
                temp=start;
                start=temp->next;
                free(temp);
                printf("Node is deleted");
        }
}
void del_at_end()
{
        node *pre,*temp;
        if(start==NULL)
```

```
        {
                printf("List is empty");
                return;
        }
        else
        {
                temp=start;
                while(temp->next!=NULL)
                {
                        pre=temp;
                        temp=temp->next;
                }
                pre->next=NULL;
                free(temp);
                printf("\Node deleted");
        }
}
void del_at_mid()
{
        int pos,ctr=1,nodectr;
        node *temp,*pre;
        nodectr=countnode(start);
        if(start==NULL)
        {
                printf("List is empty");
                return;
        }
        else
        {
                printf("Enter position:");
```

```
                scanf("%d",&pos);
                if(pos>1 && pos<nodectr)
                {
                        pre=temp=start;
                        while(ctr<pos)
                        {
                                pre=temp;
                                temp=temp->next;
                                ctr++;
                        }
                        pre->next=temp->next;
                        free(temp);
                }
                else
                printf("Not a mid position");
        }
}
void main(void)
{
        int ch,n;
        clrscr();
        while(1)
        {
                ch=menu();
                switch(ch)
                {
                        case 1:
                                if(start==NULL)
                                {
                                        printf("Enter the number of nodes you want to create:");
```

```
                scanf("%d",&n);
                createlist(n);
                printf("List is created");
                break;
        }
        else
        {
                printf("List is already created:");
                break;
        }
case 2:traverse();
                break;
case 3:
                printf("The contents of the list(left to right):\n");
                rev_traverse(start);
                printf("X");
                break;
case 4:  printf("Number of nodes:%d",countnode(start));
                break;
case 5:  insert_at_beg();
                break;
case 6:  insert_at_end();
                break;
case 7:  insert_at_mid();
                break;
case 8:  del_at_beg();
                break;
case 9:  del_at_end();
                break;
case 10:        del_at_mid();
```

```
                                        break;
                        case 11:        exit(0);
                }
        }
}
```

Output:

```
*****IMPLIMENTATION OF LINKED LIST*****
                        ---------------------------------------

                        1.Create list
                        2.Traverse the list(left to right)
                        3.Traverse the list(right to left)
                        4.Number of nodes
                        5.Insertion at Beginning
                        6.Insertion at End
                        7.Insertion at Middle
                        8.Deletion at Beginning
                        9.Deletion at End
                        10.Deletion at Middle
Enter your choice:1
Enter the number of nodes you want to create:5
Enter data:
1
Enter data:
2
Enter data:
3
Enter data:
4
Enter data:
```

5

List is created

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1 2 3 4 5 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:3

The contents of the list(left to right):

5 4 3 2 1 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:4

Number of nodes:5

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:5

Enter data:

0

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0 1 2 3 4 5 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:6

Enter data:

6

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0 1 2 3 4 5 6 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:7

Enter position:4

Enter data:

7

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0 1 2 7 3 4 5 6 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:8

Node is deleted

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1 2 7 3 4 5 6 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:9

Node deleted

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1 2 7 3 4 5 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:10

Enter position:3

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1 2 3 4 5 X

*****IMPLIMENTATION OF LINKED LIST*****

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:11

Merging two single linked lists into one list:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct slinkedlist
{
        int data;
        struct slinkedlist *next;
};
typedef struct slinkedlist node;
node *start=NULL;
int nodectr;
node *getnode()
```

```
{
        node *newnode;
        newnode=(node *)malloc(sizeof(node));
        printf("\n Enter data:");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        return newnode;
}
node * createlist(node * start,int n)
{
        node *temp,*newnode;
        int i;

        for(i=0;i<n;i++)
        {
                newnode=getnode();
                if(start==NULL)
                {
                        start=newnode;
                }
                else
                {
                        temp=start;
                        while(temp->next!=NULL)
                        {
                                temp=temp->next;
                        }
                        temp->next=newnode;
                }
        }
```

```
        return start;
}
void merge(node * start1,node *start2)
{
        node *temp;
        temp=start1;
        while(temp->next!=NULL)
                temp=temp->next;
        temp->next=start2;
        temp=start1;
        while(temp!=NULL)
        {
                printf("%5d",temp->data);
                temp=temp->next;
        }
}
int menu()
{
        int ch;
        printf("\nMerging two Single linked lists");
        printf("\n**********************");
        printf("\n1.Create lists");
        printf("\n2.Mergelists");
        printf("\n3.Exit");
        printf("\n Enter your choice:");
        scanf("%d",&ch);
        return ch;
}
void main()
{
```

```
        int n,ch;
        node * start1=NULL;node * start2=NULL;
        do
        {
                ch=menu();
                switch (ch)
                {
                        case 1:printf("Enter number of nodes you want:");
                                        scanf("%d",&n);
                                        start1=createlist(start,n);
                                        printf("Enter number of nodes you want:");
                                        scanf("%d",&n);
                                        start2=createlist(start,n);
                                        break;
                        case 2:merge(start1,start2);
                                        break;
                }
        }while(ch!=3);
        getch();
}
```

Output:

Merging two Single linked lists

****

1.Creat list 1

2.Create list 2

3.Mergelists

4.Exit

1. Enter your choice:1

Enter number of nodes you want:3

Enter data:1

Enter data:2

Enter data:3

2. Enter your choice:2

Enter number of nodes you want:4

Enter data:4

Enter data:5

Enter data:6

Enter data:7

3. Enter your choice:3

1 2 3 4 5 6 7

Merging two Single linked lists

Reversing a Single Linked List:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *link;
};
typedef struct node node;
node *start=NULL;
```

```
node * getnode()
{
        node * ptr;
        ptr=(node *)malloc(sizeof(node));
        printf("Enter data:\n");
        scanf("%d",&ptr->data);
        ptr->link=NULL;
        return ptr;
}
void create_ll(int N)
{
        node *ptr,*t;
        int i;
        printf("Enter the number of nodes you want:\n");
        scanf("%d",&N);
                for(i=0;i<N;i++)
                {
                        ptr=getnode();
                                if(start==NULL)
                                {
                                        start=ptr;
                                }
                                else
                                {
                                        t=start;
                                        while(t->link!=NULL)
                                        {
                                                t=t->link;
                                        }
                                        t->link=ptr;
```

```
                                        }
                        }


}
void reverse_ll(node *start)
{
        if(start==NULL)
        {
                return;
        }
        else
        {
                reverse_ll(start->link);
                printf("%d ",start->data);
        }
}



int menu()
{
        int ch;
        printf("1.Create list\n");
        printf("2.Reverse the list\n");
        printf("3.Exit\n");
        printf("Enter your choice:");
        scanf("%d",&ch);
        return ch;
}

void main()
```

```
{
        int n,ch;
        while(1)
        {
                ch=menu();
                switch(ch)
                {
                        case 1:
                                        create_ll(n);
                                        printf("List created\n");
                                        break;
                        case 2:
                                         printf("The list in reverse order:");
                                         reverse_ll(start);
                                         printf("\n");
                                         break;
                        case 3:
                                         exit(0);
                }
        }
}
```

Output:

```
1.Create list
2.Reverse the list
3.Exit
Enter your choice:1
Enter the number of nodes you want:
3
Enter data:
12
```

Enter data:

56

Enter data:

33

List created

1.Create list

2.Reverse the list

3.Exit

Enter your choice:2

The list in reverse order:33 56 12

1.Create list

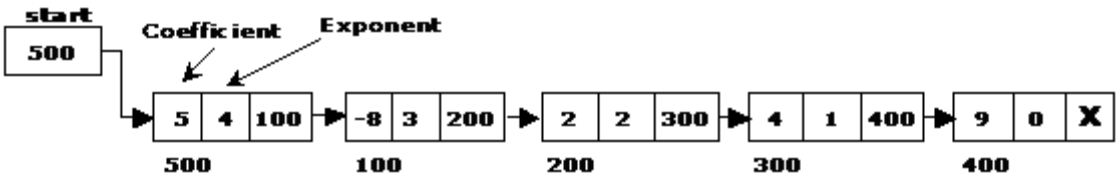
2.Reverse the list

3.Exit

Enter your choice:3

Applications of Single Linked List to Represent Polynomial Expressions:

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure ----



Source Code:

```
#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<malloc.h>

struct link

{

        float coef;

        int expo;
```

```
        struct link *next;
};
typedef struct link node;
node *getnode()
{
        node *temp;
        temp=(node*)malloc(sizeof(node));
        printf("Enter coefficient:");
        fflush(stdin);
        scanf("%f",&temp->coef);
        printf("Enter exponent:");
      fflush(stdin);
        scanf("%d",&temp->expo);
        temp->next=NULL;
        return temp;
}
node *create_poly(node *p)
{
        char ch;
        node *temp,*newnode;
        while(1)
        {
                printf("Do you want polynomial node(y/n):\n");
                ch=getch();
                if(ch=='n')
                        break;
                newnode=getnode();
                if(p==NULL)
                        p=newnode;
                else
```

```
                {
                        temp=p;
                        while(temp->next!=NULL)
                                temp=temp->next;
                        temp->next=newnode;
                }
        }
        return p;
}
void display(node *p)
{
        node *t=p;
        while(t!=NULL)
        {
                printf("+%.f",t->coef);
                printf("x^%d",t->expo);
                t=t->next;
        }
}
void main()
{
        node *poly1=NULL;
        Printf ("Enter First Polynomial..(in ascending order of exponent)\n");
        poly1=create_poly(poly1);
        printf("Polynomial 1:");
        display(poly1);
}
```

Output:

```
Enter First Polynomial..(in ascending order of exponent)

Do you want polynomial node(y/n):Y
```

Enter coefficient:12

Enter exponent:3

Do you want polynomial node(y/n):Y

Enter coefficient:45

Enter exponent:4

Do you want polynomial node(y/n):N

Polynomial 1:+12x^3+45x^4

Sparse Matrix Manipulation:

A **sparse matrix** is a matrix populated primarily with zeros as elements of the table.

Example of sparse matrix

[11 22 0 0 0 0 0]
[0 33 44 0 0 0 0]
[0 0 55 66 77 0 0]
[0 0 0 0 0 88 0]
[0 0 0 0 0 0 99]

The above sparse matrix contains only 9 nonzero elements of the 35, with 26 of those elements as zero. The basic data structure for a matrix is a two-dimensional array. Each entry in the array represents an element $a_{i,j}$ of the matrix and can be accessed by the two indices i and j. Traditionally, i indicates the row number (top-to-bottom), while j indicates the column number (left-to-right) of each element in the table. For an m×n matrix, enough memory to store up to (m×n) entries to represent the matrix is needed.

Source Code:

```
#include<stdio.h>

void main()

{

        int matrix[20][20],m,n,total_elements,total_zeros=0,i,j;

        printf("Enter number of Rows and Columns:");

        scanf("%d %d",&m,&n);

        total_elements=m*n;

        printf("Enter data for Sparse matrix\n");

        for(i=0;i<m;i++)

        {

                for(j=0;j<n;j++)
```

```
                {
                        scanf("%d",&matrix[i][j]);
                        if(matrix[i][j]==0)
                        {
                                total_zeros++;
                        }
                }
        }
        if(total_zeros>total_elements/2)
        {
                printf("Given Matrix is a Sparse matrix\n");
                printf("The Representation of Sparse matrix\n");
                printf("Row\tColumn\tValue\n");
                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                if(matrix[i][j]!=0)
                                {
                                        printf(" %d \t  %d   \t %d\n",i,j,matrix[i][j]);
                                }
                        }
                }
        }
        else
                Printf ("Given matrix is not a Sparse matrix...”);
}
```

Output:

```
Enter number of Rows and Columns:3
3
```

Enter data for Sparse matrix

2

0

0

0

0

3

0

10

0

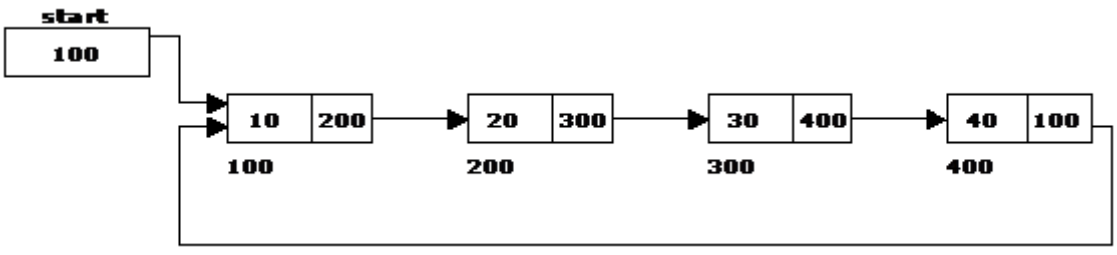
Given Matrix is a Sparse matrix

The Representation of Sparse matrix

Row	Column	Value
0	0	2
1	2	3
2	1	10

Circular Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list.



The basic operations in a circular single linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;

3. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

4. Repeat the above steps 'n' times.

```
newnode -> next = start;
```

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().

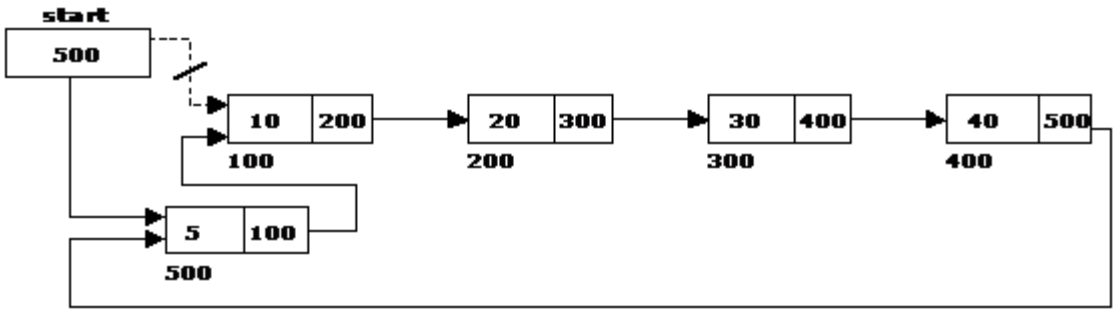
```
newnode = getnode();
```

2. If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

3. If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)

last = last -> next;

newnode -> next = start;

start = newnode;

last -> next = start;
```



Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode().

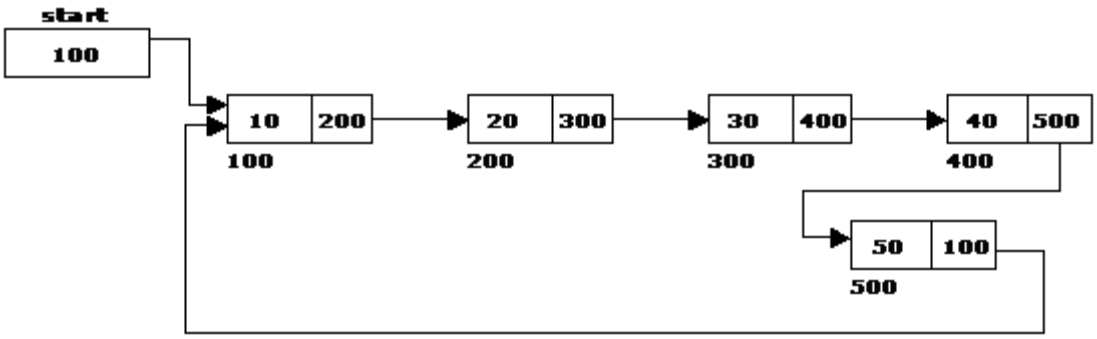
```
newnode = getnode();
```

2. If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

3. If the list is not empty follow the steps given below:

```
temp = start;

while(temp -> next != start)

        temp = temp -> next;

temp -> next = newnode;

newnode -> next = start;
```

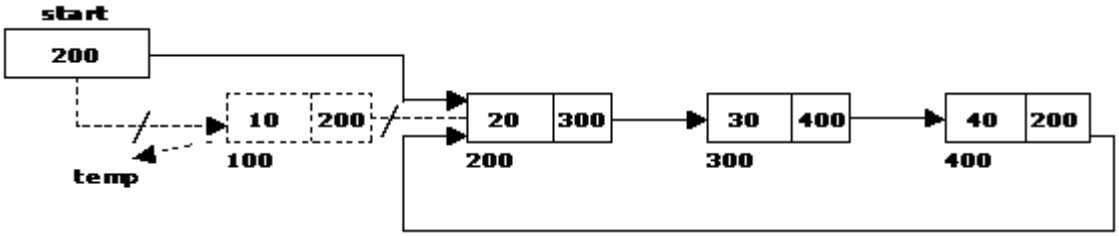


Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
last = temp = start;

while(last -> next != start)

        last = last -> next;

start = start -> next;

last -> next = start;
```



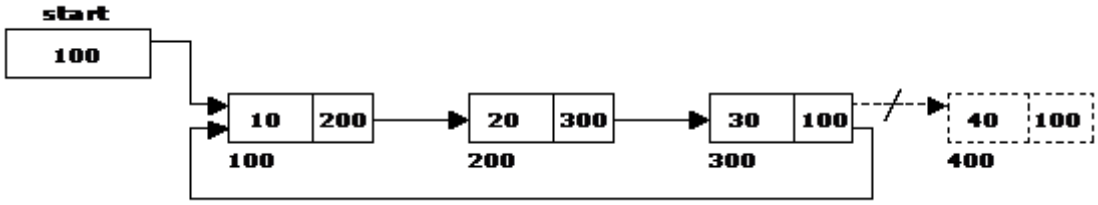
Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:

```
temp = start;

prev = start;

while(temp -> next != start)

{

prev = temp;

temp = temp -> next;

}

prev -> next = start;
```

3. After deleting the node, if the list is empty then start = NULL.



Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;

do

{



        temp = temp -> next;

} while(temp != start);
```

Source Code:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct clinklist

{

        int data;

        struct clinklist *next;

};

typedef struct clinklist node;

node *start=NULL;

int nodectr;

int menu()

{

        int ch;

        printf("\n\t *****IMPLIMENTATION OF CIRCULAR LINKED LIST*****");

        printf("\n\t ---------------------------------------\n");

        printf("\n\t 1.Create list");
```

```
        printf("\n\t 2.Dispaly the contents");
        printf("\n\t 3.Number of nodes");
        printf("\n\t 4.Insertion at Begining");
        printf("\n\t 5.Insertion at End");
        printf("\n\t 6.Insertion at Middle");
        printf("\n\t 7.Deletion at Beginning");
        printf("\n\t 8.Deletion at End");
        printf("\n\t 9.Deletion at Middle");
        printf("\nEnter your choice:");
        scanf("%d",&ch);
        return ch;
}
node *getnode()
{
        node *newnode;
        newnode=(node *)malloc(sizeof(node));
        printf("Enter data:\n");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        return newnode;
}
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        nodectr=n;
        for(i=0;i<n;i++)
        {
                newnode=getnode();
```

```
		if(start==NULL)
		{
			start=newnode;
		}
		else
		{
			temp=start;
			while(temp->next!=NULL)
				temp=temp->next;
			temp->next=newnode;
		}
	}
	newnode->next=start;
}
void display()
{
	node *temp;
	temp=start;
	printf("The contents of the list:\n");
	if(start==NULL)
		{
			printf("\n Empty list");
			return;
		}
	else
	{
		do
		{
			printf("%d\t",temp->data);
			temp=temp->next;
```

```
                }while(temp!=start);
        }
        printf("X");
}

void insert_at_beg()
{
        node *newnode,*last;
        newnode=getnode();
        if(start==NULL)
        {
                start=newnode;
        }
        else
        {
                last=start;
                while(last->next!=start)
                        last=last->next;
                newnode->next=start;
                start=newnode;
                last->next=start;
        }
                printf("\nNode is inserted ");
                nodectr++;
}
void insert_at_end()
{
        node *newnode,*temp;
        newnode=getnode();
        if(start==NULL)
```

```
        {
                start=newnode;
        }
        else
        {
                temp=start;
                while(temp->next!=start)
                        temp=temp->next;
                temp->next=newnode;
                newnode->next=start;
        }
   printf("\nNode is inserted ");
                nodectr++;

}
void insert_at_mid()
{
        node *newnode,*pre,*temp;
        int pos,ctr=1;
        printf("Enter position:");
        scanf("%d",&pos);
        if(pos>1 && pos<nodectr)
        {

                newnode=getnode();
                temp=pre=start;
                while(ctr<pos)
                {
                        pre=temp;
                        temp=temp->next;
```

```
                ctr++;
            }
            pre->next=newnode;
            newnode->next=temp;
            printf("\nNode is inserted ");
            nodectr++;
        }
        else
        {

                printf("\nNot a middle position");
        }
}
void del_at_beg()
{
        node *temp,*last;
        if(start==NULL)
        {
            printf("List is empty");
            return;
        }
        else
        {
            last=temp=start;
            while(last->next!=start)
                last=last->next;
            start=temp->next;
            last->next=start;
            free(temp);
            nodectr--;
```

```
                printf("Node is deleted");
                if(nodectr==0)
                        start=NULL;
        }
}
void del_at_end()
{
        node *temp,*pre;
        if(start==NULL)
        {
                printf("List is empty");
                return;
        }
        else
        {
                temp=start;
                pre=start;
                while(temp->next!=start)
                {

                        pre=temp;
                        temp=temp->next;
                }
                pre->next=NULL;
                free(temp);
                nodectr--;
                printf("Node is deleted");
                if(nodectr==0)
                        start=NULL;
        }
```

```
}
void del_at_mid()
{
        int pos,ctr=0;
        node *temp,*pre;
        if(start==NULL)
        {
                printf("List is empty");
                return;
        }
        else
        {
                printf("Enter position:");
                scanf("%d",&pos);
                if(pos>1 && pos<nodectr)
                {
                        pre=temp=start;
                        while(ctr<pos-1)
                        {
                                pre=temp;
                                temp=temp->next;
                                ctr++;
                        }
                        pre->next=temp->next;
                        free(temp);
                        nodectr--;
                        printf("\nNode deleted");
                }
                else
                printf("Not a mid position");
```

```
        }
}
void main(void)
{
        int ch,n;
        clrscr();
        while(1)
        {
                ch=menu();
                switch(ch)
                {
                        case 1:if(start==NULL)
                                {
                                        printf("Enter the number of nodes you want to create:");
                                        scanf("%d",&n);
                                    createlist(n);
                                        printf("List is created");
                                        break;
                                }
                                else

                                        printf("List is already created:");break;

                        case 2: display();break;
                        case 3: printf("Number of nodes:%d",nodectr);break;
                        case 4: insert_at_beg();break;
                        case 5: insert_at_end();break;
                        case 6: insert_at_mid();break;
                        case 7: del_at_beg();break;
                        case 8: del_at_end();break;
```

```
                    case 9: del_at_mid();break;
                    case 10: exit(0);
                }
            }
}
```

Output:

```
*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****
    ---------------------------------------

    1.Create list
    2.Dispaly the contents
    3.Number of nodes
    4.Insertion at Begining
    5.Insertion at End
    6.Insertion at Middle
    7.Deletion at Beginning
    8.Deletion at End
    9.Deletion at Middle
Enter your choice:1
Enter the number of nodes you want to create:5
Enter data:
2
Enter data:
3
Enter data:
4
Enter data:
6
Enter data:
7
```

List is created

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:2

The contents of the list:

2 3 4 6 7 X

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:3

Number of nodes:5

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:4

Enter data:

1

Node is inserted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:2

The contents of the list:

1 2 3 4 6 7 X

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:5

Enter data:

8

Node is inserted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:2

The contents of the list:

1 2 3 4 6 7 8 X

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:6

Enter position:5

Enter data:

5

Node is inserted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:2

The contents of the list:

1 2 3 4 5 6 7 8 X

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:7

Node is deleted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:8

Node is deleted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:9

Enter position:7

Not a mid position

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:3

Number of nodes:6

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:9

Enter position:3

Node deleted

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:9

Enter position:7

Not a mid position

*****IMPLIMENTATION OF CIRCULAR LINKED LIST*****

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:9

Enter position:3

Node deleted

Double Linked List:

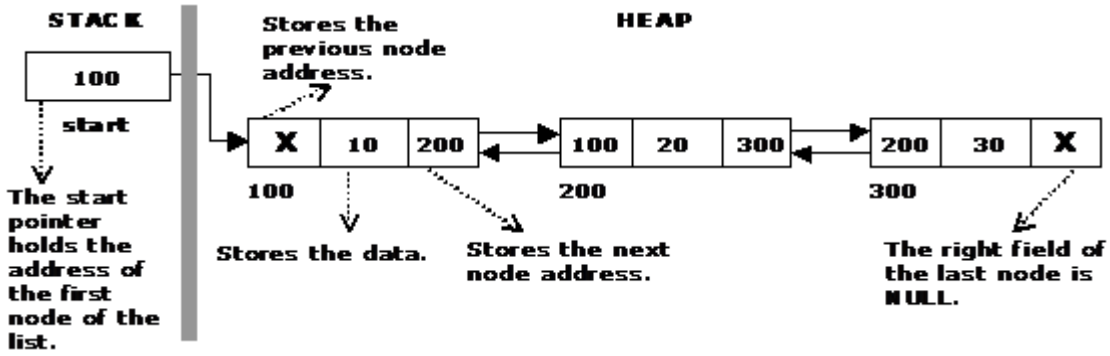
Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

1. Left link.
2. Data.
3. Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. The basic operations in a double linked list are:

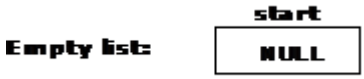
1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.



The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

node: | left | data | right |



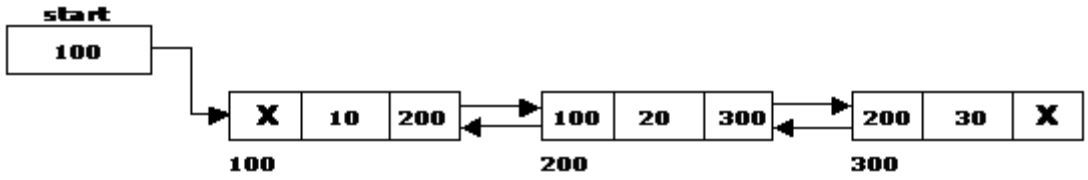
Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function.

Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().
   newnode =getnode();
2. If the list is empty then *start = newnode*.
3. If the list is not empty, follow the steps given below:
   i. The left field of the new node is made to point the previous node.
   ii. The previous nodes right field must be assigned with address of the new node.
4. Repeat the above steps 'n' times.



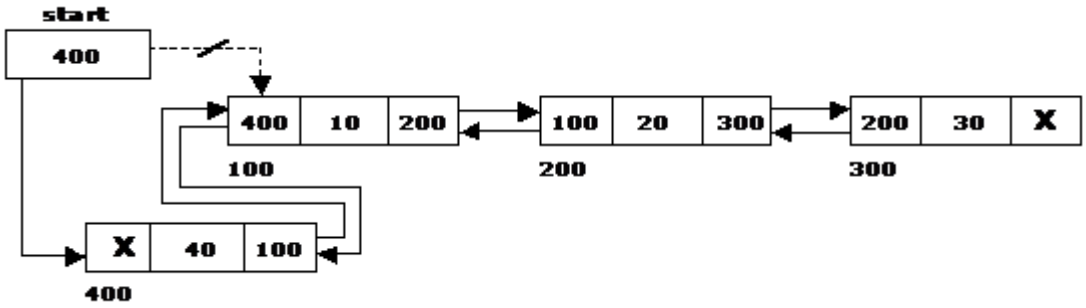
Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode().
   newnode=getnode();
2. If the list is empty then *start = newnode*.
3. If the list is not empty, follow the steps given below:
   newnode -> right = start;

   start -> left = newnode;

   start = newnode;

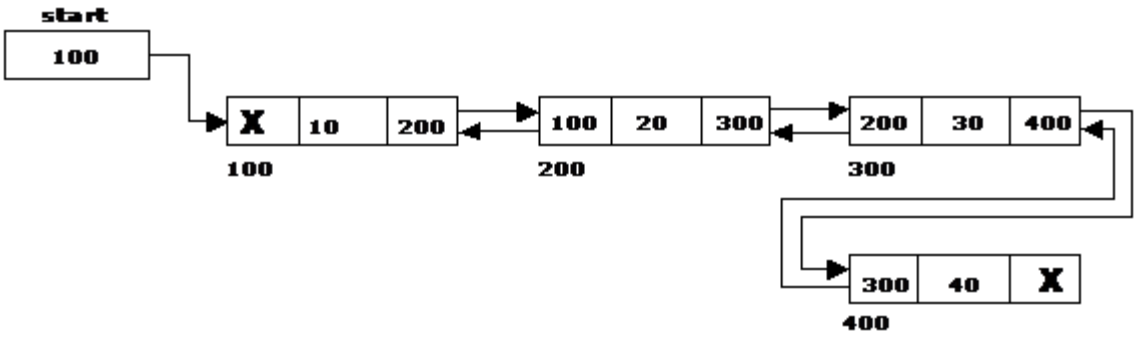


Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode()
   newnode=getnode();
2. If the list is empty then *start = newnode*.
3. If the list is not empty follow the steps given below:

```
temp = start;

while(temp -> right != NULL)

        temp = temp -> right;

temp -> right = newnode;

newnode -> left = temp;
```

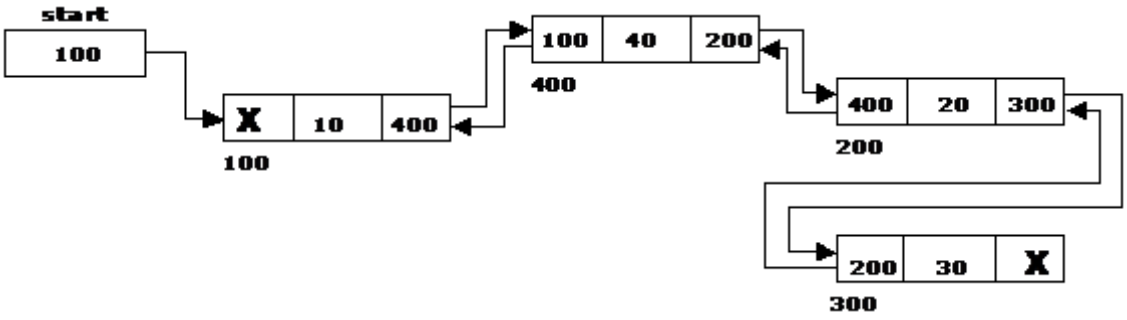


Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

1. Get the new node using getnode().
   newnode=getnode();
2. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
3. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
4. After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;

newnode -> right = temp -> right;

temp -> right -> left = newnode;

temp -> right = newnode;
```

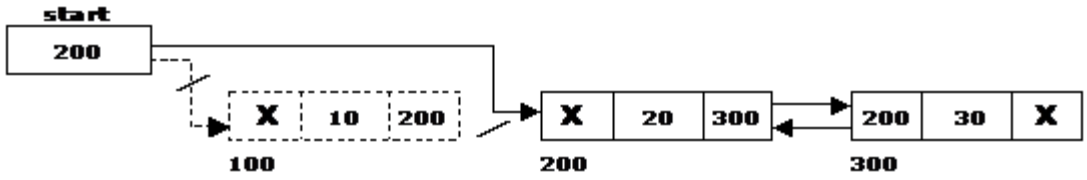


Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;

start = start -> right;

start -> left = NULL;

free(temp);
```

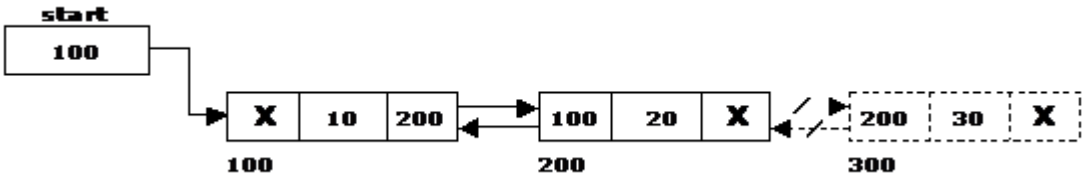


Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below:

```
temp = start;

while(temp -> right != NULL)

{

        temp = temp -> right;

}

temp -> left -> right = NULL;

free(temp);
```



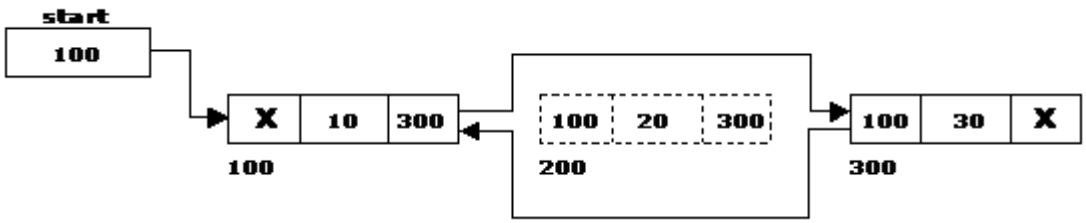
Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   i. Get the position of the node to delete.
   ii. Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
   iii. Then perform the following steps:

```
if(pos > 1 && pos < nodectr)

{

        temp = start;

        i = 1;
```

```
                while(i < pos)
                {
                        temp = temp -> right;
                        i++;
                }
                temp -> right -> left = temp -> left;
                temp -> left -> right = temp -> right;
                free(temp);
                printf("\n node deleted..");
        }
```



Traversal and displaying a list (Left to Right):

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
        temp = start;
        while(temp != NULL)
        {
                print  temp -> data;
                temp = temp -> right;
        }
```

Traversal and displaying a list (Right to Left):

The following steps are followed, to traverse a list from right to left:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
        temp = start;
```

```
        while(temp -> right != NULL)
                temp = temp -> right;
        while(temp != NULL)
        {
                print temp -> data;
                temp = temp -> left;
        }
```

Source Code:

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

struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
```

```
        return newnode;
}

int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return 1 + countnode(start -> right);
}

int menu()
{
        int ch;
        clrscr();
        printf("\n 1.Create");
        printf("\n------------------------------");
        printf("\n 2. Insert a node at beginning ");
        printf("\n 3. Insert a node at end");
        printf("\n 4. Insert a node at middle");
        printf("\n------------------------------");
        printf("\n 5. Delete a node from beginning");
        printf("\n 6. Delete a node from Last");
        printf("\n 7. Delete a node from Middle");
        printf("\n------------------------------");
        printf("\n 8. Traverse the list from Left to Right ");
        printf("\n 9. Traverse the list from Right to Left ");
        printf("\n------------------------------");
        printf("\n 10.Count the Number of nodes in the list");
        printf("\n 11.Exit ");
```

```
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}



void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                        start = newnode;
                else
                {
                        temp = start;
                        while(temp -> right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}

void traverse_left_to_right()
{
```

```
        node *temp;
        temp = start;
        printf("\n The contents of List: ");
        if(start == NULL )
                printf("\n Empty List");
        else
        while(temp != NULL)
        {
                printf("\t %d ", temp -> data);
                temp = temp -> right;
        }
}

void traverse_right_to_left()
{
        node *temp;
        temp = start;
        printf("\n The contents of List: ");
        if(start == NULL)
                printf("\n Empty List");
        else
        while(temp -> right != NULL)
                temp = temp -> right;
        while(temp != NULL)
        {
                printf("\t%d", temp -> data);
                temp = temp -> left;
        }
}
```

```
void dll_insert_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                newnode -> right = start;
                start -> left = newnode;
                start = newnode;
        }
}

void dll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                temp = start;
                while(temp -> right != NULL)
                        temp = temp -> right;
                temp -> right = newnode;
                newnode -> left = temp;
        }
}
```

```
void dll_insert_mid()
{
        node *newnode,*temp;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos - nodectr >= 2)
        {
                printf("\n Position is out of range..");
                return;
        }
        if(pos > 1 && pos < nodectr)
        {
                temp = start;
                while(ctr < pos - 1)
                {
                        temp = temp -> right;
                        ctr++;
                }
                newnode -> left = temp;
                newnode -> right = temp -> right;
                temp -> right -> left = newnode;
                temp -> right = newnode;
        }
        else
                printf("position %d of list is not a middle position ", pos);
}
```

```
void dll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty list");
                getch();
                return ;
        }
        else
        {
                temp = start;
                start = start -> right;
                start -> left = NULL;
                free(temp);
        }
}

void dll_delete_last()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty list");
                getch();
                return ;
        }
        else
        {
                temp = start;
```

```
                while(temp -> right != NULL)
                        temp = temp -> right;
                temp -> left -> right = NULL;
                free(temp);
                temp = NULL;
        }
}

void dll_delete_mid()
{
        int i = 0, pos, nodectr;
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty List");
                getch();
                return;
        }
        else
        {
                printf("\n Enter the position of the node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nthis node does not exist");
                        getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
```

```
                {
                        temp = start;
                        i = 1;
                        while(i < pos)
                        {
                                temp = temp -> right;
                                i++;
                        }
                        temp -> right -> left = temp -> left;
                        temp -> left -> right = temp -> right;
                        free(temp);
                        printf("\n node deleted..");
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch( ch)
                {
```

```
case 1 :
        printf("\n Enter Number of nodes to create: ");
        scanf("%d", &n);
        createlist(n);
        printf("\n List created..");
        break;
case 2 :
        dll_insert_beg();
        break;
case 3 :
        dll_insert_end();
        break;
case 4 :
        dll_insert_mid();
        break;
case 5 :
        dll_delete_beg();
        break;
case 6 :
        dll_delete_last();
        break;
case 7 :
        dll_delete_mid();
        break;
case 8 :
        traverse_left_to_right();
        break;
case 9 :
        traverse_right_to_left();
        break;
```