Chapter 1

Basic Concepts

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is 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. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure shows the classification of data structures.

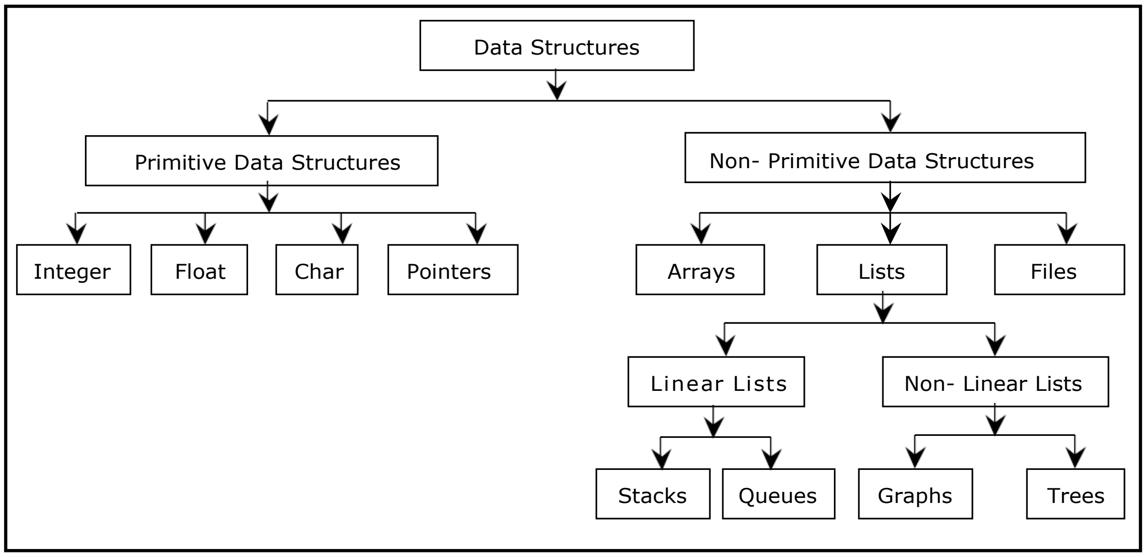


Figure 1. Classification of Data Structures

Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

An *abstract data type* in a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping.

For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is known today as an ADT.

Algorithm

An **a**lgorithm is 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. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are two basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the ***TIME COMPLEXITY*** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion.

Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data.

1.	Best Case	:	The minimum possible value of f(n) is called the best case.
2.	Average Case	:	The expected value of f(n).
3.	Worst Case	:	The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Exercises

1. Define algorithm. (infosys, 2015, 2018, 2019; tcs 2017, 2018, 2019)
2. State the various steps in developing algorithms?
3. State the properties of algorithms.
4. Define efficiency of an algorithm?
5. State the various methods to estimate the efficiency of an algorithm.
6. Define time complexity of an algorithm?
7. Define worst case of an algorithm. (wipro 2017, 2018; Infosys 2017, 2019)
8. Define average case of an algorithm.
9. Define best case of an algorithm. (tcs 2016, 2019; mphasis 2018, 2019)
10. Mention the various spaces utilized by a program
11. Define space complexity of an algorithm.
12. State the different memory spaces occupied by an algorithm.

Multiple Choice Questions

1. ______ is a step-by-step recipe for solving an instance of problem. [A]
   A. Algorithm B. Complexity
   C. Pseudocode D. Analysis

2. ______ is used to describe the algorithm, in less formal language. [C]
   A. Cannot be defined B. Natural Language
   C. Pseudocode D. None

3. ______ of an algorithm is the amount of time (or the number of steps) needed by a program to complete its task. [D]
   A. Space Complexity B. Dynamic Programming
   C. Divide and Conquer D. Time Complexity

4. ______ of a program is the amount of memory used at once by the algorithm until it completes its execution. [C]

   A. Divide and Conquer B. Time Complexity
   C. Space Complexity D. Dynamic Programming

5. ______ is used to define the worst-case running time of an algorithm. [A]

   A. Big-Oh notation C. Complexity
   B. Cannot be defined D. Analysis

Chapter 2

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursions through simpler alternatives are available. It is because recursion is elegant to use through it is costly in terms of time and space. But using it is one thing and getting involved with it is another.

In this unit we will look at "recursion" as a programmer who not only loves it but also wants to understand it! With a bit of involvement it is going to be an interesting reading for you.

Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
#include <stdio.h>
int factorial (int);
main()
{
        int num, fact;
        printf ("Enter a positive integer value: ");
        scanf ("%d", &num);
        fact = factorial (num);
        printf ("\n Factorial of %d =%5d\n", num, fact);
}

int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);

        return (result);
}
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
        int i, result = 1;
        if (n == 0)
```

```
                return (result);
        else
        {
                for (i=1; i<=n; i++)
                        result = result * i;
        }
        return (result);
}
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have a exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

Factorial of a given number:

The operation of recursive factorial function is as follows:

Start out with some natural number N (in our example, 5). The recursive definition is:

n = 0, 0 ! = 1	Base Case
n > 0, n ! = n * (n - 1) !	Recursive Case

Recursion Factorials:

5! =5 * 4! = 5 *___= _____ factr(5) = 5 * factr(4) =

4! = 4 *3! = 4 *___= _____ factr(4) = 4 * factr(3) = _

3! = 3 * 2! = 3 *____= ____ factr(3) = 3 * factr(2) =

2! = 2 * 1! = 2 *____= _____ factr(2) = 2 * factr(1) = _

1 = _

0! = 1 factr(0) = _

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

1! = 1 * 0! = 1 * = factr(1) = . fact(0)

Fibonacci Sequence Problem:

A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is 0 + 1 = 1, fourth number is 1 + 1= 2, fifth number is 1 + 2 = 3 and so on. The sequence of Fibonacci integers is given below:

0 1 1 2 3 5 8 13 21 . . . .

A recursive definition for the Fibonacci sequence of integers may be defined as follows:

Fib (n) = n if n = 0 or n = 1
Fib (n) = fib (n-1) + fib (n-2) for

n >=2 We will now use the definition to

compute fib(5):

```
                          fib(5)
                           /                  \
                    fib(4)                 fib(3)
                /         \                /        \
           fib(3)       fib(2)          fib(2)   fib(1)
         /     \        /     \         /       \
  fib(2)    fib(1)  fib(1) fib(0) fib(1) fib(0)
  /      \
fib(1) fib(0)
```

```
//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

Output:

```
fib(5) is 34
```

Exercises

1. What is the importance of the stopping case in recursive functions? (Infosys 2017,2018,2019; tcs 2018, 2019)

2. Write a function with one positive integer parameter called n. The function will write 2^n-1 integers (where ^ is the exponentiation operation). Here are the patterns of output for various values of n:

   n=1: Output is: 1
   n=2: Output is: 1 2 1
   n=3: Output is: 1 2 1 3 1 2 1
   n=4: Output is: 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

   And so on. Note that the output for n always consists of the output for n-1, followed by n itself, followed by a second copy of the output for n-1.

3. Write a recursive function for the mathematical function:
   f(n) = 1 if n = 1
   f(n) = 2 * f(n-1) if n >= 2

4. Which method is preferable in general? (wipro 2018, 2019; byju's 2019)
   a) Recursive method
   b) Non-recursive method

5. Write a function using Recursion to print numbers from *n* to 0. (magic bricks 2019)
6. Write a function using Recursion to enter and display a string in reverse and state whether the string contains any spaces. Don't use arrays/strings. (wipro 2017, 2019)

7. Write a function using Recursion to check if a number *n* is prime. (You have to check whether *n* is divisible by any number below *n*) (infosys 2018, 2019)

8. Write a function using Recursion to enter characters one by one until a space is encountered. The function should return the depth at which the space was encountered.

Multiple Choice Questions

1. In a single function declaration, what is the maximum number of statements that may be recursive calls? [ ]
   A. 1 B. 2
   C. n (where n is the argument) D. There is no fixed maximum

2. What is the maximum depth of recursive calls a function may make? [ ]
   A. 1 B. 2
   C. n (where n is the argument) D. There is no fixed maximum

3. Consider the following function: [ ]

```
void super_write_vertical (int number)
{
        if (number < 0)
        {
            printf(" - ");
            super_write_vertical(abs(number));
        }
        else if (number < 10)
            printf("%d\n", number);
        else
        {
             super_write_vertical(number/10);
             printf("%d\n", number % 10);
        }
    }
```

   What values of number are directly handled by the stopping case?
   A. number < 0 B. number < 10
   C. number >= 0 && number < 10 D. number > 10

4. Consider the following function: [ ]

```
void super_write_vertical(int number)
{
        if (number < 0)
        {
             printf(" - ");
             super_write_vertical (abs(number));
        }
        else if (number < 10)
             printf("%d\n", number);
        else
        {
             super_write_vertical(number/10);
             printf("%d\n", number % 10);
        }
    }
```

   Which call will result in the most recursive calls?
   A. super_write_vertical(-1023) B. super_write_vertical(0)
   C. super_write_vertical(100) D. super_write_vertical(1023)

5. Consider this function declaration: [ ]

```
void quiz (int i)
{
   if (i > 1)
   {
         quiz(i / 2);
         quiz(i / 2);
   }
          printf(" * ");
}
```

How many asterisks are printed by the function call quiz(5)?

A. 3 B. 4
C. 7 D. 8

6. In a real computer, what will happen if you make a recursive call without making the problem smaller? [ ]
A. The operating system detects the infinite recursion because of the "repeated state"
B. The program keeps running until you press Ctrl-C
C. The results are non-deterministic
D. The run-time stack overflows, halting the program

7. When the compiler compiles your program, how is a recursive call treated differently than a non-recursive function call? [ ]
A. Parameters are all treated as reference arguments
B. Parameters are all treated as value arguments
C. There is no duplication of local variables
D. None of the above

8. When a function call is executed, which information is not saved in the activation record? [ ]
A. Current depth of recursion.
B. Formal parameters.
C. Location where the function should return when done.
D. Local variables

9. What technique is often used to prove the correctness of a recursive function? [ ]
A. Communitivity. B. Diagonalization.
C. Mathematical induction. D. Matrix Multiplication.

Chapter 3

LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

*void *malloc (number_of_bytes)*

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is:

free (ptr);

The advantage of free() is simply memory management when we no longer need a block.

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked 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.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

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.

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the “start” node.

A single linked list is shown in figure 3.2.1.

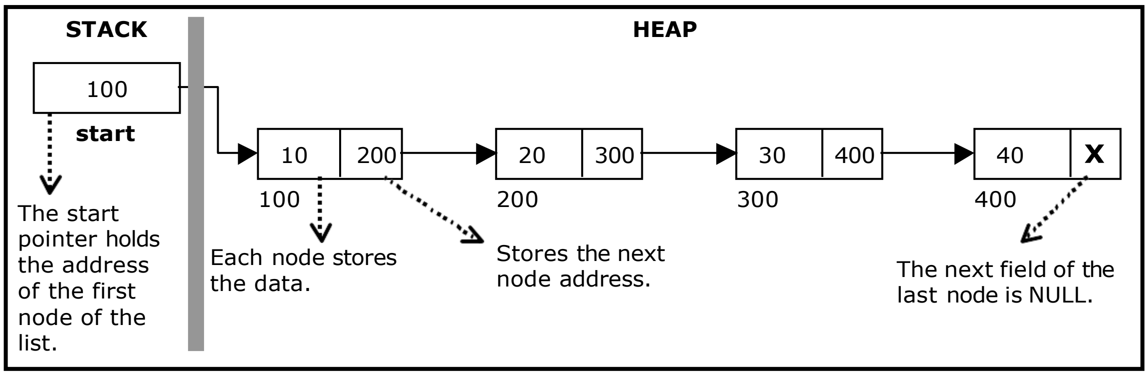

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
10
200
100
20
300
200
30
400
300
40
X
400
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Figure 3.2.1. Single Linked List

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.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

- 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.
- Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```

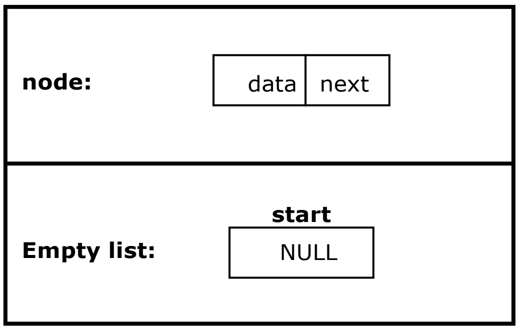


Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- 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. Figure 3.2.3 illustrates the creation of a node for single linked list.

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

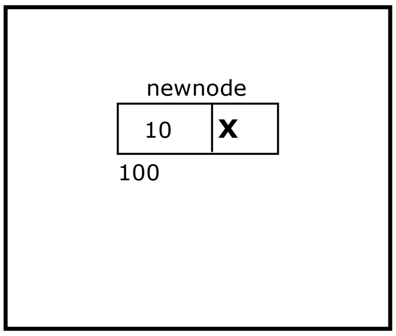


Figure 3.2.3. new node with a value of 10

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

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

- Get the new node using getnode().
  newnode = getnode();

- If the list is empty, assign new node as start.
  start = newnode;

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

  - The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.

  - The start pointer is made to point the new node by assigning the address of the new node.

- Repeat the above steps 'n' times.

Figure 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

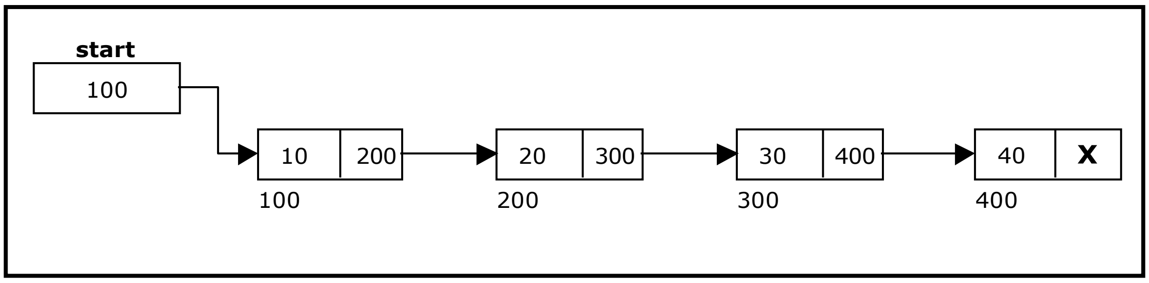


Figure 3.2.4. Single Linked List with 4 nodes

The function createlist(), is used to create 'n' number of nodes:

```
void create list (int n)
{
        int i;
        no de * ne w no de;
        no de * t e m p;
        for( i = 0 ; i < n ; i+ +)
        {
                ne w no de = get no de();
                if(st a rt = = NULL)
                {
                        start = new node;
                }
                e ls e
                {
                        te m p = st a rt;
                        while(temp -> next ! = NULL)
                               temp = temp -> next;
                        temp -> next = new node;
                }
        }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- 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:

- Get the new node using getnode().
  newnode = getnode();

- If the list is empty then *start = newnode.*

- If the list is not empty, follow the steps given below:
  newnode -> next = start;
  start = newnode;

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

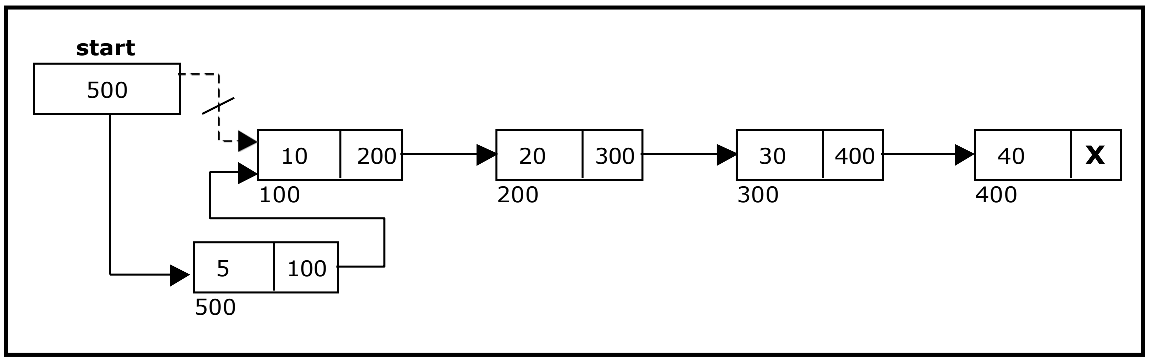


Figure 3.2.5. Inserting a node at the beginning

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
        Node *newnode;
        newnode = getnode();
        if(start == NULL){
                start = newnode;
        }
        else{
                newnode ->next = start; }
}
```

Inserting a node at the end:

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

- Get the new node using getnode()
  ```
  newnode = getnode();
  ```
- If the list is empty then *start = newnode*.
- If the list is not empty follow the steps given below:
  ```
  temp = start;
  while(temp -> next != NULL)
          temp = temp -> next;
  temp -> next = newnode;
  ```

Figure 3.2.6 shows inserting a node into the single linked list at the end.

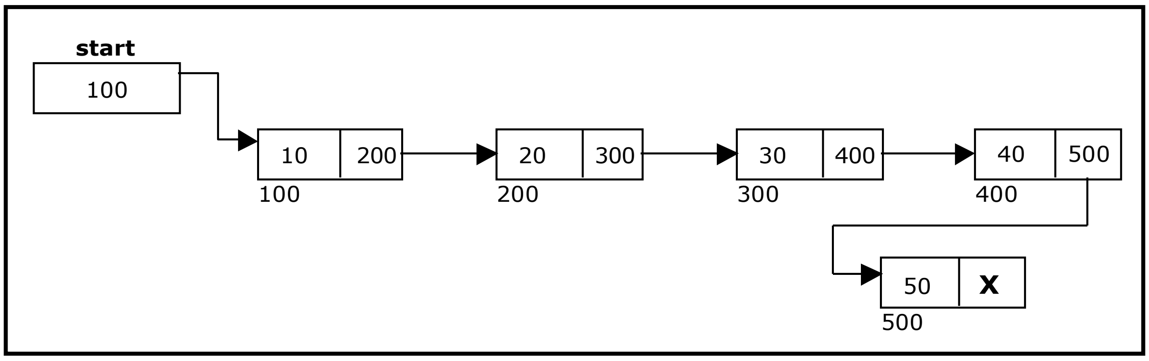


Figure 3.2.6. Inserting a node at the end.

The function insert_at_end(), is used for inserting a node at the end.

```
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;
        }
}
```

Inserting a node at intermediate position:

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

- Get the new node using getnode().
  newnode = getnode();
- 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.
- 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.
- After reaching the specified position, follow the steps given below:
  prev -> next = newnode;
  newnode -> next = temp;
- Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

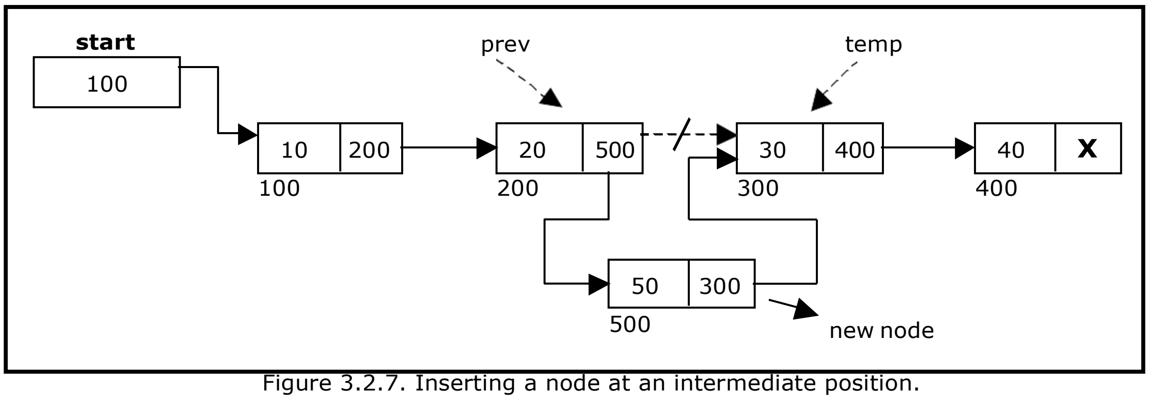


Figure 3.2.7. Inserting a node at an intermediate position.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("position %d is not a middle position", pos);
        }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- 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:

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

- If the list is not empty, follow the steps given below:
```
temp = start;
start = start -> next;
free(temp);
```

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

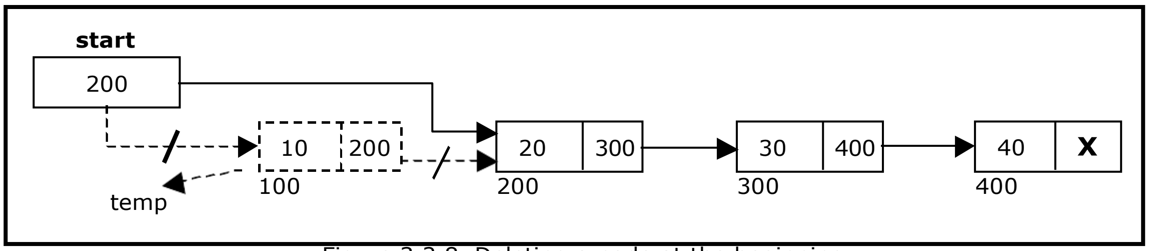


Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at the end:

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

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

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

Figure 3.2.9 shows deleting a node at the end of a single linked list.

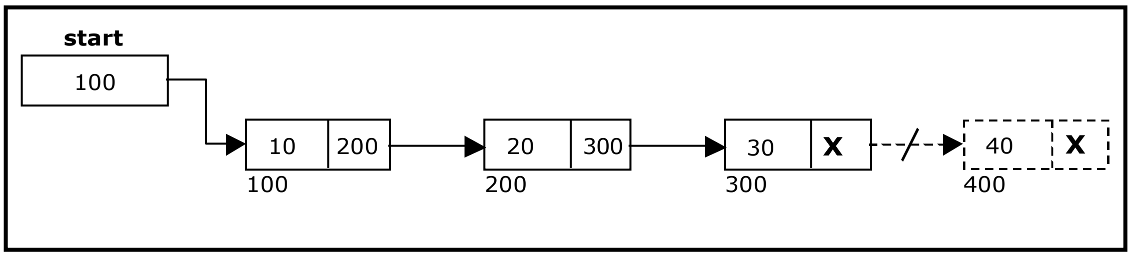


Figure 3.2.9. Deleting a node at the end.

The function delete_at_last(), is used for deleting the last node in the list.

```
void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

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).

- If list is empty then display 'Empty List' message

- 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..");
}
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

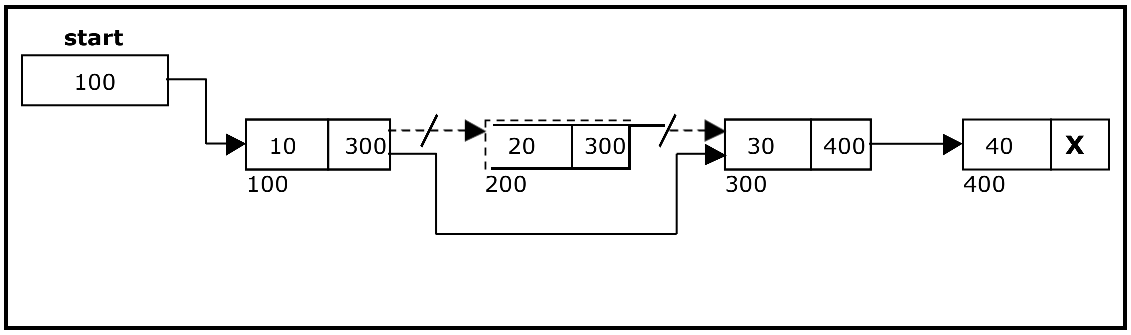


Figure 3.2.10. Deleting a node at an intermediate position.

The function delete_at_mid(), is used for deleting the intermediate node in the list.

```
void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");
                }
```

```
                        if(pos > 1 && pos < nodectr)
                        {
                                temp = prev = start;
                                while(ctr < pos)
                                {
                                        prev = temp;
                                        temp = temp -> next;
                                        ctr ++;
                                }
                                prev -> next = temp -> next;
                                free(temp);
                                printf("\n Node deleted..");
                        }
                        else
                        {
                                printf("\n Invalid position..");
                                getch();
                        }

                }
        }
```

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

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

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.

```
vo id re v_ t ra v e rs e( no de * st)
{
        if(st = = NU LL)
        {
                ret urn;
        }
        e ls e
        {
                re v_t ra v e rs e(st - > ne xt);
                printf(" % d - >" , st - > dat a);
        }
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

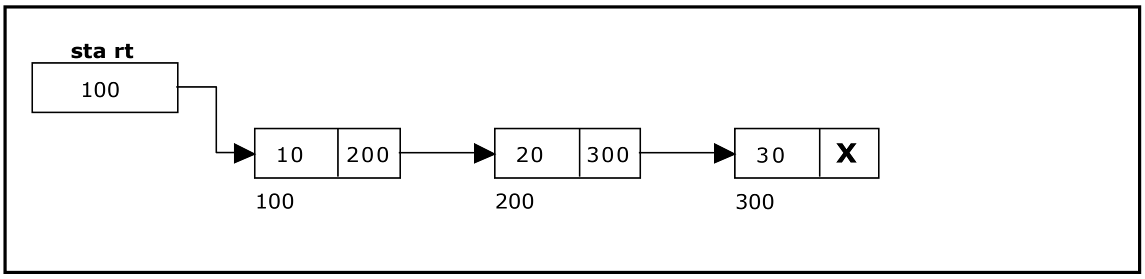
```
int co unt no de( no de * st)
{
        if(st = = NU LL)
                ret urn 0 ;
        e ls e
                ret urn(1 + co unt no de(st - > ne xt));
}
```

3.3.1. Source Code for the Implementation of Single Linked List:

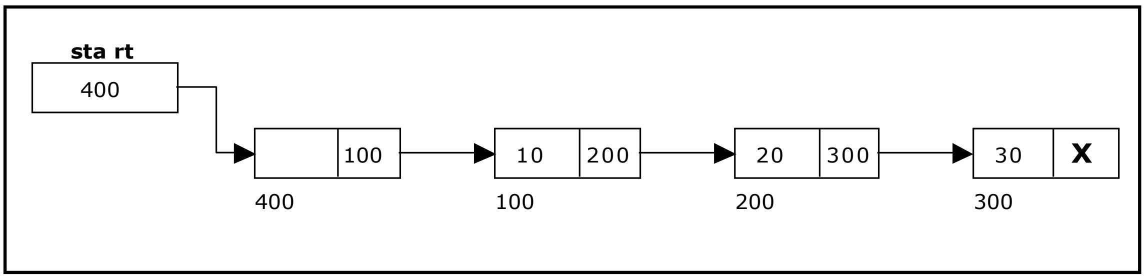
```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

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

typedef struct slinklist node;

node *start = NULL;
int menu()
{
        int ch;
        clrscr();
        printf("\n 1.Create a list ");
        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 (Left to Right)");
        printf("\n 9.Traverse the list (Right to Left)");
        printf("\n------------------------- ");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
}

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

int countnode(node *ptr)
{
        int count=0;
        while(ptr != NULL)
        {
                count++;
                ptr = ptr -> next;
        }
        return (count);
}

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("\n The contents of List (Left to Right): \n");
        if(start == NULL)
        {
                printf("\n Empty List");
                return;
        }
        else
        {
                while(temp != NULL)
                {
                        printf("%d-->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf(" X ");
}

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

void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                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, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
                printf("position %d is not a middle position", pos);
}

void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");

                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                case 1:
                        if(start == NULL)
                        {
                                printf("\n Number of nodes you want to create: ");
                                scanf("%d", &n);
                                createlist(n);
                                printf("\n List created..");
                        }
                        else
                                printf("\n List is already created..");
                                break;
                case 2:
                        insert_at_beg();
                        break;
                case 3:
                        insert_at_end();
                        break;
                case 4:
                        insert_at_mid();
                        break;
                case 5:
                        delete_at_beg();
                        break;
                case 6:
                        delete_at_last();
                        break;
                case 7:
                        delete_at_mid();
                        break;
                case 8:
                        traverse();
                        break;
                case 9:
                        printf("\n The contents of List (Right to Left): \n");
                        rev_traverse(start);
                        printf(" X ");
                        break;
                case 10:
                        printf("\n No of nodes : %d ", countnode(start));
                        break;
                case 11 :
                        exit(0);
                }
                getch();
        }
}
```

Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Sing le Linke d List w it h o ut a he a d er no d e



Sing le Linke d List w it h he a d er no d e

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node ***n*** can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node ***n***.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly.

Array based linked lists:

Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list. Figure 3.5.1 shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.

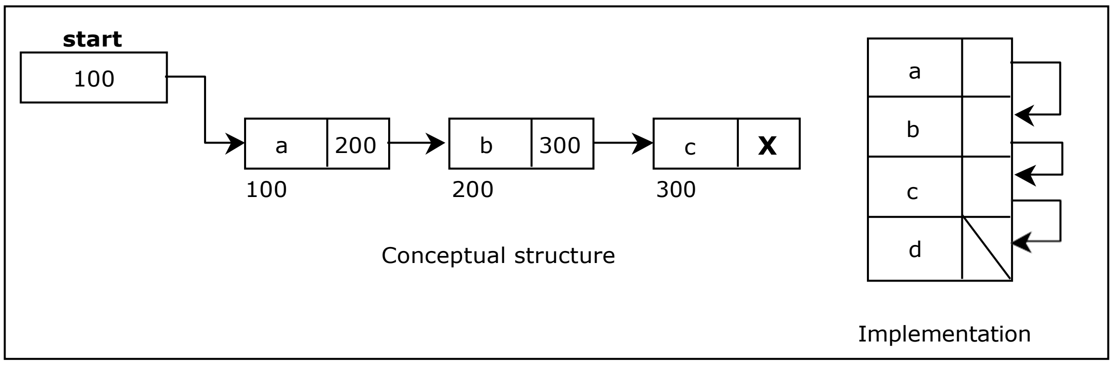


Figure 3.5.1. An array based linked list

Double Linked List:

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:

- Left link.
- Data.
- 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.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

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

A double linked list is shown in figure 3.3.1.

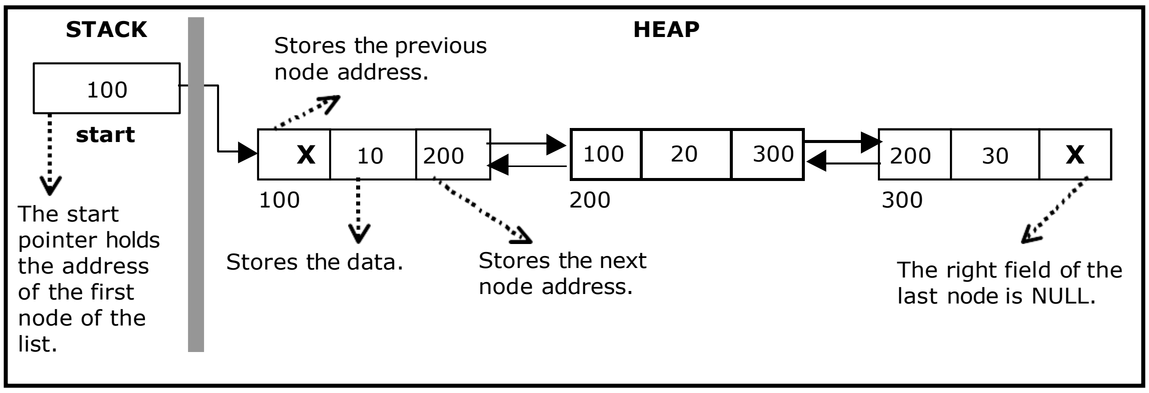

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure 3.3.1. Double Linked List

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.

The following code gives the structure definition:

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

};

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

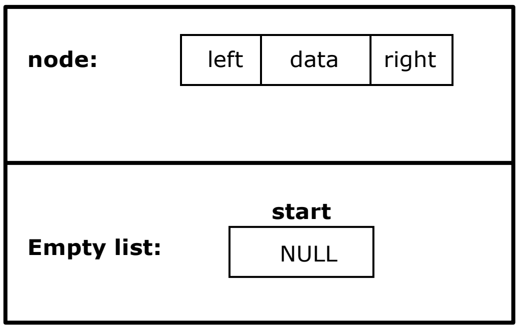


Figure 3.4.1. Structure definition, double link node and empty list

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. 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 and set left field to NULL and right field also set to NULL (see figure 3.2.2).

```
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;
}
```

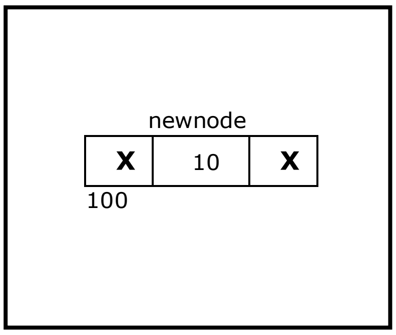


Figure 3.4.2. new node with a value of 10

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

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

- Get the new node using getnode().

  newnode =getnode();

- If the list is empty then *start* = *newnode*.
- If the list is not empty, follow the steps given below:
  - The left field of the new node is made to point the previous node.
  - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps 'n' times.

The function createlist(), is used to create 'n' number of nodes:

```
vo id c re at e list( int n)
{
        int i;
        no de * ne w no de;
        no de * t e m p;
        fo r( i = 0 ; i < n; i++)
        {
                ne w no de = get no de();
                if(st a rt = = NU LL)
                {
                        sta rt = ne w no de;
                }
                e ls e
                {
                        te m p = st a rt;
                        w hile(t e m p - > right)
                                te m p = t e m p - > right;
                        te m p - > right = ne w no de;
                        ne w no de - > left = t e m p;
                }
        }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

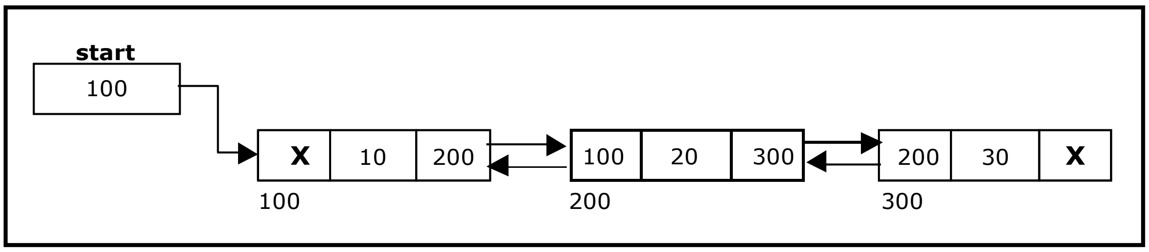


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

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

- Get the new node using getnode().

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

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

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

The function dbl_insert_beg(), is used for inserting a node at the beginning. Figure shows inserting a node into the double linked list at the beginning.

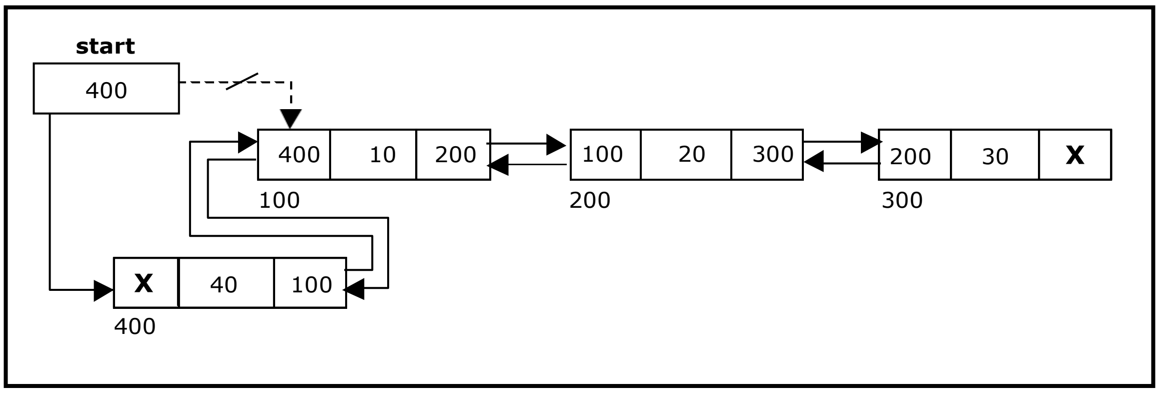


Figure 3.4.4. Inserting a node at the beginning

Inserting a node at the end:

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

- Get the new node using getnode()

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

- If the list is empty then *start = newnode*.
- 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;
```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

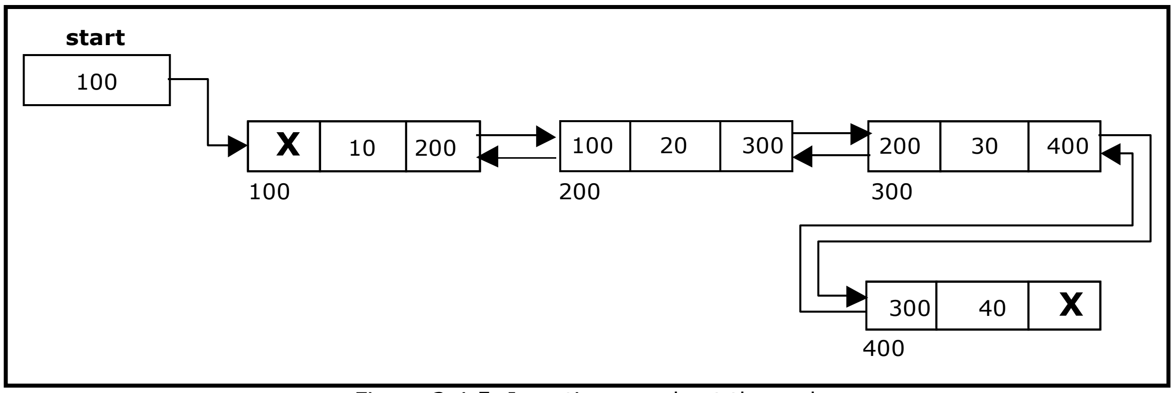


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

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

- Get the new node using getnode().

  newnode=getnode();

- 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.

- 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.

- After reading the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

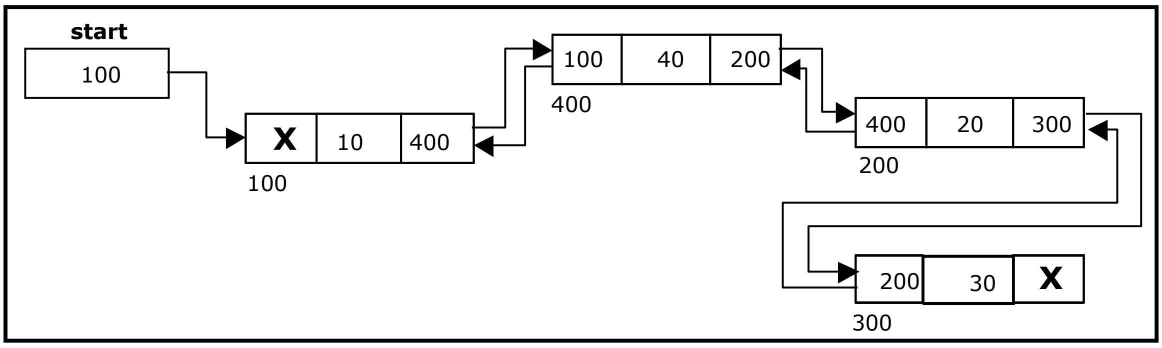


Figure 3.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

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

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

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure shows deleting a node at the beginning of a double linked list.

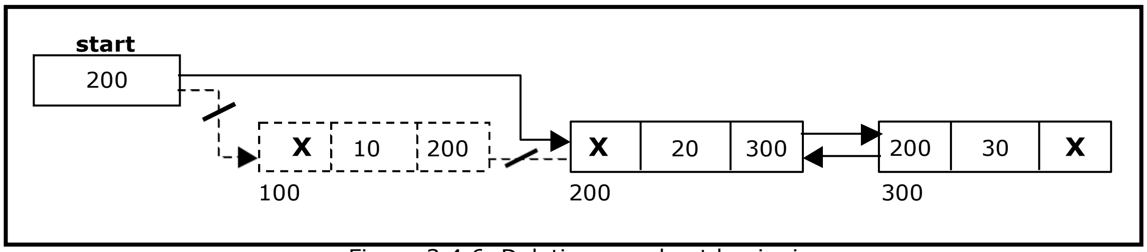


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

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

- If list is empty then display 'Empty List' message
- 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);
```

The function dbl_delete_last(), is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

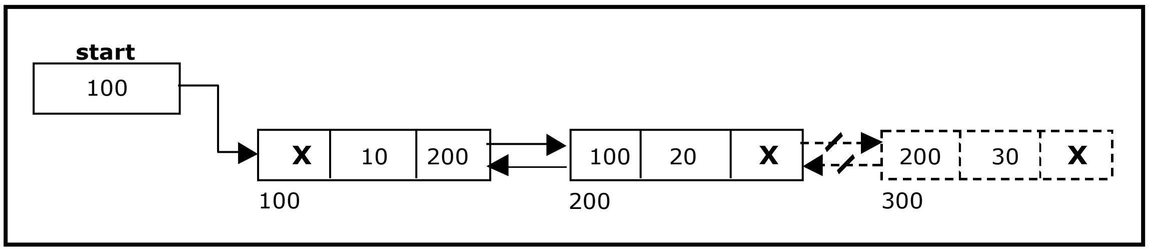


Figure 3.4.7. Deleting a node at the end

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).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
    - Get the position of the node to delete.
    - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
    - 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..");
    }
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

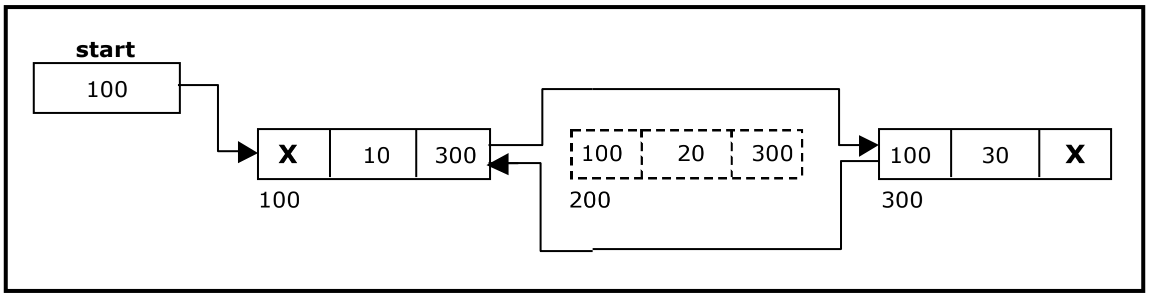


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_left_right*() is used for traversing and displaying the information stored in the list from left to right.

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

- If list is empty then display 'Empty List' message.
- 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):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

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

- 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;
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```
int co untno de( no de * sta rt)
{
        if(st a rt = = NU LL)
                ret urn 0 ;
        e ls e
                ret urn(1 + co unt no de(st a rt - > right ));
}
```

3.5. A Complete Source Code for the Implementation of Double Linked List:

```
#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;

                                case 10 :
                                        printf("\n Number of nodes: %d", countnode(start));
                                        break;
                                case 11:
                                        exit(0);
                        }
                        getch();
                }
        }
```

Circular Single 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. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

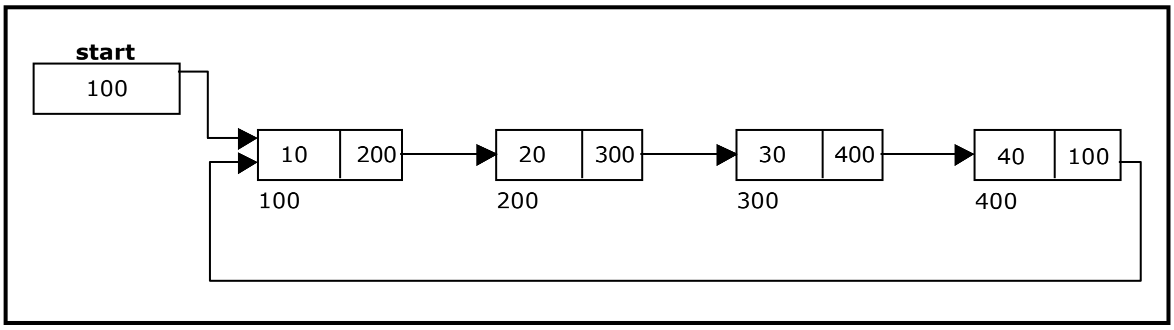


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- 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:

- Get the new node using getnode().

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

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

```
start = newnode;
```

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

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

- Repeat the above steps 'n' times.

- newnode -> next = start;

The function createlist(), is used to create 'n' number of nodes:

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:

- Get the new node using getnode().

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

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

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

- 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;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure shows inserting a node into the circular single linked list at the beginning.

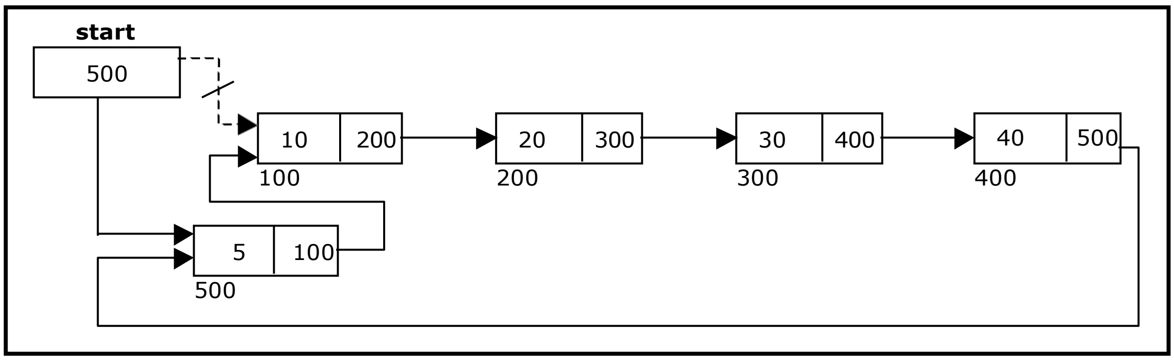


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

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

- Get the new node using getnode().

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

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

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

- 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;
```

The function cll_insert_end(), is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

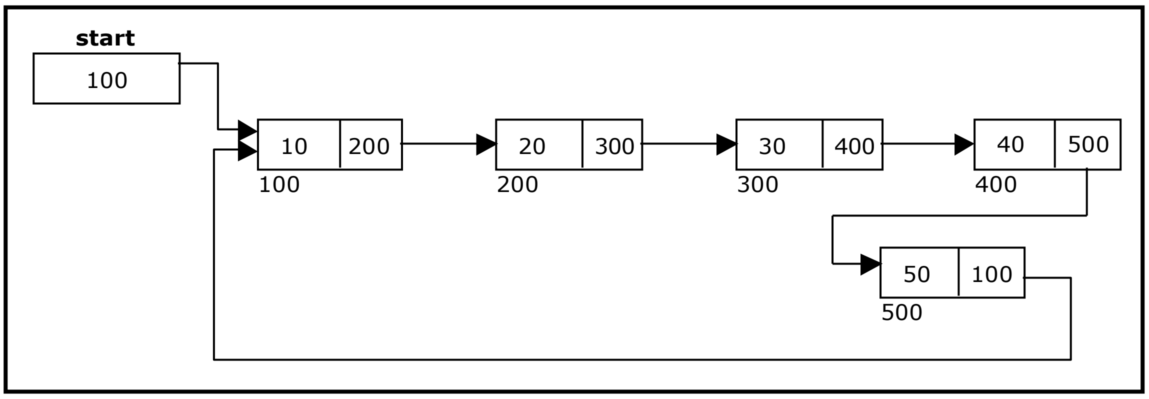


Figure 3.6.3 Inserting a node at the end.

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;
```

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

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

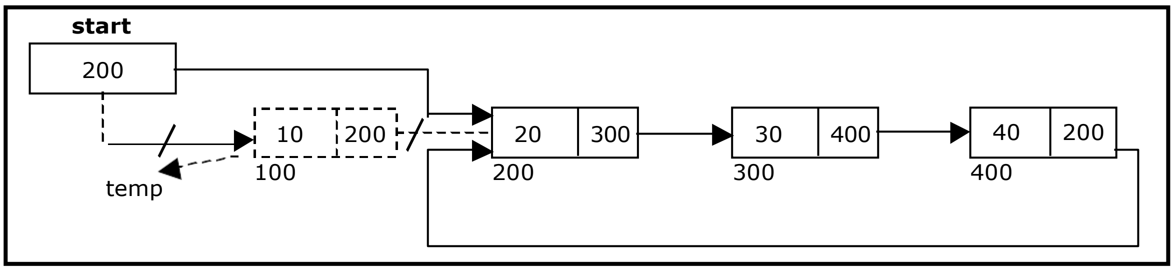


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

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

- If the list is empty, display a message 'Empty List'.
- 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;
```

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

The function cll_delete_last(), is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

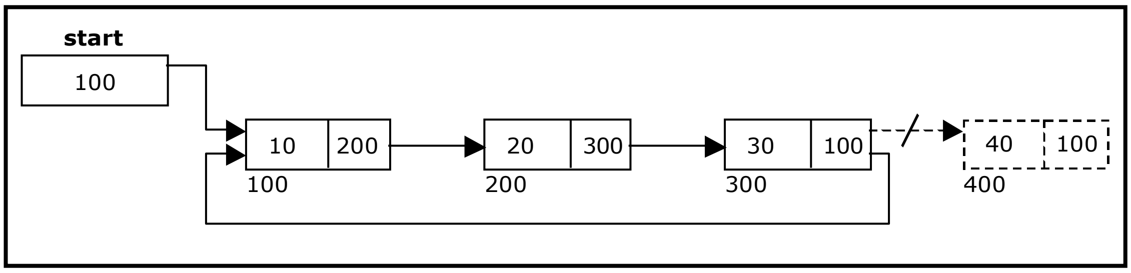


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

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

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

```
temp = start;
do
{
        printf("%d ", temp -> data);
        temp = temp -> next;
} while(temp != start);
```

3.7.1. Source Code for Circular Single Linked List:

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

struct cslinklist
{
        int data;
        struct cslinklist *next;
};

typedef struct cslinklist 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;
}
```

```
int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create a list ");
        printf("\n\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\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\n------------------------ ");
        printf("\n 8. Display the list");
        printf("\n 9. Exit");
        printf("\n\n------------------------ ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

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;         /* last node is pointing to starting node */
}

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

```
void cll_insert_beg()
{
        node *newnode, *last;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
                newnode -> next = start;
        }
        else
        {
                last = start;
                while(last -> next != start)
                        last = last -> next;
                newnode -> next = start;
                start = newnode;
                last -> next = start;
        }
        printf("\n Node inserted at beginning..");
        nodectr++;
}

void cll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL )
        {
                start = newnode;
                newnode -> next = start;
        }
        else
        {
                temp = start;
                while(temp -> next != start)
                        temp = temp -> next;
                temp -> next = newnode;
                newnode -> next = start;
        }
        printf("\n Node inserted at end..");
        nodectr++;
}

void cll_insert_mid()
{
        node *newnode, *temp, *prev;
        int i, pos ;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        if(pos > 1 && pos < nodectr)
        {
                temp = start;
                prev = temp;
                i = 1;
                while(i < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        i++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
                nodectr++;
                printf("\n Node inserted at middle..");
        }
        else
        {
                printf("position %d of list is not a middle position ", pos);
        }
}

void cll_delete_beg()
{
        node *temp, *last;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return ;
        }
        else
        {
                last = temp = start;
                while(last -> next != start)
                        last = last -> next;
                start = start -> next;
                last -> next = start;
                free(temp);
                nodectr--;
                printf("\n Node deleted..");
                if(nodectr == 0)
                        start = NULL;
        }
}

void cll_delete_last()
{
        node *temp,*prev;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != start)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = start;
                free(temp);
                nodectr--;
                if(nodectr == 0)
                        start = NULL;
                printf("\n Node deleted..");
        }
}
```

```
void cll_delete_mid()
{
	int i = 0, pos;
	node *temp, *prev;

	if(start == NULL)
	{
		printf("\n No nodes exist..");
		getch();
		return ;
	}
	else
	{
		printf("\n Which node to delete: ");
		scanf("%d", &pos);
		if(pos > nodectr)
		{
			printf("\nThis node does not exist");
			getch();
			return;
		}
		if(pos > 1 && pos < nodectr)
		{
			temp=start;
			prev = start;
			i = 0;
			while(i < pos - 1)
			{
				prev = temp;
				temp = temp -> next ;
				i++;
			}
			prev -> next = temp -> next;
			free(temp);
			nodectr--;
			printf("\n Node Deleted..");
		}
		else
		{
			printf("\n It is not a middle position..");
			getch();
		}
	}
}

void main(void)
{
	int result;
	int ch, n;
	clrscr();
	while(1)
	{
		ch = menu();
		switch(ch)
		{
			case 1 :
				if(start == NULL)
				{
					printf("\n Enter Number of nodes to create: ");
					scanf("%d", &n);
					createlist(n);
					printf("\nList created..");
				}
```

```
                                        else
                                                printf("\n List is already Exist..");
                                        break;
                                case 2 :
                                        cll_insert_beg();
                                        break;
                                case 3 :
                                        cll_insert_end();
                                        break;
                                case 4 :
                                        cll_insert_mid();
                                        break;
                                case 5 :
                                        cll_delete_beg();
                                        break;
                                case 6 :
                                        cll_delete_last();
                                        break;
                                case 7 :
                                        cll_delete_mid();
                                        break;
                                case 8 :
                                        display();
                                        break;
                                case 9 :
                                        exit(0);
                        }
                        getch();
                }
        }
```

Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

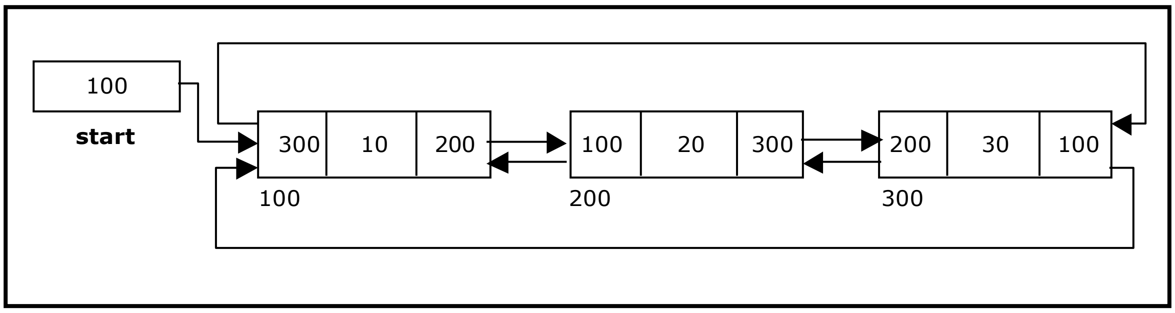


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

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

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

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

- Get the new node using getnode().
```
newnode = getnode();
```
- If the list is empty, then do the following
```
start = newnode;
newnode -> left = start;
newnode ->right = start;
```
- If the list is not empty, follow the steps given below:
```
newnode -> left = start -> left;
newnode -> right = start;
start -> left->right = newnode;
start -> left = newnode;
```
- Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

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

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

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure shows inserting a node into the circular double linked list at the beginning.

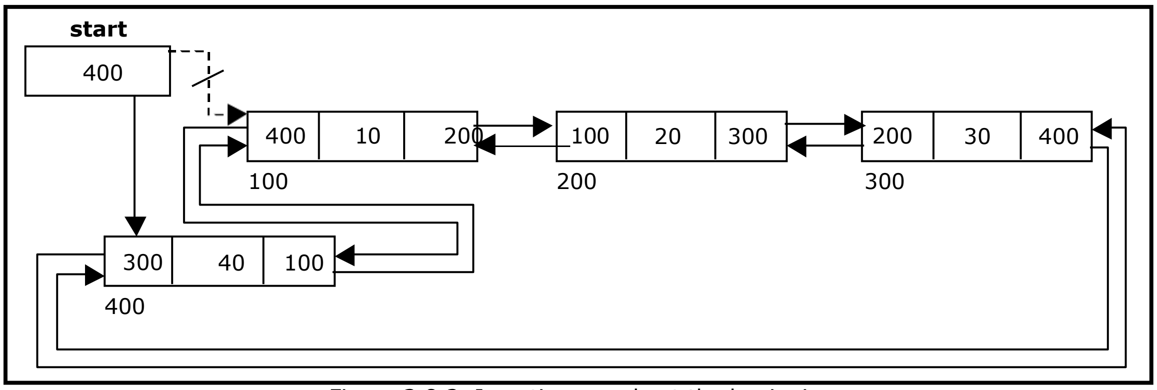


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

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

- Get the new node using getnode()
  newnode=getnode();

- If the list is empty, then
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;

- If the list is not empty follow the steps given below:
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;

The function cdll_insert_end(), is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

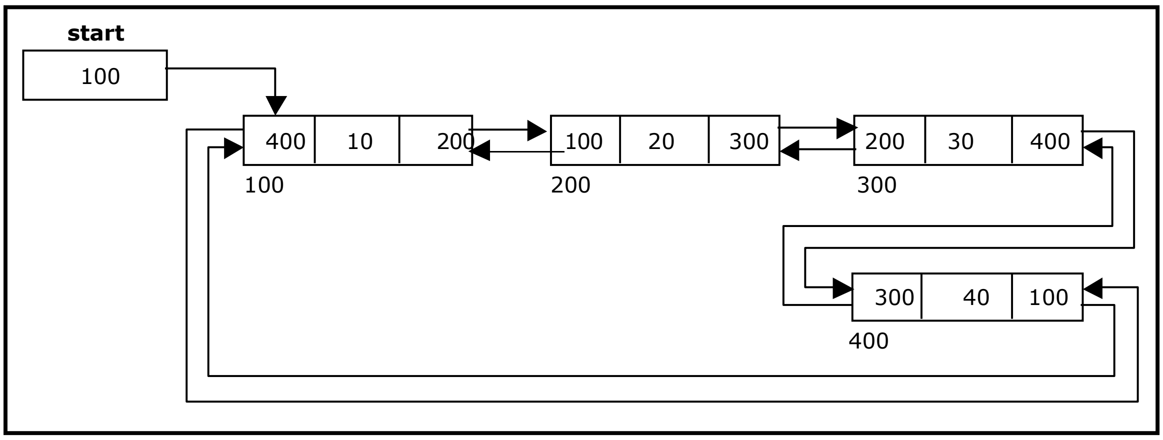


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

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

- Get the new node using getnode().
  newnode=getnode();

- 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.

- Store the starting address (which is in start pointer) in temp. Then traverse the temp pointer upto the specified position.

- After reaching the specified position, follow the steps given below:
  newnode -> left = temp;
  newnode -> right = temp -> right;
  temp -> right -> left = newnode;
  temp -> right = newnode;
  nodectr++;

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

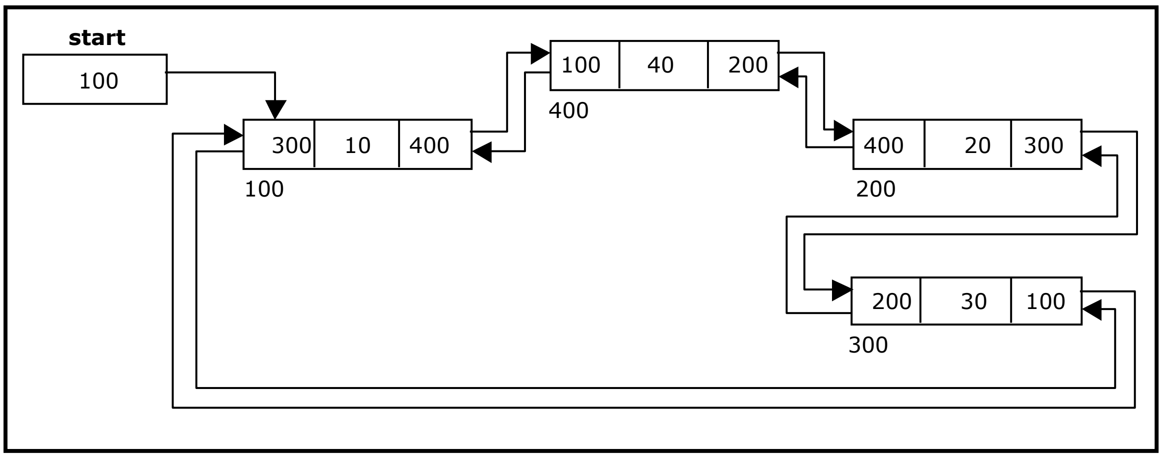


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

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

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

```
temp = start;
start = start -> right;
temp -> left -> right = start;
start -> left = temp -> left;
```

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure shows deleting a node at the beginning of a circular double linked list.

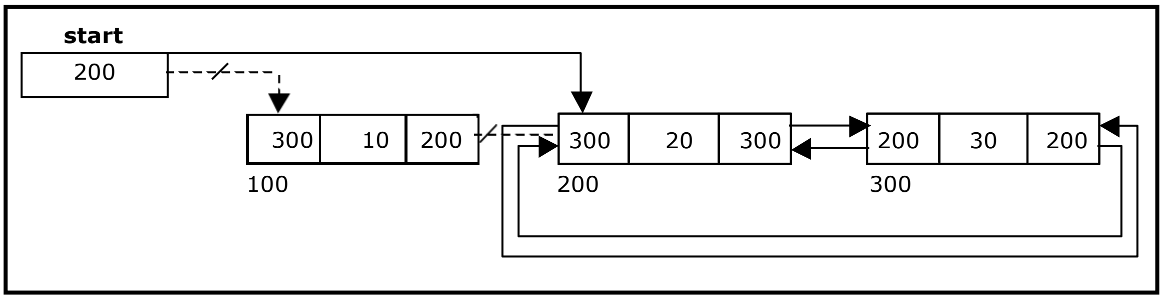


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

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

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

```
temp = start;
while(temp -> right != start)
{
        temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;
```

The function cdll_delete_last(), is used for deleting the last node in the list. Figure shows deleting a node at the end of a circular double linked list.

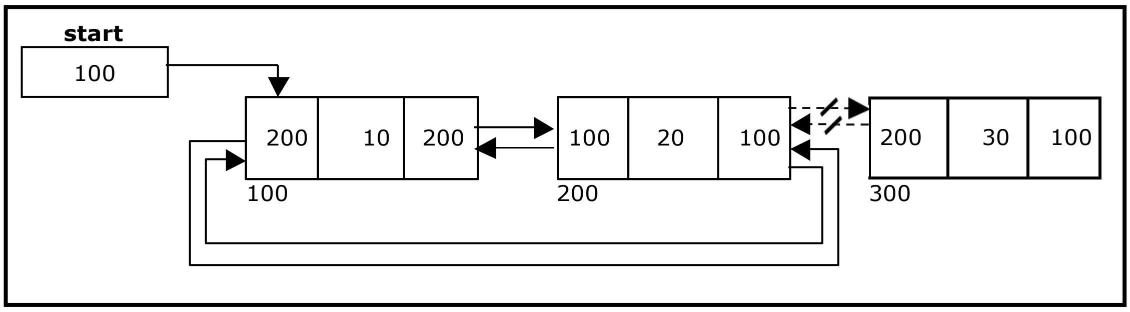


Figure 3.8.6. Deleting a node at the end

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).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
    - Get the position of the node to delete.
    - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
    - 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..");
        nodectr--;
}
```

The function cdll_delete_mid(), is used for deleting the intermediate node in the list.

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

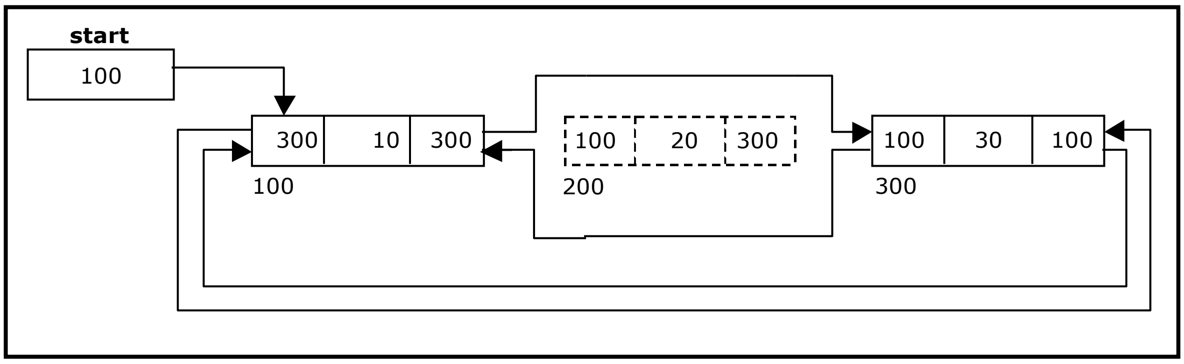


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

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

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

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

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

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:

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

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

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

```
temp = start;
do
{
        temp = temp -> left;
        print temp -> data;
} while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

3.8.1. Source Code for Circular Double Linked List:

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

```
struct cdlinklist
{
        struct cdlinklist *left;
        int data;
        struct cdlinklist *right;
};

typedef struct cdlinklist node;
node *start = NULL;
int nodectr;

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 menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create ");
        printf("\n\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\n------------------------ ");
        printf("\n 5. Delete a node from Beginning");
        printf("\n 6. Delete a node from End");
        printf("\n 7. Delete a node from Middle");
        printf("\n\n------------------------ ");
        printf("\n 8. Display the list from Left to Right");
        printf("\n 9. Display the list from Right to Left");
        printf("\n 10.Exit");
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

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

```
                                        newnode -> right = start;
                                        start -> left->right = newnode;
                                        start -> left = newnode;
                                }
                        }
                }
                else
                        printf("\n List already exists..");
        }

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

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

        void cdll_insert_beg()
        {
                node *newnode;
                newnode = getnode();
                nodectr++;
                if(start == NULL)
                {
                        start = newnode;
                        newnode -> left = start;
                        newnode -> right = start;
                }
                else
                {
                        newnode -> left = start -> left;
                        newnode -> right = start;
                        start -> left -> right = newnode;
                        start -> left = newnode;
                        start = newnode;
                }
        }

        void cdll_insert_end()
        {
                node *newnode,*temp;
                newnode = getnode();
                nodectr++;
                if(start == NULL)
                {
                        start = newnode;
                        newnode -> left = start;
                        newnode -> right = start;
                }
                else
                {
                        newnode -> left = start -> left;
                        newnode -> right = start;
                        start -> left -> right = newnode;
                        start -> left = newnode;
                }
                printf("\n Node Inserted at End");
        }

        void cdll_insert_mid()
        {
                node *newnode, *temp, *prev;
                int pos, ctr = 1;
                newnode = getnode();
                printf("\n Enter the position: ");
                scanf("%d", &pos);
                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;
                        nodectr++;
                        printf("\n Node Inserted at Middle.. ");
                }
                else
                        printf("position %d of list is not a middle position", pos);
                }
        }

        void cdll_delete_beg()
        {
                node *temp;
                if(start == NULL)
                {
                        printf("\n No nodes exist..");
                        getch();
                        return ;
                }
                else
                {
                        nodectr--;
                        if(nodectr == 0)
                        {
                                free(start);
                                start = NULL;
                        }
                        else
                        {
                                temp = start;
                                start = start -> right;
                                temp -> left -> right = start;
                                start -> left = temp -> left;
                                free(temp);
                        }
                        printf("\n Node deleted at Beginning..");
                }
        }

        void cdll_delete_last()
        {
                node *temp;
                if(start == NULL)
                {
                        printf("\n No nodes exist..");
                        getch();
                        return;
                        }
                        else
                        {
                        nodectr--;
                        if(nodectr == 0)
                        {
                                free(start);
                                start = NULL;
                        }
                        else
                        {
                                temp = start;
                                while(temp -> right != start)
                                        temp = temp -> right;
                                temp -> left -> right = temp -> right;
                                temp -> right -> left = temp -> left;
                                free(temp);
                        }
                        printf("\n Node deleted from end ");
                }
        }

        void cdll_delete_mid()
        {
                int ctr = 1, pos;
                node *temp;
                if( start == NULL)
                {
                        printf("\n No nodes exist..");
                        getch();
                        return;
                }
```

```
	else
	{
		printf("\n Which node to delete: ");
		scanf("%d", &pos);
		if(pos > nodectr)
		{
			printf("\nThis node does not exist");
			getch();
			return;
		}
		if(pos > 1 && pos < nodectr)
		{
			temp = start;
			while(ctr < pos)
			{
				temp = temp -> right ;
				ctr++;
			}
			temp -> right -> left = temp -> left;
			temp -> left -> right = temp -> right;
			free(temp);
			printf("\n node deleted..");
			nodectr--;
		}
		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);
				cdll_createlist(n);
				printf("\n List created..");
				break;
			case 2 :
				cdll_insert_beg();
				break;
			case 3 :
				cdll_insert_end();
				break;
			case 4 :
				cdll_insert_mid();
				break;
			case 5 :
				cdll_delete_beg();
				break;
			case 6 :
				cdll_delete_last();
				break;
```

```
                        case 7 :
                                cdll_delete_mid();
                                break;
                        case 8 :
                                cdll_display_left_right();
                                break;
                        case 9 :
                                cdll_display_right_left();
                                break;
                        case 10:
                                exit(0);
                }
                getch();
        }
}
```

Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

Polynomials:

A polynomial is of the form: $\sum_{i=0}^{n} c_i \, x^i$

Where, c_i is the coefficient of the ith term and
n is the degree of the polynomial

Some examples are:

$5x^2 + 3x + 1$
$12x^3 - 4x$
$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials 1 + x and x + 1 are equivalent.

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 3.10.1.

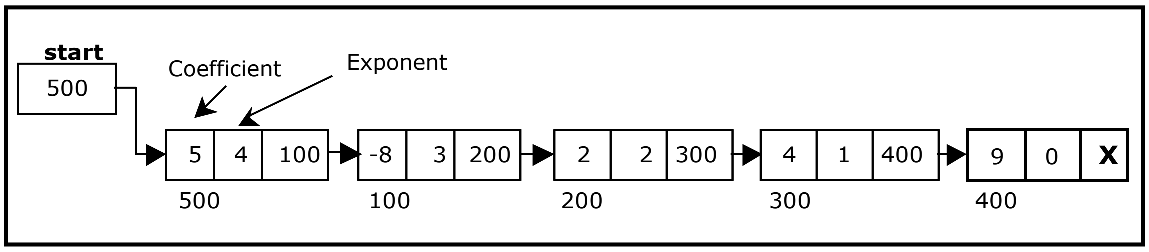


Figure 3.10.1. Single Linked List for the polynomial F(x) = $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

Source code for polynomial creation with help of linked list:

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

struct link
{
        float coef;
        int expo;
        struct link *next;
};

typedef struct link node;
node * getnode()
{
                node *tmp;
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin);
                scanf("%f",&tmp->coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}
node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n): ");
                        ch = getche();
                        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("+ %.2f", t -> coef);
                        printf("X^ %d", t -> expo);
                        t =t -> next;
                }
}

void main()
{
                node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
                clrscr();
                printf("\nEnter First Polynomial..(in ascending-order of exponent)");
                poly1 = create_poly (poly1);
                printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
                poly2 = create_poly (poly2);
                clrscr();
                printf("\n Enter Polynomial 1: ");
                display (poly1);
                printf("\n Enter Polynomial 2: ");
                display (poly2);
                getch();
}
```

Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- Read two polynomials.
- Add them.
- Display the resultant polynomial.

Source code for polynomial addition with help of linked list:

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

struct link
{
        float coef;
        int expo;
        struct link *next;
};

typedef struct link node;

node * getnode()
{
                node *tmp;
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin);
                scanf("%f",&tmp->coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}

node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n): ");
                        ch = getche();
                        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("+ %.2f", t -> coef);
                        printf("X^ %d", t -> expo);
                        t = t -> next;
                }
}

void add_poly(node *p1,node *p2)
{

        node *newnode;
        while(1)
        {
                if( p1 == NULL || p2 == NULL )
                        break;
                if(p1->expo == p2->expo )
                {
                                printf("+ %.2f X ^%d",p1->coef+p2->coef,p1->expo);
                                p1 = p1->next; p2 = p2->next;
                }
                else
                {
                        if(p1->expo < p2->expo)
                        {
                                        printf("+ %.2f X ^%d",p1->coef,p1->expo);
                                        p1 = p1->next;
                        }
                        else
                        {
                                        printf(" + %.2f X ^%d",p2->coef,p2->expo);
                                        p2 = p2->next;
                        }
                }
        }
        while(p1 != NULL )
        {
                printf("+ %.2f X ^%d",p1->coef,p1->expo);
                p1 = p1->next;
        }
        while(p2 != NULL )
        {

                printf("+ %.2f X ^%d",p2->coef,p2->expo);
                p2 = p2->next;
        }
}

void main()
{
                node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
                clrscr();
                printf("\nEnter First Polynomial..(in ascending-order of exponent)");
                poly1 = create_poly (poly1);
                printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
                poly2 = create_poly (poly2);
                clrscr();
                printf("\n Enter Polynomial 1: ");
                display (poly1);
                printf("\n Enter Polynomial 2: ");
                display (poly2);
                printf( "\n Resultant Polynomial : ");
                add_poly(poly1, poly2);
                display (poly3);
                getch();
}
```

Exercise

1. Write a "C" functions to split a given list of integers represented by a single linked list into two lists in the following way. Let the list be L = (l_0, l_1, ….., l_n). The resultant lists would be R_1 = (l_0, l_2, l_4, …..) and R_2 = (l_1, l_3, l_5, …..).

2. Write a "C" function to insert a node "t" before a node pointed to by "X" in a single linked list "L". (tcs 2017, 2018; Accenture 2016, 2017)

3. Write a "C" function to delete a node pointed to by "p" from a single linked list "L".(jaro education 2015)

4. Suppose that an ordered list L = (l_0, l_1, …..,l_n) is represented by a single linked list. It is required to append the list L = (l_n, l_0, l_1, ….., l_n) after another ordered list M represented by a single linked list.
5. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is the number of occurrences of 42 in the data field of a node on the linked list. The list itself is unchanged.

6. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is true if the list has at least one occurrence of the number 42 in the data part of a node.

7. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is the sum of all the data components of all the nodes. NOTE: If the list is empty, the function returns 0.

8. Write a "C" function to concatenate two circular linked lists producing another circular linked list. (byju's 2018, 2019)

9. Write "C" functions to compute the following operations on polynomials represented as singly connected linked list of nonzero terms.

   1. Evaluation of a polynomial
   2. Multiplication of two polynomials.

10. Write a "C" function to represent a sparse matrix having "m" rows and "n" columns using linked list.

11. Write a "C" function to print a sparse matrix, each row in one line of output and properly formatted, with zero being printed in place of zero elements.

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]

   A. Queue C. Linked List
   B. Stack D. all the above

2. Which among the following is a dynamic data structure: [A]

   A. Double Linked List C. Stack
   B. Queue D. all the above

3. The link field in a node contains: [A]

   A. address of the next node C. data of next node
   B. data of previous node D. data of current node

4. Memory is allocated dynamically to a data structure during execution by ------- function. [D]

   A. malloc() C. realloc()
   B. Calloc() D. all the above

Chapter 4

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push().

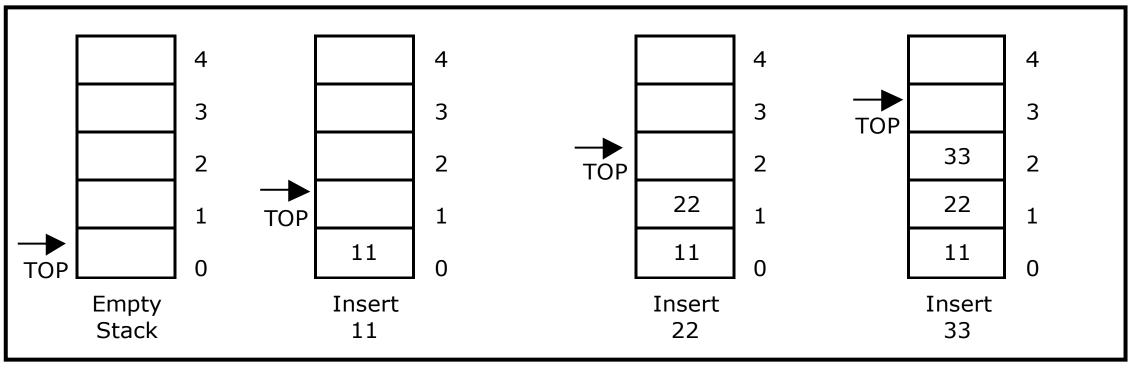


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

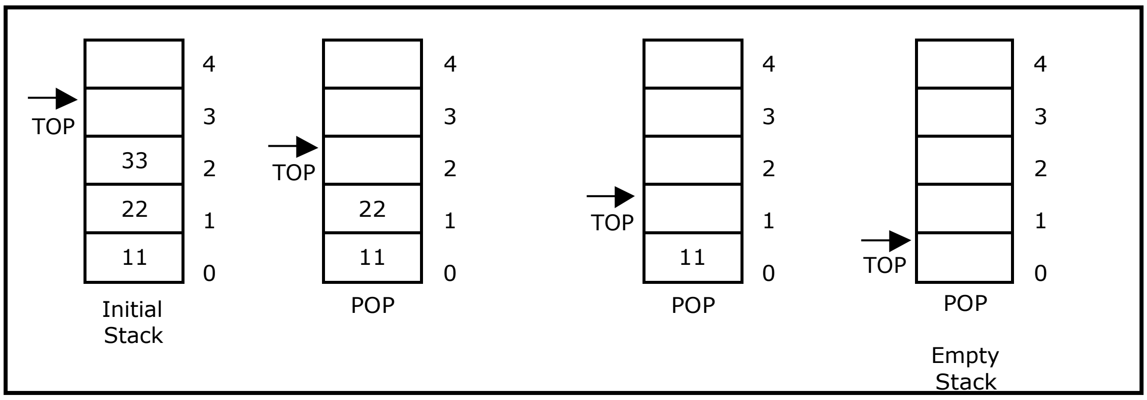


Figure 4.2. Pop operations on stack

Source code for stack operations, using array:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
        int ch;
        clrscr();
        printf("\n … Stack operations using ARRAY... ");
        printf("\n -----------********** ------------ \n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void display()
{
        int i;
        if(top == 0)
        {
                printf("\n\nStack empty..");
```

```
		return;
	}
	else
	{
		printf("\n\nElements in stack:");
		for(i = 0; i < top; i++)
			printf("\t%d", stack[i]);
	}
}

void pop()
{
	if(top == 0)
	{
		printf("\n\nStack Underflow..");
		return;
	}
	else
		printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{
	int data;
	if(top == MAX)
	{
		printf("\n\nStack Overflow..");
		return;
	}
	else
	{
		printf("\n\nEnter data: ");
		scanf("%d", &data);
		stack[top] = data;
		top = top + 1;
		printf("\n\nData Pushed into the stack");
	}
}

void main()
{
	int ch;
	do
	{
		ch = menu();
		switch(ch)
		{
			case 1:
				push();
				break;
			case 2:
				pop();
				break;
			case 3:
				display();
				break;

			case 4:
				exit(0);
		}
		getch();
	} while(1);
}
```

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

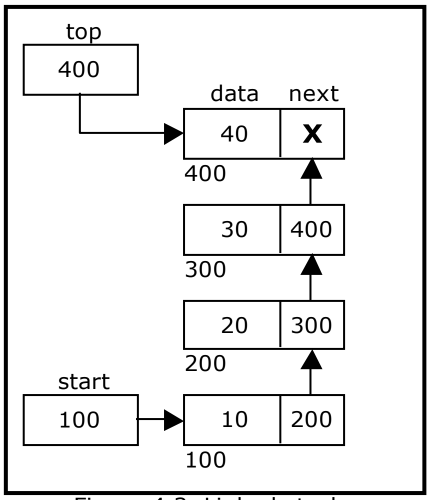


Figure 4.3. Linked stack representation

Source code for stack operations, using linked list:

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

struct stack
{
        int data;
        struct stack *next;
};

void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node *) malloc( sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void push(node *newnode)
{
        node *temp;
        if( newnode == NULL )
        {
                printf("\n Stack Overflow..");
                return;
        }
        if(start == NULL)
        {
                start = newnode;
                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
                top = newnode;
        }
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t Stack underflow");
                return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
                top = NULL;
        }
        else
        {
                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;
                printf("\n\n\n\t\t Elements in the stack: \n");
                printf("%5d ", temp -> data);
                while(temp != top)
                {
                        temp = temp -> next;
                        printf("%5d ", temp -> data);
                }
        }
}
```

```
char menu()
{
        char ch;
        clrscr();
        printf("\n \tStack operations using pointers.. ");
        printf("\n -----------********** ------------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode);
                                break;
                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while( ch != '4' );
}
```

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.

2. The parentheses are not needed to designate the expression un-ambiguously.

3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.

2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50];                    /* operator stack */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                     /* priority value of op  */
        int pv2;                     /* priority value of op_at_stack */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

```
void push(char op)        /* op - operator  */
{
    if(top == 0)
    {
        opstack[top] = op;
        top++;
    }
    else
    {
        if(op != '(' )
        {
            while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
            {
                    postfix[j] = opstack[--top];
                    j++;
            }
        }
        opstack[top] = op;      /* pushing onto stack */
        top++;
    }
}
```

/* before pushing the operator 'op' into the stack check priority of op with top of opstack if less then pop the operator from stack then push into postfix string else push op onto stack itself */

```
pop()
{
    while(opstack[--top] != '(' )              /* pop until '(' comes  */
    {
        postfix[j] = opstack[top];
        j++;
    }
}

void main()
{
    char ch;
    clrscr();
    printf("\n Enter Infix Expression : ");
    gets(infix);
    while( (ch=infix[i++]) != '\0')
    {
        switch(ch)
        {
            case ' ' : break;
            case '(' :
            case '+' :
            case '-' :
            case '*' :
            case '/' :
            case '^' :
            case '%' :
                push(ch);                 /* check priority and push */
                break;
            case ')' :
                pop();
                break;
            default :
                postfix[j] = ch;
                j++;
        }
    }
    while(top >= 0)
    {
        postfix[j] = opstack[--top];
        j++;
    }
    postfix[j] = '\0';
    printf("\n Infix  Expression    : %s ", infix);
    printf("\n Postfix Expression : %s ", postfix);
    getch();
}
```

Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression (A + B) * (C - D) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
D	D	)	
-	D	) -	
C	C D	) -	
(	- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(	+ A B – C D	*	
End of string	* + A B – C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

Program to convert an infix to prefix expression:

```
# include <conio.h>
# include <string.h>

char prefix[50];
char infix[50];
char opstack[50];                    /* operator stack */
int j, top = 0;

void insert_beg(char ch)
{
        int k;
        if(j == 0)
                prefix[0] =  ch;
        else
        {
                for(k = j + 1; k > 0; k--)
                        prefix[k] = prefix[k - 1];
                prefix[0] = ch;
        }
        j++;
}
```

```
int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                /* priority value of op  */
        int pv2;                /* priority value of op_at_stack */
        char operators[] = {'+', '-', '*', '/', '%', '^', ')'};
        int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
        if(op_at_stack == ')' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if( op_at_stack == operators[k] )
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}

void push(char op)              /* op – operator */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != ')')
                {
                        /* before pushing the operator 'op' into the stack check priority of op
                with top of operator stack if less pop the operator from stack then push into
                postfix string else push op onto stack itself  */

                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        insert_beg(opstack[--top]);
                        }
                }
                opstack[top] = op;                      /* pushing onto stack */
                top++;
        }
}

void pop()
{
        while(opstack[--top] != ')')                    /* pop until ')' comes; */
                insert_beg(opstack[top]);
}


void main()
{
        char ch;
        int l, i = 0;
        clrscr();
        printf("\n Enter Infix Expression : ");
        gets(infix);
        l = strlen(infix);
        while(l > 0)
        {
                ch = infix[--l];
                switch(ch)
                {
                        case ' ' : break;
                        case ')' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push */
                                break;
                        case '(' :
                                pop();
                                break;
                        default :
                                insert_beg(ch);
                }
        }
        while( top > 0 )
        {
                insert_beg( opstack[--top] );
                j++;
        }
        prefix[j] = '\0';
        printf("\n Infix Expression    : %s ", infix);
        printf("\n Prefix Expression : %s ", prefix);
        getch();
}
```

Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

4.4.1. Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
        if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
                return 1;
        else
                return 0;
}
```

```
void main(void)
{
        char postfix[MAX];
        int val;
        char ch;
        int i = 0, top = 0;
        float val_stack[MAX], val1, val2, res;
        clrscr();
        printf("\n Enter a postfix expression: ");
        scanf("%s", postfix);
        while((ch = postfix[i]) != '\0')
        {
                if(isoperator(ch) == 1)
                {
                        val2 = val_stack[--top];
                        val1 = val_stack[--top];
                        switch(ch)
                        {
                                case '+':
                                        res = val1 + val2;
                                        break;
                                case '-':
                                        res = val1 - val2;
                                        break;
                                case '*':
                                        res = val1 * val2;
                                        break;
                                case '/':
                                        res = val1 / val2;
                                        break;
                                case '^':
                                        res = pow(val1, val2);
                                        break;
                        }
                        val_stack[top] = res;
                }
                else
                        val_stack[top] = ch-48; /*convert character digit to integer digit */
                top++;
                i++;
        }
        printf("\n Values of %s is : %f ",postfix, val_stack[0] );
        getch();
}
```

Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor’s stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Queue:

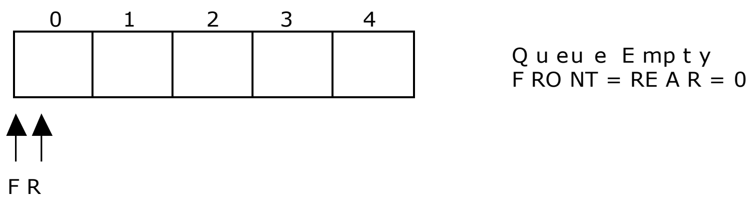
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

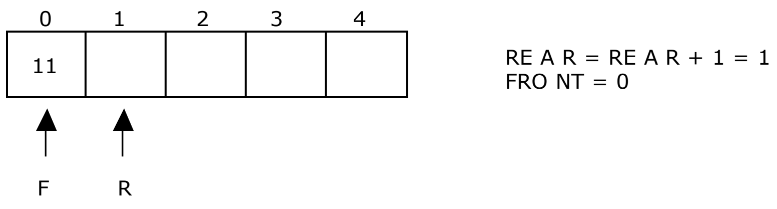
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Representation of Queue:

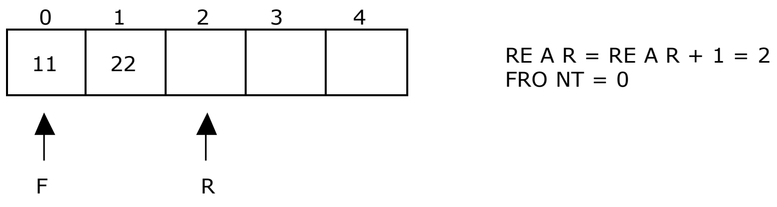
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



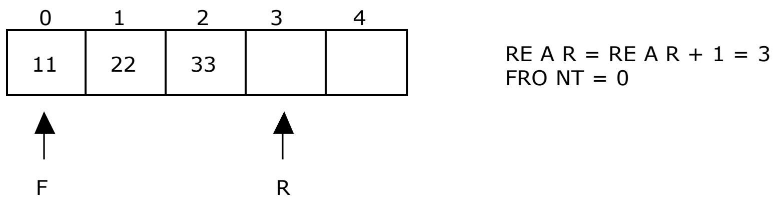
Now, insert 11 to the queue. Then queue status will be:



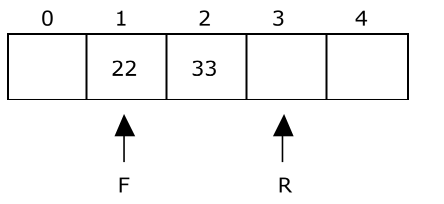
Next, insert 22 to the queue. Then the queue status is:



Again insert another element 33 to the queue. The status of the queue is:

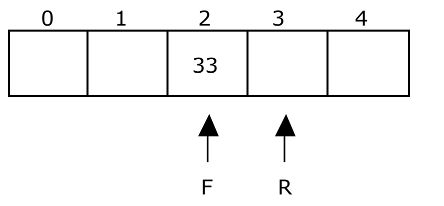


Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



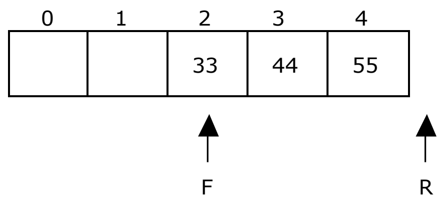
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



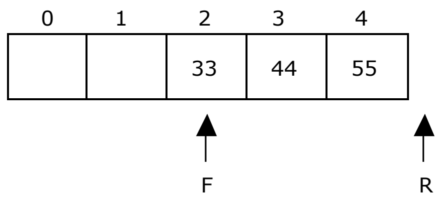
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



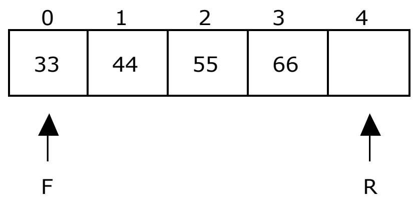
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



REAR = 4
FRONT = 0

This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

```
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
        int data;
        if(rear == MAX)
        {
                printf("\n Linear Queue is full");
                return;
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                Q[rear] = data;
                rear++;
                printf("\n Data Inserted in the Queue ");
        }
}
void deleteQ()
{
        if(rear == front)
        {
                printf("\n\n Queue is Empty..");
                return;
        }
        else
        {
                printf("\n Deleted element from Queue is %d", Q[front]);
                front++;
        }
}
void displayQ()
{
        int i;
        if(front == rear)
        {
                printf("\n\n\t Queue is Empty");
                return;
        }
        else
        {
                printf("\n Elements in Queue are: ");
                for(i = front; i < rear; i++)
                {
                        printf("%d\t", Q[i]);
                }
        }
}
int menu()
{
        int ch;
        clrscr();
        printf("\n \tQueue operations using ARRAY..");
        printf("\n -----------********** ------------ \n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }
                getch();
        } while(1);
}
```

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

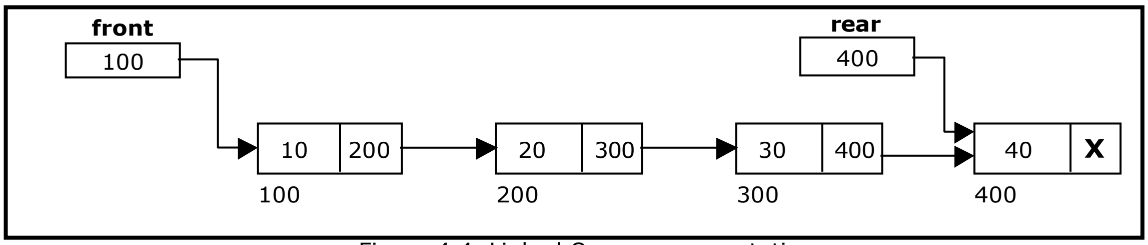


Figure 4.4. Linked Queue representation

Source code for queue operations using linked list:

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

struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear =  NULL;

node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp -> data);
        free(temp);
}
```

```
void displayQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t\t Empty Queue ");
        }
        else
        {
                temp = front;
                printf("\n\n\n\t\t Elements in the Queue are: ");
                while(temp != NULL )
                {
                        printf("%5d ", temp -> data);
                        temp = temp -> next;
                }
        }
}

char menu()
{
        char ch;
        clrscr();
        printf("\n \t..Queue operations using pointers.. ");
        printf("\n\t -----------**********------------ \n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                insertQ();
                                break;
                        case '2' :
                                deleteQ();
                                break;
                        case '3' :
                                displayQ();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while(ch != '4');
}
```

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

3. Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue:

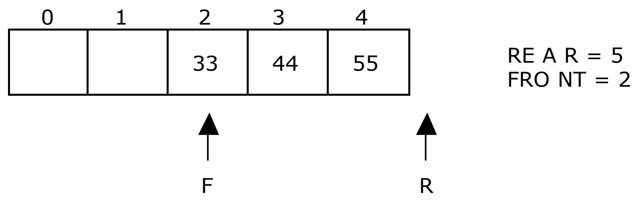
A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

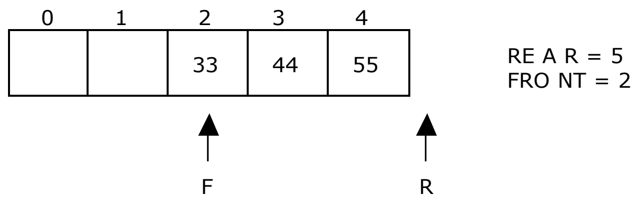
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.

- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

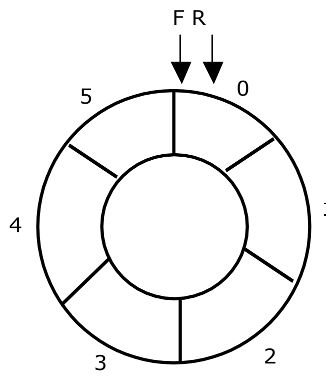


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

Representation of Circular Queue:

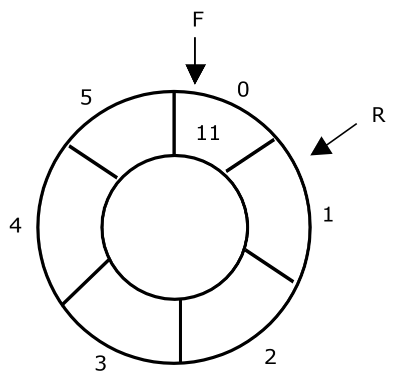
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Queue Empty
MA X = 6
F RO NT = RE A R = 0
CO U NT = 0

Circ u lar Q u e u e

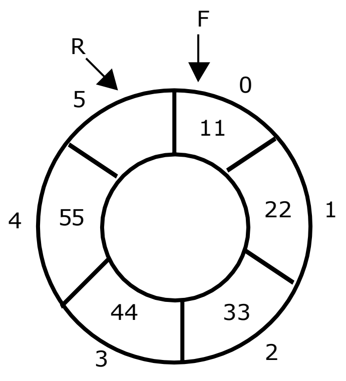
Now, insert 11 to the circular queue. Then circular queue status will be:



FRO NT = 0
RE A R = (RE A R + 1) % 6 = 1
CO U NT = 1

Circ u lar Q u e u e

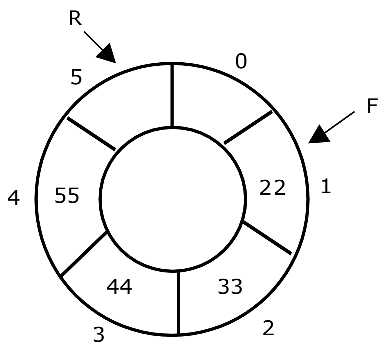
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

Circular Queue

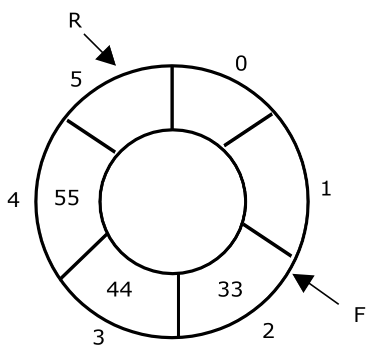
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



F RO NT = (F R O NT + 1) % 6 = 1
RE A R = 5
CO U NT = CO U NT - 1 = 4

Circ u lar Q u e u e

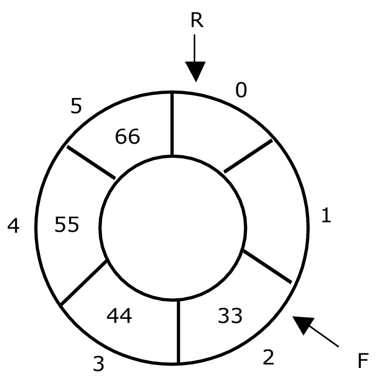
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



F RO NT = (F R O NT + 1) % 6 = 2
RE A R = 5
CO U NT = CO U NT - 1 = 3

Circ u lar Q u e u e

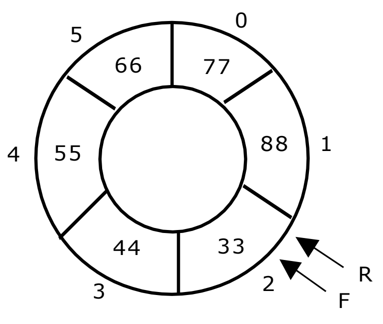
Again, insert another element 66 to the circular queue. The status of the circular queue is:



FRO NT = 2
RE A R = (RE A R + 1) % 6 = 0
C O U NT = C O U NT + 1 = 4

Circ u lar Q u e u e

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



F RO NT = 2, RE A R = 2
RE A R = RE A R % 6 = 2
CO U NT = 6

Circ u lar Q u e u e

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
        int data;
        if(count == MAX)
        {
                printf("\n Circular Queue is Full");
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                CQ[rear] = data;
                rear = (rear + 1) % MAX;
                count ++;
                printf("\n Data Inserted in the Circular Queue ");
        }
}

void deleteCQ()
{
        if(count == 0)
        {
                printf("\n\nCircular Queue is Empty..");
        }
        else
        {
                printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
                front = (front + 1) % MAX;
                count --;
        }
}
```

```
void displayCQ()
{
        int i, j;
        if(count == 0)
        {
                printf("\n\n\t Circular Queue is Empty ");
        }
        else
        {
                printf("\n Elements in Circular Queue are: ");
                j = count;
                for(i = front; j != 0; j--)
                {
                        printf("%d\t", CQ[i]);
                        i = (i + 1) % MAX;
                }
        }
}

int menu()
{
        int ch;
        clrscr();
        printf("\n \t Circular Queue Operations using ARRAY..");
        printf("\n -----------********** ------------ \n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter Your Choice: ");
        scanf("%d", &ch);
        return ch;
}

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertCQ();
                                break;
                        case 2:
                                deleteCQ();
                                break;
                        case 3:
                                displayCQ();
                                break;
                        case 4:
                                return;
                        default:
                                printf("\n Invalid Choice ");
                }
```

```
            getch();
        } while(1);
}
```

Exercises

1. What is a linear data structure? Give two examples of linear data structures. (infosys 2017, 2018; tcs 2017, 2019; nucleus 2018)

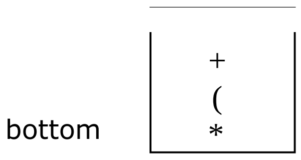
2. Is it possible to have two designs for the same data structure that provide the same functionality but are implemented differently? (wipro 2017, 2018; Mphasis 2018; tcs 2017, 2019)

3. What is the difference between the logical representation of a data structure and the physical representation? (byju’s 2018, 2019)

4. Transform the following infix expressions to reverse polish notation:
   - a) A ↑ B * C – D + E / F / (G + H)
   - b) ((A + B) * C – (D – E)) ↑ (F + G)
   - c) A – B / (C * D ↑ E)
   - d) (a + b ↑ c ↑ d) * (e + f / d))
   - f) 3 – 6 * 7 + 2 / 4 * 5 – 8
   - g) (A – B) / ((D + E) * F)
   - h) ((A + B) / D) ↑ ((E – F) * G)

5. Evaluate the following postfix expressions:
   - a) P_1: 5, 3, +, 2, *, 6, 9, 7, -, /, -
   - b) P_2: 3, 5, +, 6, 4, -, *, 4, 1, -, 2, ↑, +
   - c) P_3 : 3, 1, +, 2, ↑, 7, 4, -, 2, *, +, 5, -

6. Consider the usual algorithm to convert an infix expression to a postfix expression. Suppose that you have read 10 input characters during a conversion and that the stack now contains these symbols:



   Now, suppose that you read and process the 11th symbol of the input. Draw the stack for the case where the 11th symbol is:
   - A. A number:
   - B. A left parenthesis:
   - C. A right parenthesis:
   - D. A minus sign:
   - E. A division sign:

7. Write a program using stack for parenthesis matching. Explain what modifications would be needed to make the parenthesis matching algorithm check expressions with different kinds of parentheses such as (), [] and {}'s.

8. Evaluate the following prefix expressions:
   a) + * 2 + / 14 2 5 1
   b) - * 6 3 – 4 1
   c) + + 2 6 + - 13 2 4

9. Convert the following infix expressions to prefix notation:
   - a) ((A + 2) * (B + 4)) -1
   - b) Z – ((((X + 1) * 2) – 5) / Y)
   - c) ((C * 2) + 1) / (A + B)
   - d) ((A + B) * C – (D - E)) ↑ (F + G)

e) A – B / (C * D ↑ E)

10. Write a "C" function to copy one stack to another assuming (tcs 2016, 2018, 2019)

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]
   A. Queue
   B. Stack
   C. Linked List
   D. all the above

2. Which among the following is a Dynamic data structure: [A]
   A. Double Linked List C. Stack
   B. Queue D. all the above

3. Stack is referred as: [A]
   A. Last in first out list C. both A and B
   B. First in first out list D. none of the above

4. A stack is a data structure in which all insertions and deletions of entries are made at: [A]
   A. One end C. Both the ends
   B. In the middle D. At any position

5. A queue is a data structure in which all insertions and deletions are made respectively at: [A]
   A. rear and front C. front and rear
   B. front and front D. rear and rear

6. Transform the following infix expression to postfix form: [D]
   (A + B) * (C – D) / E
   A. A B * C + D / - C. A B + C D * - / E
   B. A B C * C D / - + D. A B + C D - * E /

7. Transform the following infix expression to postfix form: [B]
   A - B / (C * D)
   A. A B * C D - / C. / - D C * B A
   B. A B C D * / - D. - / * A B C D

8. Evaluate the following prefix expression: * - + 4 3 5 / + 2 4 3 [A]

   A. 4 C. 1
   B. 8 D. none of the above

9. Evaluate the following postfix expression: 1 4 18 6 / 3 + + 5 / + [C]
   A. 8 C. 3
   B. 2 D. none of the above

Chapter 5

Binary Trees

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

In this chapter in particular, we will explain special type of trees known as binary trees, which are easy to maintain in the computer.

TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

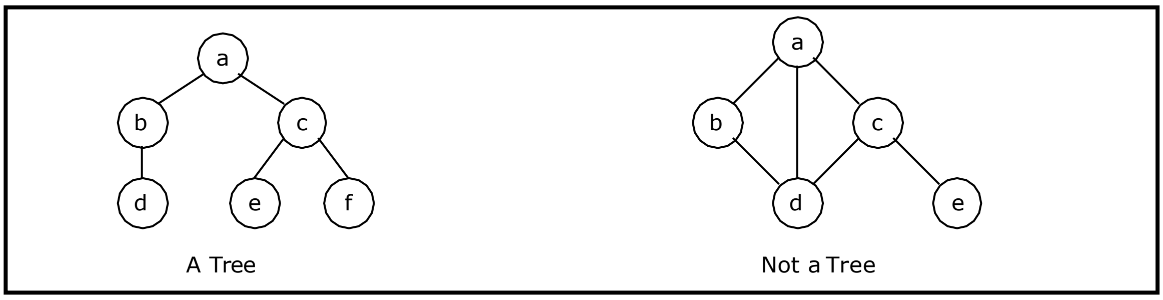


Figure 5.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

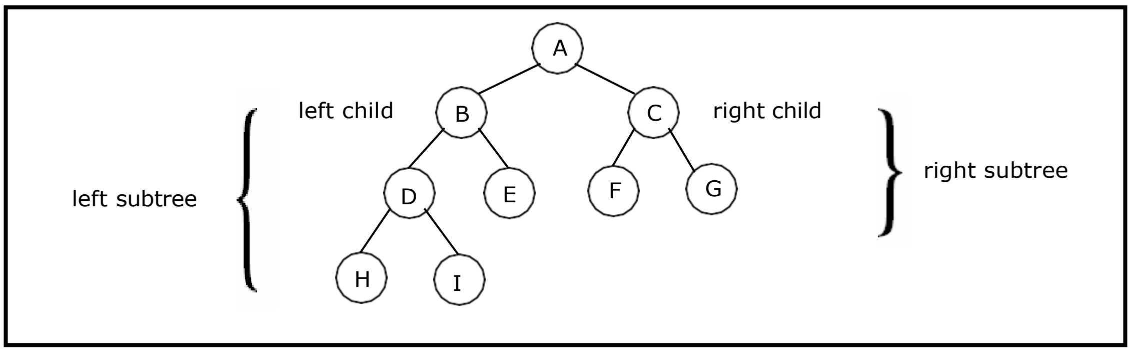


Figure 5.2.1. Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes n_1, n_2, . . ., n_k, such that n_i is the parent of n_{i+1} for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is* 2^n.

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

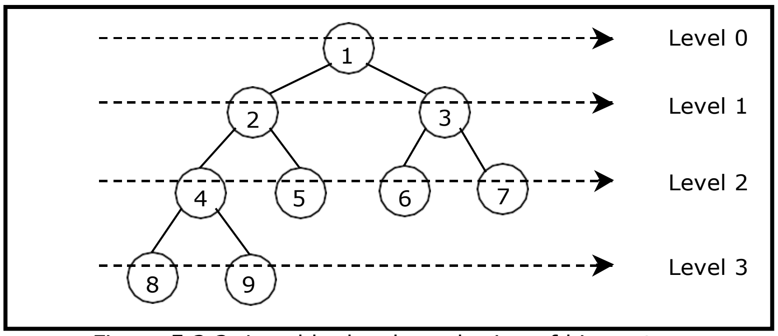


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then
   a. Maximum number of leaves = 2^h
   b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has 2^{h+1} - 1 nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.
A full binary tree of height h contains 2^h leaves and, 2^h - 1 non-leaf nodes.

Thus by induction, total number of nodes $(tn) = \sum_{l=0}^{h} 2^{l} = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

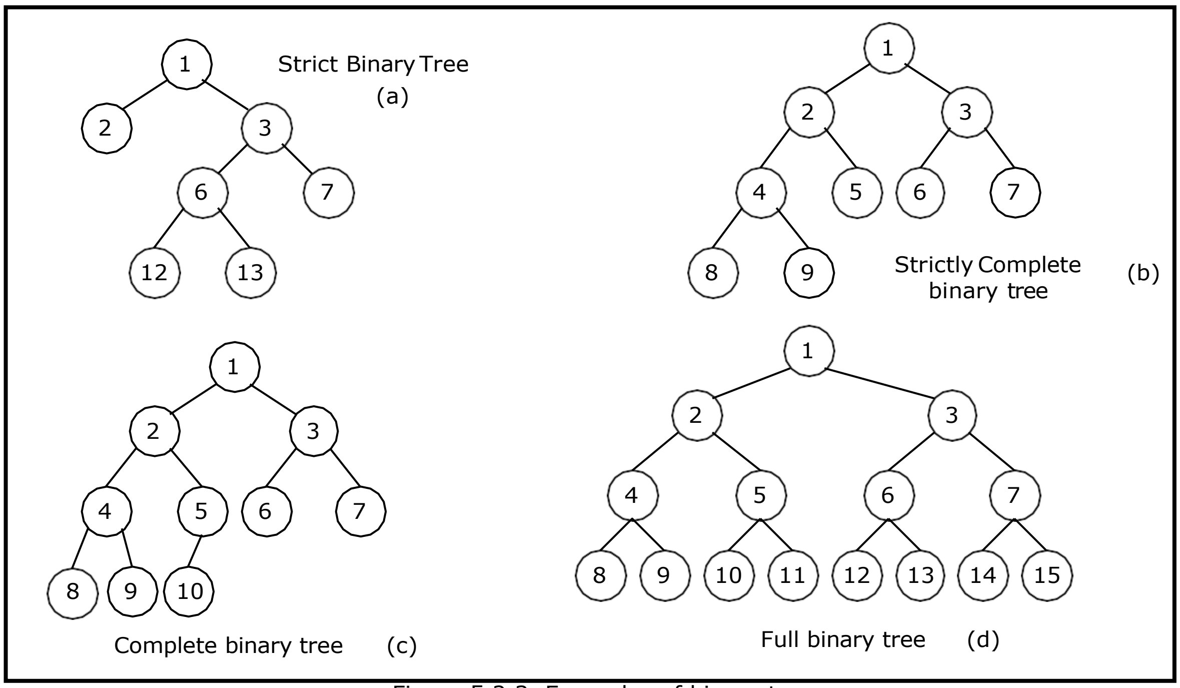


Figure 5.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

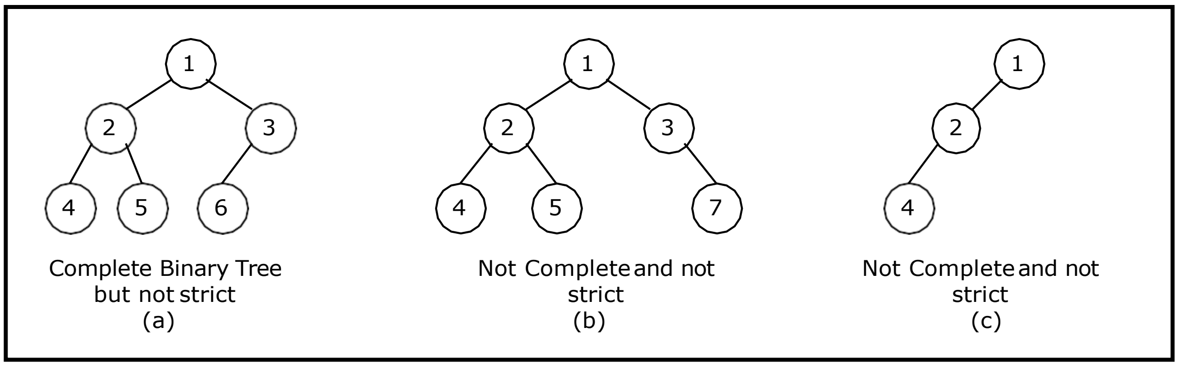

1
2
3
4
5
6
Complete Binary Tree but not strict
(a)
1
2
3
4
5
7
Not Complete and not strict
(b)
1
2
4
Not Complete and not strict
(c)

Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

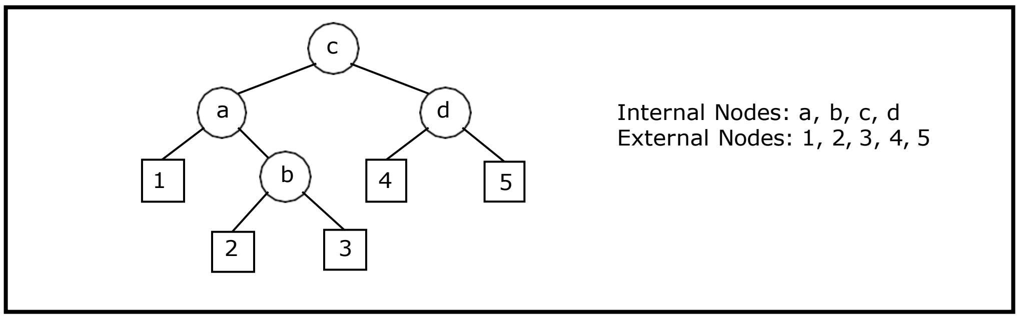


Figure 5.2.6. Internal and external nodes

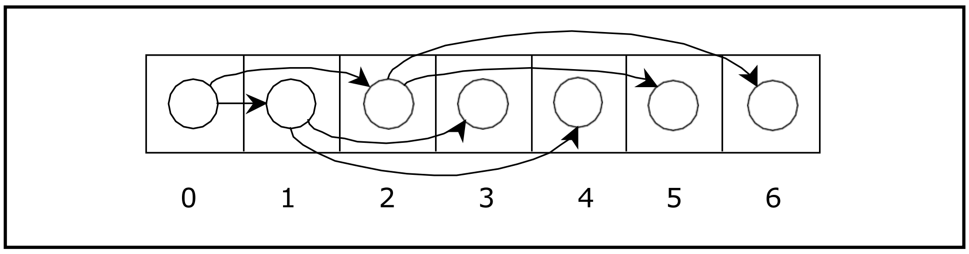
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to 2^h - n for a tree of height h with n nodes.



Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.
In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.

- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};

typedef struct binarytree node;

node *root = NULL;
```

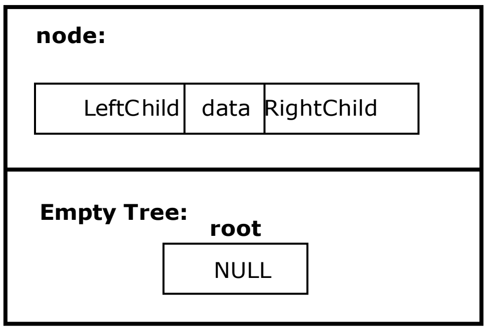


Figure 5.2.6. Structure definition, node representation and empty tree

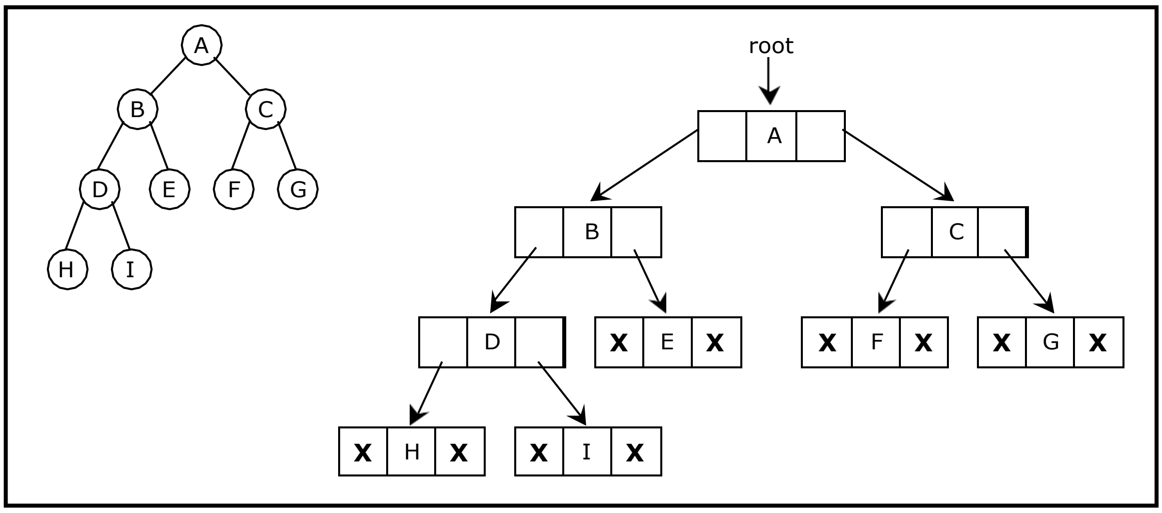


Figure 5.2.7. Linked representation for the binary tree

Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
                print root -> data;
                inorder(root->rchild);
        }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
        if( root != NULL )
        {
                print root -> data;
                preorder (root -> lchild);
                preorder (root -> rchild);
        }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
        if( root != NULL )
        {
                postorder (root -> lchild);
                postorder (root -> rchild);
                print (root -> data);
        }
}
```

Level order Traversal:

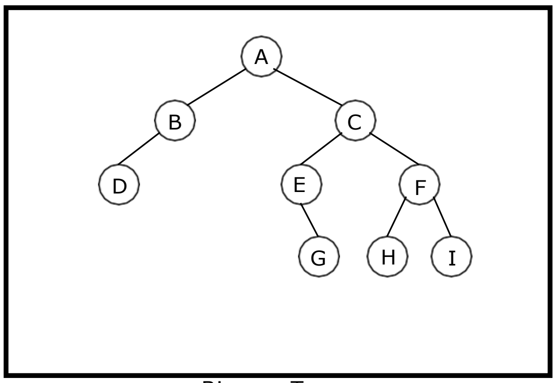
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        print tree[j] -> data;
        }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



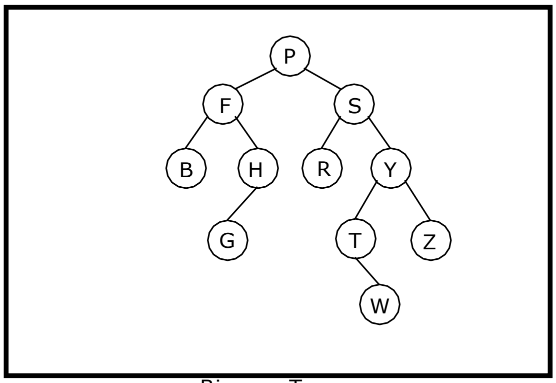
Binary Tree

- Preorder traversal yields:
  A, B, D, C , E, G , F , H, I
- Postorder traversal yields:
  D, B, G , E, H, I, F , C , A
- Inorder traversal yields:
  D, B, A, E, G , C , H, F , I
- Level order traversal yields:
  A, B, C , D, E, F , G , H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



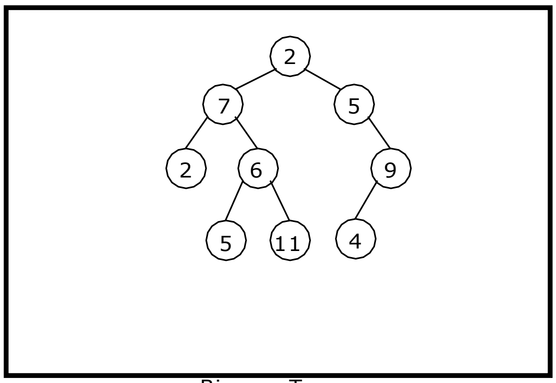
Bin ary Tree

- Preorder traversal yields:
  P , F , B, H, G , S, R, Y, T, W , Z
- Postorder traversal yields:
  B, G , H, F , R, W , T, Z, Y, S, P
- Inorder traversal yields:
  B, F , G , H, P , R, S, T, W , Y, Z
- Level order traversal yields:
  P , F , S, B, H, R, Y, G , T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



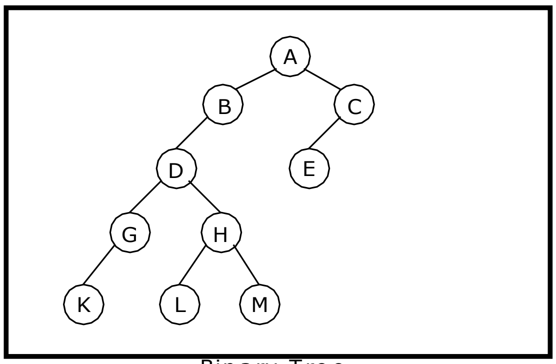
Binary Tree

- Preorder traversal yields:
  2 , 7, 2, 6, 5, 11 , 5, 9, 4
- Postorder travarsal yields:
  2 , 5, 11 , 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2 , 7, 5, 6, 11 , 2, 5, 4, 9
- Level order traversal yields:
  2 , 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  A, B, D, G , K, H, L, M , C , E
- Postorder travarsal yields:
  K, G , L, M , H, D, B, E, C , A
- Inorder travarsal yields:
  K, G , D, L, H, M , B, A, E, C
- Level order traversal yields:
  A, B, C , D, E, G , H, K, L, M

Pre, Post, Inorder and level order Traversing

Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

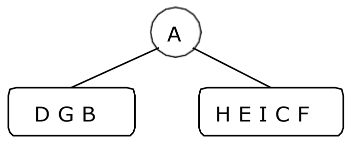
From Preorder sequence ***A*** *B D G C E H I F, the root is: A*

From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

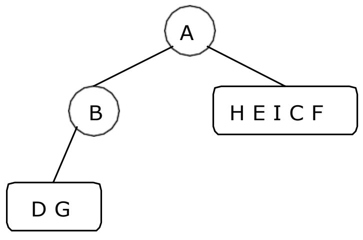


To find the root, left and right sub trees for D G B:

From the preorder sequence ***B*** *D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

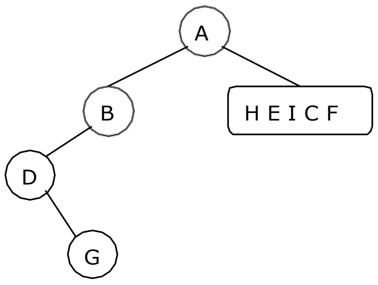


To find the root, left and right sub trees for D G:

From the preorder sequence **D** *G, the root of the tree is: D*

From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

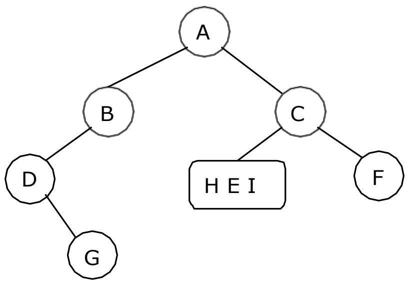


To find the root, left and right sub trees for H E I C F:

*From the preorder sequence **C** E H I F, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

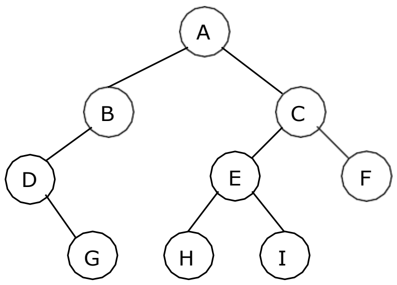


To find the root, left and right sub trees for H E I:

*From the preorder sequence **E** H I, the root of the tree is: E*

From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

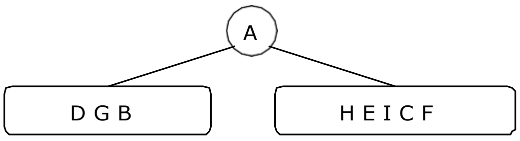
Solution:

From Postorder sequence G D B H I E F C **A**, *the root is: A*

From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B
Right sub tree is: H E I C F

The Binary tree upto this point looks like:

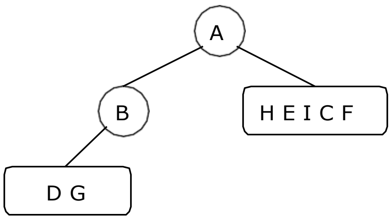


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

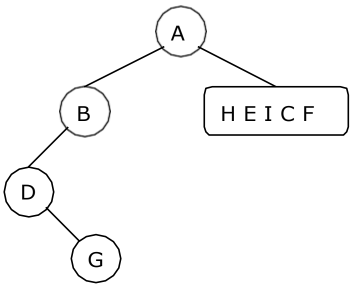


To find the root, left and right sub trees for D G:

From the postorder sequence G ***D****, the root of the tree is: D*

From the inorder sequence **D** G, we can find that is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:

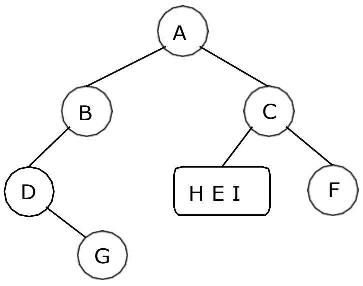


To find the root, left and right sub trees for H E I C F:

From the postorder sequence H I E F ***C****, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

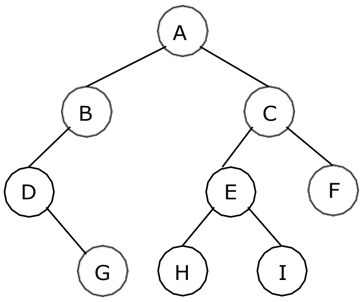


To find the root, left and right sub trees for H E I:

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

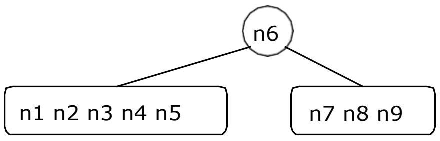
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

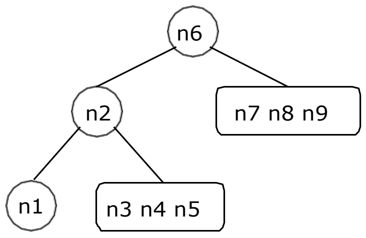
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

*From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

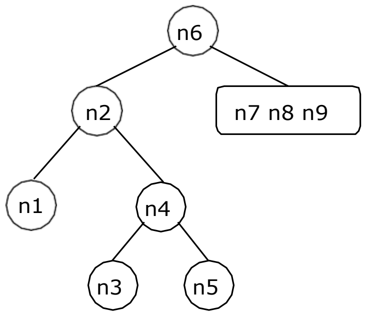


To find the root, left and right sub trees for n3 n4 n5:

*From the preorder sequence **n4** n3 n5, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

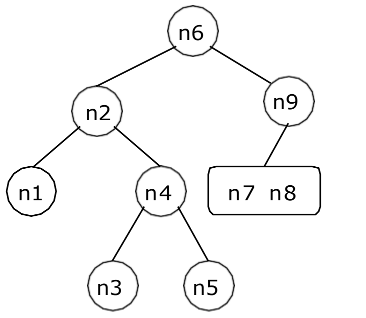


To find the root, left and right sub trees for n7 n8 n9:

*From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

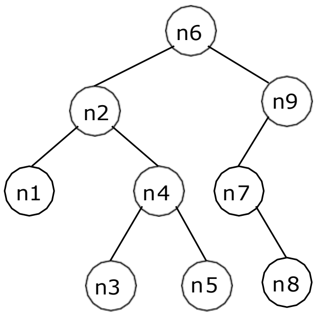


To find the root, left and right sub trees for n7 n8:

*From the preorder sequence **n7** n8, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

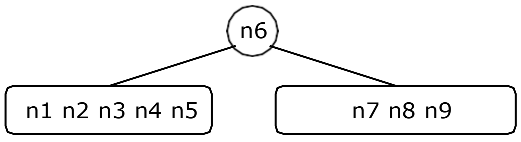
Solution:

From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

> *Left sub tree is: n1 n2 n3 n4 n5*
> *Right sub tree is: n7 n8 n9*

The Binary tree upto this point looks like:

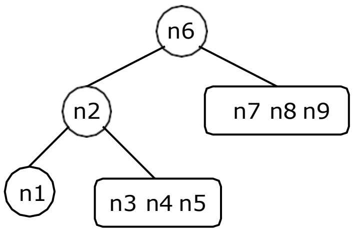


To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:

From the postorder sequence n1 n3 n5 n4 **n2***, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

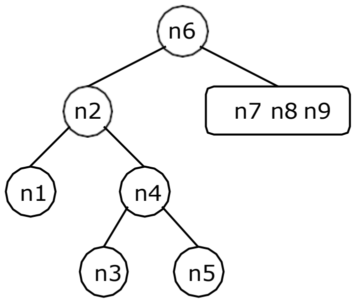
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence n3 n5 **n4***, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

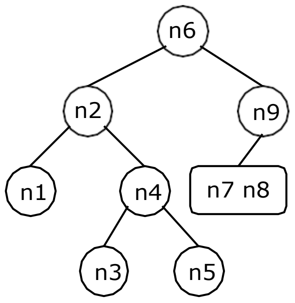


To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence n8 n7 **n9***, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

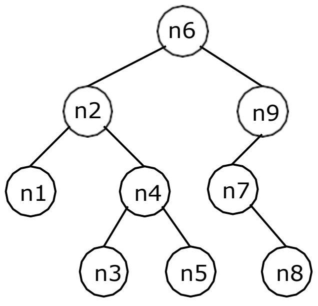
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

From the postorder sequence n8 **n7***, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



Binary Tree Creation and Traversal Using Pointers:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created
8. Deletes last node
9. Finds height of the tree created

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

struct tree
{
        struct tree* lchild;
        char data[10];
        struct tree* rchild;
};

typedef struct tree node;
node *Q[50];
int node_ctr;

node* getnode()
{
        node *temp ;
        temp = (node*) malloc(sizeof(node));
        printf("\n Enter Data: ");
        fflush(stdin);
        scanf("%s",temp->data);
        temp->lchild = NULL;
        temp->rchild = NULL;
        return temp;
}
```

```
void create_binarytree(node *root)
{
        char option;
        node_ctr = 1;
        if( root != NULL )
        {
                printf("\n Node %s has Left SubTree(Y/N)",root->data);
                fflush(stdin);
                scanf("%c",&option);
                if( option=='Y' || option == 'y')
                {
                        root->lchild = getnode();
                        node_ctr++;
                        create_binarytree(root->lchild);
                }
                else
                {
                        root->lchild = NULL;
                        create_binarytree(root->lchild);
                }

                printf("\n Node %s has Right SubTree(Y/N) ",root->data);
                fflush(stdin);
                scanf("%c",&option);
                if( option=='Y' || option == 'y')
                {
                        root->rchild = getnode();
                        node_ctr++;
                        create_binarytree(root->rchild);
                }
                else
                {
                        root->rchild = NULL;
                        create_binarytree(root->rchild);
                }
        }
}

void make_Queue(node *root,int parent)
{
        if(root != NULL)
        {
                node_ctr++;
                Q[parent] = root;
                make_Queue(root->lchild,parent*2+1);
                make_Queue(root->rchild,parent*2+2);
        }
}

delete_node(node *root, int parent)
{
        int index = 0;
        if(root == NULL)
                printf("\n Empty TREE ");
        else
        {
                node_ctr = 0;
                make_Queue(root,0);
                index = node_ctr-1;
                Q[index] = NULL;
                parent = (index-1) /2;
                if( 2* parent + 1 == index )
                        Q[parent]->lchild = NULL;
```

```
            else
                  Q[parent]->rchild = NULL;
      }
      printf("\n Node Deleted ..");
}

void inorder(node *root)
{
      if(root != NULL)
      {
            inorder(root->lchild);
            printf("%3s",root->data);
            inorder(root->rchild);
      }
}

void preorder(node *root)
{
      if( root != NULL )
      {
            printf("%3s",root->data);
            preorder(root->lchild);
            preorder(root->rchild);
      }
}

void postorder(node *root)
{
      if( root != NULL )
      {
            postorder(root->lchild);
            postorder(root->rchild);
            printf("%3s", root->data);
      }
}

void print_leaf(node *root)
{
      if(root != NULL)
      {
            if(root->lchild == NULL && root->rchild == NULL)
              printf("%3s ",root->data);
              print_leaf(root->lchild);
              print_leaf(root->rchild);
      }
}

int height(node *root)
{
      if(root == NULL)
            return -1;
      else
            return (1 + max(height(root->lchild), height(root->rchild)));
}

void print_tree(node *root, int line)
{
      int i;
      if(root != NULL)
      {
            print_tree(root->rchild,line+1);
            printf("\n");
            for(i=0;i<line;i++)
            printf(" ");
        printf("%s", root->data);
        print_tree(root->lchild,line+1);
    }
}

void level_order(node *Q[],int ctr)
{
    int i;
    for( i = 0; i < ctr ; i++)
    {
        if( Q[i] != NULL )
            printf("%5s",Q[i]->data);
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create Binary Tree ");
    printf("\n 2. Inorder Traversal ");
    printf("\n 3. Preorder Traversal ");
    printf("\n 4. Postorder Traversal ");
    printf("\n 5. Level Order Traversal");
    printf("\n 6. Leaf Node ");
    printf("\n 7. Print Height of Tree ");
    printf("\n 8. Print Binary Tree ");
    printf("\n 9. Delete a node ");
    printf("\n 10. Quit ");
    printf("\n Enter Your choice: ");
    scanf("%d", &ch);
    return ch;
}
void main()
{
    int i,ch;
    node *root = NULL;
    do
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                if( root == NULL )
                {
                    root = getnode();
                }   create_binarytree(root);
                else
                {

                }   printf("\n Tree is already Created ..");
                break;
            case 2 :
                printf("\n Inorder Traversal: ");
                inorder(root);
                break;
            case 3 :
                printf("\n Preorder Traversal: ");
                preorder(root);
                break;
```

```
                    case 4 :
                            printf("\n Postorder Traversal: ");
                            postorder(root);
                            break;
                    case 5:
                            printf("\n Level Order Traversal ..");
                            make_Queue(root,0);
                            level_order(Q,node_ctr);
                            break;
                    case 6 :
                            printf("\n Leaf Nodes: ");
                            print_leaf(root);
                            break;
                    case 7 :
                            printf("\n Height of Tree: %d ", height(root));
                            break;
                    case 8 :
                            printf("\n Print Tree \n");
                            print_tree(root, 0);
                            break;
                    case 9 :
                                    delete_node(root,0);
                                    break;
                    case 10 :
                            exit(0);
            }
            getch();
    }while(1);
}
```

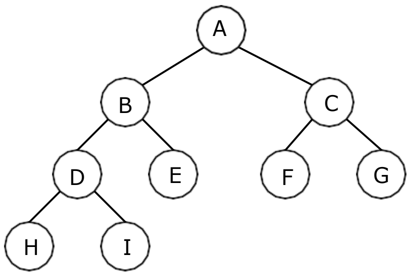
Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are 2n total links, there are n+1 null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

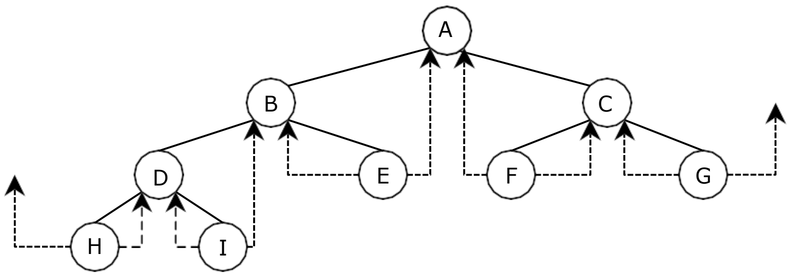
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the RCHILD(p) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:

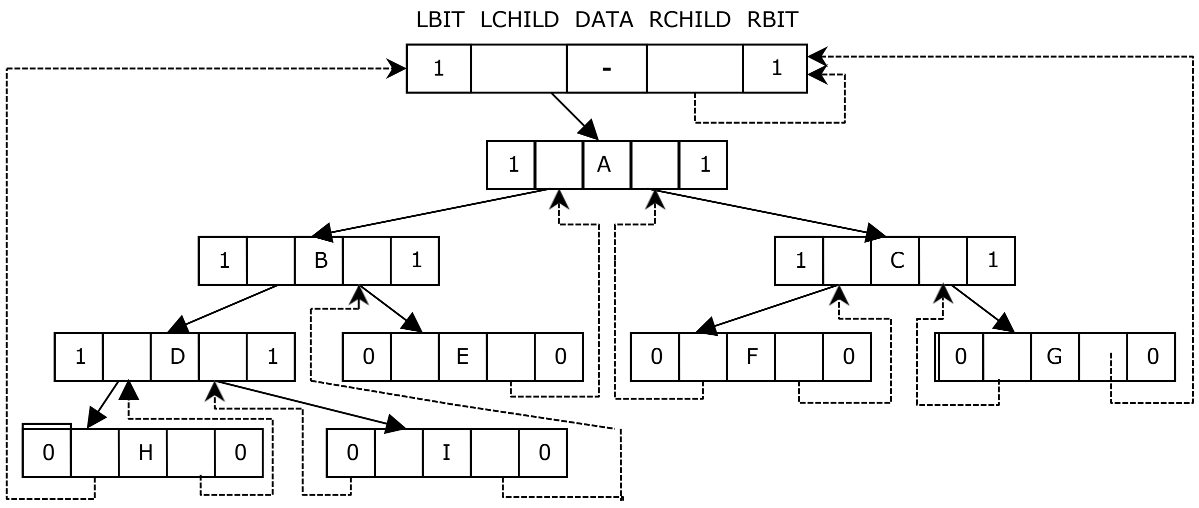


The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

LBIT(P) = 1	if LCHILD(P) is a normal pointer
LBIT(P) = 0	if LCHILD(P) is a Thread
RBIT(P) = 1	if RCHILD(P) is a normal pointer
RBIT(P) = 0	if RCHILD(P) is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

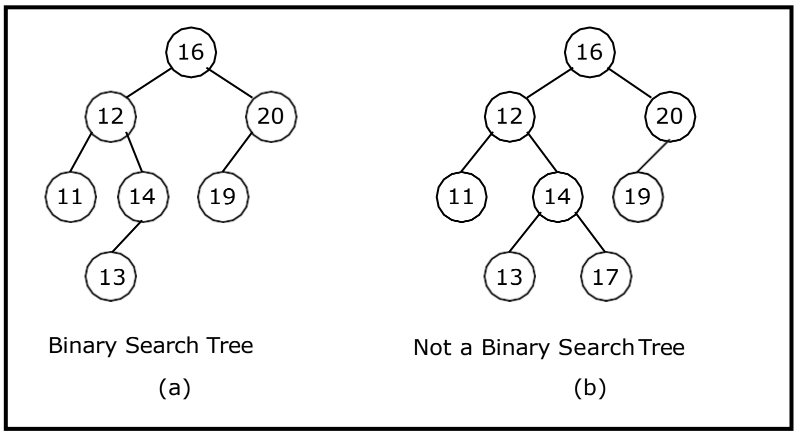


Figure 5.2.5. Examples of binary search trees

General Trees (m-ary tree):

If in a tree, the outdegree of every node is less than or equal to ***m***, the tree is called general tree. The general tree is also called as an *m-ary* tree. If the outdegree of every node is exactly equal to m or zero then the tree is called *a full or complete m-ary* tree. For m = 2, the trees are called *binary* and *full binary trees.*

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

Converting a *m-ary* tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

Conversion from general tree to binary can be done in two stages.

Stage 1:

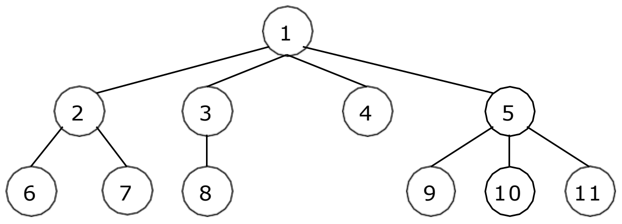
- As a first step, we delete all the branches originating in every node except the left most branch.
- We draw edges from a node to the node on the right, if any, which is situated at the same level.

Stage 2:

- Once this is done then for any particular node, we choose its left and right sons in the following manner:
  - The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

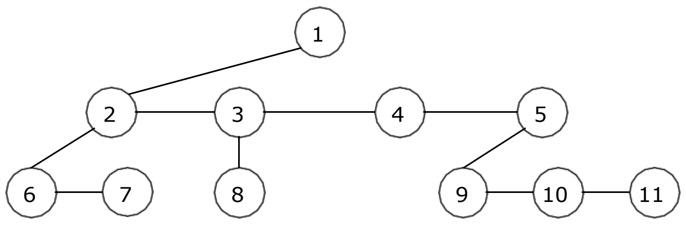
Example 1:

Convert the following ordered tree into a binary tree:

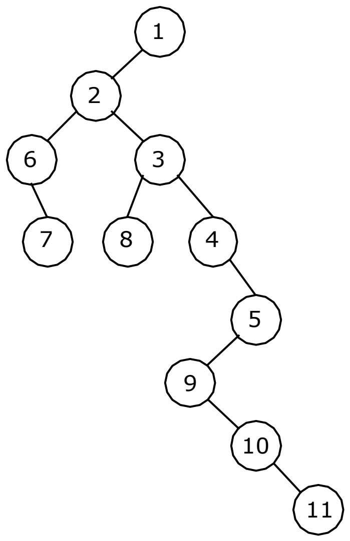


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:

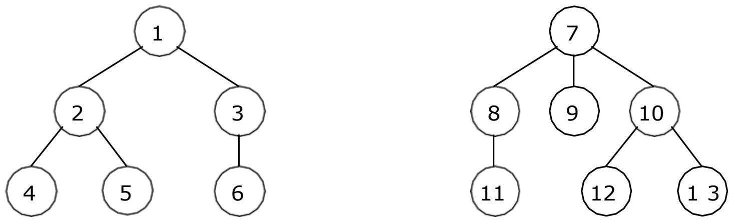


Stage 2 tree by using the above mentioned procedure is as follows:



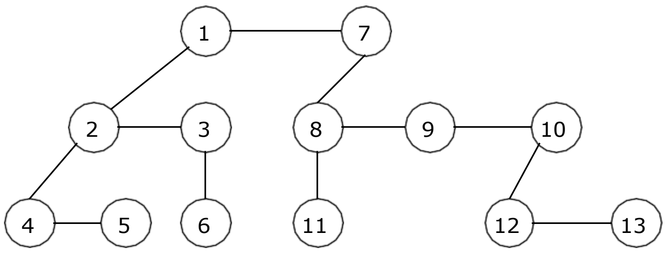
Example 2:

Construct a unique binary tree from the given forest.

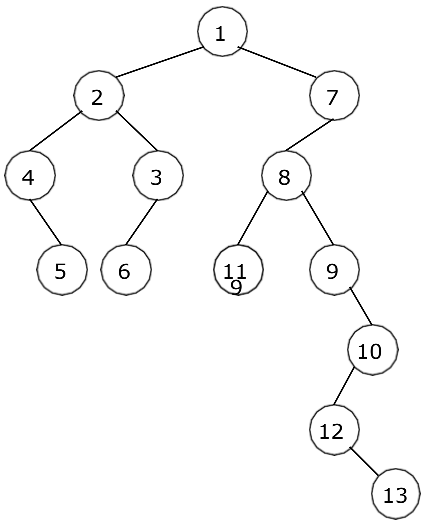


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



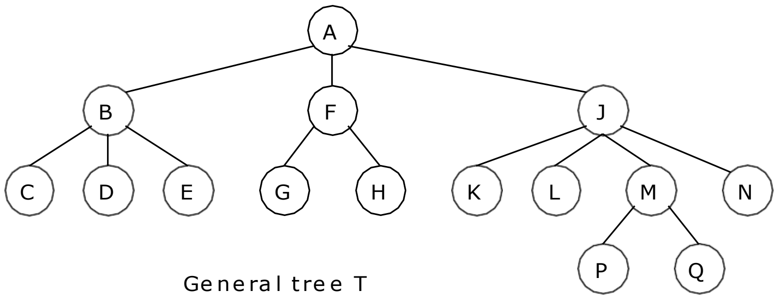
Stage 2 tree by using the above mentioned procedure is as follows (binary tree representation of forest):



Example 3:

For the general tree shown below:

1. Find the corresponding binary tree T′.
2. Find the preorder traversal and the postorder traversal of T.
3. Find the preorder, inorder and postorder traversals of T′.
4. Compare them with the preorder and postorder traversals obtained for T′ with the general tree T.

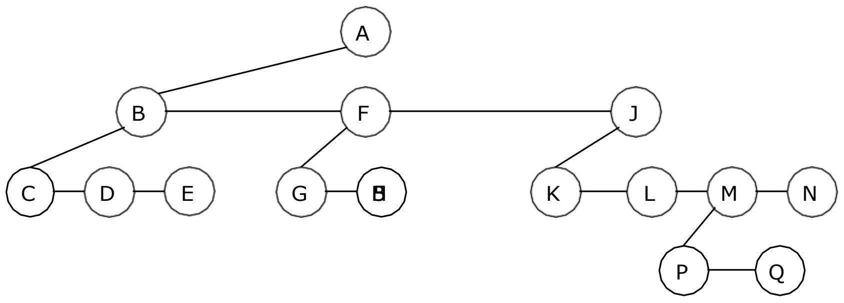


General tree T

Solution:

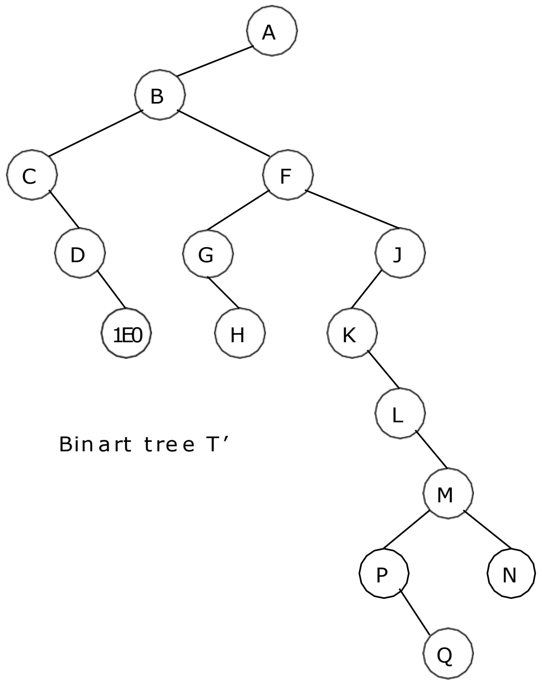
1. Stage 1:

The tree by using the above-mentioned procedure is as follows:



Stage 2:

The binary tree by using the above-mentioned procedure is as follows:



2. Suppose T is a general tree with root R and subtrees T_1, T_2, ………., T_M. The preorder traversal and the postorder traversal of T are:

Preorder:	1) Process the root R.
	2) Traverse the subtree T_1, T_2, ……., T_M in preorder.
Postorder:	1) Traverse the subtree T_1, T_2, ……., T_M in postorder.
	2) Process the root R.

The tree T has the root A and subtrees T_1, T_2 and T_3 such that:

T_1 consists of nodes B, C, D and E.

T_2 consists of nodes F, G and H.

T_3 consists of nodes J, K, L, M, N, P and Q.

A. The preorder traversal of T consists of the following steps:

(i) Process root A.

(ii) Traverse T_1 in preorder: Process nodes B, C, D, E.

(iii) Traverse T_2 in preorder: Process nodes F, G, H.

(iv) Traverse T_3 in preorder: Process nodes J, K, L, M, P, Q, N.

The preorder traversal of T is as follows:

A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

B. The postorder traversal of T consists of the following steps:

(i) Traverse T_1 in postorder: Process nodes C, D, E, B.

(ii) Traverse T_2 in postorder: Process nodes G, H, F.

(iii) Traverse T_3 in postorder: Process nodes K, L, P, Q, M, N, J.

(iv) Process root A.

The postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

3. The preorder, inorder and postorder traversals of the binary tree T' are as follows:

Preorder:	A, B, C, D, E, F, G, H, J, K, M, P, Q, N
Inorder:	C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A
Postorder:	E, D, C, H, G, Q, P, N, M, L, K, J, F, B, A

4. Comparing the preorder and postorder traversals of T' with the general tree T:

We can observer that the preorder of the binary tree T' is identical to the preorder of the general T.

The inorder traversal of the binary tree T' is identical to the postorder traversal of the general tree T.

There is no natural traversal of the general tree T which corresponds to the postorder traversal of its corresponding binary tree T'.

Search and Traversal Techniques for m-ary trees:

Search involves visiting nodes in a tree in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

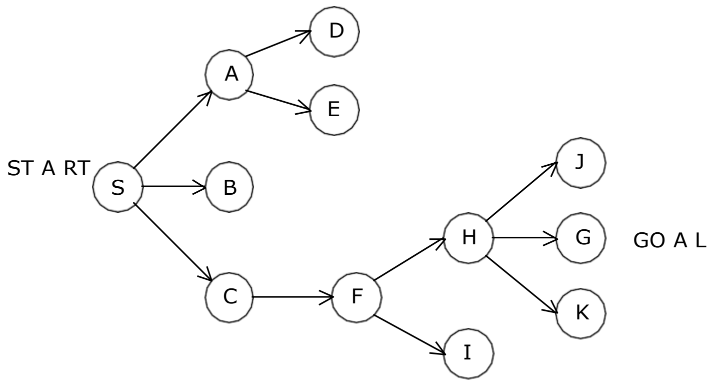
1. Depth first search or traversal.
2. Breadth first search or traversal.

Depth first search:

In Depth first search, we begin with root as a start state, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state. One simple way to implement depth first search is to use a stack data structure consisting of root node as a start state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

To illustrate this let us consider the tree shown below.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state.

So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

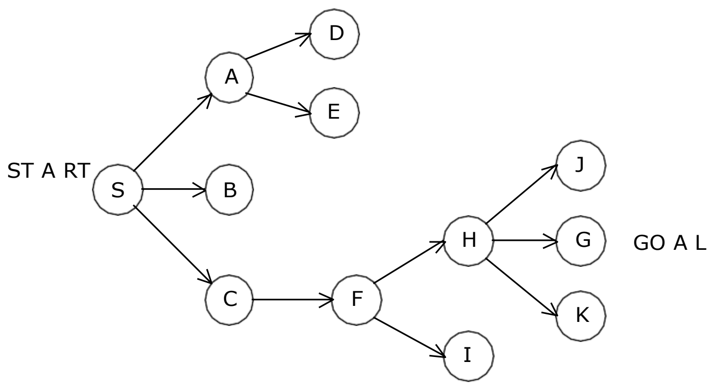
   To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. We cannot come up with shortest solution to the problem.

Breadth first search:

Breadth-first search starts at root node S and "discovers" which vertices are reachable from S. Breadth-first search discovers vertices in increasing order of distance. Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4.

So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state.

Sparse Matrices:

A sparse matrix is a two–dimensional array having the value of majority elements as null. The density of the matrix is the number of non-zero elements divided by the total number of matrix elements. The matrices with very low density are often good for use of the sparse format. For example,

$$A = \begin{bmatrix} 0 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{bmatrix}$$

As far as the storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise technique such that only non-null elements will be stored. The matrix A produces:

```
        (3, 1)  1
        (2, 2)  2
S =     (3, 2)  3
        (4, 3)  4
        (1, 4)  5
```

The printed output lists the non-zero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.
In large number of applications, sparse matrices are involved. One approach is to use the linked list.

EXCERCISES

1. How many different binary trees can be made from three nodes that contain the key value 1, 2, and 3? (tcs 2017; byju's 2019)

2. a. Draw all the possible binary trees that have four leaves and all the nonleaf nodes have no children.
   b. Show what would be printed by each of the following.
      An inorder traversal of the tree
      A postorder traversal of the tree
      A preorder traversal of the tree

3. a. Draw the binary search tree whose elements are inserted in the following order:
      50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95 (magic bricks 2018)

   b. What is the height of the tree?
   c. What nodes are on level?
   d. Which levels have the maximum number of nodes that they could contain?
   e. What is the maximum height of a binary search tree containing these nodes? Draw such a tree?
   f. What is the minimum height of a binary search tree containing these nodes? Draw such a tree?

   g. Show how the tree would look after the deletion of 29, 59 and 47?

   h. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9 and 10 (in that order).

4. Write a "C" function to determine the height of a binary tree. (infosys 2017)

5. Write a "C" function to count the number of leaf nodes in a binary tree. (infosys 2017, 2018)

6. Write a "C" function to swap a binary tree.

7. Write a "C" function to compute the maximum number of nodes in any level of a binary tree. The maximum number of nodes in any level of a binary tree is also called the width of the tree.

8. Construct two binary trees so that their postorder traversal sequences are the same.

9. Write a "C" function to compute the internal path length of a binary tree.

10. Write a "C" function to compute the external path length of a binary tree.

Multiple Choice Questions

1. The node that has no children is referred as: [C]
   A. Parent node C. Leaf node
   B. Root node D. Sibblings

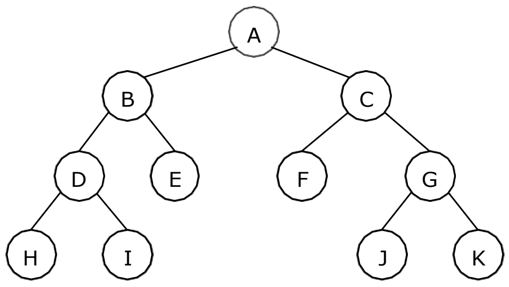
2. A binary tree in which all the leaves are on the same level is called as: [B]
   A. Complete binary tree C. Strictly binary tree
   B. Full binary tree D. Binary search tree

3. How can the graphs be represented? [C]
   A. Adjacency matrix
   B. Adjacency list
   C. Incidence matrix
   D. All of the above

4. The children of a same parent node are called as: [C]
   A. adjacent node C. Sibblings
   B. non-leaf node D. leaf node

5. A tree with n vertices, consists of________edges. [A]
   A. n – 1 C. n
   B. n - 2 D. log n

6. The maximum number of nodes at any level is: [B]
   A. n C. n + 1
   B. 2^n D. 2n

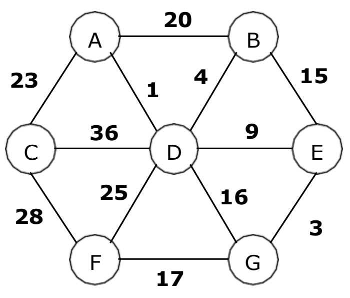


FI GUR E 1

7. For the Binary tree shown in fig. 1, the in-order traversal sequence is: [C]
   A. A B C D E F G H I J K C. H D I B E A F C J G K
   B. H I D E B F J K G C A D. A B D H I E C F G J K

8. For the Binary tree shown in fig. 1, the pre-order traversal sequence is: [D]
   A. A B C D E F G H I J K C. H D I B E A F C J G K
   B. H I D E B F J K G C A D. A B D H I E C F G J K

9. For the Binary tree shown in fig. 1, the post-order traversal sequence is: [B]
   A. A B C D E F G H I J K C. H D I B E A F C J G K
   B. H I D E B F J K G C A D. A B D H I E C F G J K



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 2 and its adjacency list

10. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 2 shown above:
   A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
   B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
   C. both A and B
   D. none of the above

Chapter 6

Graphs

Introduction to Graphs:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote n = |V|, e = |E|.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be complete if every node a in G is adjacent to every other node v in G. A complete graph with n nodes will have n(n-1)/2 edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u. On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u. For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

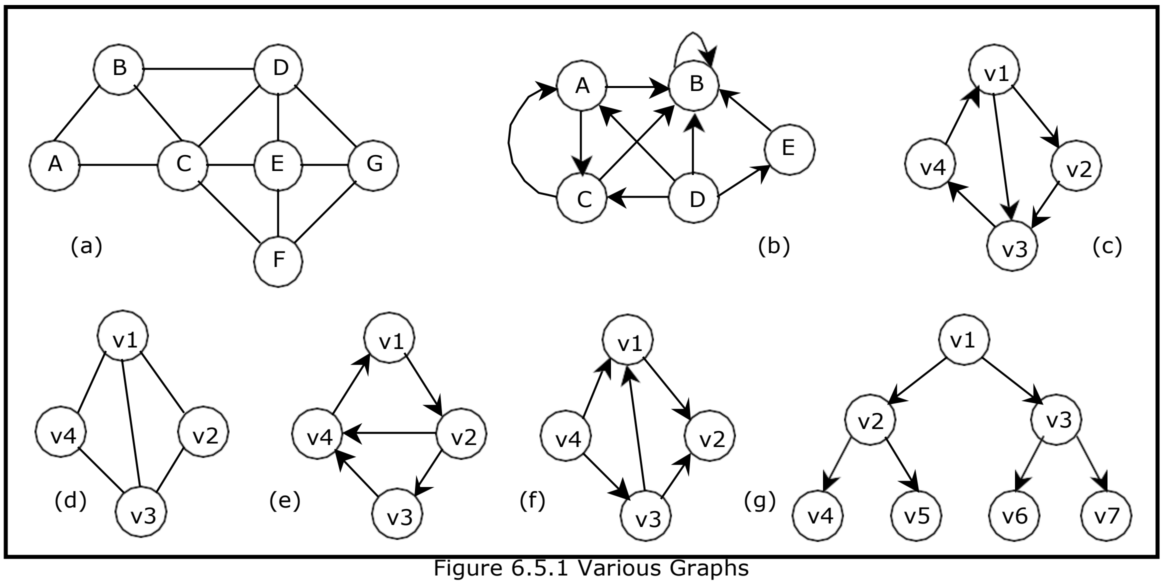


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge e $\in$ E. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in−degree of the vertex (denote indeg(v)). The number of outgoing edges from a vertex is called out-degree (denote outdeg(v)). For example, let us consider the digraph shown in figure 6.5.1(f),

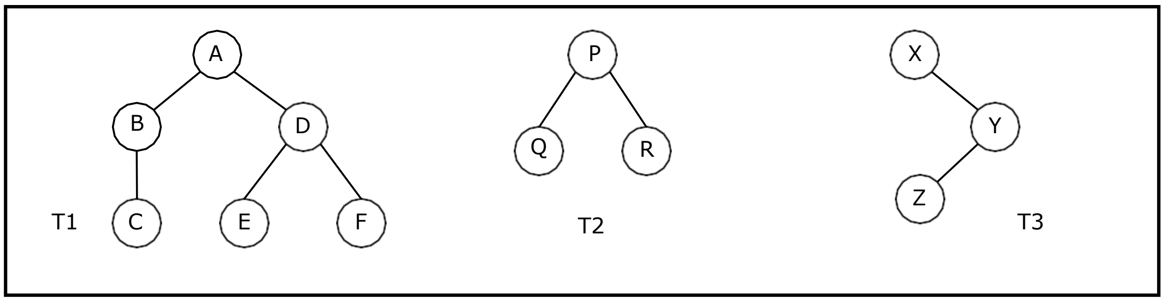
indegree(v_1) = 2	outdegree(v_1) = 1
indegree(v_2) = 2	outdegree(v_2) = 0

A path is a sequence of vertices (v_1, v_2, , v_k), where for all i, (v_i, v_{i+1}) ε E. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph G′ = (V′, E′) is a sub-graph of graph G = (V, E) iff V′ $\subseteq$ V and E′ $\subseteq$ E.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T1, T2 and T3.



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G. Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

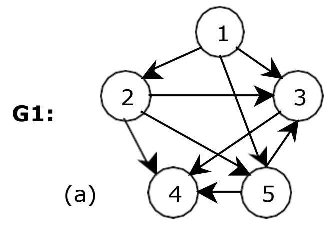
Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \textit{if there is an edge from } v_i \textit{ to } v_j \\ 0 & \textit{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

G1:



(a) (b)

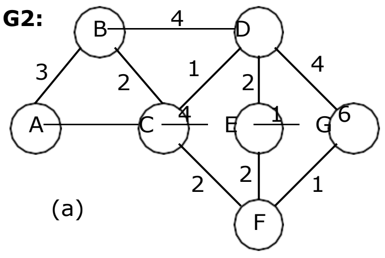
	1	**2**	**3**	**4**	**5**
1	0	1	1	0	1
2	0	0	1	1	1
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	1	1	0

Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G2 shown in figure 6.5.3(a).

G2:



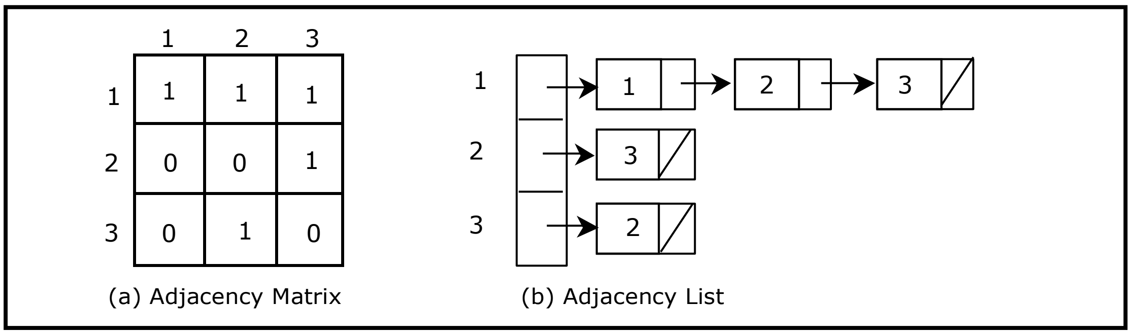
(a) (b)

	A	**B**	**C**	**D**	**E**	**F**	**G**
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝
D	∝	4	1	0	2	∝	4
E	∝	∝	4	2	0	2	1
F	∝	∝	2	∝	2	0	1
G	∝	∝	∝	4	1	1	0

Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, ... n] of pointers where for $1 \leq v \leq n$, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list in shown in figure 6.5.4 (b).



$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}$$

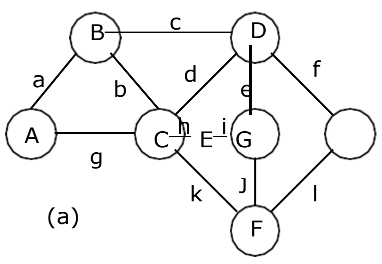
Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \textit{if there is an edge } j \textit{ incident to } v_i \\ 0 & \textit{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.



(b)

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0
D	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	0	1	0	0	1	1	1	0	0
F	0	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	0	1	0	0	1	0	0	1

Figure 6.5.4 Graph and its incidence matrix

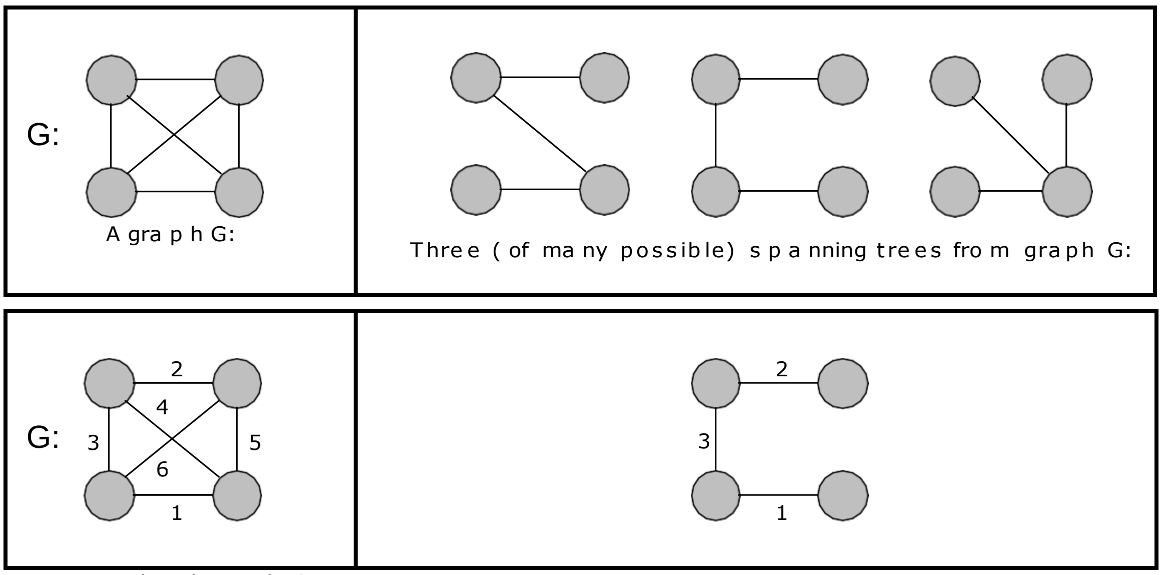
Figure 6.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 6.5.4(a).

Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w(T) is the sum of weights of all edges in T. Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



A graph G:

Three (of many possible) spanning trees from graph G:

A weighted graph G:

The minimal spanning tree from weighted graph G:

Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal’s algorithm and
2. Prim’s algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim’s algorithm uses vertex connections* in determining the MST. In *Prim’s algorithm at any instance of output it represents tree* whereas in *Kruskal’s algorithm at any instance of output it may represent tree or not*.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

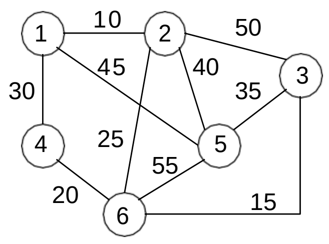
1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T

   then Add (v, w) to T

   else discard (v, w)
6. If T contains fewer than n - 1 edges then print no spanning tree.

Example 1:

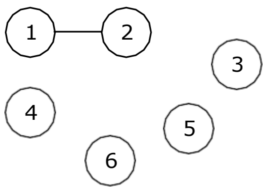
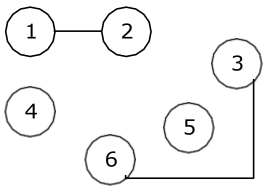
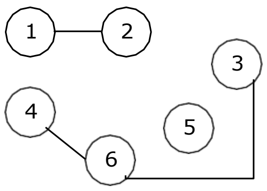
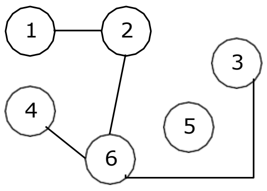
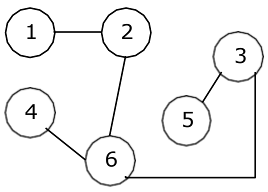
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

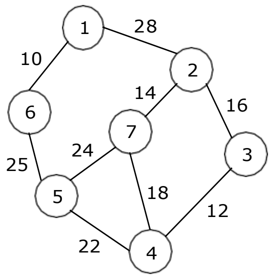
Cost	***10***	***15***	***20***	***25***	***30***	***35***	***40***	***45***	***50***	***55***
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	**COST**	**STAGES IN KRUSKAL'S ALGORITHM**	**REMARKS**
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:

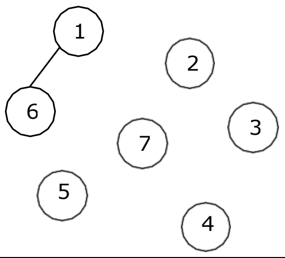
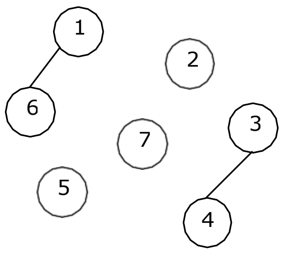
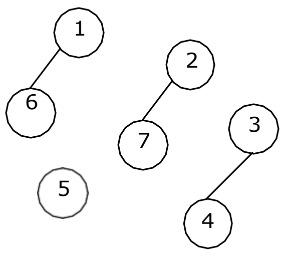


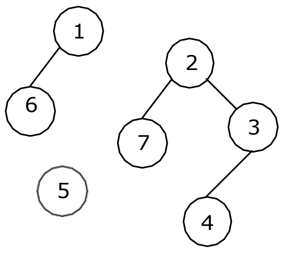
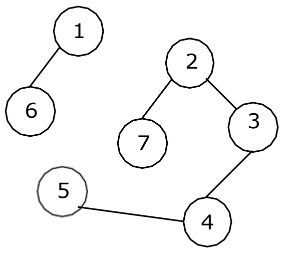
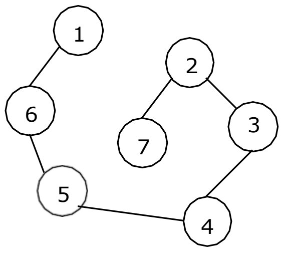
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***12***	***14***	***16***	***18***	***22***	***24***	***25***	***28***
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	**COST**	**STAGES IN KRUSKAL'S ALGORITHM**	**REMARKS**
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

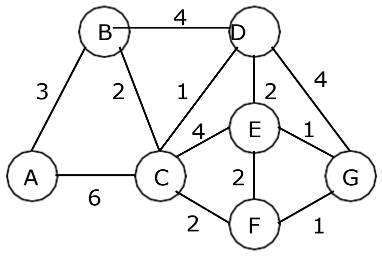
E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or $\propto$ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost [k, l];
    t [1, 1] := k; t [1, 2] := l;
    for  i :=1 to n do                          // Initialize near
        if (cost [i, l] < cost [i, k]) then near [i] := l;
        else near [i] := k;
    near [k] :=near [l] := 0;
    for  i:=2 to n - 1 do                       // Find n - 2 additional edges for t.
    {
        Let j be an index such that near [j] $\neq$ 0 **and**
        cost [j, near [j]] is minimum;
        t [i, 1] := j; t [i, 2] := near [j];
        mincost := mincost + cost [j, near [j]];
        near [j] := 0
        for  k:= 1 to n do                      // Update near[].
            if ((near [k] $\neq$ 0) **and** (cost [k, near [k]] > cost [k, j]))
                then near [k] := j;
    }
    return mincost;
}
```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.

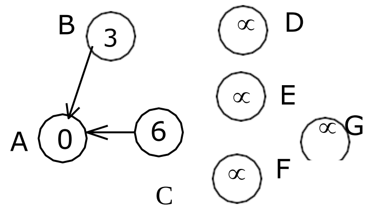


Solution:

The cost adjacency matrix is $\begin{bmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{bmatrix}$

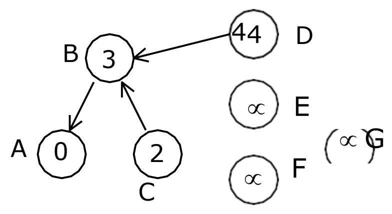
The stepwise progress of the prim's algorithm is as follows:

Step 1:



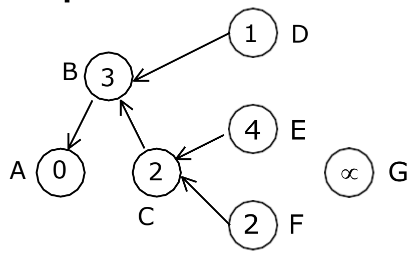
Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



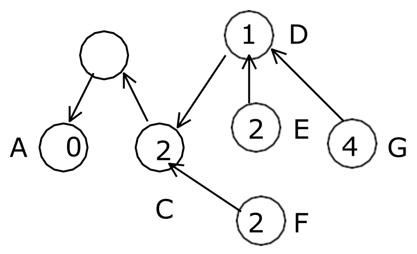
Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



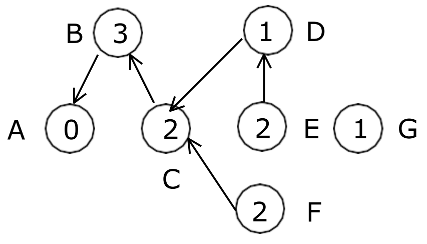
Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step 4:



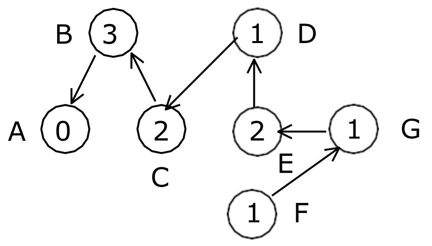
Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



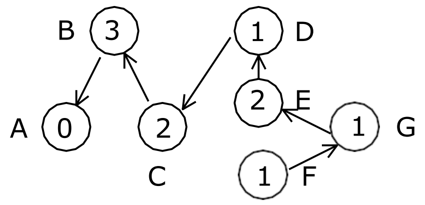
Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

1. STATUS = 1 (Ready state): The initial state of the node N.
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of neighbors of A. And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A.

```
Initialize all nodes to the ready state (STATUS = 1).

1.  Put the starting node A in QUEUE and change its status to the waiting
    state (STATUS = 2).

2.  Repeat the following steps until QUEUE is empty:

    a.  Remove the front node N of QUEUE. Process N and change the
        status of N to the processed state (STATUS = 3).

    b.  Add to the rear of QUEUE all the neighbors of N that are in the
        ready state (STATUS = 1), and change their status to the waiting
        state (STATUS = 2).

3.  Exit
```

Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFT except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

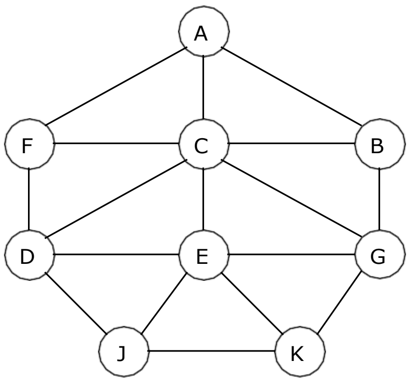
The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

```
4.  Initialize all nodes to the ready state (STATUS = 1).

5.  Push the starting node A into STACK and change its status to the waiting state
    (STATUS = 2).

6.  Repeat the following steps until STACK is empty:

    a.  Pop the top node N from STACK. Process N and change the status of N to
        the processed state (STATUS = 3).

    b.  Push all the neighbors of N that are in the ready state (STATUS = 1), and
        change their status to the waiting state (STATUS = 2).
7.  Exit.
```

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Gra p h G

Node	*Adjacency List*
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Depth-first search and traversal:

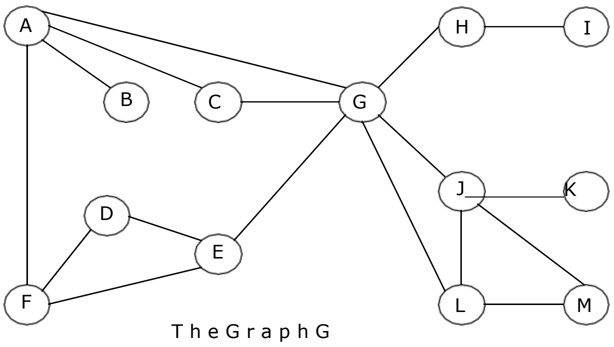
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

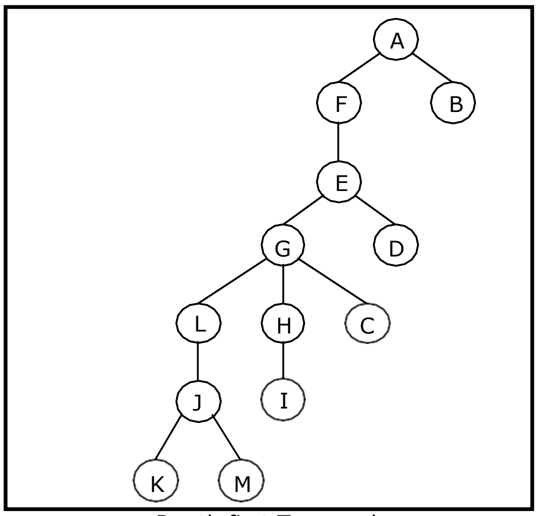


The Graph G

Node	***Adjacency List***
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

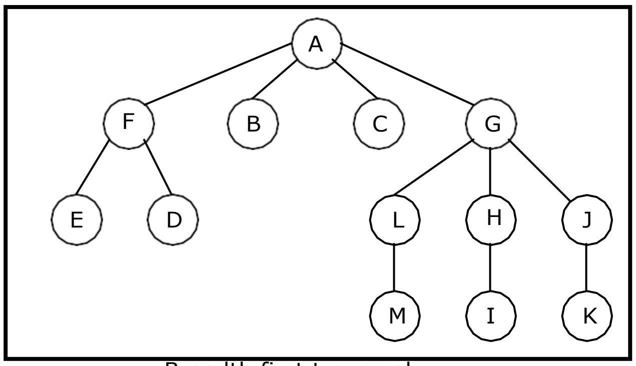
The adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F E G L J K M H I C D B***. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

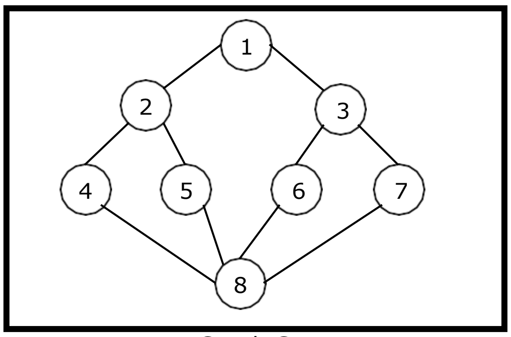
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F B C G E D L H J M I K***. The breadth first spanning tree is shown in the figure given below:



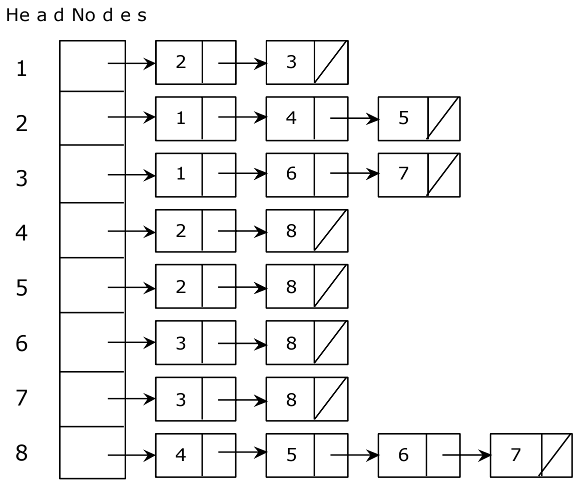
Breadth first traversal

Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



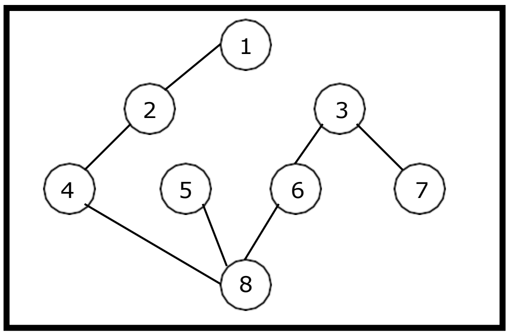
Graph G



Adjacency list for graph G

Depth first search and traversal:

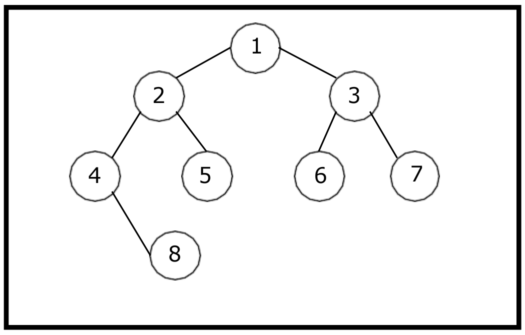
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:



Breadth First Spanning Tree

EXCERCISES

1. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges. (wipro 2016, 2018)
2. Show that the number of vertices of odd degree in a finite graph is even.
3. How many edges are contained in a complete graph of "n" vertices.
4. Show that the number of spanning trees in a complete graph of "n" vertices is $2^{n-1} - 1$.
5. Prove that the edges explored by a breadth first or depth first traversal of a connected graph from a tree.
6. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
7. Write a "C" function to generate the incidence matrix of a graph from its adjacency matrix.
8. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
9. Rewrite the algorithms "BFSearch" and "DFSearch" so that it works on adjacency matrix representation of graphs.
10. Write a "C" function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)

Multiple Choice Questions

1. How can the graphs be represented? [D]
   A. Adjacency matrix C. Incidence matrix
   B. Adjacency list D. All of the above

2. The depth-first traversal in graph is analogous to tree traversal: [C]
   A. In-order C. Pre-order
   B. Post-order D. Level order

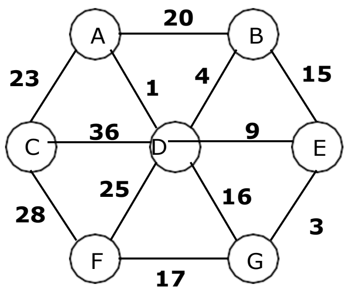
3. The children of a same parent node are called as: [C]
   A. adjacent node C. Sibblings
   B. non-leaf node D. leaf node

4. Complete graphs with n nodes will have__________edges. [C]

A. n - 1 C. n(n-1)/2
B. n/2 D. (n – 1)/2

5. A graph with no cycle is called as: [C]
A. Sub-graph C. Acyclic graph
B. Directed graph D. none of the above

6. The maximum number of nodes at any level is: [B]
A. n C. n + 1
B. 2^n D. 2n



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 1 and its adjacency list

7. For the figure 1 shown above, the depth first spanning tree visiting sequence is: [B]
A. A B C D E F G C. A B C D E F G
B. A B D C F G E D. none of the above

8. For the figure 1 shown above, the breadth first spanning tree visiting sequence is: [B]
A. A B D C F G E C. A B C D E F G
B. A B C D E F G D. none of the above

9. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 1 shown above: [B]
A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
C. both A and B
D. none of the above

10. For the figure 1 shown above, the cost of the minimal spanning tree is: [A]
A. 57 B. 68 C. 48 D. 32

Chapter 7

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are ‘n’ elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison’s to search an element. If search is not successful, you would need ’n’ comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element ‘x’ is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf(“Data found at %d position“, index);
        else
                printf(“data not found”);

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:
45, we’ll look at 1 element before success
39, we’ll look at 2 elements before success
8, we’ll look at 3 elements before success
54, we’ll look at 4 elements before success
77, we’ll look at 5 elements before success
38 we’ll look at 6 elements before success
24, we’ll look at 7 elements before success
16, we’ll look at 8 elements before success
4, we’ll look at 9 elements before success
7, we’ll look at 10 elements before success
9, we’ll look at 11 elements before success
20, we’ll look at 12 elements before success

For any element not in the list, we’ll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7	Search successful, data found at 3rd position.
2. Searching for x = 82	Search successful, data found at 7th position.
3. Searching for x = 42	Search un-successful, data not found.

A non-recursive program for Linear Search:

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

main()
{
        int number[25], n, data, i, flag = 0;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be Searched: ");
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", i+1);
        else
                printf("\n Data not found ");
}
```

A Recursive program for linear search:

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

void linear_search(int a[], int data, int position, int n)
{
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}

void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ≤ (low + high)/2 ƒ
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8*, **found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element:

> 20 – requires 1 comparison;
> 8 and 39 – requires 2 comparisons;
> 4, 9, 24, 54 – requires 3 comparisons and
> 7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

A program for binary search:

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

main()
{
        int number[25], n, data, i, flag = 0, low, high, mid;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        low = 0; high = n-1;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(number[mid] == data)
                {
                        flag = 1;
                        break;
                }
                else
                {
                        if(data < number[mid])
                                high = mid - 1;
                        else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
        else
                printf("\n Data Not Found ");
}
```

Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
        int i, j, temp;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j];
                                x[j] = x[j+1];
                                x[j+1] = temp;
                        }
                }
        }
}

main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:");
        for(i = 0; i < n ; i++)
                scanf("%d", &x[i]);
        bubblesort(x, n);
        printf ("\n Array Elements after sorting: ");
        for (i = 0; i < n; i++)
                printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = $n^2 – 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also ***O(n²)*** for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remarks*
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Program for selection sort:

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

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num, i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until a[up] >= pivot.
2. Repeatedly decrease the pointer 'down' until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ...x[j-1] and x[j+1], x[j+2], ... x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∝

```
quicksort (p, q)
{
        if ( p < q ) then
        {
              call j = PARTITION(a, p, q+1);      // j is the position of the partitioning element
              call quicksort(p, j – 1);
              call quicksort(j + 1 , q);
         }
}

partition(a, m, p)
{
          v = a[m]; up = m; down = p;          // a[m] is the partition element
          do
          {
                    repeat
                              up = up + 1;
                    until (a[up] ≥ v);
```

```
                repeat
                        down = down – 1;
                until (a[down] ≤ v);
                if (up < down) then call interchange(a, up, down);
         } while (up ≥ down);

        a[m] = a[down];
        a[down] = v;
        return (down);
}
interchange(a, up, down)
{
        p = a[up];
        a[up] = a[down];
        a[down] = p;
}
```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down

	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>
void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];
int main(){
        int num, i = 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf( "%d", &num);
        printf( "Enter the elements: " );
        for(i=0; i < num; i++)
                scanf( "%d", &array[i] );
        quicksort(0, num -1);
        printf( "\nThe elements after sorting are: " );
        for(i=0; i < num; i++)
                printf("%d ", array[i]);
        return 0;
}
void quicksort(int low, int high){
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}
int partition(int low, int high){
        int pivot = array[low];
        int up = low, down = high;

        do{
                do
                        up = up + 1;
                while(array[up] < pivot );

                do
                        down = down - 1;
                while(array[down] > pivot);

                if(up < down)
                        interchange(up, down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}
void interchange(int i, int j){
        int temp; temp = array[i]; array[i] = array[j]; array[j] = temp;
}
```