Unit - 2

Arrays and Matrices:

Array is a collection of similar data items in which each element is unique and located in separate continuous memory locations. The amount of storage required for holding elements of the array depends on its type and size.

For example, an internal has been conducted to 4th Semester ECE students, they are 70 in number. So 70 IA marks have to be stored in an array. Let array name be **IA**. It can be declared as

```
int IA[70];          /* IA is an Array Name, 70 is the size of the array. */
```

Here, members of the array IA are IA[0], IA[1], ... IA[69], the roll number of the student from 1 to 70. The subscripted variable IA[0] stores the marks of the 6th student.

Always an array index from 0th location and ends with n - 1 (Here, n is the size of the array).

Array Types:

Arrays are classified as 2 types.

1) One-Dimentional Array 2) Multi-Dimentional Array

One Dimensional Array:

One-dimensional array is a linear list of fixed number of data items of the same type. .All these data items are accessed using the same name using a single subscript.

An array name occurs with single subscript ([ ]) is called as One-dimentional Array. One pair of Square bracket is called subscript.

Declaration of One Dimensional Array:

Array must be declared before like declaring an ordinary variable. At the same time size of an array must be specified. This allows the compiler to decide on how much memory is to be reserved for the array.

The syntax: Data_type arrayname[size];

```
Ex:   int IA[70];        /* size of array is 70  */
      float avg[30];     /* size of array is 30  */
```

Initialization of One Dimensional Array:

As we know that C variables can be initialized to some values when they are declared.

Ex: int count=0;

Similarly, the arrays can also be initialized to some initial values.

Ex: int marks[6] = {35,40,81,56,48,71};

a. Arrays elements must be included in a pair of curly braces and separated by comma, initial values of the array elements are assigned to array **marks[6]**.
b. If the array is declared to contain integer numbers, all the initial values must be integers only.
c. The marks[0] = 35, marks[1]=40 ..... ,marks[5]=71

Multidimensional Arrays:

Arrays of more than one dimension is called multi-dimensional arrays. It can be declared merely by adding more subscripts. The two dimensional array (contains 2 subscripts) is made up of rows (1st subscript) and columns (2nd subscript).

Declaration of Two Dimensional Arrays:

Two dimensional arrays have to be declared with two subscripts and declaration must specify the size and data type of the array.

Ex: int table[3][2];

It specifies the table is two dimensional array with 3 row and 2 columns consisting of 3 x 2 = 6 elements of integer type. Here, Array index starts with table[0][0], table[0][1], table[0][2], table[1][0], table[1][1], table[1][2], table[2][0], table[2][1], and array index ended with table[2][2].

Initialization of Two-Dimensional Array:

a) Two dimensional arrays can be initialized like one-dimensional array. For eg.,

```
int table[3][2] = {   {14,8},
                      {10,12},
                      {15,14}
                  };
```

Two-dimensional array named **table** elements are initialized row by row. Thus each row elements are enclosed in a pair of brace.

Observe that each pair of braces is separated by comma to indicate separate rows.

Note that if the array named **table** is initialized completely for all of its elements then inner pairs of braces are not required. Thus it is sufficient to write.

```
int table[3][2] ={14,8,10,12,15,14};
```

b) Like one-dimensional arrays, it is not compulsory to initialize all members of the two-dimensional arrays also.

```
Ex: int table[3][2] = { {14},
                        {10,12},
                        {}
                      };
```

Remaining elements of the array report are considered to be zeros.

Array Operations:

Majorly used for Matrices operations and their applications.

Data Structure:

The data structure is logically or mathematically organized data items. Representing or Storing the data in a systematic way in computer memory is called data structure.

Operations on Data Structures:

- Creation of data structure.
- Deletion of data structure.
- Insertion of data structure.
- Accessing elements within a data structure, called selection operation.

- Modification of contents in the data structure.
- Sort operation to arrange elements within a structure.

The overall intention of all these operations is to access data elements to provide useful results.

Classification of Data structures:

A systematic grouping of data structure broadly classified into

Primitive data structure and Non-primitive data structure

Primitive data structure:

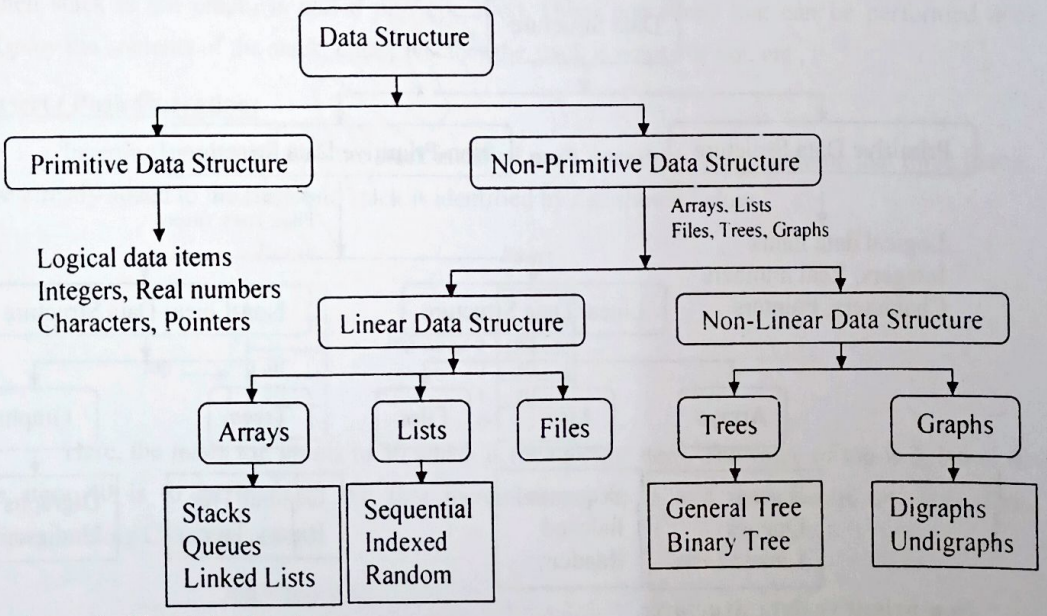
Primitive data types include integers, real numbers, characters, logical data items and pointer data types.

Non-primitive data structure

Non-primitive data structure include arrays, lists, files, trees, graphs. Non-primitive data structure further classified into

- Linear data structure
- Non-linear data structure.

Arrays, lists and files are classified under linear data structure whereas trees and graphs belong to non-linear data structure. The data structure classification details are as shown below:



Non-primitive data structure:

A data structure, which is derived from primitive data types, is referred to as non-primitive data types or non-primitive data structures. For example an array consisting if integer elements (primitive data type) is classified as Non-primitive data structure.

Operations on Non-primitive Data Structures:

The type of operation that can be applied on a particular non-primitive data structure varies functionally depending upon their logical organisation and storage structure.

- Create or define an array / list / file
- Search for an element in an array / list, locate a record in a file
- Sort the elements of an array into ascending/descending order.
- Combine or merge two or more lists to form another new list.
- Insert elements into and delete elements from a list
- Copy a list, Update a file, rewind a file, delete records etc.,
- Delete a node from a binary tree, insert a new mode in a binary tree.
- Traverse a tree in inorder, postorder, preorder.

Stacks

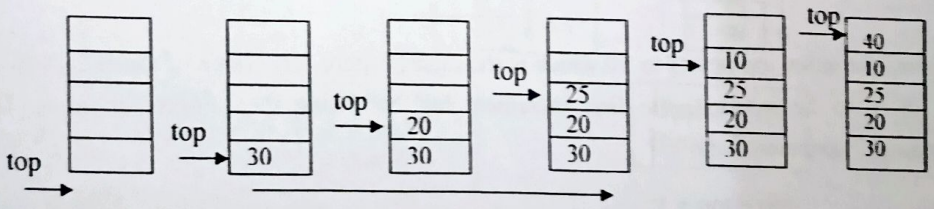
We know that in a cafeteria the plates are placed one above the other and every new plate is added at top. When a plate is required, it is taken off from the top and it is used. We call this process as *stacking* of plates. It was natural, that when man started programming data, stack was one of the first structures that he thought of when faced with the problem of maintaining data in an orderly fashion.

Definition and Example:

A stack is a data structure in which addition of new element or deletion of an existing always takes place at the same end. This is often known as **top**. Here, the last item inserted will be on **top** of stack. Since the deletion is done from the same end, ***L***ast item ***I***nserted is the ***F***irst item to be deleted ***O***ut from the stack and so, stack is also called **L**ast **I**n **F**irst **O**ut (LIFO) data structure. The various operations that can be performed on stacks are:

- Insert an item into the stack
- Delete an item from the stack
- Display the contents of the stack

From the definition of the of the *stack*, it is clear that it is a collection of similar type of items and naturally we can use an array (an array is a collection of similar data types) to hold the items of stack. Since array is used, its size is fixed. So, let us assume that 5 items 30, 20, 25, 10 and 40 are to be place on the stack. The items can be inserted one by one as shown below.

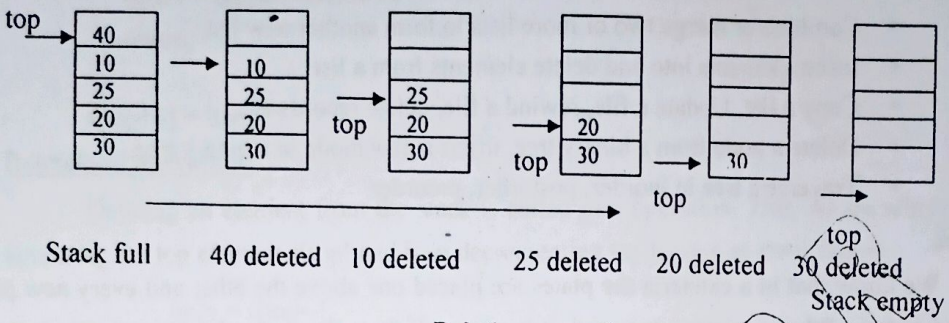


Empty stack insert 30 insert 20 insert 25 insert 10 insert 40

Insertion operation

It is clear from this figure that initially stack is empty and top point at the bottom of the stack. As the items are inserted top pointer is incremented and it points to the topmost item. Here, the items 30, 20, 25, 10 and 40 are inserted one after the other. After inserting 40 the stack is full. In this situation it is not possible to insert any new item. This situation is called *stack overflow*.

When an item is to be deleted, it should be deleted from the top as shown in delete operation



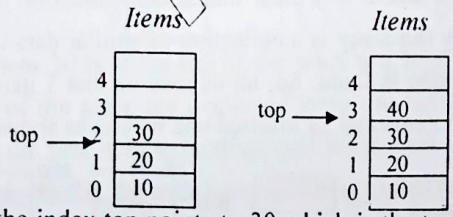
Deletion operation

Since items are inserted from one end, in stack deletions should be done from the same end. So, the items are deleted, the item below the top item becomes the new top item and so the position of the top most item is decrements as shown in the figure. The items deleted in order are 40, 10, 25, 20 and 30. Finally, when all items are deleted, top points to bottom of stack. When the stack is empty, it is not possible to delete any item and this situation is called *under flow of stack.*

So, the main operations to be performed on stacks are insertion and deletion. Inserting an item into stack when stack is not full is called *push* operation and deleting an item from the stack when stack is not empty is called *pop* operation. Other operations that can be performed are *display* the contents of the stack, check whether the stack is empty or not, etc.,

Insert / Push Operation:

Inserting an element into the stack is called **push** operation. Let us assume that three items are already added to the stack and stack is identified by *s* as shown below.



Here, the index top points to 30 which is the topmost item. The value of top is 2. Now, if an item 40 is to be inserted, the first increment top by 1 and then inserts an *item*. The corresponding C statements are:

```
top = top + 1;
s[top] = item;
```

These two statements can also be written as

```
s[top+1] = item;        or        s[++top] = item;
```

stack_size should be **max = some integer value** (Ex: 5) and is called symbolic constant, the value of which cannot be modified. As we insert an item we must take care of the overflow situation i.e., when top reaches **stack_size–1**, stack results in *overflow condition* and appropriate error message has to be returned and items has to be inserted as shown below:

```
if(top==stack_size-1)
 {
   cout<<"Stack overflow\n";
   return;
 }
s[top+1] = item;
```

Delete/pop operation:

Deleting an element from the stack is called **pop** operation. This can be achieved by first accessing the top element s[top] and then decrementing top by one as show below:

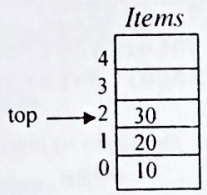
```
item = s[top--];
```

If items are present in the stack, an item is always deleted from the top of the stack. Trying to delete an item from the empty stack results in *stack underflow*. The above statement has to be executed only if stack is not empty. Hence, the code to delete an item from stack can be written as

```
if(top < 0)
{
   cout<<"Stack is empty";;
}
item =s[top--];
cout<<item;
```

Display:

Assume that the stack contains three elements as shown below:



The item 30 is at the top of the stack and item 10 is at the bottom of the stack. Usually, the contents of the stack are displayed from the bottom of the stack till the top of the stack is reached. So the first item to be displayed is 10, next item to be displayed is 20 and final item to be displayed is 30. So, the code corresponding to this can take the following form

```
for(i=0;i<=top; i++)
{
   cout<<s[i];
}
```

But, the above statement should not be executed when stack is empty i.e., when top takes the value lessthan 0. So, the modified code can be written as shown below.

```
void display()
{
        int i;
        if(top < 0)
        {
                cout<<"Stack is empty\n";
        }
```

```
        cout<<"Contents of the stack\n";
        for(i=0; i<=top; i++)
        {
            cout<< s[i];
        }
    }
```

Write a complete C++ program to implement stack operations using switch statement

```
#include<iostream.h>
int max = 5
class stack
{
    int top, stk[max];
    public:
        stack()
        {
            top = -1;
        }
        void push(int x)
        {
            if(top>max-1)
            {
                cout<<"Stack is Full";
                return;
            }
            stk[++top] = x;
        }
        void pop()
        {
            if(top<0)
            {
                cout<<"Stack is Empty";
            }
            item = stk[top--];
        }
        void display()
        {
            if(top<0)
            {
                cout<<"Stack is Empty";
            }
            cout<<"Contents of the stack\n";
            for(int i = 0; i<=top; i++)
            cout<<stk[i];
        }
};
void main()
{
    stack st;
```

OUTPUT:

```
1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:1
Enter data:25

1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:1
Enter data:50


1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:3

Elements in stack:
25
50

1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:4
      Exit!!
```

```
    int choice, a;
    do
    {
        cout<<"1: PUSH\n 2. POP\n 3. DISPLAY\n 4. EXIT\n";
        cout<<"Enter your choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: cout<<"Enter data";
                    cin>>a;
                    st.push(a);
                    break;
            case 2: st.pop();
                    break;
            case 3: st.display();
                    break;
            case 4: cout<< "Exit!\n";
        }
    }
    while(choice!=4);
    getch();
}
```

Abstract Datatype stack
{

instances

linear lists of elements; one end is called the bottom; the other end is the top.

operations

Create(): Create an empty stack;

Isempty(): Return true if stack is empty, return false otherwise;

Isfull():Return true if stack is full, return false otherwise;

Top(): Returns the top element of the stack;

Add(x) / Push(x): Add element x to the stack;

Delete(x) / Pop(x): Delete top element from the stack & put it in x;

}

Formula / Array based representation:

The top element of the stack is stored in element[MaxSize-1] and the bottom element in element [0].

```
template <class T>
class Linearlist
{
    public:
        Linearlist(int maxsize = 10);          //Constructor
        ~Linearlist()
```

```
        {
                delete [] element;              //Destructor
        }
        bool Isempty()
        {
                return length = 0;
        }
        int length()
        {
                return length;
        }
        bool find(int k, T& x)                  //return the kth element of list in x
        int search(T& x)                        //return position of x
        LinearList<T>  & Delete(int k, T& x);   //Delete kth element & return in x
        LinearList<T>  & Insert(int k, T& x);   //Insert x just after kth element
        void operator(Ostream &out);
    private:
        int length, maxsize;
        T * element;            //Dynamic 1 D Array.
};
```

Formula / Array based class Linearlist:

```
template <class T>
class stack : private Linearlist<T>
{
    public:
        stack(int maxsize = 10) : Linearlist<T>
        bool Isempty()
        {
                return Linearlist<T>::Isempty();
        }
        bool Isfull()
        {
                return (Length() = = GetMaxSize(x));
        }
        T Top()
        {
                if( Isempty()) throw outofBounds();
                T x;
        }
        stack <T> & Add(Const T& x)
        {
                insert(Length(s, x));
                return *this;
        }
};
```

Since the inheritance made is private, the public and private members of Linearlist become private to stack. However, the private members of Linearlist are not accessible to members of stack.

Pointer / Linked Representation:

While the array representation of stack in the previous section is considered both elegant and efficient but is wasteful of memory when multiple stacks are coexist.

Multiple stacks can be represented efficiently using a chain for each stack. This representation incurs a space penalty of one pointer field for each stack element. When using a chain to represent a stack, we must decide which end of the chain corresponds to the stack top. If we associate the right end of the chain with the stack top, then insertions and deletion operations takes $\Theta(n)$ time.

On the other hand, If we associate the left end of the chain with the stack top, then insertions and deletion operations takes $\Theta(1)$ time. This shows that we should use the left end of the chain to represent the stack top.

Write a complete C++ program to implement stack operations using linked representation.

```
#include<iostream.h>
using namespace std;
class node
{
        int data;
        node *top, *link;
        public:
                node()
                {
                        top = NULL;
                        link = NULL;
                }
                void push(int x)
                {
                        node *n = new node;
                        n->data = x;
                        n->link = top;
                        top = n;
                        cout<<"Pushed"<<n->data<<endl;
                }
                void pop()
                {
                        node *n = new node;
                        n = top;
                        top = top->link;
                        n->link=NULL;
                        cout<<"Poped"<<n->data<<endl;
                        delete n;
                }
                void display()
                {
                        node *n = new node;
                        n = top;
                        while(n!=NULL)
                        {
```

```
                cout<<n->data<<endl;
                n=n->link;
            }
            delete n;
        }
};
int main()
{
    node stack;
    int choice, a;
    do
    {
        cout<<"1. PUSH\n 2. POP\n ";
        cout<<"3. DISPLAY\n 4. EXIT\n";
        cout<<"Enter your choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: cout<<"Enter data";
                    cin>>a;
                    stack.push(a);
                    break;
            case 2: stack.pop();
                    break;
            case 3: stack display();
                    cout<<"Elements in stack";
                    break;
            case 4: cout<<"Exit!!\n";
        }
    }
    while(choice!=4);
    getch();
}
```

OUTPUT:

```
1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:1
Enter data:25
Pushed 25

1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:1
Enter data:50
Pushed 50


1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
Enter your choice:3

Elements in stack:
25
50

1.    PUSH
2.    POP
3.    DISPLAY
4.    EXIT
```

Applications of Stack:

Following are the some of the various applications of stacks.

1. To match left & right parenthesis in an expression.
2. Tower of Hanoi problem.
3. Shutting tracks in a rail yard.
4. CAD of circuit field.
5. Offline equivalence class problem.
6. Rat in maze problem
7. Conversion of infix expressions
8. Evaluation of postfix expressions
9. Recursion
10. Other applications like, to find whether the string is a palindrome or not, topological sort, etc...

Parenthesis Matching:

Here, In a character string, we are to match the left and right parenthesis of type { } and ( ). For example, string (a*(b+c)+d) contains left parenthesis at 0^{th} and 3^{rd} positions and right parenthesis at 7^{th} and 10^{th} positions. The parenthesis 0^{th} and 10^{th} positions are matches, while the parenthesis 3^{rd} and 7^{th} positions are matches. Our main objective is to write a C++ program that inputs a string and outputs the pair of matched parenthesis as well as no matching parenthesis.

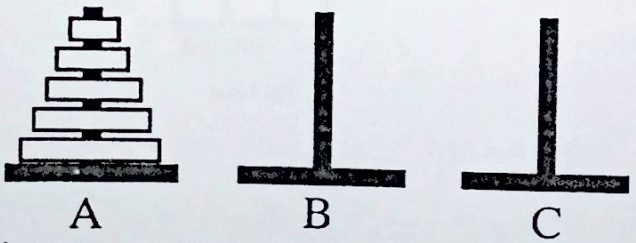
We notice that, if we scan an input expression fron left to right, then each right parenthesis is matched to the most recently seen unmatched left parenthesis. This motivates us to save the left parenthesis position on stack (because of left to right scan). If a right parenthesis is occurred, it is matched to left parenthesis at the top of the stack. Matched left parenthesis is deleted from stack.

Algorithm for Parenthesis Matching:

```
while(not end of line)
{
        read the current character;
        if(No parenthesis at this place)
                continue;
        if(it is a left parenthesis)
                push it on to the stack;        //may or may not get overflow error.
        if(it is a right parenthesis)
        {
                pop the stack;                  //may or may not get underflow error.
                if(poped left bracket doesnot match the current right bracket)
                {
                report malformed expression error;
                exit;
                }
        }
}
```


Write a C++ code of your own to match the parenthesis in an expression

Tower of Hanoi Problem:



This problem is fashioned after the ancient Tower of Brahma ritual. There was a diamond tower (tower A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top. Next to this tower are 2 other diamond towers (towers B and C). Since the time of creation, attempting to move the disks from tower A to B, using C for intermediate storage. As the

disks are very heavy, they can be moved only one at a time and also at no time can a disk be on the top of a smaller disk.

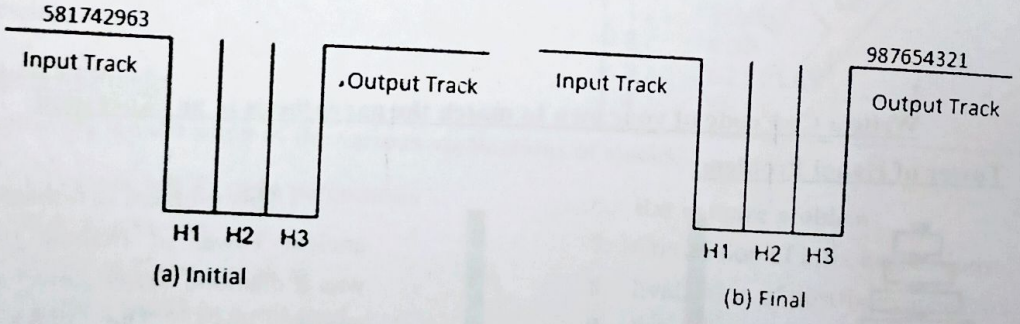
A very elegant solution results from the use of recursion. To get the largest disk to the bottom of tower B, we move the remaining n-1 disks to C, then move the largest to tower B. Now there are n-1 disks to be moved from C to B. To perform this task, we can use tower A and B. We can safely ignore the fact that tower B has a disk on it as this disk is larger than disks being moved from tower C. Hence, we can place any disk on top of the disk on tower B.

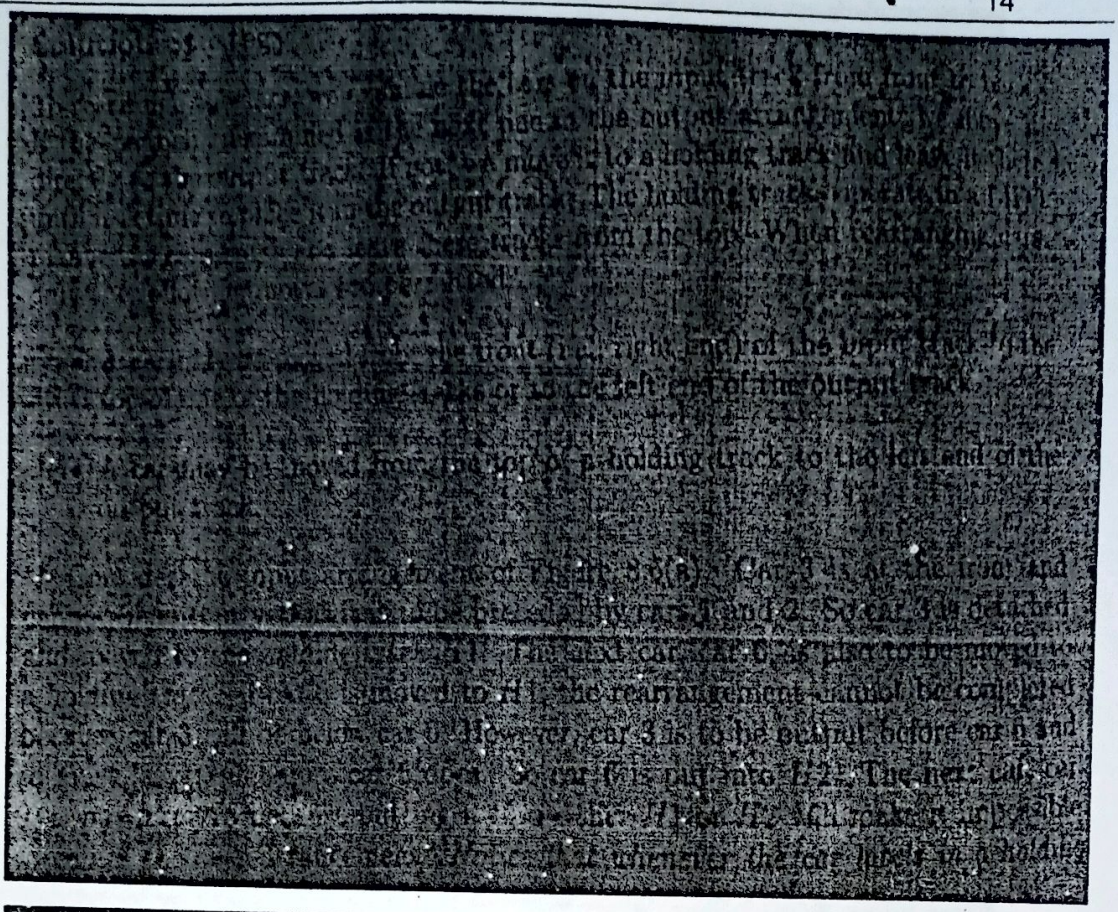
```
void towerofhanoi(int n, int x, int y, int z)
{
        if(n>0)
        {
                towerofhanoi(n-1, x, z, y)
                cout<<"Move top disk from tower" << x << to top of tower"<<y<<endl;
                towerofhanoi(n-1, z, y, x);
        }
}
```

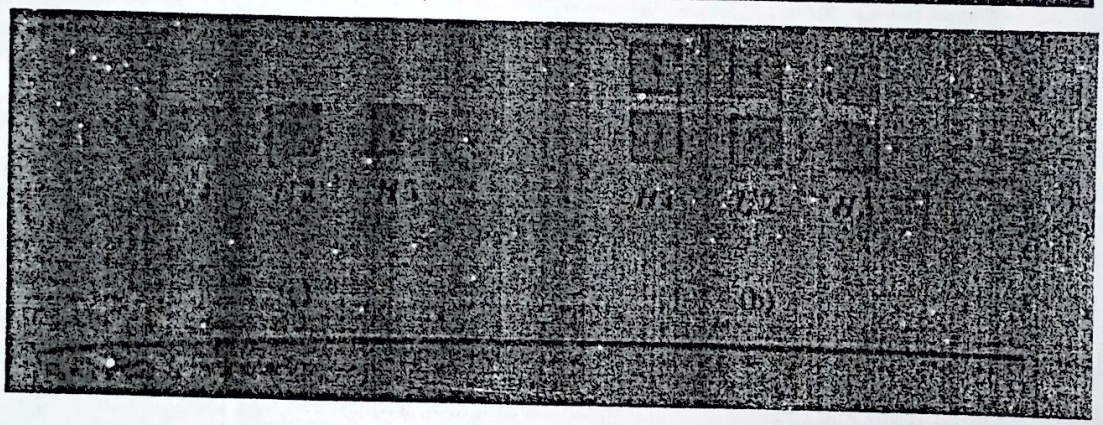
Rearranging Railroad Cars:

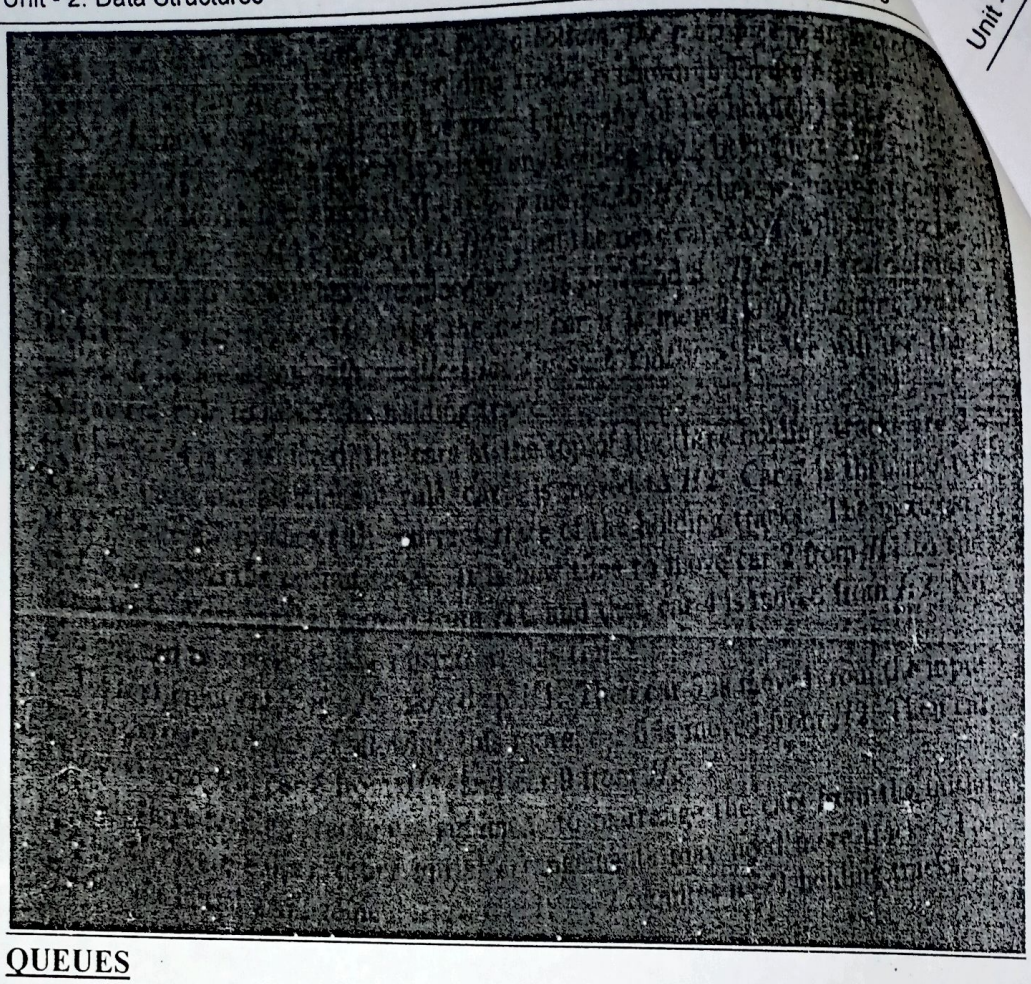
A train has n railroad cars. Each has to be left at a different station. Consider n stations are numbered from 1 through n and that the train visits these stations in an order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we should reorder the cars so that they are in the order 1 through n from the back. When the cars are in this order, the last cas is detached at each station. We rearrange the cars at shunting yard that has an input track, an output track, and k holding tracks between the input and output tracks.

Figure shows the shunting yard with k=3 holding tracks H1, H2, and H3. The n cars of the train begin in the input track and are to end up in the output track in the order 1 through n from right to left. See figure (a) and (b) for initial and final order.









QUEUES

Whether it is a railway reservation counter, a movie theatre or print jobs submitted to a network printer there is only one way to bring order to chaos – from a queue. If you wait your turn patiently there is more likelihood that you would get a better service.

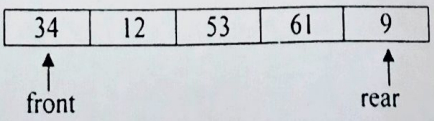
Definition:

Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called **front** and the end at which insertion of a new element can take place is called **rear**.

The first element that gets added into the queue is the first one to get removed from the list. Hence, queue is also referred as First-In-First-Out **(FIFO)** list.

Example:

Consider a queue of bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. The first person in the line is the first person to leave.



In the above figure 34 is the first element 9 is the last element added to the queue. Similarly, 34 would be the first element to get removed and the 9 would be the last element to get removed.

Sequential Representation:

Queue, being a linear data structure can be sequentially represented in various ways such as arrays and linked lists. An array is a data structure that can store a fixed number of elements. The size of the array should be fixed before using it. Queue on the other hand keeps on changing as we remove element from the front end. Or add new element at the end. Declaring an array with maximum size would solve this problem.

Different types of Queues:

The different types of queues are:

- Ordinary Queue
- Circular Queue
- Double ended Queue (Dequeue)
- Priority Queue

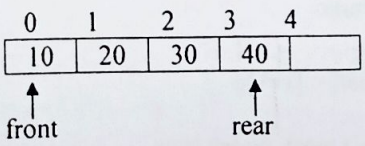
Queues and their operations:

An ordinary queue operates on first come first serve basis. Items will be inserted from one end and they are deleted at the other end in the same order in which they are inserted. Here first element inserted is the first element to go out of the queue. The operations that can be performed on these queues are

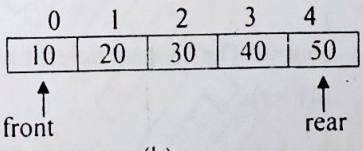
- Insert an element at the rear end
- Delete an element at the front end
- Display the contents of the queue

Insert an element at the rear end:

Consider a queue, with Q_SIZE as 5 and assume 4 items are present as shown in fig (a). Here, two variable *f* and *r* are used to access the elements located at the *front end* and *rear end* respectively. It is clear from this figure that at most 5 elements can be inserted into the queue. Any new item should be inserted from the *rear end* of queue. So, if an item 50 has to be inserted, it has to be inserted to the right of item 40, i.e., at q[4].



(a)



(b)

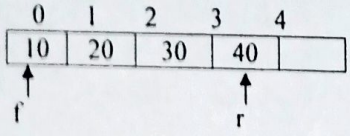
It is possible if increment **r** by 1 so as to point to next location and insert the item 50. The queue after inserting an item to is shown in fig (b). When queue is full, it is not possible to insert any element into queue and this condition is called *overflow* i.e., when **r** = = Q_SIZE – 1, queue becomes full. It can be written as:

```
void insert_rear (int item, int q[], int r)
{
        if(r= =Q_SIZE-1)
        {
```

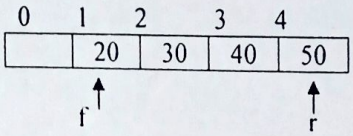
```
        cout<<"Queue overflow\n";
        return;
    }
    q[++r] = item;
}
```

Delete an element at the front end:

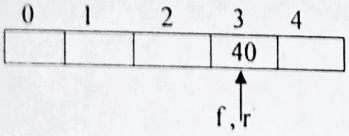
The first item to be deleted from the queue is the item, which is at the front end of the queue. It is clear from the queue shown in fig (a) that the first item to be deleted is 10. Once this item is deleted, next item i.e., 20 in the queue will be the first element and the resulting queue is of the form shown in the fig (b). So, the variable *f* should point to 20 indicating that 20 is at the front of the queue. This can be achieved by accessing the item 10 first and then incrementing the variable *f*.



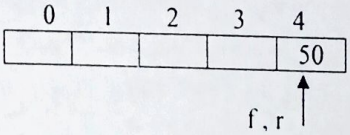
(a)



(b)



(c)



(d)

But, as we keep on deleting one after the other, finally queue will be empty. Consider the queue shown in (c). Once the item 40 is deleted, *f* points to next location and queue is empty. This condition is called *underflow* of queue. It can be written as:

```
void delete_front (int q[], int f, int r)
{
    if(f > r)
    {
        cout<<"Queue underflow\n";
        return;
    }
    cout<<"The element deleted is "<< q[f++] ;
    if(f > r)
    {
        f = 0, r = – 1 ;
    }
}
```

Consider the situation shown in (d). Deleting an item 50 from queue results in an empty queue. At this stage, suppose an item has to be inserted. While inserting when queue rear pointer *r* reaches Q_SIZE – 1 it displays the message "Queue overflow". It is clear from the queue that queue is not full and even then, an item cannot be inserted. Hence, whenever queue is empty, we reset the front end identified by *f* to 0 and rear end identified by *r* to –1. So, the initial values of front pointer *f* and rear pointer *r* should be 0 and –1 respectively.

Display Queue contents:

The contents of queue can be displayed only if queue is not empty. If queue is empty an appropriate message is displayed.

```
void display (int q[], int f, int r)
{
        inti;
        if(f > r)
        {
                cout<<"Queue is empty\n";
                return;
        }
        cout<<"Contents of the queue is\n";
        for(i = f; i<=r; i++)
        cout<<q[i];
}
```

Write a C++ program to implement queue operations (Passing by parameters)

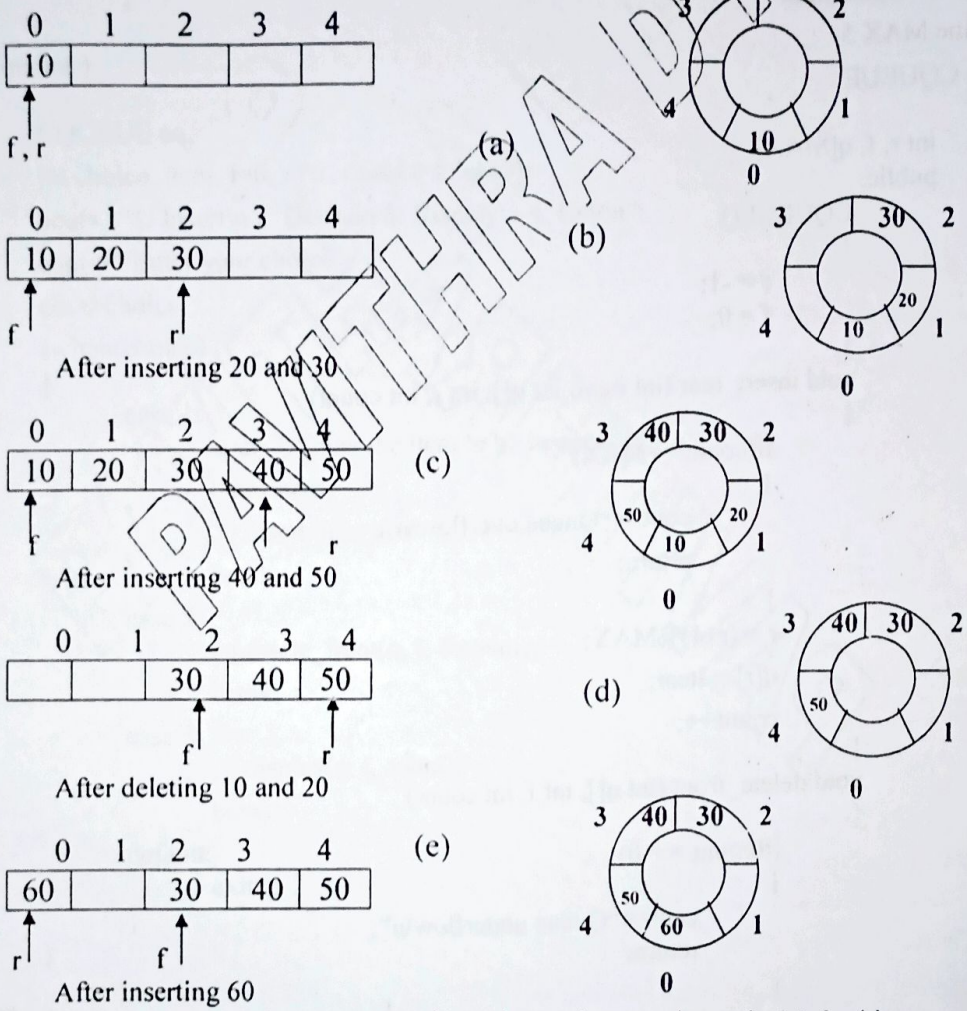
```
#include<iostream.h>
#define MAX 5
class QUEUE
{
    int r, f, q[MAX];
    public:
        QUEUE()
        {
                r = -1;
                f = 0;
        }
        void insert_rear (int item, int q[], int r)
        {
                if(r= =MAX-1)
                {
                        cout<<"Queue overflow\n";
                        return;
                }
                q[++r] = item;
        }
        void delete_front (int q[], int f, int r)
        {
                if(f > r)
                {
                        cout<<"Queue underflow\n";
                        return;
                }
                cout<<"The element deleted is "<< q[f++] ;
                if(f > r)
                {
```

```
                f = 0, r = – 1 ;
            }
        }
        void display (int q[], int f, int r)
        {
            int i;
            if(f > r)
            {
                cout<<"Queue is empty\n";
                return;
            }
            cout<<"Contents of the queue is\n";
            for(i = f; i<=r; i++)
            cout<< q[i];
        }
void main( )
{
    QUEUE Q;
    int choice, item, f=0, r=-1, q[10];
    cout<< "1: Insert\n 2: Delete\n 3: Display\n 4: Exit\n";
    cout<< "Enter your choice";
    cin>>choice;
    switch(choice)
    {
        case 1:
            cout<<"Enter the item to be inserted\n";
            cin>>item;
            Q.insert_rear(item, r, q);
            break;
        case 2:
            Q.delete_front(q, f, r);
            break;
        case 3:
            Q.display(q, r, f);
            break;
        default:
            exit();
        }
    }
}
```

Circular Queues:

In an ordinary queue, as an item is inserted, the rear end identified by **r** is incremented by 1. Once **r** reaches MAX – 1 it displays the message "Queue overflow". It is clear from the queue that queue is not full and even then, an item can not be inserted. **This disadvantage is overcome using *circular queue*.** Here, as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array (provided if it is free). Suppose an

array **a** of **n** elements is used to implement a circular queue. If we go on adding elements to the queue we may reach **a[n-1]**. Since we have reached the end of the array, we cannot add any more elements to the queue. Instead of reporting the queue as full, if some elements in the queue have been deleted then there might be empty slots at the beginning of the queue. In such a case these slots would be filled by new elements being added to the queue. The following pictorial representation represents of a circular queue.



Assume that circular queue contains only one item as shown in (a). In this case rear end r is 0 and front end f is also 0. Since **r** is incremented by inserting 20 and 30 in fib (b), **r** is incremented by 1 and 2 respectively. Hence,

r = r + 1 **or** **r = (r + 1)%Q_SIZE**

Both the statements are correct, but we prefer the second statement, because it indicates the size of the queue also.

In fig (c), the queue is full after inserting 40 and 50. Suppose, we delete 10 and 20 and try to insert 60. If the statement r = r + 1 is used to increment rear point r will be 5. In ordinary queue, it is not possible to insert 60 in the queue size 5. But, because this is a circular queue r should point to 0. This can be achieved using the statement

$$r = (r + 1)\%Q\_SIZE$$

Here, we can use the *count* as the variable that contains the number of items in the queue at any instant. So, as an item is inserted *count* is incremented by 1 and as an item is deleted *count* is decremented by 1. Hence count contains the total number of elements in the queue. So, if queue is empty count is 0, if queue is full the count will be Q-SIZE.

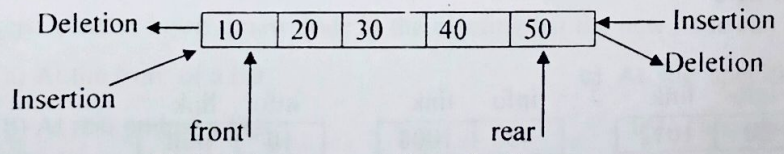
Write a C++ program to implement circular queue operations

```
#include<iostream.h>
#define MAX 5
class CQUEUE
{
    int r, f, q[MAX];
    public:
        CQUEUE()
        {
            r = -1;
            f = 0;
        }
        void insert_rear (int item, int q[], int r, int count)
        {
            if(count= =MAX)
            {
                cout<<"Queue ove, flow\n";
                return;
            }
            r = (r+1)%MAX;
            q[r] = item;
            count++;
        }
        void delete_front (int q[], int f, int count)
        {
            if(count = = 0)
            {
                cout<<"Queue underflow\n";
                return;
            }
            cout<<"The element deleted is "<< q[f];
            f=(f + 1)%MAX;
            COUNT--;
        }
        void display (int q[], int f, int count)
        {
            int i;
            if(count = = 0)
            {
                cout<<"Queue is empty\n";
                return;
            }
```

```
                cout<<"Contents of the queue is\n";
                for(i = 1; i<=count; i++)
                {
                        cout<<q[f];
                        f = (f + 1)%MAX;
                }
        }
};
void main( )
{
        CQUEUE cq;
        int choice, item, f=0, r=-1, count = 0; q[10];
        cout<<"1: Insert\n 2: Delete\n 3: Display\n 4: Exit\n";
        cout<<"Enter your choice";
        cin>>choice;
        switch(choice)
        {
                case 1:
                        cout<<"Enter the item to be inserted\n";
                        cin>>item;
                        cq.insert_rear(item, q, r, count);
                        break;
                case 2:
                        cq.delete_front(q, f, &count);
                        break;
                case 3:
                        cq.display(q, f, count);
                        break;
                default:
                        exit();
                }
        }
}
```


DeQueue:

The word **dequeue** is a short form of double-ended-queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes made elsewhere in the list. Thus a dequeue is a generation of both stack and a queue.



Pictorial representation of a Dequeue

Note: Assignment: Write a C++ program implement Dequeue.

Priority Queues:

The priority queue is a special type of data structure in which items can be inserted or deleted based on the priority. The order in which the elements should get added or removed is decided by the priority of the element. The following rules are applied to maintain priority queue.

1. Always highest priority element is processed before processing lower priority elements.
2. If the elements in the queue are of same priority, then the element, which is inserted first into the queue, would get processed.

Priority queue are used in job scheduling algorithms in the design of operating system where the jobs with highest priorities have to be processed first.

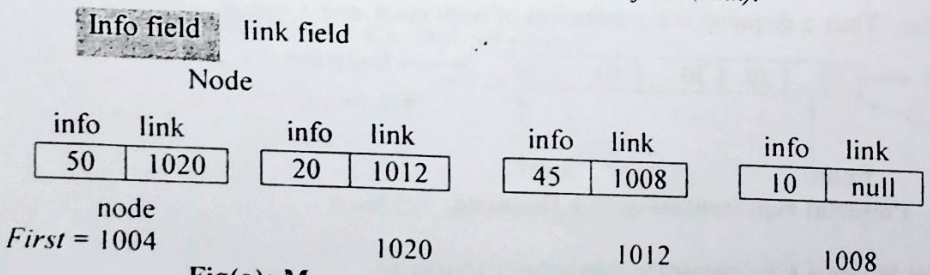
Applications of Queues:

Applications of queue as a data structure are common while performing tasks on a computer, it is often necessary to wait one's turn before having access to some device or process. Within a computer system there may be queues of tasks waiting for the line printer, or for access to disk storage, or in a time-sharing system for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue. Traffic signal light controlled insertion at cross roads or circles for movement of vehicles can be developed as traffic system simulation model. Here, the vehicles wanted to move in a particular direction are queued.

LINKED LISTS

We call stacks and queues as restricted lists because insertions and deletions are restricted to occur in these structures only at the ends of the list. In stacks and queues all the items are kept sequentially in arrays, declaring a maximum size of arrays. We cannot increase the amount of storage allotted at run time, which would cause overflow. The problems discussed on sequential storage representation can be solved by the introduction of a new data structure called linked lists.

A linked list is a series of data items with each item containing a pointer giving the location of the link item in the list. We can place the elements anywhere in the memory, but to make them a part of the same list it is required to be linked with the previous element of the list. This can be done by storing the address of the link element in the previous element itself. This requires that every element must be capable of holding the data as well as the address of the link element. Thus every data element (called *nodes*) must be a record with a minimum of two fields, one for holding the data value (info), which we call a *data (info) field*, and the other for holding the address of the link element, which we call *linked field (link)*.



Fig(a): Memory representation of singly linked list

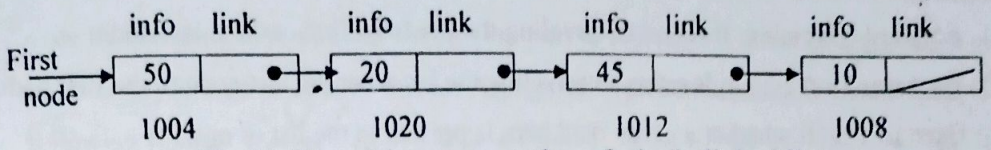


Fig (b):Pictorial representation of singly linked list

The memory representation and pictorial representation of a singly linked list is shown in figure (a) and (b) respectively. This list contains 4 nodes and each node consists of two fields, *info* and *link*. The first field of each node i.e., *info* contains the data that has to be stored in a list. In this list the data items 50, 20, 45, and 10 are stored. The second field i.e., link of each node contains address of the next node. Note the arrow originating from link field of each node. This arrow indicates that the address of the next node is stored. So, using *link* field, any node in the list can be accessed since the nodes in the list are logically adjacent. Nodes that are physically adjacent need not be logically adjacent in the list. For example, the nodes identified by the address 1004, 1020, 1012, and 1008 are physically not adjacent but they are logically adjacent.

In this list the variable *first* contains the address of the first node. Before creating a list, *first* should be initialized to NULL indicating, the list is to empty to start with. Once the list is created, variable *first* contains address of the first node of the list. The nodes in the list can be accessed using a pointer variable, which contains the address of the first node. If *first* points to NULL, it means that the list is empty.

Components of a Linked List:

1) Nodes: A linked list is a non-sequential collection of data items called nodes. These nodes have two fields called data field and link field.
2) Data (info) fields: It contains the actual value to be stored and processed.
3) Link fields: It is used to access a particular node and also known as pointers.
4) Null pointer: The link field of the last node contains zero rather than a valued address. It is known as valued a null pointer and indicates the end of the list.
5) External pointer: This points to very first node in the linked list.
6) Empty list: A list with no nodes is called empty list or null list.

Operations on Linked Lists:

The basic operation that can be performed on a linked list are

1) Create: Used to create a linked list with one or more nodes and their information field is initialized with same value.
2) Insert: Used to insert a new node in the existing list the new node can be inserted.
   a) At the front of a list.
   b) At rear end of a list.
   c) At the specified position in a linked list. This operation is termed as place.
3) Delete (Remove): Used to delete a node from the existing list. The node can be deleted from.
   a) Front of the list.
   b) Rear end of the list.
   c) From the specified position in the list.

4) Traverse: Used to visit all the nodes of the list.

   a) Forward traversing: It refers to traveling the list from first node to last node.

   b) Backward traversing: It refers to traveling the list from the last node to the first node.

5) Search: Used to search whether a given data item is present in the list or not.

6) Concatenate: Used to appends or joins a second list at the end of first list.

7) Display: Used to displays all the data items from a list.

8) Sort: Used to sort the elements of the Linked Lists in ascending or descending order

9) Reverse: Used to reverse the Linked Lists.

10) Print: Used to print the Linked Lists

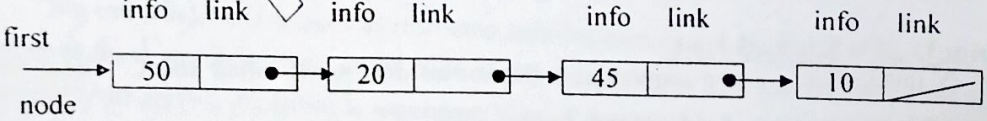
11) Decompose: Used to decomposition of a Linked Lists into Even and Odd Linked Lists.

Types of linked lists:

1) Singly Linked List
2) Doubly Linked List
3) Circular Linked List
4) Non-Integer and Non-Homogeneous Linked List

1) Singly Linked List:

In singly linked list nodes linked together by a single pointer and can be traversed only in the forward direction. It has two ends a front end and the rear end.

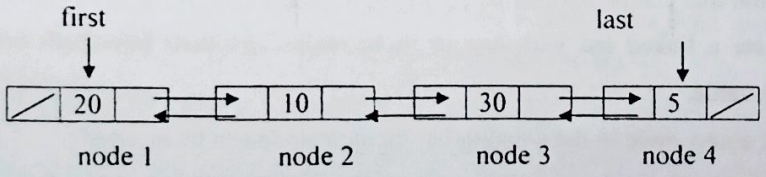


Pictorial representation of singly linked list

The disadvantage of singly list is from a current position one cannot access the previous node. The doubly linked list is used to overcome this disadvantage.

2) Doubly Linked Lists:

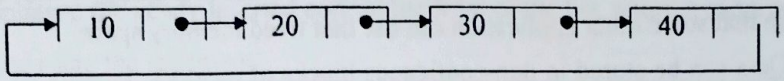
In Doubly linked list adjacent nodes are linked by two pointers a left pointer and a right pointer. The right pointer will point to the next node whereas the left pointer will point to the previous node. Doubly linked list can be traversed in both forward and backward directions and locate given node predecessor and successor.



3) Circular Linked Lists:

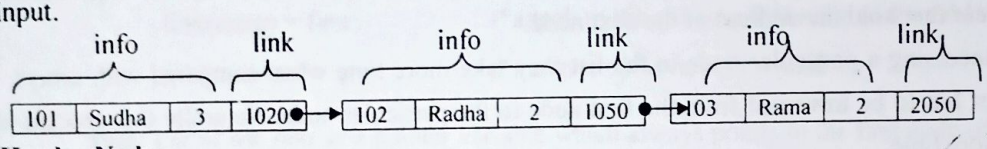
In the linked lists, the link field of the last node contained a NULL pointer. We can modify the linked list by storing address of the first node in the link field of the last node. The resulting list is circular list. From any point in such a list it is possible to reach any other point in the list. A

circular linked list does not have a first or last node. We must, therefore establish a first and last node by convention. A circular list can be used to represent a stack or a queue. A circular list can be of singly linked or doubly linked. A pictorial representation of a circular list is shown in the following figure.



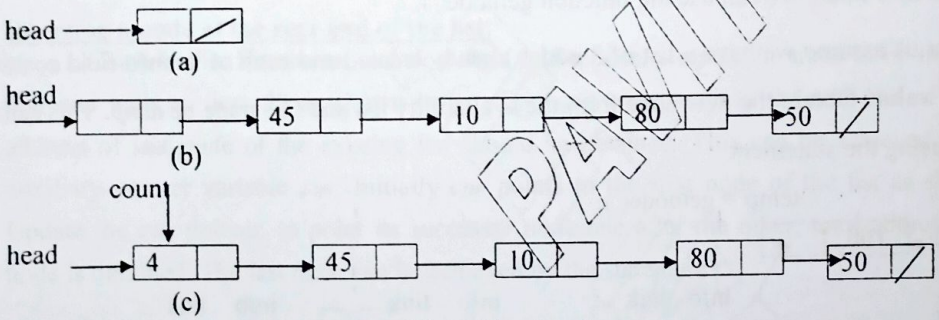

```
struct student
{
        int id;
        char name[10];
        int sem;
        struct student *link;
    } *stud;
```

Here, the student details such as name, id and semester in which he is studying are the input.



Header Nodes:

It may be useful to have a node in the beginning of the list. The purpose of which is used to simplify the design. Such a node is called a *header node*. The *info* field of such node does not represent an item in the list. Sometimes, useful information such as, number of nodes in the list can be stored in the info field as shown in figure (c). The info field of list contains 4, which represents the total number of nodes present in the list. In such a case, there is an overhead of altering the info field of this node. Each time the node is added or deleted, the *count* in this field must be readjusted so as to contain actual number of node to be accessed in the list.



List with a header node

Applications of Linked Lists:

1) Linked lists are used to perform polynomial manipulations.
2) The common operations that are performed are addition, subtraction, division, multiplication, integration and differentiation.
3) Linked lists are also used to maintain a dictionary of names.

Advantages of Linked Lists:

1. The size is not fixed: The linked list can grow or shrink depending on the data availability. If more memory space is required, the required memory space can be allocated if available. If certain portion of memory is not required, the required memory space can be allocated if available. If certain portion of memory is not required, that memory space can be de-allocated so that some other application can use that freed memory space.
2. Data can be stored in non-contiguous blocks of memory: Usually we will not fall shortage of memory space since non-contiguous blocks of memory can be used to store the data.
3. Insertion and deletion of nodes is easier: Unlike arrays, a node can be inserted into any position in the list and a node can be deleted from any position in the list. In such situations, there is no need to shift the nodes.
4. Complex problems can be easily solved by using linked list.

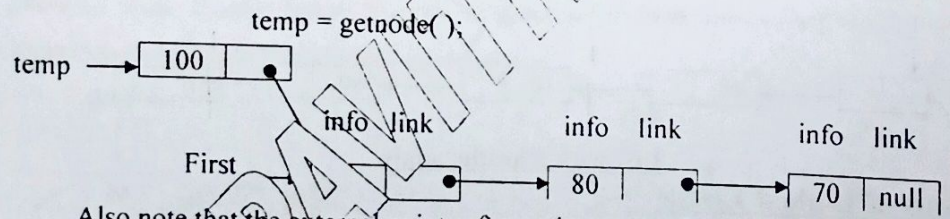
Disadvantages of linked lists:

1. The linked list requires extra space because each node in the list has a special field called *link field* that hold the address of the next node.
2. Accessing a particular node in the list may take more time when compared with arrays. The list has to be traversed from the first node to the particular node. Naturally this process takes more time.
3. Accessing to an arbitrary data item is a little bit difficult and confusing.
4. If the address of first node is lost then the entire list cannot be accessed.
5. Only one element can be accessed in one traversal

Inserting a node at the front end of the list:

Creating the linked list is nothing but repeated insertion of items into the list. Insertion can be done at the front end of the list. To insert an item, a new node is required first. This new node can be obtained from the availability list (availability list is list of free nodes). To obtain memory space for a new node, we can use the function getnode( ).

Step 1: Let us assume we have a list of 3 nodes already created and each of its info field contains an integer value. Obtain the new node from the availability list and this node be *temp*. This can be achieved using the statement



Also note that the external pointer *first* points to the first node of this list.

Step 2: Store the *item* 100 in the information field of node *temp* using the statement:

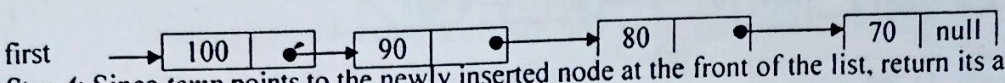
info(temp) = item;

Step 3:

Attach existing first node to the link field of *temp* using the following statement

link(temp) = first;

After insertion of an element, the list will be look like as follows:



Step 4: Since *temp* points to the newly inserted node at the front of the list, return its address to the calling routine using the statement

return temp;

where *temp* is the address of the new first node.

Hence, the algorithmic function to insert an item is as shown below:

```
insert_front(item, first)
{
        temp = getnode( );          /*get a node from the availability first */
        info(temp) = item;          /*insert the item */
        link(temp) = first;         /*attach to the existing first node of the first*/
        return temp;                /*make temp as new first node*/
}
```

Note: Let us use *first* as a pointer variable which always points to the first node of the list. This can be achieved by calling the function Insert_front( ) as shown below:

```
first = insert_front(item, first);
```

When this statement is executed, the variable *first* points to first node of the modified list. The above algorithm has been designed by assuming that the list already exists. If the list is empty, the pointer variable *first* contains NULL to start with. The function works for this case too. Insert the items by calling the function *insert_front( )* to create a list again and again.

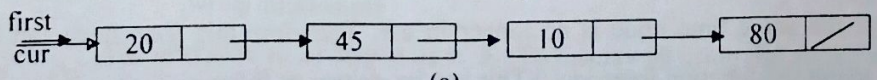
Inserting a node at the rear end of the list:

Consider the list shown in below fig (a). Suppose, *temp* is the node to be inserted at the rear end with an item 50 as shown in (b). To insert node *temp* at the rear end of the list, the address of last node of the existing list should be obtained. This can be achieved by using an auxiliary pointer variable *cur*. Initially *cur* points to the first node of the list as shown in (a). Update the *cur* pointer, to point its successor node one after the other, until address of the last node is obtained. The last node can be achieved by the statement.

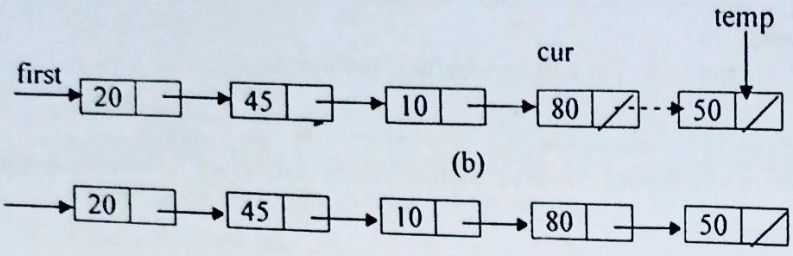
```
cur = cur ->link;
```

Once address of the last node is pointed by *cur*, the temp can be easily inserted at the end. This can be achieved by assigning the address of the node to be inserted i.e., *temp*, to the *link* field of the last node. The statement is

```
cur->link = temp;
```



(a)



(b)

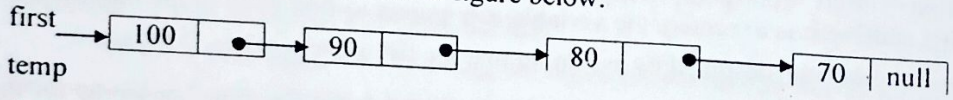
(c)

All these steps have been designed, by assuming that the linked list already exists. So, the program segment is as follows:

```
if(first!=NULL)
{
        cur = first;
        while(cur->link!=NULL)
        {
                cur = cur->link;
        }
        cur->link=temp;
}
return first;
```

Deleting nodes from the front end of the list:

Step 1: Let us assume we have a list of 4 nodes and external pointer *first* points to the very first node of this list as shown in figure below:



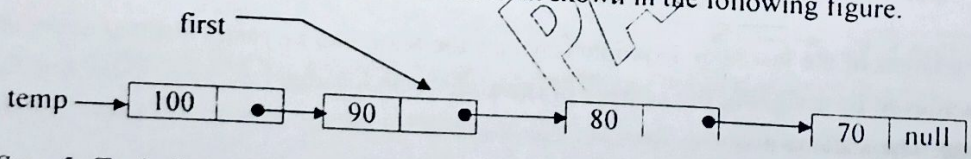
Step 2: To delete a node from the front end of list, apart from the variable *first*, temporary variable *temp* is used. Initially both *first* and *temp* pointers to point to the first node of the list. This can be achieved by the statement:

```
temp = first;
```

Step 3: After deleting the first node, the next should be first node and the variable first should be point to it. Before deleting the first node, update the pointer *first* to point the next node. This can be achieved by the following statement.

```
first = first->link;
```

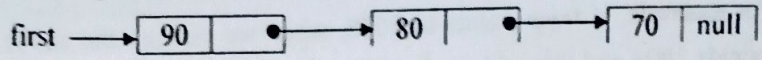
Step 4: The resulting linked list is of the form shown in the following figure.



Step 5: To indicate the completion of delete operation, we should discard the pointer variable *temp* and at the same time node is no longer in use. Therefore this *temp* node can be deleted by using the function freenode( ).This can be written as follows:

freenode(temp);

Step 6: The node pointed by *temp* is deleted and is returned to the availability list. Now, the resulting linked list consist of only three nodes as shown below:



Once the node is deleted, return the address of the new first node identified by the first, by executing the statement

return first;

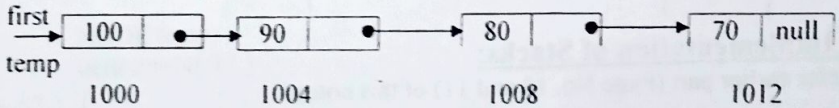
Note that all these statements have to be executed, only when the list exists. If the list is empty, display the message, "List is empty, nothing is to delete" and return NULL. Hence, the algorithmic function to delete an item is as shown below.

```
delete_front(node first)
{
        node temp;
        if(list==NULL)
        {
                cout<<"List is empty, nothing is to delete\n";
                return first;
        }
        temp = first;           /*retain address of the node to be deleted*/
        first = first->link;    /*Make the successor node as the first node*/
        cout<<"The item deleted is \n"<< list;
        freenode(temp);
        return first;
}
```

Display of singly linked list:

One the list is created, the next step is to display the contents of the list. Consider the list as shown below, the variable *first* contains address of the first node. Using *first*, the entire list can be traversed one by one. We use an auxiliary pointer variable *temp* that initially points to the first node itself.

temp=first( );



Singly linked list to be displayed

The items to be displayed are 100, 90, 80 and 70. The first item 100 can be displayed using the following statement.

write(info(temp))

The next item to be displayed is 90. To display 90 using the above statement, temp should contain

address of its successor. This can be done by assigning *link(temp)* which contains address of its successor to temp using the following statement.

```
temp = link(temp)          /*address of the node is copied to temp*/
```

Note: For example, if *temp* points to 1000, when we execute the above statement, the right hand side yields 1004 and this address is copied into temp. Since *temp* contains 1004, now temp points to the second node. Thus by executing *temp=link(temp)* we can point *temp* to point to the next node.

Now, display the item pointed to by *temp* as shown earlier. Repeating this process, all the items in the list can be displayed. The formal version of the algorithm can be written as follows.

```
write(info(temp))
temp=link(temp)
```

After executing these two statements for the first time, the 100 is displayed. Going back as shown using an arrow and executing the statements, the next 90 is displayed. If this process is repeated again and again all the items will be displayed. After displaying 90, note that *temp* points to NULL, indicating all the nodes in the list have been displayed. Now, there is no need to go back and execute those statements i.e., these statements have to be repeatedly executed, as long as *temp* is not pointing to NULL. If *first* points to NULL, it will display the message "List is empty". the algorithmic function to display the lists is shown below:

```
Algorithm display(first)
{
        if(first == NULL)
        {
                write 'list is empty'
                return;
        }
        write 'The contents of the list'
        temp = first
        while(temp!=NULL)
        {
                write(info(temp)
                temp=link(temp)
        }
}
```


Linked Implementation of Stacks:

Refer earlier part (Page No. 10 and 11) of this notes.

Linked Implementation of Queues:

A queue can be implemented using linked list is called linked queue. In a linked queue the insertion of the new node is done at end of the linked list and deletion is performed at the beginning of the linked list.

```
#include<iostream.h>
#include<alloc.h>
```

```
#include<process.h>
struct node
{       int info;
        struct node *link;
}*NODE;
class Queue
{
        public:
        NODE getnode()
        {
                NODE x = new NODE;
                if(x==NULL)
                {
                        cout<<"Out of memory\n";
                        exit(0);
                }
                return x;
        }
        void freenode(NODE x)
        {
                delete x;
        }
        insert_rear(int item, NODE first)
        {
                NODE temp, cur;
                temp = getnode();
                temp->info = item;
                temp->link = NULL;
                if(first == NULL)
                return temp;
                cur = first;                    //if list exists obtain the address of first node
                while(cur->link != NULL)
                {
                        cur = cur->link;
                }
                cur->link = temp;
                return first;
        }
        delete_front(NODE first)
        {
                NODE temp;
                if(first==NULL)
                {
                        cout<<"List is empty, cannot delete\n";
                        return first;
                }
                temp = first;
                first = first->link;
                cout<<"The deleted element is \n" <<temp->info;
                freenode(temp);
                return first;
        }
```

```
        display(NODE first)
        {
                NODE temp;
                if(first==NULL)
                {
                        cout<<"List is empty\n";
                        return;
                }
                cout<<"The contents of queue linked list\n";
                temp = first;
                while(temp != NULL)
                {
                        cout<<temp->info;
                        temp = temp->link;    /*point to next node*/
                }
                cout<<"\n";
        }
};
void main()
{
        Queue Q;
        NODE first;
        int choice, item;
        do
        {
                cout<<"\n1.Insert rear\n2.Delete front\n3.Display\n4.Exit\n";
                cout<<Enter your choice:";
                cin>>choice;
                switch(choice)
                {
                        case 1: cout<<"Enter the item to be inserted\n";
                                cin<<item;
                                first=Q.insert_rear(item, first);
                                break;
                        case 2: first=Q.delete_front(first);
                                break;
                        case 3: Q.display(first);
                                break;
                                default:
                                exit(0);
                }
        }
        while(choice!=4);
        getch();
}
```

The difference between singly linked lists and doubly linked lists

Singly Linked Lists	Doubly Linked Lists
1. Traversing is possible in only one direction	1. Traversing is possible in both the direction
2. To delete a specific node address of its	2. The address of the previous node

predecessor should be known. To find the address of the predecessor, it is required to traverse the list from the very first node.	can be obtained from the left link of the node to be deleted.

The difference between Circular Queue and Circular Linked list

Circular Queue	Circular Linked List
1) Size of the array is fixed	1) No size of the circular linked list is fixed
2) Stores in contiguous memory spaces	2) Stores in non-contiguous memory spaces
3) Since the data items stores in contiguous memory locations there is front end and rear end	3) Since the data items stores in non-contiguous memory locations there is no front end and rear end
4) In insertions and deletions are in FIFO order	4) In insertions and deletions are in any order ie., front element, rear element, and specified location
5) Manipulation of stored data is sequential	5) Manipulation of stored data is not sequential. Any desired data can be accessed
6) Spent more time in shifting of array elements	6) There is no shifting of nodes

Write a program to implement singly linked list

```
#include<stdio.h>
#include<alloc.h>
struct node
{
        int info;
        struct node *link;
}*first;
void main()
{
        int item, i;
        first = NULL;
        printf("1.Add at the beginning\n, 2.Add at the middle\n, 3.Add  at the end\n, 4.Display the
        list\n, 5.Delete the node\n, Enter your choice:");
        scanf("%d", &i);
        if(i==1)
        {
                printf("Enter the number to insert at the beginning of the list \n");
                scanf("%d", &item);
                insertbeg(&first, item);
        }
        else if(i==2)
        {
                printf("Enter the number to insert at the middle of the sorted list\n");
                scanf("%d", &item);
```

```
            middle(&first, item);
        }
        else if(i==3)
        {
                printf("Enter the number to insert at the end of the list\n");
                scanf("%d", &item);
                end(&first, item);
        }
        else if(i==4)
        {
                printf("Delete the node at the beginning of the list\n");
                delete_beginning();
        }
        else if(i==5)
        {
                printf("Delete the node of the given info in the list\n");
                scanf("%d", &item);
                delete_given_info(&first, item);
        }
        else if(i==6)
        {
                printf("Delete the node at the end of the list\n");
                delete_end();
        }
        else if(i==7)
        {
                printf("Sort the elements in the linked list\n");
                sorting();
        }
        else if(i==8)
        {
                printf("Reverse the linked list\n");
                reverse();
        }
        else if(i==9)
        {
                printf("\n**********Display the elements of the list **********\n");
                display(first);
        }
        else
                exit(0);
getch();
}
```

Write a function to insert at the beginning of the list

```
void insert_at_beg(struct node **first, int num)
{
        struct node *temp;
        temp = malloc(sizeof(struct node));
        temp->info = num;
        temp->link = *first;
        *first=temp;
```

```
        return;
}
```

Write a function to insert at the middle of the sorted list

```
void insert_at_middle(struct node **first, int num)
{
        struct node *mid, *temp=*first;
        mid = malloc(sizeof(struct node));
        mid->info=num;
        while(temp!=NULL)
        {
                if((temp->info <= num)&&(temp->link->info > num)||(temp->link==NULL))
                {
                        mid->link = temp->link;
                        temp->link = mid;
                        return;
                }
                temp=temp->link;
        }
        return;
}
```

Write a function to insert at the end of the list

```
void insert_at_end(struct node **first, int num)
{
        struct node *temp, *last;
        temp=*first;
        if(first==NULL)
        {
                first=malloc(sizeof(struct node));
                first->info = num;
                first->link = NULL;
                return;
        }
        else
        {
                while(temp->link!=NULL)
                {
                        temp = temp->link;
                }
                last = malloc(sizeof(struct node));
                last->info = num:
                last = temp->link;
                last->link = NULL;
                return;
        }
}
```

Write a function to delete the node from the beginning of the list

```
void delete_beginning( )
{
        struct node *ptr, *first;
        ptr = first;
        first = ptr->link;
        free(ptr);
```

```
}
```

Write a function to delete the given info node from the list

```
void delete_given_info(struct node **first, int num)
{
        struct node *ptr, *cpt;
        ptr = first;
        while(ptr->info !=num)
        {
                cpt = ptr;
                ptr = ptr->link;
        }
        cpt->link = ptr->link;
        free(ptr);
}
```

Write a function to delete the node at the end of the list

```
void delete_end( )
{
        struct node *ptr, *cpt, *first;
        ptr = first;
        while(ptr->link !=NULL)
        {
                cpt = ptr;
                ptr = ptr->link;
        }
        cpt->link = NULL;
        free(ptr);
}
```

Write a function to display the list

```
void display(struct node *first)
{
        printf("The list element are\n");
        while(first!= NULL)
        {
                printf("%d\n", first->info);
                first=first->link;
        }
        printf("List is empty\n");
        return;
}
```

Write a function to sort the linked list

```
void sorting( )
{
        int temp;
        struct node *ptr, *cpt, *first;
        ptr = first;
        while(ptr->link !=NULL)
        {
                cpt = ptr->link;
                while(cpt !=NULL)
                {
                        if(ptr->info > cpt->info)
                        {
```

```
                        temp = ptr -> info;
                        ptr -> info = cpt ->info;
                        cpt - > info = temp;
                    }
                    cpt = cpt -> link;
                }
                ptr = ptr->link;
        }
}
```

Write a function to reverse the linked list

```
static void reverse(struct Node **first)
{
        struct Node *prev   = NULL;
        struct Node *current = *first;
        struct Node *next;
        while (current != NULL)
        {
                next  = current->link;
                current->link = prev;
                prev = current;
                current = next;
        }
        *first = prev;
}
```

Write a Program to implement doubly linked list

```
#include<alloc.h>
#include<stdio.h>
struct dnode
{
        struct dnode *prev,*next;
        int info;
}*start, *end;
void main()
{
        int ch, num;
        q=NULL;
        do
        {
                printf("\n1.Insert at the beginning \n 2.Insert at the End  \n 3. Insert node after
                particular node \n 4.Delete a desired node\n  5. Display \n 6.Exit \nEnter your
                choice:");
                scanf("%d", &ch);
                 switch(ch)
                 {
                        case 1:
                                printf("\nEnter an element to be inserted:");
                                scanf("%d", &num);
                                insert_beginning(&first, num);
                                break;
                        case 2:
                                printf("\nEnter an element to be inserted:");
                                scanf("%d", &num);
```

```
                    insert_end(&first, num);
                    break;
                case 3:
                    printf("\nEnter an element to be inserted:");
                    scanf("%d", &num);
                    insert_after(&first, num);
                    break;

                case 4:
                    printf("Enter the element to be deleted:");
                    scanf("%d", &num);
                    delete(&first, num);
                    break;
                case 5:
                    display(&first);
                    break;
                case 6:
                    traverse();
                    break;
                case 7:
                    count();
                    break;
                case 8:
                    search();
                    break;
            }
        }
        while(ch != 8);
}
```

/*Create a first node in a list which contains the info & 2 link fields (prev & next) points to NULL*/

```
void create()
{
        int data;
        start = (struct dnode *) malloc(sizeof(struct dnode));
        printf("Enter the data you want to insert: ");
        scanf("%d", &data);
        start->info = data;
        start->prev = NULL;
        start->next = NULL;
        end = start;
}
```

/* Function to insert a node at the beginning of the doubly linked list*/

```
void insert_beginning(struct dnode **first, int data)
{
        struct dnode *temp;
        temp = (struct dnode *) malloc(sizeof(struct dnode));
        temp->info = data;
        temp->prev = NULL;
        temp->next = start;
        start = temp;
}
```

```
/* Function to insert a node at the end of the doubly linked list*/
void insert_end(struct dnode **first, int data)
{
        struct dnode *temp;
        temp = (struct dnode *) malloc(sizeof(struct dnode));
        temp->info = data;
        temp->next = NULL;
        temp->prev = end;
        end->next = temp;
        end = temp;
}
/* Function to insert a node after comparing the value of the node in doubly linked list*/
void insert_after(struct dnode **first, int num)
{
        struct dnode *temp, *ptr;
        ptr = start;
        while(ptr -> info == num)              /*Num is passed from main*/
        {
                temp = (struct dnode *) malloc(sizeof(struct dnode));
                temp->info = num;
                temp->next = ptr->next;
                temp->prev = ptr;
                ptr->next = temp;
                return;
        }
        ptr = ptr->next;
        if(ptr -> next == NULL && ptr -> info == num)
        {
                start = ptr;
                insert_after(first, num);
        }
}
/* Function to delete a desired node from the doubly linked list*/
void delete_node(struct dnode **start, int data)
{
        struct dnode *ptr;
        ptr = start;
        if(start -> info == num)
        {
                start = start ->next;
                start -> prev = NULL;
                free(ptr);
                return;
        }
        else if(ptr ->next != NULL)
        {
                if(ptr -> info != num)
                {
                        ptr -> prev -> next = ptr -> next;
                        ptr -> next -> prev = ptr -> prev;
                        free(ptr);
                        return;
```

```
            }
            ptr = ptr -> next;
        }
        else if(ptr -> info == num)
        {
                end = end -> prev;
                end -> next = NULL;
                return;
        }
}
/* Function to display the contents of the doubly linked list*/
void display(struct dnode *first)
{
        struct dnode *temp=*first;
        printf("\nContents of the doubly linked list are: ");
        while(temp != NULL)
        {
                printf("%d\t", temp->info);
                temp=temp->next;
        }
        printf("\n No Contents to Display in the list.\n ");
}
/* Function to traverse the doubly linked list*/
void traverse (struct dnode *first)
{
        struct dnode *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\t");
                printf("%d\t", ptr->info);
                ptr = ptr -> next;
        }
        printf("\n No Contents to Traverse the list.\n ");
}
/* Function to count the number of nodes in the doubly linked list*/
void counter (struct dnode *first)
{
        int count = 0;
        struct dnode *ptr;
        ptr = start;
```

(next page)

```
        while(ptr != NULL)
        {
                count++;
                ptr = ptr -> next;
        }
        printf("Number of nodes in the list is %d: " count);
}
```

/* Function to count untill given info node in the doubly linked list*/

```
void count (struct dnode *first, int num)
{
        int count = 0;
        struct dnode *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                count++;
                if( ptr -> info == num)
                {
                        return (count);
                        ptr = ptr -> next;
                }
        }
        return (0);
}
```

Assignment : Write a Program to implement circular linked list (Write your own code)