UNIT III
LINEAR DATA STRUCTURES

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation — singly linked lists –Polynomial Manipulation - Stack ADT – Queue ADT - Evaluating arithmetic expressions

3. INTRODUCTION

Data Structures

Data refers to a single value or a set of values.

Data Structures is the representation of the logical relationship between individual elements of data. This is a way of organizing all data items that considers not the elements stored but also their relationship to each other.

Types of Data Structures

1. *Linear Data Structures (Arrays, Lists, Stacks, Queues) :* This is an ordered set consisting of a variable number of elements to which insertions and deletion can be made. A List which displays the relationship of adjacency between elements is said to be linear.
2. *Non- Linear Data Structures (Tree, Graphs) :* This is an unordered set consisting of a variable number of elements to which insertions and deletion can be made.

3.1. Abstract Data Types (ADTs)

Abstract Data Types (ADT) is a set of objects(lists, sets and graphs) together with a set of operations. E.g. List ADT, Tree ADT, Stack ADT, Set ADT and so on.

➢ For example, In ***list ADT***, we have the operations such as insert, delete, and find, count and so on. The ***Set ADT*** the operations are Union, Intersection, size, complement and so on.

The basic idea is that the implementation of these operations is written once, further needs of this ADT we can call the appropriate function.

3.2. List ADT

A list is a sequence of zero or more elements of a given type. It can be of the form A_1, A_2, A_3, …, An , where the size of the list is N. If the size is 0 then the list is called as *empty list.*

For any list except the empty list the element A_{i+1} follows the element A_i ($i<n$) and that A_{i-1} precedes A_i ($i>1$) The first element of the list is A_1 and the last element of the list is A_n. A_1 has no predecessor and A_n has no successor. The position of the element A_i in a list is i.

Some **Operations**

1. ***PrintList*** **–** Print the elements of the list in the order of occurrence
2. ***Find*** – Returns the position of the first occurrence of the element in the list
3. ***Insert*** **–** Insert an element into the list at the specified position
4. ***Delete*** **–** Delete an element from the list
5. ***MakeEmpty*** **–** Empty the list.

The Last cell"s Next pointer point to NULL.

3.3. Types of Implementation

1. **Array-based Implementation -** In this implementation, the list elements are stored in contiguous cells of an array. All the list operation can be implemented by using the array.

   Disadvantages of Array Implementation

   i. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high overestimate, which waste considerable space.

   ii. Insertion and deletion operations are expensive, because insertion at the beginning of the array requires pushing the entire array elements one step downwards.
   As like the deleting the first element of the array requires, shifting all elements up one position. So the worst case operation requires the computation time O(n).

2. **Linked List Implementation -** Linked list consist of a series of structures, which are not necessarily in adjacent in memory. The structure is called as Node. Each structure (node) contains

   i. Element

   ii. Pointer to a structure containing its successor. – It is called as Next Pointer

Types of Linked List

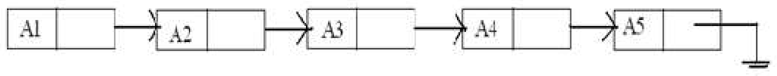
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List.

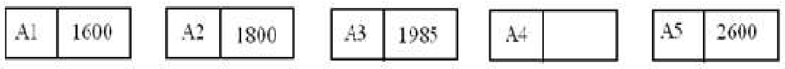
3.4. Singly Linked List

A singly linked list is a list in which each node contains only one link field pointing to the next node in the list. A node in this type of linked list contains two types of fields.

Data – This holds the list element

Next – Pointer to the next node in the list





The list contains five structures. The structures are stored at memory locations 800, 1600, 1800, 1985, 2600 respectively. The Next pointer in the first structure contains the value 1600.

Basic linked List Operations

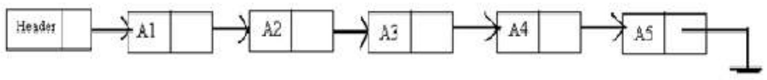
The basic operations to be performed on linked lists are as

1. Creation - Create a linked list
2. Insertion - insert a new node at the specified position
3. Deletion - delete the specified node
4. Traversing - to display every node information
5. Find - Search a particular data

Implementation

To access the list, we must know the address of the header Node. To insert a new node at the beginning of the list, we have to change the pointer of the head node. If we miss to do this we can lose the list. Likewise deleting a node from the front of the list is also a special case, because it changes the head of the list.

To solve the above mentioned problem, we will keep a sentinel node. A linked list with header representation is shown below,

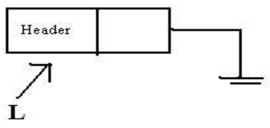


For easy implementation of all linked list operation a *sentinel node* is maintained to point the beginning of the list. This node is sometimes referred to as a header or *dummy node*.

Singly-Linked List Internal Representation

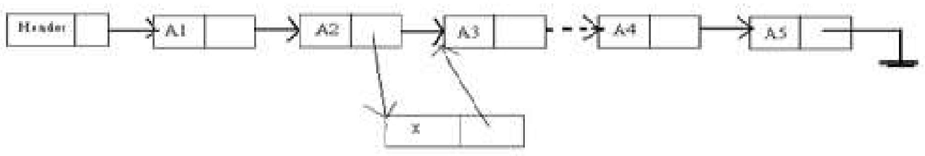
Every node of a singly-linked list contains following information:

- a value (user's data)
- a link to the next element (auxiliary data)



```
class SinglyLinkedListNode
{
      public:
            int value;
            SinglyLinkedListNode *next;
            SinglyLinkedListNode(int value)
            {
                  this->value = value;
                  next = NULL;
            }
};
class SinglyLinkedList
{
      private:
            SinglyLinkedListNode *head;
            SinglyLinkedListNode *tail;
      public:
            SinglyLinkedList()
            {
                  head = NULL;
                  tail = NULL;
            }
};
```

Singly Linked List Addition (insertion) operation.



Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

```
void SinglyLinkedList::addLast(SinglyLinkedListNode *newNode)
{
        if(newNode==NULL)
                return;
        else
        {
                newNode->next=NULL;
                if(head==NULL)
                 {
                        head=newNode;
                        tail=newNode;
                }
                 else
                 {
                        tail->next=newNode;
                        tail=newNode;
                }
        }
}
void SinglyLinkedList::addFirst(SinglyLinkedListNode *newNode)
{
        if(newNode==NULL)
                return;
        else
        {
                if(head==NULL)
                 {
                        newNode->next=NULL;
                        head=newNode;
                        tail=newNode;
```

```
            }
             else
             {
                  newNode->next=head;
                  head=newNode;
            }
      }
}
void SinglyLinkedList::insertAfter(SinglyLinkedListNode
*previous,
            SinglyLinkedListNode *newNode)
{
      if(newNode==NULL)
            return;
      else
      {
            if(previous==NULL)
                  addFirst(newNode);
            else if(previous==tail)
                  addLast(newNode);
            else
             {
                  SinglyLinkedListNode *next=previous->next;
                  previous->next=newNode;
                  newNode->next=next;
            }
      }
}
```

Singly Linked List Traversal

Assume, that we have a list with some nodes. Traversal is the very basic operation, which presents as a part in almost every operation on a singly-linked list. For instance, algorithm may traverse a singly-linked list to find a value, find a position for insertion, etc. For a singly-linked list, only forward direction traversal is possible.

Traversal algorithm

Beginning from the head,

1. check, if the end of a list hasn't been reached yet
2. do some actions with the current node, which is specific for particular algorithm
3. current node becomes previous and next node becomes current. Go to the step 1.

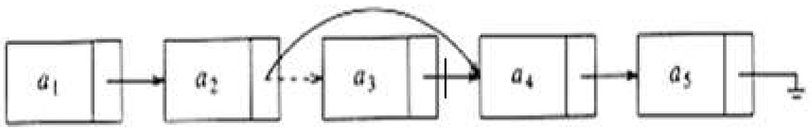
```
int SinglyLinkedList::traverse()
{
      int sum=0;
      SinglyLinkedListNode *current=head;
      SinglyLinkedListNode *previous=NULL;
      while(current!=NULL)
      {
            sum+=current->value;
            previous=current;
            current=current->next;
      }
      return sum;
}
```

Singly Linked List Removal (deletion) operation

There are four cases, which can occur while removing the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite.

List has only one node

When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to *NULL*.



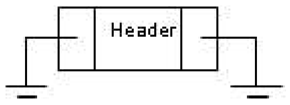
```
void SinglyLinkedList::removeFirst()
```

```
{
      if(head==NULL)
            return;
      else
      {
            SinglyLinkedListNode *removedNode;
            removedNode=head;
            if(head==tail)
            {
                  head=NULL;
                  tail=NULL;
            }
            else
            {
                  head=head->next;
            }
            delete removedNode;
      }
}
void SinglyLinkedList::removeLast()
{
      if(tail==NULL)
            return;
      else
      {
            SinglyLinkedListNode *removedNode;
            removedNode=tail;
            if(head==tail)
            {
                  head=NULL;
                  tail=NULL;
            }
            else
            {
                  SinglyLinkedListNode *previousToTail=head;
                  while(previousToTail->next!=tail)
                        previousToTail=previousToTail->next;
                  tail=previousToTail;
                  tail->next=NULL;
            }
            delete removedNode;
      }
}
void SinglyLinkedList::removeNext(SinglyLinkedListNode
*previous)
{
      if(previous==NULL)
            removeFirst();
      else if(previous->next==tail)
      {
            SinglyLinkedListNode *removedNode=previous->next;
            tail=previous;
            tail->next=NULL;
            delete removedNode;
      }
      else if(previous==tail)
            return;
      else
      {
            SinglyLinkedListNode *removedNode=previous->next;
            previous->next=removedNode->next;
            delete removedNode;
      }
}
```

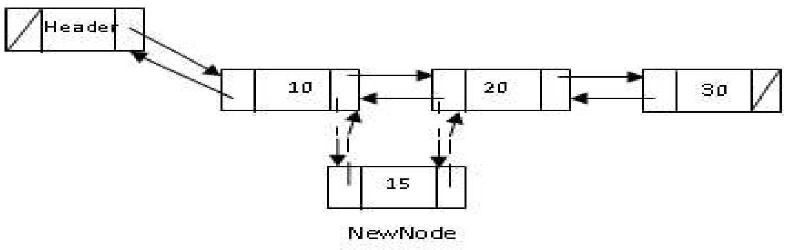
Doubly Linked List

A node contains pointers to previous and next element. One can move in both directions.

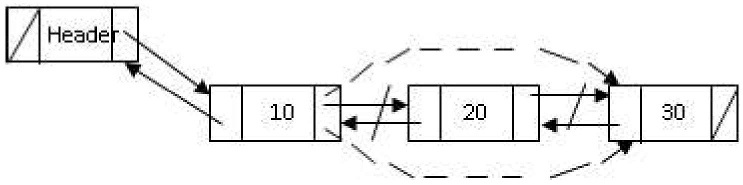
Node Declaration



To insert an element in a doubly Linked List



To delete an element



```
typedef string Elem; // list element type
class DNode
{ // doubly linked list node
        private:
        Elem elem; // node element value
        DNode* prev; // previous node in list
        DNode* next; // next node in list
        friend class DLinkedList; // allow DLinkedList access
};
class DLinkedList
{ // doubly linked list
        public:
                DLinkedList() // constructor
                { // constructor
                        header = new DNode; //create sentinels
```

```
      trailer = new DNode;
      header->next=trailer;//  have  them  point  to
      each other
      trailer->prev = header;
}
~DLinkedList() // destructor
{
      while  (!empty())  removeFront()  //  remove  all
      but sentinels
      delete header; // remove the sentinels
      delete trailer;
}
bool empty() const // is list empty?
{
      return (header->next == trailer);
}
const Elem& front() const // get front element
{
      return header->next->elem;
}
const Elem& back() const // get back element
{
      return trailer->prev->elem;
}
void addFront(const Elem& e) // add to front of list
{
      add(header->next, e);
}
void addBack(const Elem& e) // add to back of list
{
      add(trailer, e);
}
```

```
            void removeFront() // remove from front
            {
                  remove(header->next);
            }
            void removeBack() // remove from back
            {
                  remove(trailer->prev);
            }
            private: // local type definitions
                  DNode* header; // list sentinels
                  DNode* trailer;
            protected: // local utilities
                  void add(DNode* v, const Elem& e) // insert
            new node before v
                  {
                        DNode* u = new DNode;
                        u->elem = e; // create a new node for e
                        u->next = v; // link u in between v
                        u->prev = v->prev; // . . .and v->prev
                        v->prev->next = v->prev = u;
                  }
                  void remove(DNode* v) // remove node v
                  { // remove node v
                        DNode* u = v->prev; // predecessor
                        DNode* w = v->next; // successor
                        u->next = w; // unlink v from list
                        w->prev = u;
                        delete v;
                  }
};
```

Advantages

1. It is more efficient.

Disadvantages of Doubly Linked list over Single Linked list

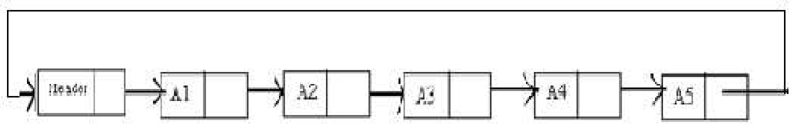
1. The location of the preceding node is needed.
2. The two-way list contains this information, whereas with a one-way list we must traverse the list.
3. A two-way list is not much more useful than a one-way list except in special circumstances.

Circular Linked List

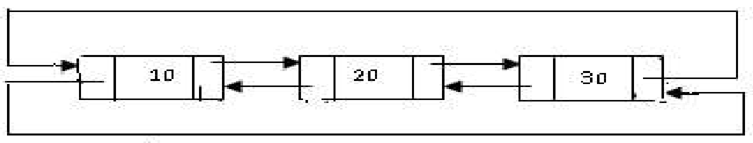
The last node points to the first one.

Types of Circular linked List

1. Circular Singly Linked List



2. Circular Doubly Linked List



3.5. Polynomial Manipulation

It is a simple way to represent single-variable polynomials. A polynomial, P(x) is an expression, the variable x of the form ($ax^n+bx^{n-1}+\ldots+jx+k$) where a,b,c,…k are real numbers.

n is a non-negative number (n-degrees of the polynomial)

In is a non-negative number (n-degrees of the polynomial)

In this polynomial expression consists of two parts one is a co-efficient and the other is an exponent. Consider the following polynomial.

$1000x^4+200x^3+100x^2+50x$ Here (1000, 200, 100, 50) these are co-efficient and (4,3,2,1) are exponent.

```
#include<conio.h>
#include<iostream.h>
#include<process.h>
```

```
//   Creating a NODE Structure
struct node
{
   int coe,exp;         // data
   struct node *next;   // link to next node and previous node
};
// Creating a class Polynomial
class polynomial
{
   struct node *start,*ptrn,*ptrp;
   public:
     void get_poly(); // to get a polynomial
     void show();     // show
     void multiply(polynomial p1,polynomial p2); // Multiply
polynomials
     void sort();  // Sort Polynomials
};
void polynomial::get_poly()      //  Get Polynomial
{
   char c='y';
   ptrn=ptrp=start=NULL;
   while(c=='y' || c=='Y')
   {
      ptrn=new node;
      if(ptrp!=NULL)
      ptrp->next=ptrn;
      if(start==NULL)
      start=ptrn;
      ptrp=ptrn;
      cout<<"\nEnter the coefficient: ";
      cin>>ptrn->coe;
      cout<<"Enter the exponent: ";
      cin>>ptrn->exp;
      ptrn->next=NULL;
      cout<<"Enter y to add more nodes: ";
      cin>>c;
   }
   return;
}
void polynomial::show()  // Show Polynomial
{
   struct node *ptr;
   ptr=start;
   while(ptr!=NULL)
   {
      cout<coe<<"X^"<exp<<" + ";
      ptr=ptr->next;
   }
   cout<<"\b\b ";
}
// Multiplying
void polynomial::multiply(polynomial p1,polynomial p2)
{
   struct node *p1ptr,*p2ptr,*ptri,*ptrj;
   int exp,coe;
   ptrn=ptrp=start=NULL;
   p1ptr=p1.start;
   p2ptr=p2.start;
   while(p1ptr!=NULL)   // Start multiplying
   {
      p2ptr=p2.start;
      while(p2ptr!=NULL)
      {
       ptrn=new node;
```

```
         ptrn->next=NULL;
         if(start==NULL)
         start=ptrn;
         if(ptrp!=NULL)
         ptrp->next=ptrn;
         ptrn->coe=p1ptr->coe*p2ptr->coe;
         ptrn->exp=p1ptr->exp+p2ptr->exp;
         ptrp=ptrn;
         p2ptr=p2ptr->next;
        }
        p1ptr=p1ptr->next;
    }                    // end of multiplication
    ptri=ptrj=start;     // Normalising
    ptrn=ptrp=start=NULL;
    while(ptri!=NULL)
    {
        exp=ptri->exp;
        coe=ptri->coe;
        ptrj=ptri->next;
        while(ptrj!=NULL)
        {
         if(ptrj->exp==ptri->exp)
         {
            coe=coe+ptrj->coe;
            ptrj->coe=0;//ptrj_p->next=ptrj->next;
         }
         ptrj=ptrj->next;
        }
        if(coe!=0)
        {
         ptrn=new node;
         ptrn->next=NULL;
        if(start==NULL)
        start=ptrn;
        if(ptrp!=NULL)
        ptrp->next=ptrn;
        ptrn->coe=coe;
        ptrn->exp=exp;
        ptrp=ptrn;
       }
       ptri=ptri->next;
   }
   return;
}
void polynomial::sort()  // Sort Polynomials
{
   struct node *ptri,*ptrj;
   int coe,exp;
   ptri=ptrj=start;
   while(ptri->next!=NULL)
   {
      ptrj=ptri->next;
      while(ptrj!=NULL)
      {
       if(ptri->expexp)
       {
          coe=ptri->coe;
          exp=ptri->exp;
          ptri->coe=ptrj->coe;
          ptri->exp=ptrj->exp;
          ptrj->coe=coe;
          ptrj->exp=exp;
       }
       ptrj=ptrj->next;
      }
      ptri=ptri->next;
   }
   return;
}
int main()
{
   clrscr();
   polynomial p1,p2,product;
   cout<<"First Polynomial.\n";
   p1.get_poly();
   cout<<"\nSecond polynomial.\n";
   p2.get_poly();
   clrscr();
   cout<<"\nThe First polynomial is: ";
   p1.show();
   cout<<"\nThe second polynomial is: ";
   p2.show();
   cout<<"\n\nThe product of two polynomials is: \n";
   product.multiply(p1,p2);
   product.sort();
   product.show();
   getch();
   return 0;
}
```

3.6. Stack ADT

A *stack* is a data structure that *inserts* and *deletes* can be performed in only one position, namely the end of the list called the *top*.

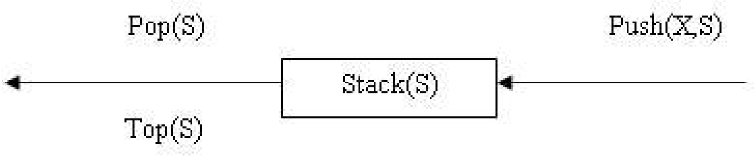
A stack is a data structure in which only the top element can be accessed. As data is stored in the stack, each data is pushed downward, leaving the most recently added data on top.

The fundamental operations on a stack are

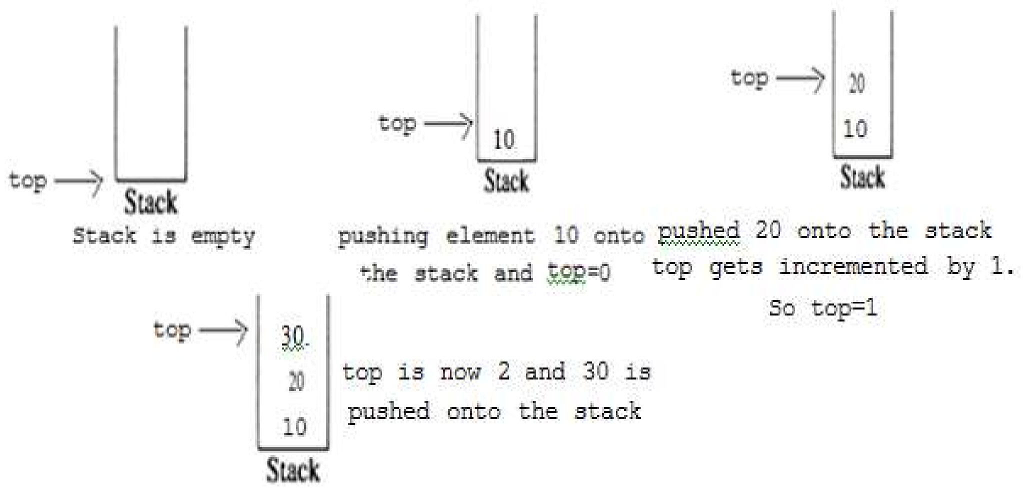
- ***push***, which is equivalent to an insert,

- ***pop***, which deletes the most recently inserted element.

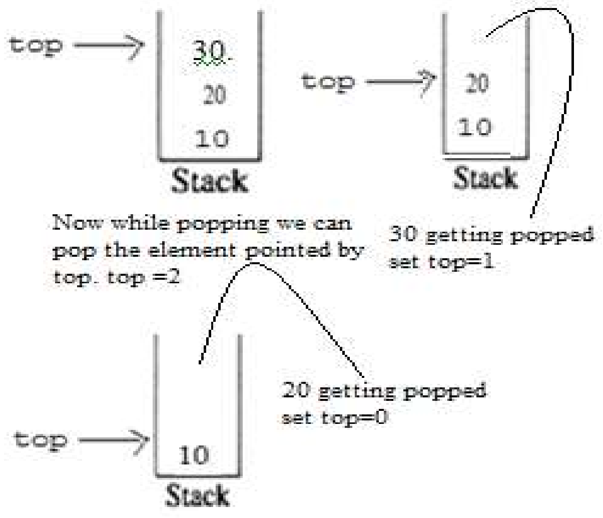
Stack Model



Insert the following elements into stack. The elements are, 10, 20, 30.


top
Stack
Stack is empty
top
10
Stack
pushing element 10 onto the stack and top=0
top
20
10
Stack
pushed 20 onto the stack top gets incremented by 1. So top=1
top
30
20
10
Stack
top is now 2 and 30 is pushed onto the stack

Pop operation can be shown below,Pop(20)


top
30
20
10
Stack
Now while popping we can pop the element pointed by top. top =2
top
20
10
Stack
30 getting popped set top=1
20 getting popped set top=0
top
10
Stack

3.6.1. Implementation of Stack

3.6.1.1. Array Based Implementation of Stacks

Implementation of array-based stack is very simple. It uses **top** variable to point to the topmost stack's element in the array.

1. Initialy **top = -1;**
2. **push** operation increases top by one and writes pushed element to **storage[top];**
3. **pop** operation checks that top is not equal to -1 and decreases top variable by 1;
4. **peek** operation checks that top is not equal to -1 and returns **storage[top];**
5. **isEmpty** returns boolean **(top == -1).**

```
class Stack
{
      private:
            int top;
            int capacity;
            int *storage;
      public:
            Stack(int capacity)
            {
                  if (capacity <= 0)
                        throw string("Stack's capacity must be
positive");
                  storage = new int[capacity];
                  this->capacity = capacity;
                  top = -1;
            }
            void push(int value)
            {
                  if (top == capacity)
                        throw string("Stack's underlying storage
is overflow");
                  top++;
                  storage[top] = value;
            }
            int peek()
            {
                  if (top == -1)
                        throw string("Stack is empty");
                  return storage[top];
            }
            void pop()
            {
                  if (top == -1)
                        throw string("Stack is empty");
                  top--;
            }

            bool isEmpty()
            {
                  return (top == -1);
            }
            ~Stack()
            {
                  delete[] storage;
            }
};
```

3.6.1.2. Linked List Implementation of Stacks

Inserting an element into linked list contains 3 types .

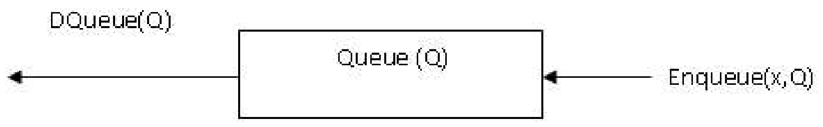
1. Insertion at beginning of the Linked list
2. Insertion at the middle of the linked list
3. Insertion at the end of the linked list

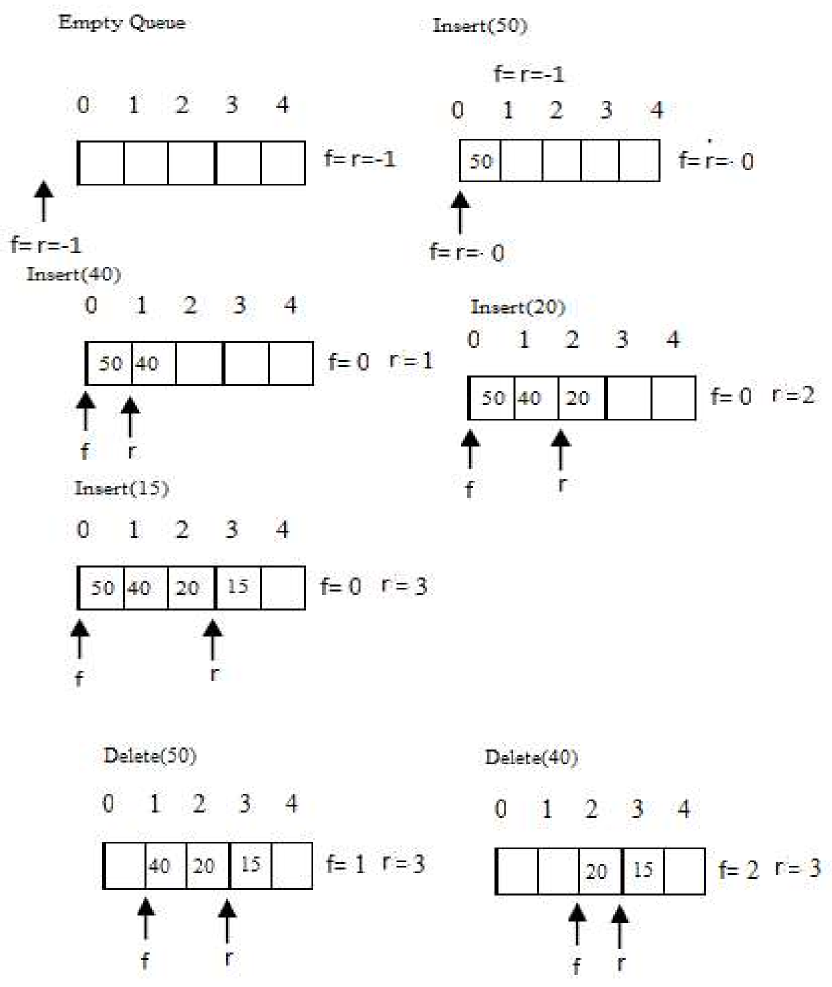
```
class stack
{
      int element;
      stack* next;
      public:
            stack* push(stack*,int);
            stack* pop(stack*);
}*head,object;
stack* stack::push(stack* head,int key)
{
      stack* temp,*temp1;
      temp1=head;
      temp=new stack;
      temp->element=key;
      temp->next=NULL;
      if(head==NULL)
            head=temp;
      else
      {
            while(head->next!=NULL)
                  head=head->next;
            head->next=temp;
            head=temp1;
      }
      return head;
}
stack* stack::pop(stack* head)
{
      stack* temp;
      if(head!=NULL)
      {
            temp=head;
            if(head->next==NULL)
            {
                        return NULL;
            }
            while(head->next->next!=NULL)
                  head=head->next;
            head->next=head->next->next;
            head=temp;
            return head;
      }
}
void stack::stack_display(stack* head)
{
      if(head!=NULL)
      {
            while(head->next!=NULL)
                  head=head->next;
            cout<<head->element;
      }
 }
```

3.7. Queue ADT

Queue is an ordered collection of elements in that insertion is done at one end called rear, whereas deletion is performed at the other end called front. The basic operations on a queue are

- enqueue, which inserts an element at the end of the list (called the rear)
- dequeue, which deletes (and returns) the element at the start of the list (known as the front).





3.7.1. Implementation of Stack

3.7.1.1. Array Implementation of Queues

For each queue data structure, we keep an array, *QUEUE*[], and the positions *q_front* and *q_rear,* which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, *q_size*. All this information is part of one structure, and as usual, except for the queue routines themselves, no routine should ever access these directly.

```
class queue
{
        public :
                int *a,f,r,size;
        queue()
        {
                f=0;
                r=-1;
              a = new int [size];
        }
        int isempty();
        int isfull();
        int push();
        int pop();
};
int queue :: isempty()
{
       if(r==-1)
               return 1;
       return 0;
}
int queue :: isfull()
{
       if(r==(size-1))
            return 1;
       return 0;
}
int queue :: push()
{
       if(isfull())
        return 0;
       else
             a[++r];

}
```

```
int queue :: pop()
{
     if(isempty())
             return 0;
     for(int i=0;i<=r;i++)
            a[i]=a[i+1];
        --r;
}
```

3.7.1.2. Linked List Implementation of Stacks

```
class queue
{
      int element;
      queue* next;
      public:
            queue* enqueue(queue*,int);
            queue* dequeue(queue*);
}*head,*tail,object;
queue* queue::enqueue(queue* head,int key)
{
      queue* temp;
      temp=new queue;
      temp->element=key;
      temp->next=NULL;
      if(head==NULL)
            head=temp;
      else
            tail->next=temp;
      tail=temp;
      return head;
}
queue* queue::dequeue(queue* head)
{
        queue* temp;
        if(head==NULL)
        {
                cout<<"\nit is impossible to dequeue an element as
";
                return NULL;
        }
        else if(head->next==NULL)
                return NULL;
        else
        {
                temp=head->next;
                head=temp;
                return head;
        }
}
void queue::queue_display(queue* head)
{
        if(head!=NULL)
        {
                while(head->next!=NULL)
                        head=head->next;
        }
        cout<<head->element;
        cout<<endl;
 }
```

3.7.1.3. Double-Ended Queues

A queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. The functions of the deque ADT are as follows, where *D* denotes the deque:

insertFront(*e*)	: Insert a new element *e* at the beginning of the deque.
insertBack(*e*)	: Insert a new element *e* at the end of the deque.
eraseFront()	: Remove the first element of the deque; an error occurs if the deque is empty.

eraseBack()	: Remove the last element of the deque; an error occurs if the deque is empty.

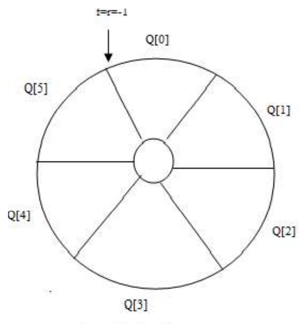
Additionally, the deque includes the following support functions:

front()	: Return the first element of the deque; an error occurs if the deque is empty.
back()	: Return the last element of the deque; an error occurs if the deque is empty.
size()	: Return the number of elements of the deque.
empty()	: Return true if the deque is empty and false otherwise.

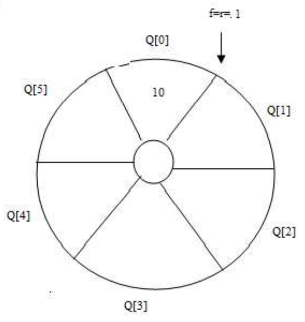
```
typedef string Elem; // deque element type
class LinkedDeque
{ // deque as doubly linked list
      public:
            LinkedDeque(); // constructor
            int size() const; // number of items in the deque
            bool empty() const; // is the deque empty?
            const                Elem&                front()const
      throw(DequeEmpty);/*thefirst element
            const  Elem&  back()  const  throw(DequeEmpty);/*the
      last element
            void insertFront(const Elem& e); // insert new first
            element
            void  insertBack(const  Elem&  e);  //  insert  new  last
            element
            void   removeFront()   throw(DequeEmpty);   //   remove
            first element
            void removeBack() throw(DequeEmpty); // remove last
            element
      private: // member data
            DLinkedList D; // linked list of elements
            int n; // number of elements
};
void LinkedDeque::insertFront(const Elem& e)
{
      D.addFront(e);
      n++;
} // insert new last element
void LinkedDeque::insertBack(const Elem& e)
{
      D.addBack(e);
      n++;
} // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty)
{
      if   (empty())throw   DequeEmpty("removeFront   of   empty
deque");
      D.removeFront();
      n--;
} // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty)
{
      if (empty())throw DequeEmpty("removeBack of empty deque");
      D.removeBack();
}
```

3.8. Circular Queue

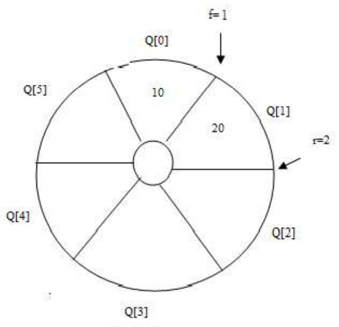
In circular queue, the insertion of a new element is performed at the very first location of the queue. If the last location of the queue is full in which the first element comes just after the last element. Insert the following elements into the circular Queue 10,20,30,40,50,60. Queue Size is 6.



Empty Circular Queue



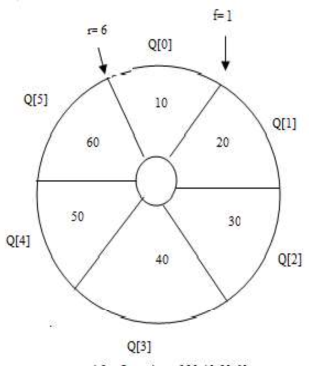
Insert(10)



Insert(20)

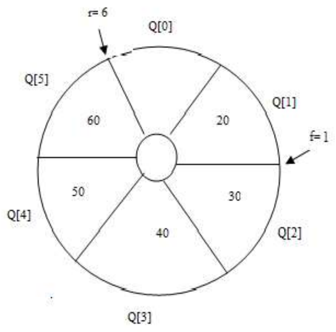
Advantages

To overcome the problem of unutilized space in linear queues when it is implemented using an array. Insert the following element into the circular queue. Elements are, 10,20,30,40,50,60.



After Insertion of 30,40,50,60

```
#define size 6
struct cqueue
{
            int CQ[size];
            int rear,front;
};
void insertion(int X,CQUEUE cq)
{
            int item;
            if(front=(rear+1)%size)
               error("Queue is full");
            else
            {
                        if(front==-1)
                                    front=rear=-1;
                        else
                                    rear=(rear+1)%size;
                                    CQ[rear]=item;
            }
}
void delete()
{
            int item;
            if(front==-1)
             error("Queue is Empty");
            else
            {
```



Delete(10)

```
                item=CQ[front];
                if(front==rear)
                front=rear=-1
                else
                        front=(front+1)%size;
        }
}
```

3.8. Evaluating arithmetic expressions

The stack is used to convert the infix expression to postfix expression.

Infix

In Infix notation, the arithmetic operator appears between the two operands to which it is being applied. For example: A / B + C

Postfix

The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation. ((A/B) + C) For example: AB / C +

Algorithm

1. Read the infix expression one character at a time until we reach the end of input
    a. If the character is an operand, place it on to the output.
    b. If the character is a left parenthesis, push it onto the stack.
    c. If the character is a right parenthesis, pop all the operators from the stack until we encounters a left parenthesis, discard both the parenthesis in the output.
    d. If the character is an operator, then pop the entries from the stack until we find an entry of lower priority (never pop („„). The push the operator into the stack.
2. Pop the stack until it is empty, writing symbols onto the output.

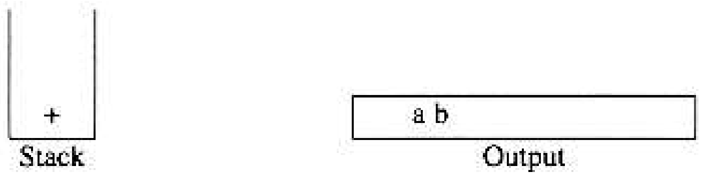
Operator	Precedence in stack	Precedence in Expression
^	3	4
*, /	2	2
+, -	1	1
(	0	4

Example

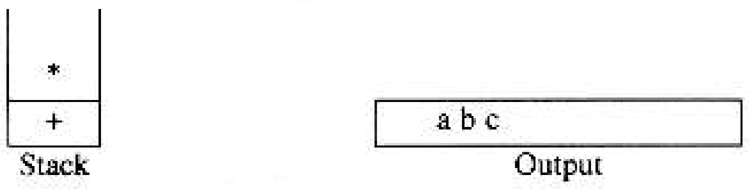
Suppose we want to convert the infix expression

a+b*c+(d*e+f)*g into postfix expression.

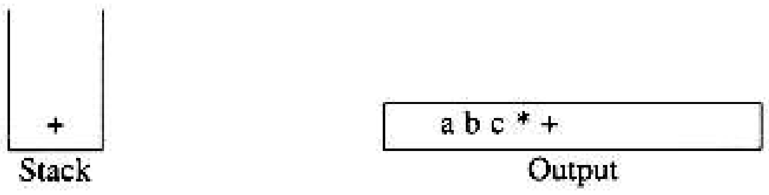
First, the symbol *a* is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next *b* is read and passed through to the output. The state is as follows:



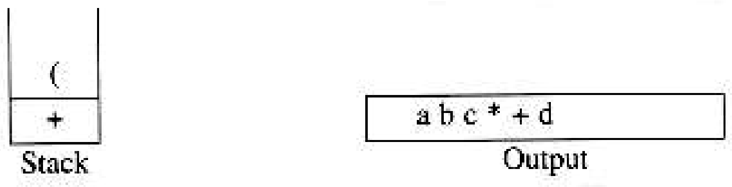
Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, *c* is read and output. Thus far, we have



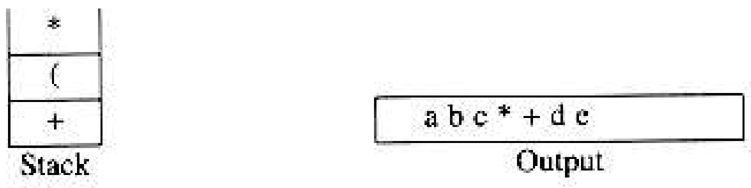
The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output, pop the other '+', which is not of *lower* but equal priority, on the stack, and then push the '+'.



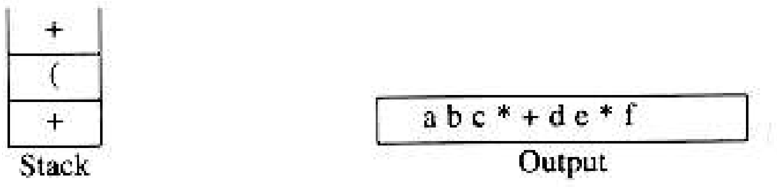
The next symbol read is an '(', which, being of highest precedence, is placed on the stack. Then *d* is read and output.



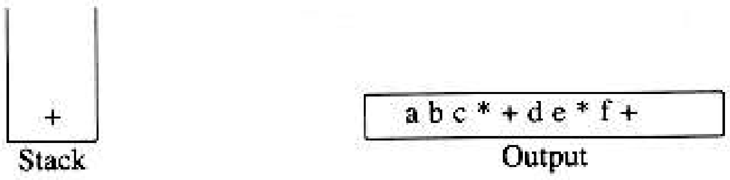
We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, *e* is read and output.



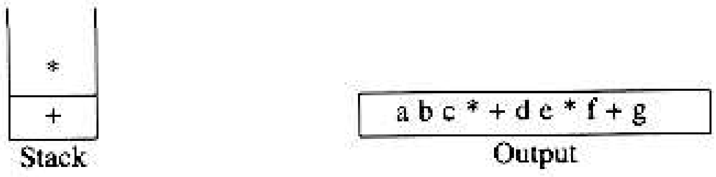
The next symbol read is a '+'. We pop and output '*' and then we push '+'. Then we read and output



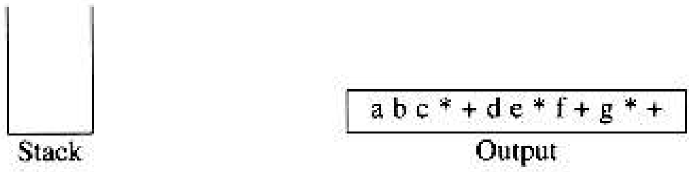
Now we read a ')', so the stack is emptied back to the '('. We output a '+'.



We read a '*' next; it is pushed onto the stack. Then *g* is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



```
class EXP
{
        private:
                char post[40];
        int top,st[20];
    public:
        void postfix(char info[40])
        {
            int i,j=0;
            for(i=0;inf[i]!='\o';i++)
            {
                switch(inf[i])
                {
                    case '+':while(st[top]>=1)
                        post[j++]=pop();
                        push(1);
                        break;
                    case '-':while(st[top]>=1)
                        post[j++]=pop();
                        push(2);
                        break;
                    case '*':while(st[top]>=3)
                        post[j++]=pop();
                        push(3);
                        break;
                    case '/':while(st[top]>=3)
                        post[j++]=pop();
                        push(4);
                        break;
                    case '^':while(st[top]>=4)
                        post[j++]=pop();
                        push(5);
                        break;
                    case '(':push(0);
                        break;
                    case '^':while(st[top]!=0)
```

```
                                        post[j++]=pop();
                                        top--;
                                        break;
                                default:post[j++]=inf[i];
                        }
`               }
                while(top>0)
                        post[j++]=pop();
        }
        void push(int ele)
        {
                top++;
                st[top]=ele;
        }
        char pop()
        {
                int el;
                char e;
                el=st[top];
                top--;
                switch(el)
                {
                        case 1:e='+';break;
                        case 2:e='-';break;
                        case 3:e='*';break;
                        case 4:e='/';break;
                        case 5:e='^';break;
                }
        }
        EXP()
        {
                top=0;      }};
```