UNIT- II
STACKS

The data structures seen so far, allows insertion and deletion of elements at any place. But sometimes it is required to permit the addition and deletion of elements only at one end that is either at the beginning or at the end.

Stacks: *A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop. Stack is also called as Last-In-First-Out (LIFO) list.*

Operations on Stack:

There are two possible operations done on a stack. They are pop and push operation.

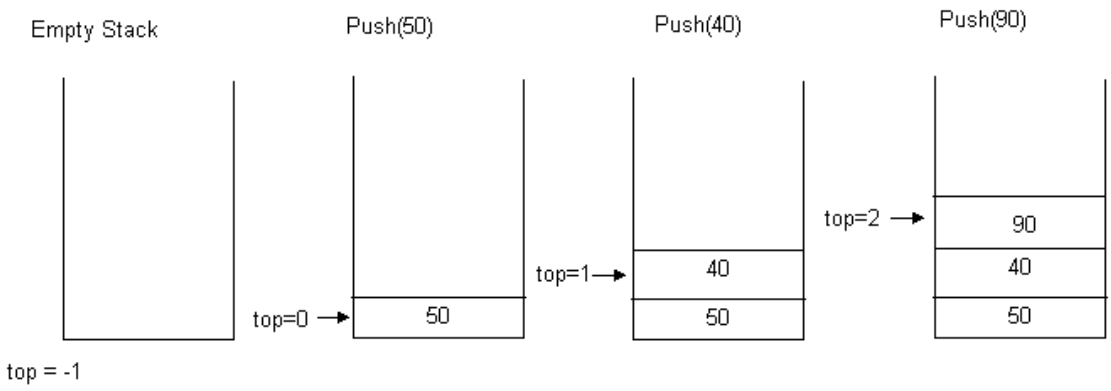
- ***Push:*** *Allows adding an element at the top of the stack.*
- ***Pop:*** *Allows removing an element from the top of the stack.*

The Stack can be implemented using both arrays and linked lists. When dynamic memory allocation is preferred we go for linked lists to implement the stacks.

ARRAY IMPLEMENTATION OF THE STACK

Push operation:

If the elements are added continuously to the stack using the push operation then the stack grows at one end. Initially when the stack is empty the top = -1. The top is a variable which indicates the position of the topmost element in the stack.

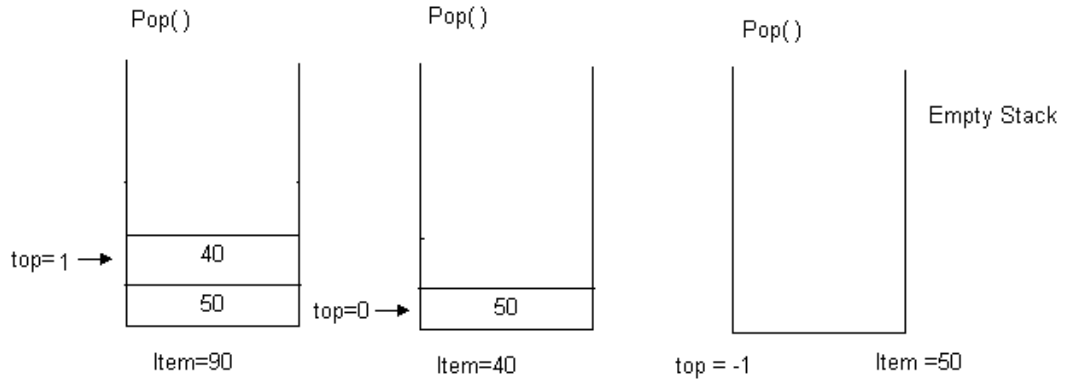


```
PUSH(x)

If top = MAX – 1
Then
      Print "Stack is full"
      Return
Else
       Top = top + 1
       A[top] = x
End if
End PUSH( )
```

Pop operation:

On deletion of elements the stack shrinks at the same end, as the elements at the top get removed.




```
POP( )

If top = -1
Then
      Print "Stack is empty"
      Return
Else
       Item = A[top]
       A[Top] = 0
       Top = top – 1
       Return item
End if
End POP( )
```

If arrays are used for implementing the stacks, it would be very easy to manage the stacks. However, the problem with an array is that we are required to declare the size of the array before using it in a program. This means the size of the stack should be fixed. We can declare the array with a maximum size large enough to manage a stack. As result, the stack can grow or shrink within the space reserved for it. The following program implements the stack using array.

Program:

```
// Stack and various operations on it

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

const int MAX=20;
class stack
{
private:
      int a[MAX];
      int top;
public:
      stack();
      void push(int x);
      int pop();
      void display();

};

stack::stack()
{
      top=-1;
}

void stack::push(int x)
{
      if (top==MAX-1)
      {
            cout<<"\nStack is full!";
            return;
      }
      else
      {
            top++;
            a[top]=x;
      }
}

int stack::pop()
{
      if (top==-1)
      {
            cout<<"\nStack is empty!";
            return NULL;
      }
      else
      {
            int item=a[top];
            top--;
            return item;
```

```
        }
}

void stack::display()
{
        int temp=top;
        while (temp!=-1)
                cout<<"\n"<<a[temp--];
}
void main()
{
        clrscr();
        stack s;
        int n;
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
        s.display();
        n=s.pop();
        cout<<"\nPopped item:"<<n;
        n=s.pop();
        cout<<"\nPopped item:"<<n;
        s.display();
        getch();
}
```

Output:

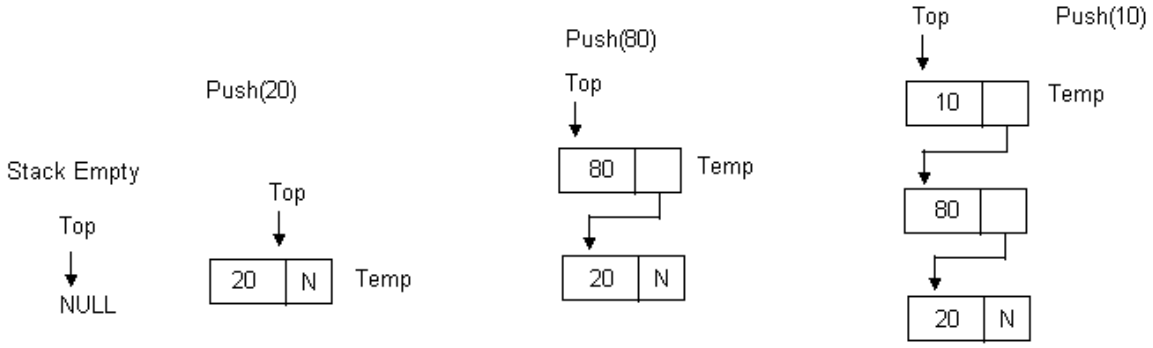
```
40
30
20
10
Popped item:40
Popped item:30
20
10
```

LINKED LIST IMPLEMENTATION OF STACK

Initially, when the stack is empty, top points to NULL. When an element is added using the push operation, top is made to point to the latest element whichever is added.

Push operation:

Create a temporary node and store the value of x in the data part of the node. Now make link part of temp point to Top and then top point to Temp. That will make the new node as the topmost element in the stack.

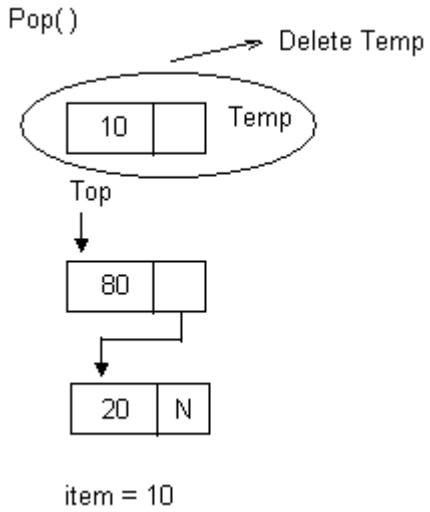



```
PUSH(x)

Info(temp) = x
Link(temp) = top
Top = temp
End PUSH( )
```

Pop operation

The data in the topmost node of the stack is first stored in a variable called item. Then a temporary pointer is created to point to top. The top is now safely moved to the next node below it in the stack. Temp node is deleted and the item is returned.




```
POP( )

If Top = NULL
Then
      Print "Stack is empty"
      Return
Else
      Item = info(top)
      Temp = top
```

```
        Top = link(top)
        Delete temp
        Return item
End if
End POP( )
```

The following program implements the stack using linked lists.

Program:

```
// Stack implemented using linked list

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

class stack
{
private:
        struct node
        {
                int data;
                node *link;
        };
        node *top;
public:
        stack();
        ~stack();
        void push(int x);
        int pop();
        void display();
};

stack::stack()
{
        top=NULL;
}

stack::~stack()
{
        node *temp;
        while (top!=NULL)
        {
                temp=top->link;
                delete top;
                top=temp;
        }
}

void stack::push(int x)
{
        node *temp;
```

```
      temp=new node;
      temp->data=x;
      temp->link=top;
      top=temp;
}

int stack::pop()
{
      if (top==NULL)
      {
            cout<<"\nStack is empty!";
            return NULL;
      }
      node *temp=top;
      int item=temp->data;
      top=temp->link;
      delete temp;
      return item;
}

void stack::display()
{
      node *temp=top;
      while (temp!=NULL)
      {
            cout<<"\n"<<temp->data;
            temp=temp->link;
      }
}

void main()
{
      clrscr();
      stack s;
      int n;
      s.push(10);
      s.push(20);
      s.push(30);
      s.push(40);
      s.display();
      n=s.pop();
      cout<<"\nPopped item:"<<n;
      n=s.pop();
      cout<<"\nPopped item:"<<n;
      s.display();
      getch();
}
```

Output:

40

```
30
20
10
Popped item:40
Popped item:30
20
10
```

APPLICATION OF STACKS

Conversion of Infix Expression to Postfix Expression

The stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. The operands can be numeric values or numeric variables. The operators used in an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentiation.

The arithmetic expression expressed in its normal form is said to be Infix notation, as shown:

A + B

The above expression in prefix form would be represented as follows:

+ AB

The same expression in postfix form would be represented as follows:

AB +

Hence the given expression in infix form is first converted to postfix form and then evaluated to get the results.

The function to convert an expression from infix to postfix consists following steps:

1. Every character of the expression string is scanned in a while loop until the end of the expression is reached.
2. Following steps are performed depending on the type of character scanned.
    (a) If the character scanned happens to be a space then that character is skipped.
    (b) If the character scanned is a digit or an alphabet, it is added to the target string pointed to by t.
    (c) If the character scanned is a closing parenthesis then it is added to the stack by calling push( ) function.
    (d) If the character scanned happens to be an operator, then firstly, the topmost element from the stack is retrieved. Through a while loop, the priorities of the character scanned

and the character popped 'opr' are compared. Then following steps are performed as per the precedence rule.

i. If 'opr' has higher or same priority as the character scanned, then opr is added to the target string.

ii. If opr has lower precedence than the character scanned, then the loop is terminated. Opr is pushed back to the stack. Then, the character scanned is also added to the stack.

(e) If the character scanned happens to be an opening parenthesis, then the operators present in the stack are retrieved through a loop. The loop continues till it does not encounter a closing parenthesis. The operators popped, are added to the target string pointed to by t.

2. Now the string pointed by t is the required postfix expression.

Program:

```
// Program to convert an Infix form to Postfix form

#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>

const int MAX=50;

class infix
{
      private:

            char target[MAX], stack[MAX];
            char *s, *t;
            int top;

      public:

            infix();
            void push(char c);
            char pop();
            void convert(char *str);
            int priority (char c);
            void show();
};

infix::infix()
{
      top=-1;
      strcpy(target,"");
      strcpy(stack,"");
      t=target;
      s="";
```

```
}

void infix::push(char c)
{
	if (top==MAX-1)
		cout<<"\nStack is full\n!";
	else
	{
		top++;
		stack[top]=c;
	}
}


char infix::pop()
{
	if (top==- 1)
	{
		cout<<"\nStack is empty\n";
		return -1;
	}
	else
	{
		char item=stack[top];
		top--;
		return item;
	}
}
void infix::convert(char *str)
{
	s=str;
	while(*s!='\0')
	{
		if (*s==' '||*s=='\t')
		{
			s++;
			continue;
		}
		if (isdigit(*s) || isalpha(*s))
		{
			while(isdigit(*s) || isalpha(*s))
			{
				*t=*s;
				s++;
				t++;
			}
		}
		if (*s=='(')
		{
			push(*s);
			s++;
```

```
            }
            char opr;
            if (*s=='*'||*s=='+'||*s=='/'||*s=='%'||*s=='- '||*s=='^')
            {
                  if (top!=- 1)
                  {
                        opr=pop();
                        while (priority(opr)>=priority(*s))
                        {
                              *t=opr;
                              t++;
                              opr=pop();
                        }
                        push(opr);
                        push(*s);
                  }
                  else
                        push (*s);
                  s++;
            }

            if (*s==')')
            {
                  opr=pop();
                  while ((opr)!='(')
                  {
                        *t=opr;
                        t++;
                        opr=pop();
                  }
                  s++;
            }
      }

      while (top!=- 1)
      {
            char opr=pop();
            *t=opr;
            t++;
      }

      *t='\0';
}


int infix::priority(char c)
{
      if (c=='^')
            return 3;
      if (c=='*'||c=='/'||c=='%')
            return 2;
```

```
        else
        {
                if (c=='+'||c=='-')
                        return 1;
                else
                        return 0;
        }
}

void infix::show()
{
   cout<<target;
}

void main()
{
        clrscr();
        char expr[MAX], *res[MAX];
        infix q;

        cout<<"\nEnter an expression in infix form: ";
        cin>>expr;
        q.convert(expr);


        cout<<"\nThe postfix expression is: ";
        q.show();
        getch();
}
```

Output:

Enter an expression in infix form: 5^2-5

Stack is empty

The postfix expression is: 52^5-

Evaluation of Expression entered in postfix form

The program takes the input expression in postfix form. This expression is scanned character by character. If the character scanned is an operand, then first it is converted to a digit form and then it is pushed onto the stack. If the character scanned is a blank space, then it is skipped. If the character scanned is an operator, then the top two elements from the stack are retrieved. An arithmetic operation is performed between the two operands. The type of arithmetic operation depends on the operator scanned from the string s. The result is then pushed back onto the stack. These steps are repeated as long as the string s is not exhausted. Finally the value in the stack is the required result and is shown to the user.

Program:

```
// Program to evaluate an expression entered in postfix form

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>

const int MAX=50;

class postfix
{
      private:

            int stack[MAX];
            int top, n;
            char *s;

      public:

            postfix();
            void push(int item);
            int pop();
            void calculate(char *str);
            void show();
};

postfix::postfix()
{
      top=-1;
}

void postfix::push(int item)
{
      if (top==MAX-1)
            cout<<endl<<"Stack is full";
      else
      {
            top++;
            stack[top]=item;
      }
}

int postfix::pop()
{
      if (top==-1)
      {
            cout<<endl<<"Stack is empty";
```

```
                return NULL;
        }
        int data=stack[top];
        top--;
        return data;
}
void postfix::calculate(char *str)
{
        s=str;
        int n1, n2, n3;
        while (*s)
        {
                if (*s==' '||*s=='\t')
                {
                        s++;
                        continue;
                }
                if (isdigit(*s))
                {
                        n=*s-'0';
                        push(n);
                }
                else
                {
                        n1=pop();
                        n2=pop();
                        switch(*s)
                        {
                                case '+':
                                        n3=n2+n1;
                                        break;
                                case '-':
                                        n3=n2-n1;
                                        break;
                                case '/':
                                        n3=n2/n1;
                                        break;
                                case '*':
                                        n3=n2*n1;
                                        break;
                                case '%':
                                        n3=n2%n1;
                                        break;
                                case '^':
                                        n3=pow(n2, n1);
                                        break;
                                default:
                                        cout<<"Unknown operator";
                                        exit(1);
                        }
                        push(n3);
```

```
            }
            s++;
      }
}
void postfix::show()
{
      n=pop();
      cout<<"Result is: "<<n;
}
void main()
{
      clrscr();
      char expr[MAX];
      cout << "\nEnter postfix expression to be evaluated :";
      cin>>expr;
      postfix q ;
      q.calculate(expr);
      q.show();
      getch();
}
```

Output:

Enter postfix expression to be evaluated : 53^5-
Result is: 120

QUEUE

Queue: *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 deletion or insertion of elements can take place only at the front or rear end of the list respectively.*

The first element that gets added into the queue is the first one to get removed from the list. Hence, queue is also referred to as First-In-First-Out list (FIFO). Queues can be represented using both arrays as well as linked lists.

ARRAY IMPLEMENTATION OF QUEUE

If queue is implemented using arrays, the size of the array should be fixed maximum allowing the queue to expand or shrink.

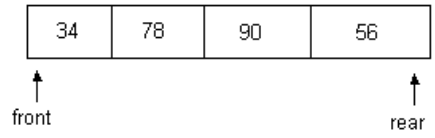
Operations on a Queue

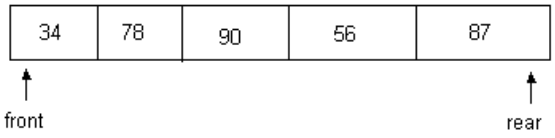
There are two common operations one in a queue. They are addition of an element to the queue and deletion of an element from the queue. Two variables front and rear are used to point to the ends of the queue. The front points to the front end of the queue where deletion takes place and rear points to the rear end of the queue, where the

addition of elements takes place. Initially, when the queue is full, the front and rear is equal to -1.

Add(x)

An element can be added to the queue only at the rear end of the queue. Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the rear side.



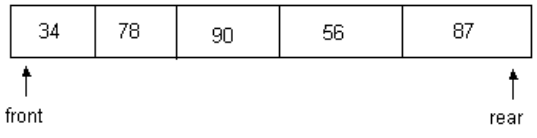


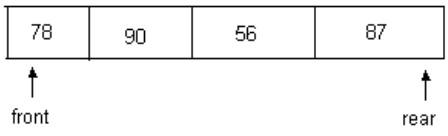

```
ADDQ(x)

If rear = MAX – 1
Then
      Print "Queue is full"
      Return
Else
      Rear = rear + 1
      A[rear] = x
      If front = -1
      Then
            Front = 0
      End if
End if
End ADDQ( )
```

Del( )

The del( ) operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.






```
DELQ( )

If front = -1
Then
      Print "Queue is Empty"
      Return
```

```
Else
      Item = A[front]
      A[front] = 0
      If front = rear
      Then
            Front = rear = -1
      Else
            Front = front + 1
      End if
      Return item
End if
End DELQ( )
```

Program:

```
// Queues and various operations on it – Using arrays

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

const int MAX=10;
class queue
{
private:
      int a[MAX], front, rear;
public:
      queue();
      void addq(int x);
      int delq();
      void display();
};

queue::queue()
{
      front=rear=- 1;
}

void queue::addq(int x)
{
      if (rear==MAX-1)
      {
            cout<<"Queue is full!";
            return;
      }
      rear++;
      a[rear]=x;
      if (front==- 1)
            front=0;
}

int queue::delq()
```

```
{
      if (front==- 1)
      {
            cout<<"Queue  is empty!";
            return NULL;
      }
      int item=a[front];
      a[front]=0;
      if (front==rear)
            front=rear=- 1;
      else
            front++;
      return item;
}

void queue::display()
{
      if (front==- 1)
            return;
      for (int i=front; i<=rear; i++)
            cout<<a[i]<<"\t";
}

void main()
{
      clrscr();
      queue q;
      q.addq(50);
      q.addq(40);
      q.addq(90);
      q.display();
      cout<<endl;
      int i=q.delq();
      cout<<endl;
      cout<<i<<" deleted!";
      cout<<endl;
      q.display();
      i=q.delq();
      cout<<endl;
      cout<<i<<" deleted!";
      cout<<endl;
      i=q.delq();
      cout<<i<<" deleted!";
      cout<<endl;
      i=q.delq();

      getch();
}
```

Output:

```
50     40     90

50 deleted!
40     90
40 deleted!
90 deleted!
Queue is empty!
```