DATA STRUCTURES THROUGH C++

UNIT III

RUPA DEVI T

UNIT- III

- **Stacks:** Definition, ADT, standard stack operations- array and linked list implementations, applications-infix to postfix conversion, postfix expression evaluation, parsing parenthesis, reverse of a string using stack, history of a browser, etc
- **Queues:** Definition, ADT**,** standard queue operations - array and linked implementations, Circular queues - Insertion and deletion operations. Real Time Applications of Queue: Operating Systems and Task Scheduling, Networking and Message Queues, etc

STACK

STACK ADT

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

•A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack.

•A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO)-an element inserted last will be removed first.

•Stack uses a single index or pointer to keep track of the information in the stack.

•The basic operations associated with the stack are: a) push(insert) an item onto the stack. b) pop(remove) an item from the stack.

The general terminology associated with the stack is as follows:

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be **pushed** on the stack.

When an object is removed from the stack, it is said to be **popped** off the stack.

Two additional terms almost always used with stacks are **overflow-** which occurs when we try to push more information on a stack that it can hold, and **underflow-**which occurs when we try to pop an item off a stack which is empty.

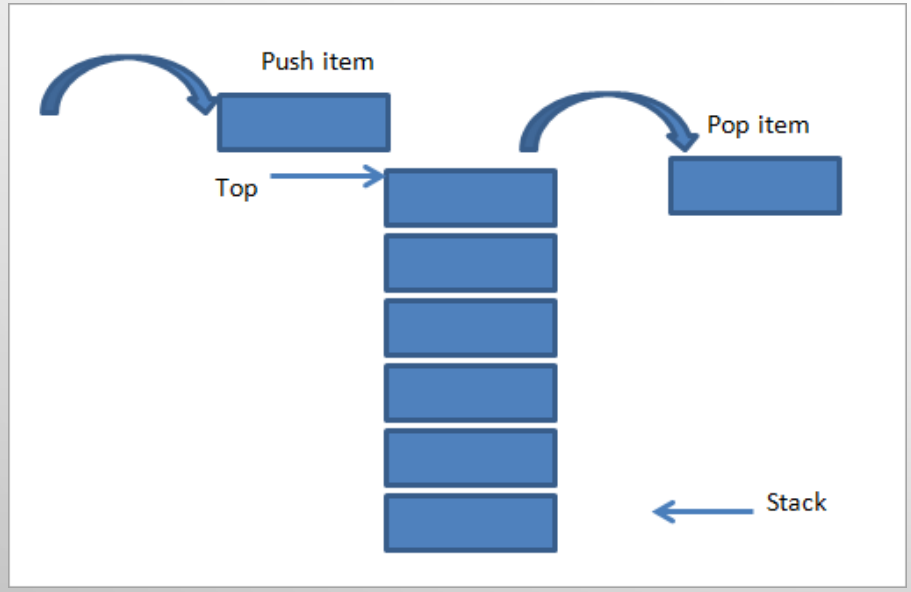
Pushing items onto the stack: Assume that the array elements begin at 0
(because the array subscript starts from 0) and the maximum elements that can be placed in stack is max.

The stack pointer, top, is considered to be pointing to the top element of the stack.

A push operation thus involves **adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack.**

Initially the top is initialized to -1.

- A stack can be implemented by means of Array, Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.



```
//Code to push an element on to
stack;
void stack::push()
{
 if(top==max-1)
 cout<<"Stack Overflow...\n";
else
{
cout<<"Enter an element to be
pushed:";
top++;
cin>>data;
stk[top]=data;
 cout<<"Pushed Successfully....\n";
}
 }
```

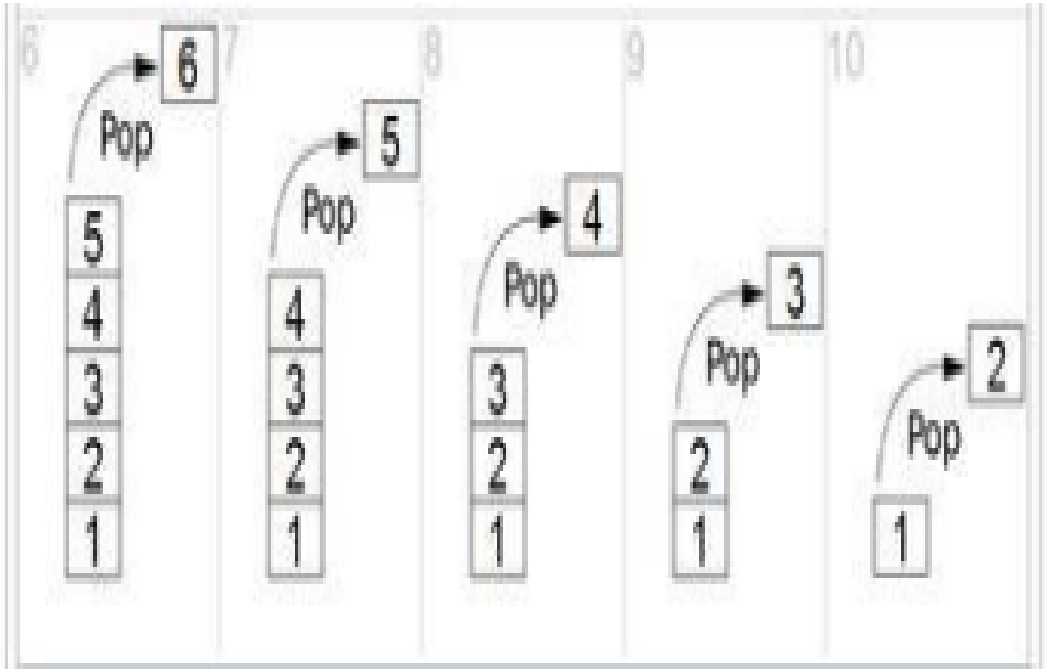
Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.

Popping an element from stack:

To remove an item, **first extract the data from top position in the stack** and then decrement the stack pointer, top.



Pop Operation

Accessing the content if required while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.

A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses data element at which top is
- Step 4 – Decreases the value of top by 1.

```
//Code to pop an element from a stack;

void stack::pop()
{
if(top==-1)
cout<<“stack underflow”;
else
{
cout<<“the elelment popped is:”<<stk[top];

top–;

}
}
```

4.a.(i). Write a program that implement stack (its operations) using Arrays

```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;
void push(int val) {
  if(top>=n-1)
  cout<<"Stack Overflow"<<endl;
  else {
    top++;
    stack[top]=val;
  }
}
void pop() {
  if(top<=-1)
  cout<<"Stack Underflow"<<endl;
  else {
    cout<<"The popped element is "<<
stack[top] <<endl;
    top--;
  }
}
void display() {
  if(top>=0) {
    cout<<"Stack elements are:";
    for(int i=top; i>=0; i--)
    cout<<stack[i]<<" ";
cout<<endl;
  } else
  cout<<"Stack is empty";
}
int main() {
  int ch, val;
  cout<<"1) Push in stack"<<endl;
  cout<<"2) Pop from stack"<<endl;
  cout<<"3) Display stack"<<endl;
  cout<<"4) Exit"<<endl;
  do {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
 switch(ch) {
      case 1: {
        cout<<"Enter value to be pushed
:"<<endl;
        cin>>val;
        push(val);
        break;
      }
case 2: {
        pop();
        break;
      }
      case 3: {
        display();
break;
      }
      case 4: {
        cout<<"Exit"<<endl;
        break;
      }
      default: {
        cout<<"Invalid Choice"<<endl;
      }
```

Applications of Stack:

1. Stacks are used in conversion of infix to postfix expression.
2. Stacks are also used in evaluation of postfix expression.
3. Stacks are used to implement recursive procedures.
4. Stacks are used in compilers.
5. Reverse String

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

Expression	Example	Note
Infix	a + b	Operator Between Operands
Prefix	+ a b	Operator before Operands
Postfix	a b +	Operator after Operands

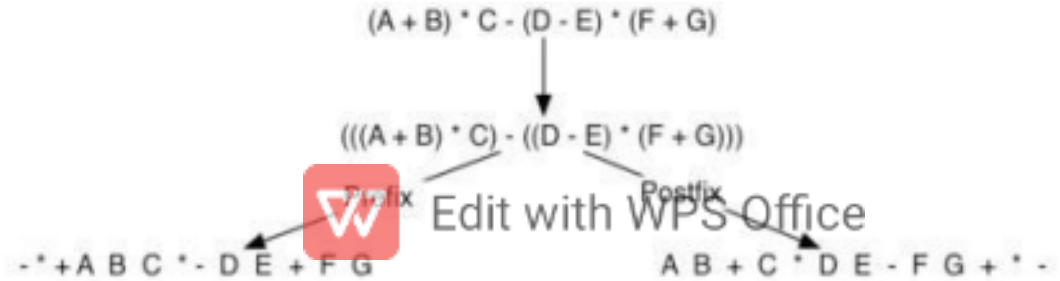
Conversion of Infix Expressions to Prefix and Postfix

Conversion of Infix Expressions to Prefix and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

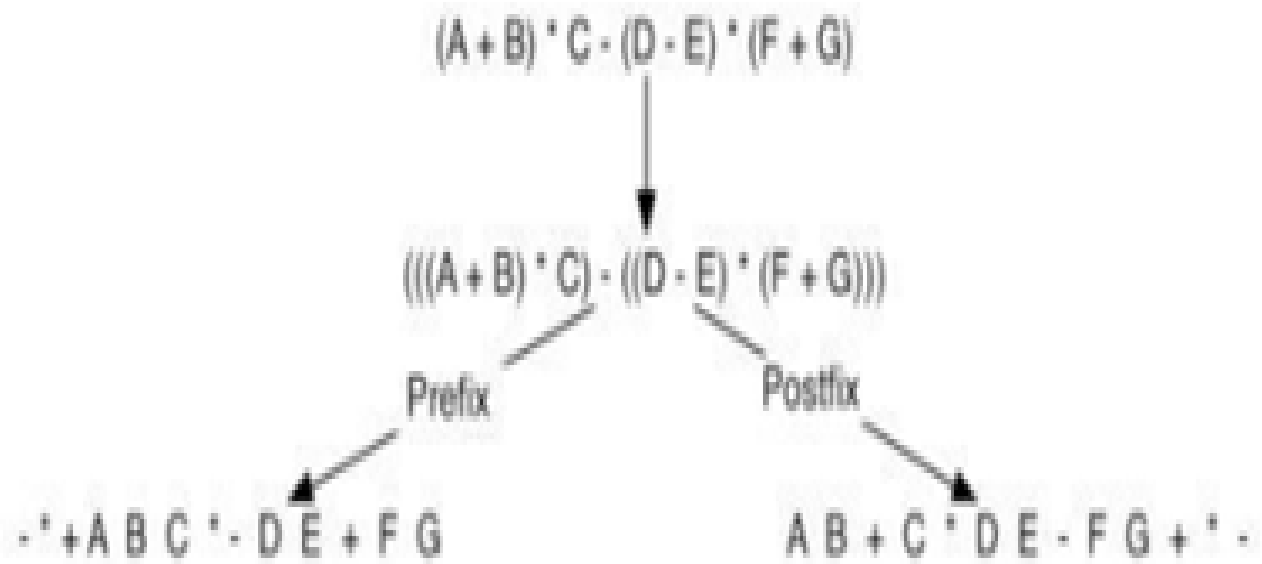
Convert following infix expression to prefix and postfix

(A + B) * C - (D - E) * (F + G)



Convert following infix expression to prefix and postfix

(A + B) * C - (D - E) * (F + G)



INFIX TO POSTFIX CONVERSION USING STACKS

Basic:-

Infix Expression: The operator is in between the two operands

Example: A + B is known as infix expression.

Postfix Expression: The operator is after the two operands

Example: BD + is known as postfix expression.

Steps needed for infix to postfix conversion using stack in C++:-

1. First Start scanning the expression from left to right
2. If the scanned character is an operand, output it, i.e. print it
3. Else
    - If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'('), then push operator in the stack
    - Else, Pop all the operators, that have higher or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Now, we should repeat steps 2 – 6 until the whole infix i.e. whole characters are scanned.
7. Print output
8. Do the pop and output (print) until the stack is not empty

Infix to Postfix

Expression : A + B * C / D - F + A ^ E

Scanned Symbol	Stack	Output	Reason	
A		A	Step 2	
+	+	A	Step 3.1	
B	+	AB	Step 2	
*	+*	AB	Step 3.1	
C	+*	ABC	Step 2	
/	+/	ABC*	Step 3.2	/ prec is equal to * So not higher, thus going to Step 3.2
D	+/	ABC*D	Step 2	
-	-	ABC*D/+	Step 3.2	/ will be popped, added to o/p & then + popped & added to o/p, then - will be pushed
F	-	ABC*D/+F	Step 2	
+	+	ABC*D/+F-	Step 3.2	- will be popped, added to the o/p and then + added to the stack
A	+	ABC*D/+F-A	Step 2	
^	+^	ABC*D/+F-A	Step 2	
E	+^	ABC*D/+F-AE	Step 2	
(Empty)		ABC*D/+F-AE^+	Step 8	

Infix to Postfix Conversion

Infix Expression: (A/(B-C)*D+E)

Symbol Scanned	Stack	Output
(	(	-
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
+	(+	ABC-/D*
E	(+	ABC-/D*E
)	Empty	ABC-/D*E+

Postfix Expression: ABC-/D*E+

Benefits of Postfix expression over infix expression

- In postfix any formula can be expressed without parenthesis.
- It is very useful for evaluating formulas on computers with stacks.
- Infix operators have precedence

4. b. Write a program to implement Infix to Postfix Conversion using Stacks

```
#include <iostream>
using namespace std;

class Stack {
    int top;
    int MAX;
    int* a;

public:

    Stack(int size){
        top = -1;
        MAX = size;
        a = new int [MAX];
    }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    bool isFull();
};
```

```
bool Stack::isEmpty(){
    return (top < 0);
}
bool Stack::isFull(){
    return (top == MAX - 1);

}

int Stack::peek(){
    return a[top];

}
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        top++;
        a[top] = x;
        return true;
    }
}
```

```
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return INT_MIN;
    }
    else {
        int x = a[top];
        top--; return x;
    }
}
int priority (char alpha)
{
    if(alpha == '+' || alpha =='-')
        return 1;
    if(alpha == '*' || alpha =='/')
        return 2;
    if(alpha == '^')
        return 3;

    return 0;
}
```

```
string convert(string infix)
{
   int i = 0;
   string postfix = "";

   Stack s(20);
   while(infix[i]!='\0')
   {
   // if operand add to the postfix expression
     if(infix[i]>='a' && infix[i]<='z'||
infix[i]>='A'&& infix[i]<='Z')
     {
        postfix += infix[i];
         i++;
     }
   // if opening bracket then push the stack
     else if(infix[i]=='(') {
        s.push(infix[i]);
        i++;

     }
   // if closing bracket encounted then
keep popping from stack until
   // closing a pair opening bracket is not
encountered
   else if(infix[i]==')') {
      while(s.peek()!='(')
         postfix += s.pop();
      s.pop();
      i++;
      }
   else {
      while (!s.isEmpty() && priority(infix[i])
<= priority(s.peek())){
         postfix += s.pop();
      }
      s.push(infix[i]);
      i++;
   }
   }
while(!s.isEmpty()){
      postfix += s.pop();
      }
   cout << "Postfix is : " << postfix;
   //it will print postfix conversion
   return postfix;

}

int main() {
      cout<<"Enter Infix Expression:
\n";
   string infix;
      cin>>infix;
   string postfix;
   postfix = convert(infix);
   return 0;
}
```

EVALUATION OF POSTFIX EXPRESSION

1. Start reading the expression from left to right.
2. If the element is an operand then, push it in the stack.
3. If the element is an operator, then pop two elements from the stack and use the operator on them.
4. Push the result of the operation back into the stack after calculation.
5. Keep repeating the above steps until the end of the expression is reached.
6. The final result will be now left in the stack, display the same.

```
#include<iostream>
#include<stack>
#include<math.h>
using namespace std;
// The function calculate_Postfix returns the final answer of the expression
after calculation
int calculate_Postfix(string post_exp)
{
stack <int> stack;
int len = post_exp.length();
// loop to iterate through the expression
for (int i = 0; i < len; i++)
{
// if the character is an operand we push it in the stack
// we have considered single digits only here
if ( post_exp[i] >= '0' && post_exp[i] <= '9')
{
stack.push( post_exp[i] - '0');
}
// if the character is an operator we enter else block
```

```
else
{
// we pop the top two elements from the
stack and save them in two integers
int a = stack.top();
stack.pop();
int b = stack.top();
stack.pop();
//performing the operation on the operands
switch (post_exp[i])
{
case '+': // addition
stack.push(b + a);
break;
case '-': // subtraction
stack.push(b - a);
break;
case '*': // multiplication
stack.push(b * a);
break;
case '/': // division
stack.push(b / a);
break;
case '^': // exponent
stack.push(pow(b,a));
break;
}
}
}
//returning the calculated result
return stack.top();
}
//main function/ driver function
int main()
{
//we save the postfix expression to calculate in
//postfix_expression string
string postfix_expression;
cout<<"Enter postfix expression: ";
cin>>postfix_expression;
cout<<"The answer after calculating the postfix
expression is : ";
cout<<calculate_Postfix(postfix_expression);
return 0;
}
```

Output

The answer after calculating the postfix expression is: -4

The working of the above code is as:

- Push ‘5’ and ‘9’ in the stack.
- Pop ‘5’ and ‘9’ from the stack, add them and then push ‘14’ in the stack.
- Push ‘3’ and ‘3’ in the stack.
- Pop ‘3’ and ‘3’ from the stack, and push ‘27’ (3^3) in the stack.
- Push ‘4’ in the stack.
- Pop ‘4’ and ‘27’ from the stack, multiply them and then push ‘108’ in the stack.
- Push ‘6’ in the stack.
- Pop ‘6’ and ‘108’ from the stack, divide 108 by 6 and then push ‘18’ in the stack.
- Pop ‘18’ and ‘14’ from the stack, subtract 18 from 14 and then push ‘-4’ in the stack.
- Print -4 as the final answer.

Parsing Parenthesis

- Check for Balanced Bracket expression using Stack:
- The idea is to put all the opening brackets in the stack.
- Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature.
- If this holds then pop the stack and continue the iteration.
- In the end if the stack is empty, it means all brackets are balanced or well-formed.
- Otherwise, they are not balanced.

Initially :

Step 1:

Step 2:

Step 3:

Step 4:

Step 5:

- Follow the steps mentioned below to implement the idea:
- Declare a character stack (say **temp**).
- Now traverse the string exp.
  - If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.
  - If the current character is a closing bracket (**')' or '}' or ']'**) then pop from the stack and if the popped character is the matching starting bracket then fine.
  - Else brackets are **Not Balanced**.
- After complete traversal, if some starting brackets are left in the stack then the expression is **Not balanced**, else **Balanced**.

// C++ program to check for balanced brackets.

```
// #include <bits/stdc++.h>
#include <iostream>
#include<stack>
using namespace std;
// Function to check if brackets are balanced
bool areBracketsBalanced(string expr)
{
        // Declare a stack to hold the previous brackets.
        stack<char> temp;
        for (int i = 0; i < expr.length(); i++) {
                if (temp.empty()) {
                        // If the stack is empty
                        // just push the current bracket
                        temp.push(expr[i]);
                }
                else if ((temp.top() == '(' && expr[i] == ')')
                                || (temp.top() == '{' && expr[i] ==
'}')
                                || (temp.top() == '[' && expr[i] ==
']')) {
                        // If we found any complete pair of
bracket
                        // then pop
                        temp.pop();
                }
                else {
                        temp.push(expr[i]);
                }
        }
        if (temp.empty()) {
                // If stack is empty return true
                return true;
        }
        return false;
}
// Driver code
int main()
{
        string expr = "{()}[]";
        // Function call
        if
(areBracketsBalanced(expr)
)
                cout << "Balanced
";
        else
                cout << "Not
Balanced";
        return 0;
```

Reverse Of A String Using Stack

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
void reverseString(string &str) {
   stack<int> stk;
   for (int i = 0; i < str.length(); i++) stk.push(str[i]);
   for (int i = 0; i < str.length(); i++) {
      str[i] = stk.top();
      stk.pop();
   }
}
int main() {
   string str = "hello kmit";
   cout << "Original String - " << str << endl;
   reverseString(str);
   cout << "Reversed String - " << str;
}
```

The algorithm contains the following steps:

1.Create an empty stack.

2.Push all the characters of the string into the stack.

3.One by one, pop each character from the stack until the stack is empty.

4.Place the popped characters back into the string from the zeroth position.

- Time Complexity - O(N)
- Space Complexity - O(N)

History of a Browser

- Implementing browser history using a stack data structure is a common approach. Here's how it can be done:
- Start with an empty stack to represent the browser history.
- Whenever the user visits a new webpage, push the URL of that webpage onto the stack.
- If the user clicks the back button, pop the top URL from the stack. This represents going back to the previously visited webpage.
- If the user clicks the forward button, if there are URLs in the stack that were previously popped (indicating a back operation), push the popped URL back onto the stack. This represents going forward to a previously visited webpage.
- If the user visits a completely new webpage after using the back button, clear the forward history by emptying the stack.
- You can also include additional information in the stack nodes, such as the webpage title or timestamp, to provide more functionality and context to the browser history.
- By using a stack, the most recently visited URLs are stored at the top, allowing for efficient back and forward operations. When the back button is clicked, the most recently visited webpage is popped from the stack.
- When the forward button is clicked, if there are URLs in the stack, the previously popped URLs are pushed back onto the stack.
- Overall, using a stack data structure for browser history allows for easy navigation between visited web pages, providing a familiar browsing experience for users.

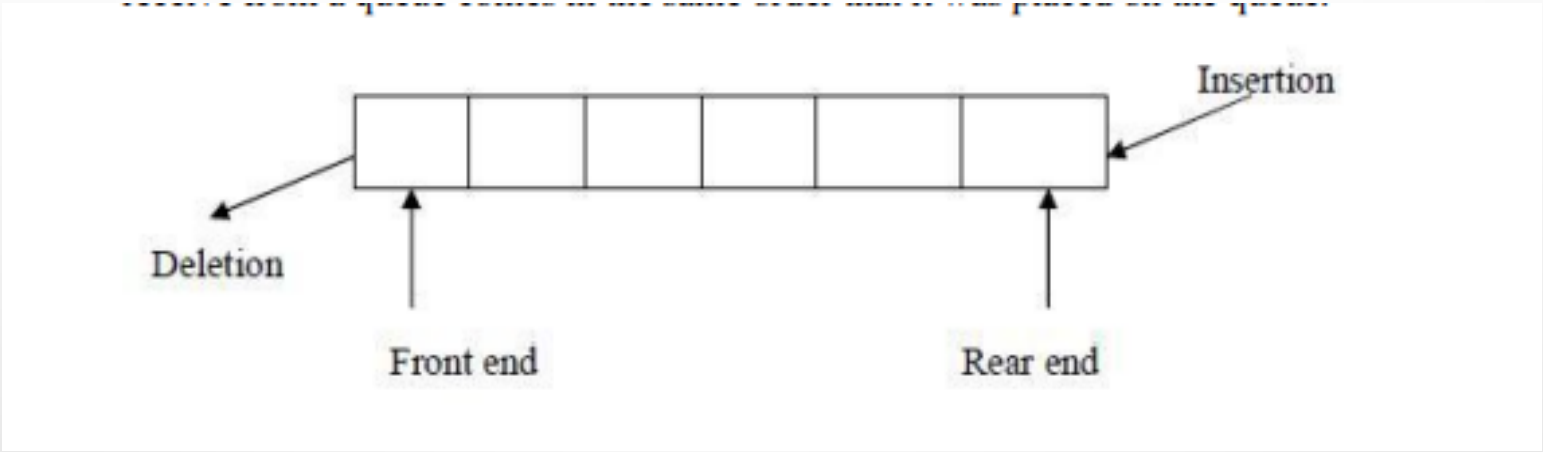
```
// C++ Implementation of the
approach
#include <bits/stdc++.h>
using namespace std;

class BrowserHistory {
public:
	stack<string> backStack,
forwardStack;

	// Constructor to initialize
object with
	// homepage
	BrowserHistory(string
homepage)
	{
	backStack.push(homepag
e);
	}
// Visit current url
	void visit(string url)
	{		while (!
forwardStack.empty()) {
			forwardStack.pop
();
		}
		string back(int steps)
	{
		while (backStack.size() > 1 &&
steps--) {
		forwardStack.push(backStack.to
p());
			backStack.pop();
		}
		return backStack.top();
	}
	// 'steps' move forward and return
	// current page
	string forward(int steps)
	{
		while (!forwardStack.empty() &&
steps--) {
			backStack.push(forwardSta
ck.top());
			forwardStack.pop();
		}
		return backStack.top();
	}};
// History
BrowserHistory obj(homepage);

string url = "google.com";
obj.visit(url);
url = "facebook.com";
obj.visit(url);
url = "youtube.com";
obj.visit(url);
cout << obj.back(1) << endl;
cout << obj.back(1) << endl;
cout << obj.forward(1) << endl;

obj.visit("linkedin.com");

cout << obj.forward(2) << endl;

cout << obj.back(2) << endl;

cout << obj.back(7) << endl;

return 0;
}
// } Driver Code Ends
```

QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end.

The key difference when compared stacks is that in a queue the information stored is processed First-in First-out or **FIFO-an element inserted first, will be removed first.**

In other words the information received from a queue comes in the same order that it was placed on the queue.



Representing a Queue:

One of the most common way to implement a queue is using array.

An easy way to do so is to def ne an array Queue, and two additional variables front and rear.

The rules for manipulating these variables are simple:

- Each time information is added to the queue, increment rear.
- Each time information is taken from the queue, increment front.
- Whenever front >rear or front=rear=-1 the queue is empty.

Array implementation of a Queue do have drawbacks:

- The maximum queue size has to be set at compile time, rather than at run time.
- Space can be wasted, if we do not use the full capacity of the array.

Operations on Queue: A queue have two basic operations: a)adding new item to the queue
b) removing items from queue.

The operation of adding new item on the queue occurs only at one end of the queue called the rear or back.

The operation of removing items of the queue occurs at the other end called the front.

For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize.

The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue.

Assume that initially the front and rear variables are initialized to -1. Like stacks, underf bw and overf bw conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

Applications of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

5.A.(i). Write a program that implement Queue (its operations) using Arrays

```
#include <iostream>
#define MAX_SIZE 3
using namespace std;
class Queue
{
private:
int myqueue[MAX_SIZE], front, rear;
public:
Queue()
{ front = -1;
rear = -1;
}

bool isFull(){
if(front == 0 && rear == MAX_SIZE - 1)
{ return true;
}
return false;
}
bool isEmpty(){
if(front == -1) return true;
else return false;
}

void enQueue(int value)
{
if(isFull()){
cout << endl<< "Queue is full!!";
}
else {
if(front == -1)
front = 0;
rear++;
myqueue[rear] = value;
cout << value << " ";
}
}
int deQueue()
{
int value;
if(isEmpty())
{
cout << "Queue is empty!!" <<
endl; return(-1);
}else
{
value = myqueue[front];

if(front >= rear){
//only one element in queue
front = -1;
rear = -1;
}
else {

front++;
}
cout << endl << "Deleted => " << value << "
from myqueue";
return(value);
}
}
```

```
/* Function to display elements of Queue */
void displayQueue()
{
int i;
if(isEmpty()) {
cout << endl << "Queue is Empty!!" << endl;
}
else {
        cout << endl << "Front = " << front;
        cout << endl << "Queue elements : ";
        for(i=front; i<=rear; i++)
cout << myqueue[i] << "\t";
cout << endl << "Rear = " << rear << endl;
}
}
};
```

```
int main()
{

Queue myq;
int ch,ele;
while(1)
{
cout<<"\n1.Enqueue(insertion) 2.Dequeue(deletion)
3.display 4. Exit \nEnter ur choice ";
cin>>ch; switch(ch){
case 1: cout<<"enter the element ";
cin>>ele;
myq.enQueue(ele); break;
case 2: myq.deQueue();
break;
case 3: myq.displayQueue(); break;
case 4: exit(0);
}
}
return 0;
}
```

Containership in C++

When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
———
———
};
Class class_name2
{
———
———
};
Class class_name3
{
class_name1 obj1;     // object of class_name1
class_name2 obj2;     // object of class_name2
———-
————
};
```

```
//Sample Program to demonstrate Containership
#include < iostream.h >
#include < conio.h >
#include < iomanip.h >
#include< stdio.h >
const int len=80;

class employee
{
private:
char name[len];
int number;
public:
void get_data()
{
cout << "\n Enter employee name: ";
cin >> name;
cout << "\n Enter employee number: ";
cin >> number;
}
void put_data()
{
cout << " \n\n Employee name: " << name;
cout << " \n\n Employee number: " << number;
}
};
class manager
{
private:
char dept[len];
int numemp;
employee emp;
public:
void get_data()
{
emp.get_data();
cout << " \n Enter department: ";
cin >> dept;
cout << "\n Enter number of employees: ";
cin >> numemp;
}
```

```
void put_data()
{
emp.put_data();
cout << " \n\n Department: " << dept;
cout << " \n\n Number of employees: " <<
numemp;
}
};
```

```
class scientist
{
private:
int pubs,year;
employee emp;
public:
void get_data()
{
emp.get_data();
cout << " \n Number of publications: ";
cin >> pubs;
cout << " \n Year of publication: ";
cin >> year;
}
void put_data()
{
emp.put_data();
cout << "\n\n Number of publications: " <<
pubs;
cout << "\n\n Year of publication: "<< year;
}
};
```

```
void main()
{
manager m1;
scientist s1;
int ch;
clrscr();
do
{
cout << "\n 1.manager\n 2.scientist\n";
cout << "\n Enter your choice: ";
cin >> ch;
switch(ch)
{
case 1:
cout << "\n Manager data:\n";
m1.get_data();
cout << "\n Manager data:\n";
m1.put_data();
break;
case 2:
cout << " \n Scientist data:\n";
s1.get_data();
cout << " \n Scientist data:\n";
s1.put_data();
break;
}
cout << "\n\n To continue Press 1 -> ";
cin >> ch;
}
while(ch==1);
getch();
}
```

Difference between Inheritance and Containership :

Containership: Containership is the phenomenon of using one or more classes within the def nition of other class. When a class contains the def nition of some other classes, it is referred to as composition, containment or aggregation. The data member of a new class is an object of some other class. Thus the other class is said to be composed of other classes and hence referred to as containership. Composition is often referred to as a "has-a" relationship because the objects of the composite class have objects of the composed class as members.

Inheritance: Inheritance is the phenomenon of deriving a new class from an old one. Inheritance supports code reusability. Additional features can be added to a class by deriving a class from it and then by adding new features to it. Class once written or tested need not be rewritten or redefined. Inheritance is also referred to as specialization or derivation, as one class is inherited or derived from the other. It is also termed as "is-a" relationship because every object of the class being defined is also an object of the inherited class.

Stack using Linked List

```
// Stack using linked list
#include <iostream>
using namespace std;
//Structure of the Node
class Node
{
public:
int data;
Node *link;
};
class Stack1
{
public:
// top pointer to keep
track of the top of the
stack
Node *top = NULL;
//Function to check if
stack is empty or not
bool isempty()
{
if(top == NULL) return
true;
else
```

```
//Function to insert an element in stack
void push (int value)
{
Node *ptr = new Node(); ptr->data = value;
ptr->link = top; top = ptr;
}
//Function to delete an element from the stack
void pop ( )
{
if ( isempty() )
cout<<"Stack is Empty";
else
{
        Node *ptr = top; top = top -> link; delete(ptr);
        }
}
// Function to show the element at the top of the
stack
void showTop()
{
if ( isempty() ) cout<<"Stack is Empty";
else
cout<<"Element at top is : "<< top->data;
}
```

```
// Function to Display the stack
void displayStack()
{
if ( isempty() ) cout<<"Stack is Empty";
else
{
Node *temp=top; while(temp!=NULL)
{
cout<<temp->data<<" "; temp=temp->link;
}
cout<<"\n";
}
}
};
```

```
// Main function
int main()
{
Stack1 s;
int choice, flag=1, value;
//Menu Driven Program using Switch
while( flag == 1)
{
cout<<"\n1.Push 2.Pop 3.showTop 4.displayStack 5.exit\n";
cin>>choice;
switch (choice)
{
case 1: cout<<"Enter Value:\n"; cin>>value;
s.push(value); break;
case 2: s.pop();
break;
case 3: s.showTop();
break;
case 4: s.displayStack(); break;
case 5: flag = 0;
break;
}
}
return 0;
}
```

Queue using Linked List

```
#include <iostream>
using namespace
std;
class QNode
{
public:
int data;
QNode* next;
QNode(int d)
{
data = d;
next = NULL;
}
};
class Queue
{
public:
QNode *front, *rear;
Queue()
{
front = rear = NULL;
}
```

```
void enQueue(int x)
{
// Create a new LL node
QNode* temp = new QNode(x);
// If queue is empty, then
// new node is front and rear both
if (rear == NULL) {
front = rear = temp;
return;
}
// Add the new node at
// the end of queue and change
rear
rear->next = temp;
rear = temp;
}
```

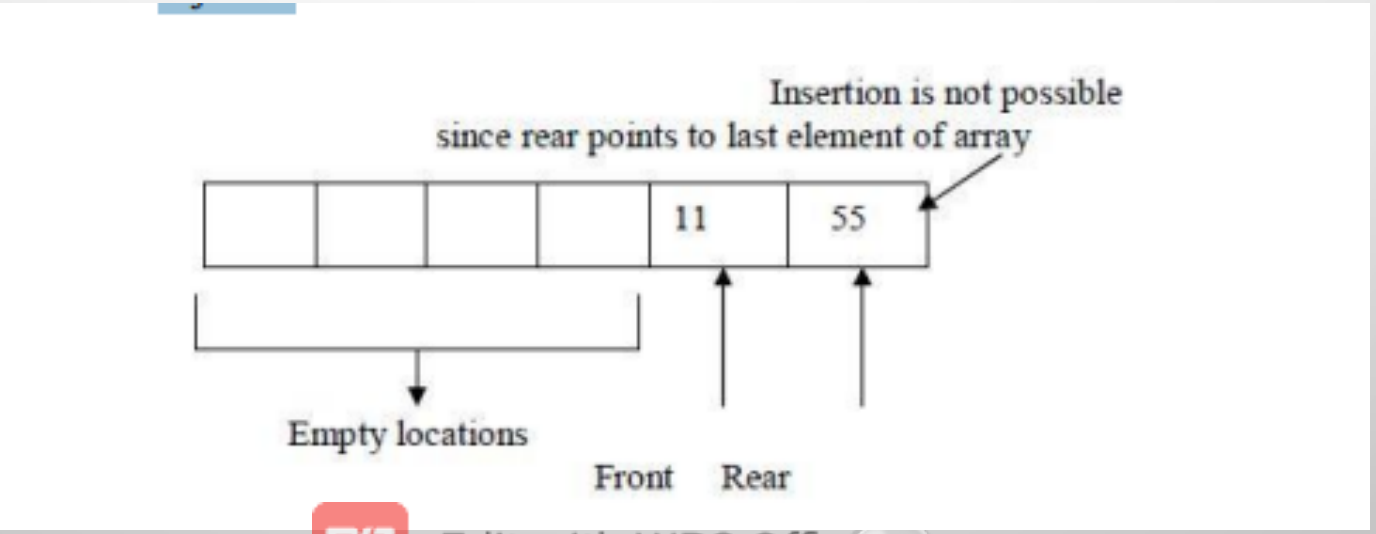
```
// Function to remove
// a key from given queue q
void deQueue()
{
// If queue is empty, return NULL.
if (front == NULL)
return;
// Store previous front and
// move front one node ahead
QNode* temp = front;
front = front->next;
// If front becomes NULL, then
// change rear also as NULL
if (front == NULL)
rear = NULL;
delete (temp);
}
```

```
void Display()
{
 QNode* temp = front;
if ((front == NULL) && (rear ==
NULL))
{
cout<<"Queue is empty"<<endl;
return;
}
cout<<"Queue elements are: ";
while (temp != NULL) {
cout<<temp->data<<" ";
temp = temp->next;
}
cout<<endl;
}
};
```

```
// Driven Program
int main()
{
Queue myq;
int ch,ele;
while(1){
        cout<<"\n1.Enqueue(insertion) 2.Dequeue(deletion)
3.display 4. Exit\nEnter ur choice ";
cin>>ch;
switch(ch)
{
case 1: cout<<"enter the element "; cin>>ele;
myq.enQueue(ele); break;
case 2: myq.deQueue(); break;
case 3: myq.Display(); break;
case 4: exit(0);
}
}
return 0;
}
```

CIRCULAR QUEUE

Once the queue gets f lled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted



When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overf bw condition satisfies. That means a lot of space is wasted.

Frequent reshuf fng of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as Circular Queues.

A circular queue is a queue in which all locations are treated as circular such that the f irst location CQ[0] follows the last location CQ[max-1].

For Program refer lab manual.