**UNIT-2: Stack & Queue**

**Stacks: Introduction, Operations, implementation, Applications.**

**Queues: Introduction,Operations, implementation, Applications, Circular Queue**

**Stack:**

The stack is a data structure that is very frequently used in computing as a kind of temporary storage. It is a list of elements to which additions and deletions can only be made at one end-the top. Consequently, the stack becomes a last-in-first-out (LIFO) data structure; the last element added is the first to be removed. When we add an item to the top of the stack, it is called a PUSH operation and when we remove an item from the top, it is called a POP operation. The stack data structure maintains a stack pointer always pointing to its top. After pushing an item, the stack pointer moves up to point always to the last item added. Similarly, after popping an item, the stack pointer moves down to the next last item in the stack.

**Stack Model**

POP (S)

TOP (S)

Stack s

Stack S

PUSH (R, S)

Stack can be implemented in either Fixed size stack or Dynamic size stack.

In fixed size stack we are using arrays.

Ex:

Shows a sample Stack

|  |
| --- |
|  |
| 3 |
| 2 |
| 1 |

After you Push 4 the Stack would be

|  |
| --- |
|  |
| 4 |
| 3 |
| 2 |
| 1 |

When you Pop the Stack would be

|  |
| --- |
|  |
| 3 |
| 2 |
| 1 |

**Stack Implementation:**

*import java.util.\*;*

*class StackN*

*{*

*int top=-1;*

*Scanner s= new Scanner(System.in);*

*int stk[];*

*int max;*

*StackN()*

*{*

*System.out.println("enter max");*

*max=s.nextInt();*

*stk=new int[max];*

*}*

*void push(int e)*

*{*

*if(!isFull())*

*stk[++top]=e;*

*else*

*System.out.println("stackoverflow");*

*}*

*void pop()*

*{*

*if(!isEmpty())*

*System.out.println("the element which is deleted from the stack is "+stk[top--]);*

*else*

*System.out.println("stackunderflow");*

*}*

*int peek()*

*{*

*if(!isEmpty())*

*return stk[top];*

*else*

*return -1;*

*}*

*boolean isEmpty()*

*{*

*if(top==-1)*

*return true;*

*else*

*return false;*

*}*

*boolean isFull()*

*{*

*if(top==max-1)*

*return true;*

*else*

*return false;*

*}*

*void display()*

*{*

*for(int i=top;i>=0;i--)*

*System.out.println(stk[i]);*

*}*

*}*

*class StackDemo*

*{*

*public static void main(String[] args)*

*{*

*StackN x=new StackN();*

*Scanner s= new Scanner(System.in);*

*int ch;*

*while(true)*

*{*

*System.out.println("enter your choice");*

*ch=s.nextInt();*

*switch(ch)*

*{*

*case 1:x.push(s.nextInt());*

*break;*

*case 2:x.pop();*

*break;*

*case 3:System.out.println(x.peek());*

*break;*

*case 4:x.display();*

*break;*

*default:return;*

*}*

*}*

*}*

*}*

**Applications of Stacks:**

1. Compilers will take the help of stack for checking syntax errors (Symbol Balancing)
2. Stacks were used in evaluating postfix expressions (Reverse polish notation).
3. Stacks are used in conversion of infix expressions to postfix expressions.
4. Stack for matching brackets in an expression.
5. Stacks were used in function call.

i.e. When there is a function call, all the important information that needs to be saved, such as register values and the return addresses are saved onto the stack. The information saved is called either an activation record or stack frame.

**Applications :**

* **Reversing a list:**
* **Factorial calculation:**
* **Infix to Postfix Conversion:**

**Infix Expression :**   
  
Any expression in the standard form like "2\*3-4/5" is an Infix(Inorder) expression.  
  
**Postfix Expression :**   
  
The Postfix(Postorder) form of the above expression is "23\*45/-".  
  
 **Conversion :**   
  
In normal algebra we use the infix notation like a+b\*c. The corresponding postfix notation is abc\*+. The algorithm for the conversion is as follows :

* Scan the Infix string from left to right.
* Initialize an empty stack.
* If the scanned character is an operand, add it to the Postfix string.
* If the scanned character is an operator and if the stack is empty Push the character to stack.
* If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack).
  + If topStack has lower precedence over the scanned character Push the scanned character to stack.
  + Else(higher or equal precedence) Pop the stack and add it to the postix string. Repeat this step as long as stack is not empty and topStack has less precedence over the character.
* If a left parenthesis is encountered, push it onto Stack.
* If a right parenthesis is encountered ,then:
  + - Repeatedly pop from Stack and add to postfix string until left parenthesis is encountered.
    - Remove the left Parenthesis.
* Repeat this step till all the characters are scanned.
* If stack is not empty Pop the stack and add it to postfix string
* Repeat this step as long as stack is not empty.
* Return the Postfix string.

**Example :**   
  
Let us see how the above algorithm will be imlemented using an example.   
  
Infix String : a+b\*c-d   
  
Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

|  |  |
| --- | --- |
| **Stack** | **Postfix String** |

Next character scanned is 'b' which will be placed in the Postfix string. Next character is '\*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '\*', so '\*' will be pushed to the stack.

|  |  |
| --- | --- |
| **Stack** | **Postfix String** |

The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '\*' which has a higher precedence than '-'. Thus '\*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.

|  |  |
| --- | --- |
| **Stack** | **Postfix String** |

Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :

|  |  |
| --- | --- |
| **Stack** | **Postfix String** |

End result :

* Infix String : a+b\*c-d
* Postfix String : abc\*+d-

***Follow the procedure in which is in your note book***

***Program for converting infix to postfix:***

*import java.util.\*;*

*class Stack1*

*{*

*char stk[]=new char[20];*

*int top=-1;*

*void push(char c)*

*{*

*stk[++top]=c;*

*}*

*char pop()*

*{*

*return stk[top--];*

*}*

*char peek()*

*{*

*return stk[top];*

*}*

*boolean isEmpty()*

*{*

*if(top==-1)*

*return true;*

*else*

*return false;*

*}*

*}*

*class InfxToPfx*

*{*

*public static void main(String[] args)*

*{*

*Stack1 x=new Stack1();*

*Scanner s=new Scanner(System.in);*

*String infx;*

*String pfx="";*

*System.out.println("enter the infix string!");*

*infx=s.nextLine();*

*for(int i=0;i<infx.length();i++)*

*{*

*if (Character.isLetter(infx.charAt(i)))*

*{*

*pfx=pfx+infx.charAt(i);*

*}*

*else if (infx.charAt(i)=='(')*

*{*

*x.push('(');*

*}*

*else if(infx.charAt(i)==')')*

*{*

*char y;*

*while((y=x.pop())!='(')*

*pfx=pfx+y;*

*}*

*else*

*{*

*if(x.isEmpty() || precedence(x.peek())<precedence(infx.charAt(i)))*

*x.push(infx.charAt(i));*

*else*

*{*

*while( !x.isEmpty()&&precedence(x.peek())>=precedence(infx.charAt(i)))*

*pfx=pfx+x.pop();*

*x.push(infx.charAt(i));*

*}*

*}*

*}*

*while(!x.isEmpty())*

*{*

*pfx=pfx+x.pop();*

*}*

*System.out.println(pfx);*

*}*

*static int precedence(char ch)*

*{*

*switch(ch)*

*{*

*case '(': return 1;*

*case '+':*

*case '-': return 2;*

*case '\*':*

*case '/':*

*case '%': return 3;*

*case '^': return 4;*

*}*

*return -1;*

*}*

*}*

**Postfix Evaluation:**

**Infix Expression :**   
  
Any expression in the standard form like "2\*3-4/5" is an Infix(Inorder) expression.  
  
**Postfix Expression :**   
  
The Postfix(Postorder) form of the above expression is "23\*45/-".  
  
 **Evaluation :**   
  
In normal algebra we use the infix notation like a+b\*c. The corresponding postfix notation is abc\*+. The algorithm for the conversion is as follows :

* Scan the Postfix string from left to right.
* Initialise an empty stack.
* If the scannned character is an operand, add it to the stack
* If the scanned character is an Operator, then pop two elements from stack. Store them to x and y. Now evaluate x operator y. Let the result of this operation be retVal. Push retVal into the stack.
* Repeat the step till all the characters are scanned.
* After all characters are scanned, we will have only one element in the stack. Return that element.

**Example :**   
  
Let us see how the above algorithm will be imlemented using an example.   
  
Postfix String : 123\*+4-   
  
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

Next character scanned is "\*", which is an operator. Thus, we pop the top two elements from the stack and perform the "\*" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

The value of the expression(2\*3) that has been evaluated(6) is pushed into the stack.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

Next character scanned is "4", which is added to the stack.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.

|  |  |
| --- | --- |
| **Stack** | **Expression** |

Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.   
  
End result :

* Postfix String : 123\*+4-
* Result : 3

**Program for postfix expression evaluation:**

*import java.util.\*;*

*class Stack1*

*{*

*char stk[]=new char[20];*

*int top=-1;*

*void push(char c)*

*{*

*stk[++top]=c;*

*}*

*char pop()*

*{*

*return stk[top--];*

*}*

*char peek()*

*{*

*return stk[top];*

*}*

*boolean isEmpty()*

*{*

*if(top==-1)*

*return true;*

*else*

*return false;*

*}*

*}*

*class InfxToPfx*

*{*

*public static void main(String[] args)*

*{*

*Stack1 x=new Stack1();*

*Scanner s=new Scanner(System.in);*

*String infx;*

*String pfx="";*

*System.out.println("enter the infix string!");*

*infx=s.nextLine();*

*for(int i=0;i<infx.length();i++)*

*{*

*if (Character.isLetter(infx.charAt(i)))*

*{*

*pfx=pfx+infx.charAt(i);*

*}*

*else if (infx.charAt(i)=='(')*

*{*

*x.push('(');*

*}*

*else if(infx.charAt(i)==')')*

*{*

*char y;*

*while((y=x.pop())!='(')*

*pfx=pfx+y;*

*}*

*else*

*{*

*if(x.isEmpty() || precedence(x.peek())<precedence(infx.charAt(i)))*

*x.push(infx.charAt(i));*

*else*

*{*

*while( !x.isEmpty()&&precedence(x.peek())>=precedence(infx.charAt(i)))*

*pfx=pfx+x.pop();*

*x.push(infx.charAt(i));*

*}*

*}*

*}*

*while(!x.isEmpty())*

*{*

*pfx=pfx+x.pop();*

*}*

*System.out.println(pfx);*

*}*

*static int precedence(char ch)*

*{*

*switch(ch)*

*{*

*case '(': return 1;*

*case '+':*

*case '-': return 2;*

*case '\*':*

*case '/':*

*case '%': return 3;*

*case '^': return 4;*

*}*

*return -1;*

*}*

*}*

**Queue**

Like Stacks, Queues are lists. With a queue, however, insertion is done at one end called rear end, where as deletion is performed at the other end called front end. Some times it is called FIFO (First in First out) List.

The basic operations on a queue are Enqueue, which inserts an element at the end of the list (called the rear), and Dequeue, which deletes the elements at the start of the list.

*Model of a Queue is*

Dequeue(Q)

Queue Q

Queue Q

Enqueue(Q)

***Array Implementation of Queue***

Consider a queue, which can add a maximum of five elements

After Create

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

0 1 2 3 4

**Front = 0, Rear=0**

After adding 5, 7, and 9 in same sequence the queue will be

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 9 |  |  |

0 1 2 3 4

Front Rear

After an Dequeue operation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 7 | 9 |  |  |

0 1 2 3 4

Front Rear

But there is one potential problem with the above implementation after some Enqueue operation the Queue appears to be full. (Suppose rear = 4 and front = 2 for 5 cell queue the queue appears to be full but not). Then you cannot insert further till the queue is empty.

The Simple solution is that whenever front and rear gets to the end of the array, it is wrapped around to the beginning. This is known as **circular Queue.**

**Applications of queues:**

1. Uses of Queues in operating systems: Operating system maintains several queues like ready queue, device queue etc. for handling multiple processes.

2. Use of Queue in Network Ex: for Queuing jobs in file Server.

3. A Whole branch of mathematics, known as queuing theory, deals with computing, probabilistically, how long users expect to wait on a line, how long the line gets, and other such questions.

**Implementation of Queue:**

*import java.util.Scanner;*

*class Queue1*

*{*

*int max,f,r;*

*int q[];*

*Queue1()*

*{*

*f=r=-1;*

*max=5;*

*q=new int[max];*

*}*

*void enQueue(int e)*

*{*

*if(!isFull())*

*{*

*q[++r]=e;*

*if(f==-1)*

*f++;*

*}*

*else*

*{*

*System.out.println("queue is full");*

*}*

*}*

*void deQueue()*

*{*

*if(!isEmpty())*

*{*

*System.out.println("the element which is going to deleted is "+q[f]);*

*if(f==r)*

*f=r=-1;*

*else*

*f++;*

*}*

*else*

*{*

*System.out.println("queue is empty");*

*}*

*}*

*boolean isEmpty()*

*{*

*if(f==-1)*

*return true;*

*else*

*return false;*

*}*

*boolean isFull()*

*{*

*if(r==max-1)*

*return true;*

*else*

*return false;*

*}*

*void display()*

*{*

*if(!isEmpty())*

*{*

*for (int i=f;i<=r;i++)*

*{*

*System.out.println(q[i]);*

*}*

*}*

*else*

*System.out.println("Queue is empty");*

*}*

*}*

*class QueueDemo*

*{*

*public static void main(String[] args)*

*{*

*Queue1 x=new Queue1();*

*Scanner s=new Scanner(System.in);*

*int ch;*

*while(true)*

*{*

*System.out.println("enter your choice ");*

*ch=s.nextInt();*

*switch(ch)*

*{*

*case 1:x.enQueue(s.nextInt());*

*break;*

*case 2:x.deQueue();*

*break;*

*case 3:x.display();*

*break;*

*default:return;*

*}*

*}*

*}*

*}*

**Implement Queue using Stacks**

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

**Method (By making deQueue operation costly)**  
In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from satck1 to stack2.

3) Pop the element from stack2 and return it.

**Program for Queue using two stacks**

*import java.util.Scanner;*

*class TwoStk*

*{*

*int top1,top2;*

*int s1[],s2[];*

*TwoStk()*

*{*

*top1=top2=-1;*

*s1=new int[20];*

*s2=new int[20];*

*}*

*void push1(int e)*

*{*

*s1[++top1]=e;*

*}*

*int pop1()*

*{*

*return s1[top1--];*

*}*

*void push2(int e)*

*{*

*s2[++top2]=e;*

*}*

*int pop2()*

*{*

*return s2[top2--];*

*}*

*void enQueue(int e)*

*{*

*push1(e);*

*}*

*void deQueue()*

*{*

*if (top1==-1 && top2==-1)*

*System.out.println("Queue underflow");*

*else if(top2==-1)*

*{*

*while(top1!=-1)*

*push2(pop1());*

*System.out.println("the element to be deleted is "+pop2());*

*}*

*else*

*System.out.println("the element to be deleted is "+pop2());*

*}*

*}*

*class QueueStack*

*{*

*public static void main(String[] args)*

*{*

*TwoStk x=new TwoStk();*

*Scanner s=new Scanner(System.in);*

*int ch;*

*while(true)*

*{*

*System.out.println("enter your choice ");*

*ch=s.nextInt();*

*switch(ch)*

*{*

*case 1:x.enQueue(s.nextInt());*

*break;*

*case 2:x.deQueue();*

*break;*

*default:return;*

*}*

*}*

*}*

*}*

**What is Round-Robin Scheduling?**

It is one of the oldest, simplest, fairest and most widely used scheduling algorithms, designed especially for time-sharing systems. A small unit of time, called timeslice or **quantum**, is defined. All runnable processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue.

The CPU scheduler picks the first process from the queue, sets a timer to interrupt after one quantum, and dispatches the process.

If the process is still running at the end of the quantum, the CPU is preempted and the process is added to the tail of the queue. If the process finishes before the end of the quantum, the process itself releases the CPU voluntarily. In either case, the CPU scheduler assigns the CPU to the next process in the ready queue. Every time a process is granted the CPU.

**Circular queue :**

One limitation of ordinary queue is, we can insert a new element into queue if rear<max . Suppose if front=4, Rear=9 then we can’t insert a new element even there are 3 empty locations

Front Rear

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 4 | 5 | 6 | 7 | 8 | 9 |

Circular queue eliminate the problem linear queue.

Circular queue is a queue but it is not linear in structure insted it is circular . In other words ,the FRONT and REAR variables which display a linear movements(left to right) over a queue, in circular queue they display a circular movements(clock wise)

Now we can place the next element in the first position in the above example using circular queue. For the circular movements FRONT and REAR can be increment like

Rear Front

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 |  |  | 4 | 5 | 6 | 7 | 8 | 9 |

For the circular movements FRONT and REAR can be increment like

Rear=(Rear%max)+1;

Fonrt=(Front%max)+1;

**Implementation of circular queue:**

*import java.util.\*;*

*class CQueue*

*{*

*int q[];*

*int f,r,size;*

*CQueue()*

*{*

*f=r=-1;*

*size=5;*

*q=new int[size];*

*}*

*void enQueue(int e)*

*{*

*if(!isFull())*

*{*

*r=(r+1)%size;*

*q[r]=e;*

*if(f==-1)*

*f=0;*

*}*

*else*

*System.out.println("Queue is full");*

*}*

*void deQueue()*

*{*

*if(!isEmpty())*

*{*

*System.out.println("the element that is to be deleted is "+q[f]);*

*if(f==r)*

*f=r=-1;*

*else*

*f=(f+1)%size;*

*}*

*else*

*System.out.println("Queue is empty");*

*}*

*boolean isEmpty()*

*{*

*if(f==-1)*

*return true;*

*else*

*return false;*

*}*

*boolean isFull()*

*{*

*if((r+1)%size==f)*

*return true;*

*else*

*return false;*

*}*

*void display()*

*{*

*if (!isEmpty())*

*{*

*if(f<=r)*

*{*

*for (int i=f;i<=r;i++)*

*System.out.println(q[i]);*

*}*

*else*

*{*

*for(int i=f; i!=r;i=(i+1)%size)*

*System.out.println(q[i]);*

*System.out.println(q[r]);*

*}*

*}*

*else*

*System.out.println("queue is empty");*

*}*

*}*

*class CQueueDemo*

*{*

*public static void main(String[] args)*

*{*

*CQueue x=new CQueue();*

*Scanner s=new Scanner(System.in);*

*int ch;*

*while(true)*

*{*

*System.out.println("enter your choice ");*

*ch=s.nextInt();*

*switch(ch)*

*{*

*case 1:x.enQueue(s.nextInt());*

*break;*

*case 2:x.deQueue();*

*break;*

*case 3:x.display();*

*break;*

*default:return;*

*}*

*}*

*}*

*}*