UNIT-II

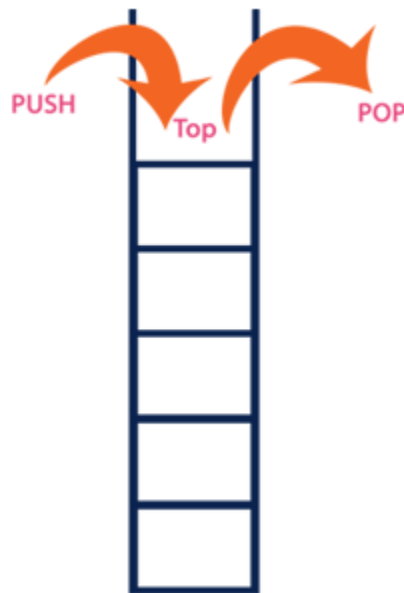## Stack Data structure:

What is a Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.

In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

> **Stack is a linear data structure in which the operations are performed based on LIFO principle.**

Stack can also be defined as

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. The last inserted element 50 is at Top of the stack as shown in the image below...

Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array stores only fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **functions** used in stack implementation.
- **Step 3 -** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4 -** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5 -** In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Push (value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- **Step 1 -** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

Pop () - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top = = -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top- -**).

Display () - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- **Step 1 - Check whether** stack **is EMPTY. (**top == -1)
- **Step 2 - If it is** EMPTY**, then display** "Stack is EMPTY!!!" **and terminate the function.**
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3 -** Repeat above step until **i** value becomes '0'.

Expressions

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc.,

we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**

2. **Postfix Expression**

3. **Prefix Expression**

Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example

Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
5. If the reading symbol is operator (+, - , *, / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Infix Expression	(5 + 3) * (8 - 2)
Postfix Expression	5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

| Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5)	Nothing
3	push(3)	Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result)	value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8) **(5 + 3)**
8	push(8)	(5 + 3)
2	push(2)	(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result)	value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6) **(8 - 2)** (5 + 3) , (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result)	value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48) **(6 * 8)** (5 + 3) * (8 - 2)
$ End of Expression	result = pop()	Display (result) **48** As final result

Infix Expression	(5 + 3) * (8 - 2) = 48
Postfix Expression	5 3 + 8 2 - * value is 48

Queue

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.
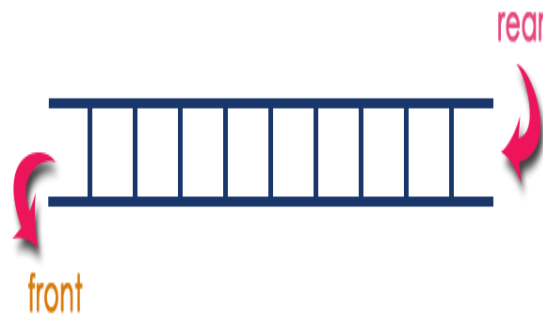
In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

Queue data structure can be defined as follows...

> **Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.**

A queue data structure can also be defined as

> **"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".**

Example

Queue after inserting 25, 30, 51, 60 and 85.

Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Queue Data structure Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle**with the

help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enqueue (value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**'(**front = rear = -1**).

Display () - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i = front+1**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i <= rear**)

Circular Queue Data structure

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example consider the queue below...

The queue after inserting all the elements into it is as follows...

Queue is Full

25	30	51	60	85	45	88	90	75	95

↑ front   ↑ rear

Now consider the following situation after deleting three elements from the queue...

Queue is Full (Even three elements are deleted)

| 25	30	51	60	85	45	88	90	75	95

↑ front   ↑ rear

This situation also says that Queue is Full and we cannot insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

What is Circular Queue?

A Circular Queue can be defined as follows...

> **Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.**

Graphical representation of a circular queue is as follows...
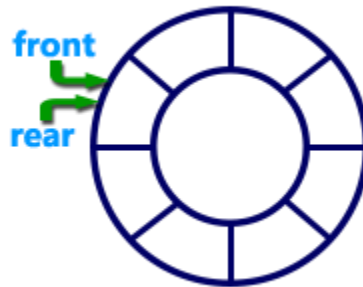
Priority Queue

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Applications of Queue Data Structure

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.
1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
See this for more detailed applications of Queue and Stack.
References: