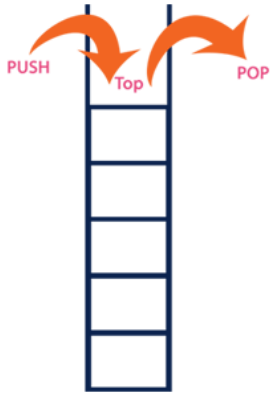
What is a Stack?

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
"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".
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 we want 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 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.

- **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 **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'.

Implementation of Stack using Array

```
#include<stdio.h>
#include<conio.h>
```

```
#define SIZE 10

void push(int);
void pop();
void display();

int stack[SIZE], top = -1;

void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
          case 1: printf("Enter the value to be insert: ");
                  scanf("%d",&value);
                  push(value);
                  break;
          case 2: pop();
                  break;
          case 3: display();
                  break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
void pop(){
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", stack[top]);
```

Applications of Stack:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)
3. Balancing of symbols
4. Redo-undo features at many places like editors, Photoshop.
5. Forward and backward feature in web browsers
6. Other applications can be Backtracking, Knight tour problem, queen problem and Sudoku

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

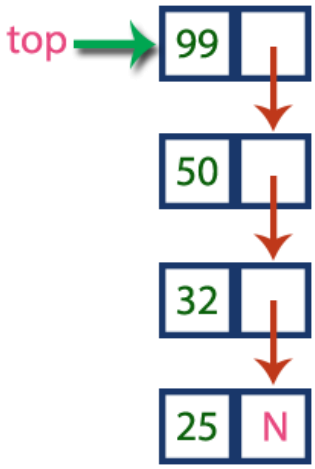
```
    top--;
  }
}
void display(){
  if(top == -1)
    printf("\nStack is Empty!!!");
  else{
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
          printf("%d\n",stack[i]);  }  }
```

Stack Using Linked List

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

To implement stack using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.

- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **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 define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

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

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **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 Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

Implementation of Stack using Linked List | C Programming

```
#include<stdio.h>
#include<conio.h>

struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();
```

```
void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(value);
                break;
        case 2: pop(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void push(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
    newNode->next = NULL;
  else
    newNode->next = top;
  top = newNode;
  printf("\nInsertion is Success!!!\n");
}
void pop()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    printf("\nDeleted element: %d", temp->data);
    top = temp->next;
    free(temp);
  }
}
```

```
void display()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    while(temp->next != NULL){
          printf("%d--->",temp->data);
          temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
}
```

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**

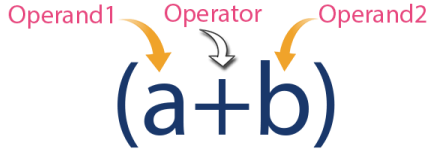
Infix Expression

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

Prefix Expression

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

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

Operator Operand1 Operand2

Example

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Infix to Postfix Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

- **Step 1 -** The Operators in the given Infix Expression : **= , + , ***
- **Step 2 -** The Order of Operators according to their preference : *** , + , =**
- **Step 3 -** Now, convert the first operator ***** ----- **D = A + B C ***
- **Step 4 -** Convert the next operator **+** ----- **D = A BC* +**
- **Step 5 -** Convert the next operator **=** ----- **D ABC*+ =**

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

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 poped and print each poped 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.**

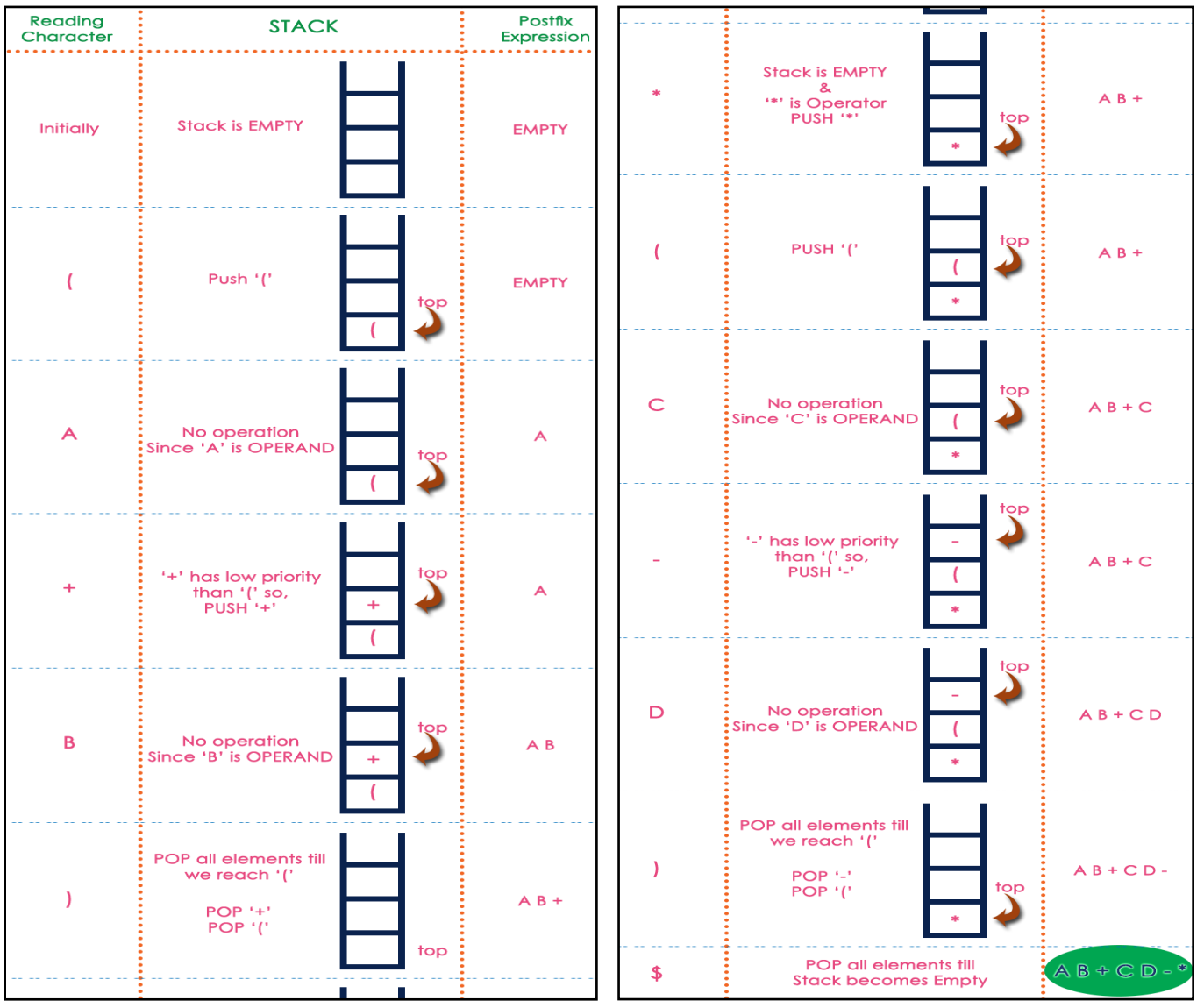
Example

Consider the following Infix Expression... **(A + B) * (C - D)**

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

The final Postfix Expression is as follows...

A B + C D - *



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.**

Example

Consider the following Expression...

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

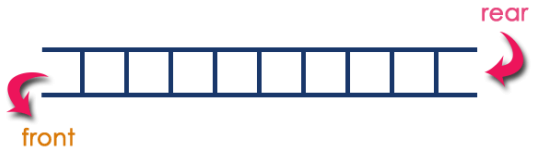
Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Queue ADT

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.

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

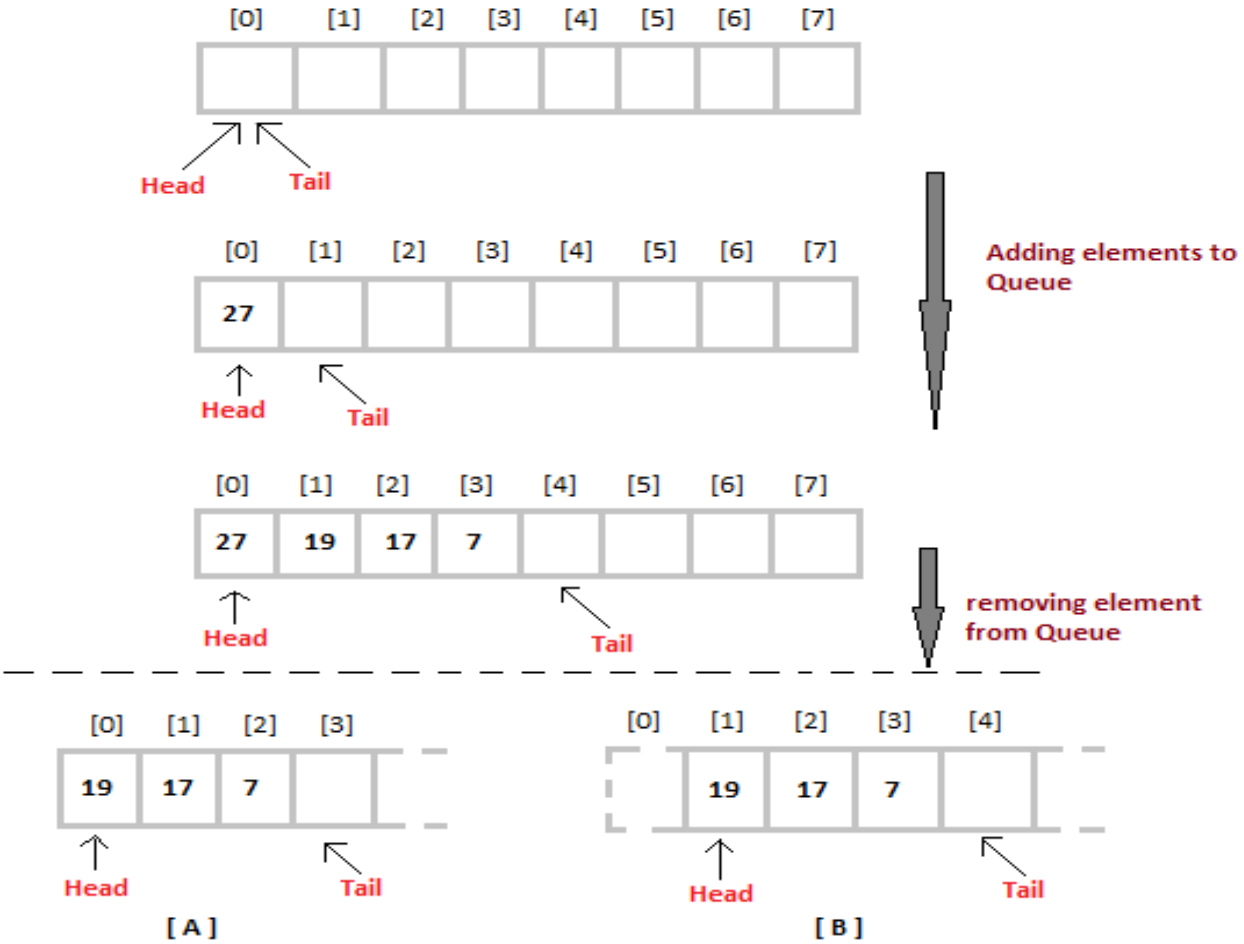
1. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail**keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.

2. When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

3. In approach [B] we remove the element from **head** position and then move **head** to the next position.

4. In approach [A] there is an **overhead of shifting the elements one position forward**every time we remove the first element.

5. In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.



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**)

- **Implementation of Queue Data structure using array - C Programming**

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void enQueue(int);
void deQueue();
void display();

int queue[SIZE], front = -1, rear = -1;

void main()
{
   int value, choice;
   clrscr();
   while(1){
      printf("\n\n***** MENU *****\n");
      printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
      printf("\nEnter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d",&value);
                 enQueue(value);
                 break;
         case 2: deQueue();
                 break;
         case 3: display();
                 break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void enQueue(int value){
   if(rear == SIZE-1)
      printf("\nQueue is Full!!! Insertion is not possible!!!");
   else{
      if(front == -1)
         front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
void deQueue(){
   if(front == rear)
      printf("\nQueue is Empty!!! Deletion is not possible!!!");
   else{
      printf("\nDeleted : %d", queue[front]);
      front++;
```

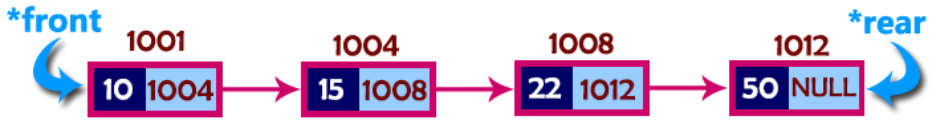
```
      if(front == rear)
         front = rear = -1;
   }
}
void display(){
   if(rear == -1)
      printf("\nQueue is Empty!!!");
   else{
      int i;
      printf("\nQueue elements are:\n");
      for(i=front; i<=rear; i++)
          printf("%d\t",queue[i]);
   }
}
```

Queue Using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data values. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode** → **next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear** → **next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front** → **next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

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

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp** → **next** != **NULL**).
- **Step 5 -** Finally! Display '**temp** → **data** ---> **NULL**'.

Implementation of Queue Datastructure using Linked List - C Programming

```
#include<stdio.h>
#include<conio.h>

struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
```

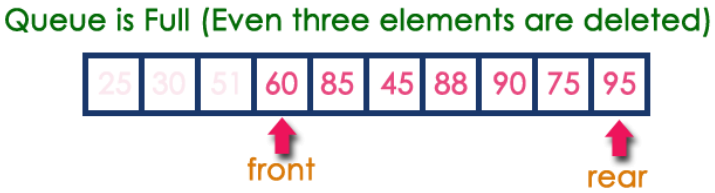
```
                scanf("%d", &value);
                insert(value);
                break;
         case 2: delete(); break;
         case 3: display(); break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n");
}
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
         printf("%d--->",temp->data);
         temp = temp -> next;
      }
      printf("%d--->NULL\n",temp->data);
   }
}
```

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...

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



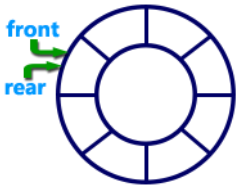
This situation also says that Queue is Full and we cannot insert the new element because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue we cannot make use of them to insert new element. This is the major problem in normal queue data structure.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

What is Circular Queue?

A Circular Queue can be defined as follows...

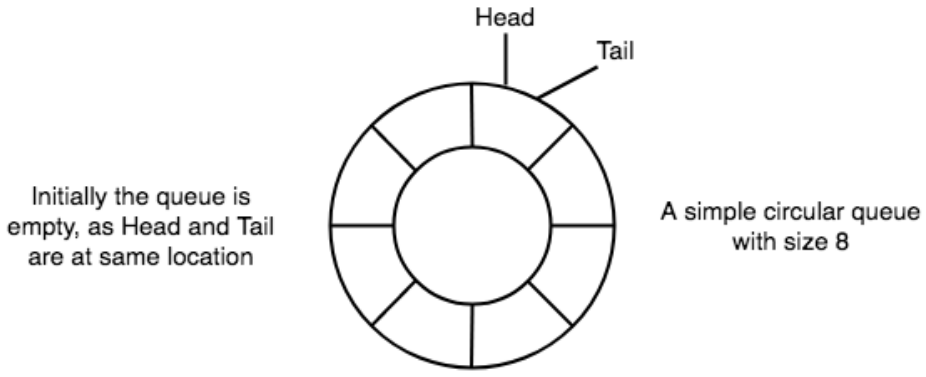
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

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

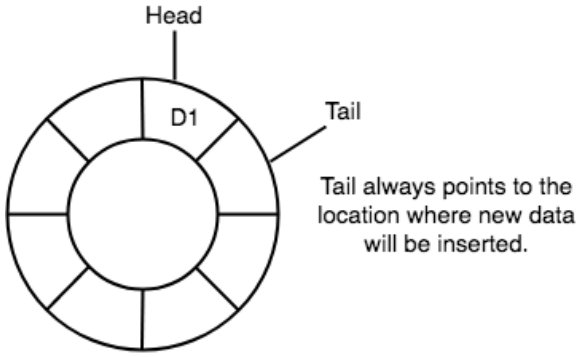


Basic features of Circular Queue

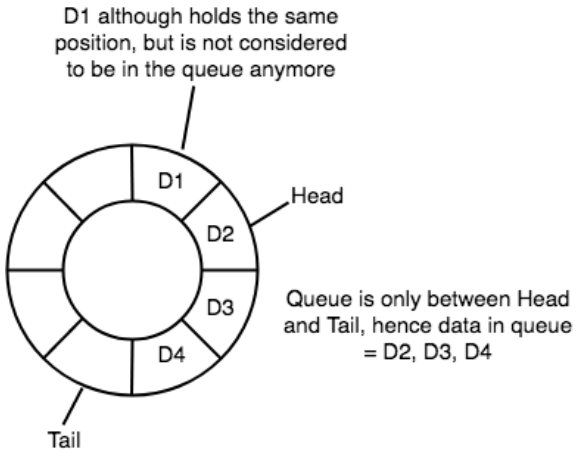
1. In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



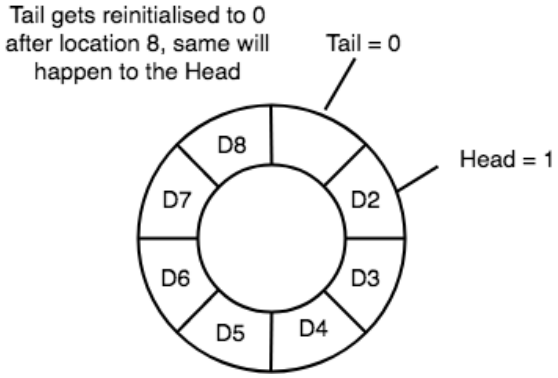
3. New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.



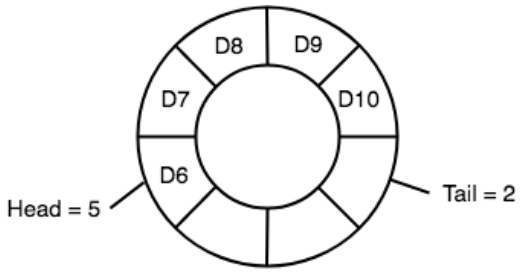
4. In a circular queue, data is not actually removed from the queue. Only the `head`pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.



5. The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.



6. Also, the `head` and the `tail` pointers can cross each other. In other words, `head`pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management

Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

- **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 **user defined functions** used in circular queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular 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 circular queue. We can use the following steps to insert an element into the circular queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+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 check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4 -** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

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

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)

- **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 display **queue[front]** as deleted element and increment the **front**value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

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

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **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**'.
- **Step 4 -** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5 -** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.
- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

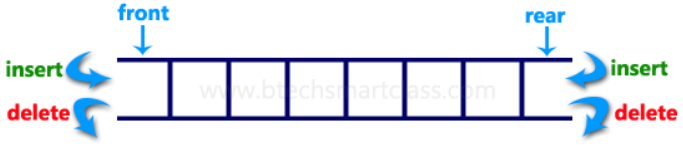
Implementation of Circular Queue Data structure using array - C Programming

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5

void enQueue(int);
void deQueue();
void display();

int cQueue[SIZE], front = -1, rear = -1;

void main()
{
   int choice, value;
   clrscr();
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("\nEnter the value to be insert:  ");
                 scanf("%d",&value);
                 enQueue(value);
                 break;
         case 2: deQueue();
                 break;
         case 3: display();
                 break;
         case 4: exit(0);
         default: printf("\nPlease select the correct choice!!!\n");
      }
   }
}
```

```
void enQueue(int value)
{
   if((front == 0 && rear == SIZE - 1) || (front == rear+1))
      printf("\nCircular Queue is Full! Insertion not possible!!!\n");
   else{
      if(rear == SIZE-1 && front != 0)
         rear = -1;
      cQueue[++rear] = value;
      printf("\nInsertion Success!!!\n");
      if(front == -1)
         front = 0;
   }
}
void deQueue()
{
   if(front == -1 && rear == -1)
      printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
   else{
      printf("\nDeleted element : %d\n",cQueue[front++]);
      if(front == SIZE)
         front = 0;
      if(front-1 == rear)
         front = rear = -1;
   }
}
void display()
{
   if(front == -1)
      printf("\nCircular Queue is Empty!!!\n");
   else{
      int i = front;
      printf("\nCircular Queue Elements are : \n");
      if(front <= rear){
         while(i <= rear)
            printf("%d\t",cQueue[i++]);
      }
      else{
         while(i <= SIZE - 1)
            printf("%d\t", cQueue[i++]);
         i = 0;
         while(i <= rear)
            printf("%d\t",cQueue[i++]);
      }
   }
}
```

Double Ended Queue Data structure

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

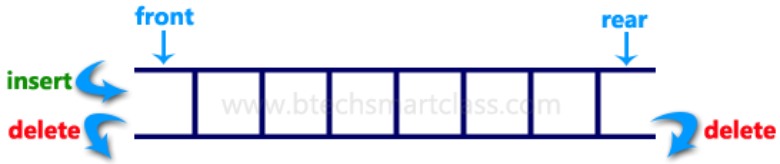


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

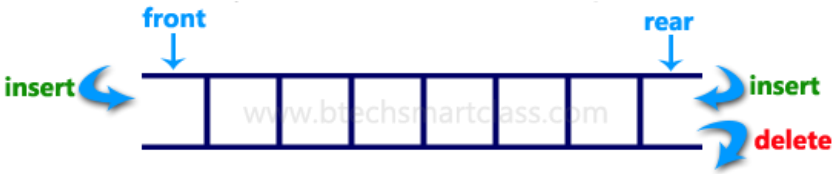
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Implementation of Double Ended Queue Data structure using array - C Programming

```
#include<stdio.h>
#include<conio.h>
#define SIZE 100

void enQueue(int);
int deQueueFront();
int deQueueRear();
void enQueueRear(int);
void enQueueFront(int);
void display();

int queue[SIZE];
int rear = 0, front = 0;

int main()
{
    char ch;
    int choice1, choice2, value;
    printf("\n******* Type of Double Ended Queue *******\n");
     do
     {
          printf("\n1.Input-restricted deque \n");
          printf("2.output-restricted deque \n");
          printf("\nEnter your choice of Queue Type : ");
          scanf("%d",&choice1);
          switch(choice1)
          {
               case 1:
                    printf("\nSelect the Operation\n");
                    printf("1.Insert\n2.Delete from Rear\n3.Delete from Front\n4. Display");
                    do
                    {
                       printf("\nEnter your choice for the operation in c deque: ");
                       scanf("%d",&choice2);
                       switch(choice2)
```

```
                        {
                           case 1: enQueueRear(value);
                                   display();
                                    break;
                            case 2: value = deQueueRear();
                                    printf("\nThe value deleted is %d",value);
                                   display();
                                    break;
                           case 3: value=deQueueFront();
                                    printf("\nThe value deleted is %d",value);
                                   display();
                                    break;
                           case 4: display();
                                   break;
                           default:printf("Wrong choice");
                        }
                        printf("\nDo you want to perform another operation (Y/N): ");
                        ch=getch();
                     }while(ch=='y'||ch=='Y');
                     getch();
                     break;

               case 2 :
                    printf("\n---- Select the Operation ----\n");
                    printf("1. Insert at Rear\n2. Insert at Front\n3. Delete\n4. Display");
                    do
                    {
                       printf("\nEnter your choice for the operation: ");
                       scanf("%d",&choice2);
                       switch(choice2)
                       {
                          case 1: enQueueRear(value);
                                  display();
                                  break;
                          case 2: enQueueFront(value);
                                  display();
                                  break;
                          case 3: value = deQueueFront();
                                  printf("\nThe value deleted is %d",value);
                                  display();
                                  break;
                          case 4: display();
                                  break;
                          default:printf("Wrong choice");
                        }
                        printf("\nDo you want to perform another operation (Y/N): ");
                        ch=getch();
                     } while(ch=='y'||ch=='Y');
                     getch();
                     break ;
            }
            printf("\nDo you want to continue(y/n):");
            ch=getch();
      }while(ch=='y'||ch=='Y');
}

void enQueueRear(int value)
{
     char ch;
```

```
     if(front == SIZE/2)
      {
            printf("\nQueue is full!!! Insertion is not possible!!! ");
            return;
      }
      do
      {
            printf("\nEnter the value to be inserted:");
            scanf("%d",&value);
            queue[front] = value;
            front++;
            printf("Do you want to continue insertion Y/N");
            ch=getch();
      }while(ch=='y');
}

void enQueueFront(int value)
{
     char ch;
     if(front==SIZE/2)
      {
            printf("\nQueue is full!!! Insertion is not possible!!!");
            return;
      }
      do
      {
            printf("\nEnter the value to be inserted:");
            scanf("%d",&value);
            rear--;
            queue[rear] = value;
            printf("Do you want to continue insertion Y/N");
            ch = getch();
      }
      while(ch == 'y');
}
int deQueueRear()
{
     int deleted;
     if(front == rear)
     {
            printf("\nQueue is Empty!!! Deletion is not possible!!!");
            return 0;
     }
     front--;
     deleted = queue[front+1];
     return deleted;
}
int deQueueFront()
{
     int deleted;
     if(front == rear)
     {
            printf("\nQueue is Empty!!! Deletion is not possible!!!");
            return 0;
     }
     rear++;
     deleted = queue[rear-1];
     return deleted;
}
```

```
void display()
{
     int i;
     if(front == rear)
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
     else{
        printf("\nThe Queue elements are:");
        for(i=rear; i < front; i++)
        {
           printf("%d\t ",queue[i]);
        }
     }
}
```

Implement Queue using Stacks

1. A Queue is defined by its property of FIFO, which means First in First Out, i.e the element which is added first is taken out first. This behaviour defines a queue, whereas data is actually stored in an array or a list in the background.
2. What we mean here is that no matter how and where the data is getting stored, if the first element added is the first element being removed and we have implementation of the functions `enqueue()`and `dequeue()` to enable this behaviour, we can say that we have implemented a Queue data structure.
3. In our previous tutorial, we used a simple array to store the data elements, but in this tutorial we will be using Stack data structure for storing the data.
4. While implementing a queue data structure using stacks, we will have to consider the natural behaviour of stack too, which is First in Last Out.
5. For performing enqueue we require only one stack as we can directly push data onto the stack, but to perform dequeue we will require two Stacks, because we need to follow queue's FIFO property and if we directly pop any data element out of Stack, it will follow LIFO approach(Last in First Out).

Implementation of Queue using Stacks

In all we will require two Stacks to implement a queue, we will call them `S1` and `S2`.

```
class Queue {
   public:
   Stack S1, S2;
         //declaring enqueue method
   void enqueue(int x);

          //declaring dequeue method
   int dequeue();
}
```

In the code above, we have simply defined a class `Queue`, with two variables `S1` and `S2` of type `Stack`.
We know that, Stack is a data structure, in which data can be added using `push()` method and data can be removed using `pop()` method.

Making the Enqueue operation costly

In this approach, we make sure that the oldest element added to the queue stays at the **top** of the stack, the second oldest below it and so on.
To achieve this, we will need two stacks. Following steps will be involved while enqueuing a new element to the queue.

NOTE: First stack(S1) is the main stack being used to store the data, while the second stack(S2) is to assist and store data temporarily during various operations.

1. If the queue is empty(means S1 is empty), directly **push** the first element onto the stack S1.
2. If the queue is not empty, move all the elements present in the first stack(S1) to the second stack(S2), one by one. Then add the new element to the first stack, then move back all the elements from the second stack back to the first stack.
3. Doing so will always maintain the right order of the elements in the stack, with the 1st data element staying always at the **top**, with 2nd data element right below it and the new data element will be added to the bottom. This makes removing an element from the queue very simple, all we have to do is call the pop()method for stack S1.