**MY DSA DOCUMENT**

**2.DATA STRUCTURES(COURSERA)**

**1.LINKED LIST**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:  


**Advantages over arrays**  
**1)** Dynamic size  
**2)** Ease of insertion/deletion

**Drawbacks:**  
**1)** Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it [here](https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/).  
**2)** Extra memory space for a pointer is required with each element of the list.  
**3)** Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

**1.creating nodes**

C

// A linked list node

struct Node {

    int data;

    struct Node\* next;

};

C++

class Node {

public:

    int data;

    Node \*next;

};

**2. creating Linked list with 3 nodes**

*// A simple CPP program to introduce   
// a linked list*#include **<bits/stdc++.h>  
using namespace** std;  
  
**class** Node {  
**public**:  
 **int** data;  
 Node\* next;  
};  
  
*// Program to create a simple linked   
// list with 3 nodes***int** main()  
{  
 Node\* head = NULL;  
 Node\* second = NULL;  
 Node\* third = NULL;  
  
 *// allocate 3 nodes in the heap* head = **new** Node();  
 second = **new** Node();  
 third = **new** Node();  
  
 */\* Three blocks have been allocated dynamically.   
 We have pointers to these three blocks as head,   
 second and third   
 head second third   
 | | |   
 | | |   
 +---+-----+ +----+----+ +----+----+   
 | # | # | | # | # | | # | # |   
 +---+-----+ +----+----+ +----+----+   
   
# represents any random value.   
Data is random because we haven’t assigned   
anything yet \*/* head->data = 1; *// assign data in first node* head->next = second; *// Link first node with   
 // the second node   
  
 /\* data has been assigned to the data part of first   
 block (block pointed by the head). And next   
 pointer of the first block points to second.   
 So they both are linked.   
  
 head second third   
 | | |   
 | | |   
 +---+---+ +----+----+ +-----+----+   
 | 1 | o----->| # | # | | # | # |   
 +---+---+ +----+----+ +-----+----+   
\*/  
  
 // assign data to second node* second->data = 2;  
  
 *// Link second node with the third node* second->next = third;  
  
 */\* data has been assigned to the data part of the second   
 block (block pointed by second). And next   
 pointer of the second block points to the third   
 block. So all three blocks are linked.   
   
 head second third   
 | | |   
 | | |   
 +---+---+ +---+---+ +----+----+   
 | 1 | o----->| 2 | o-----> | # | # |   
 +---+---+ +---+---+ +----+----+ \*/* third->data = 3; *// assign data to third node* third->next = NULL;  
  
 */\* data has been assigned to the data part of the third   
 block (block pointed by third). And next pointer   
 of the third block is made NULL to indicate   
 that the linked list is terminated here.   
  
 We have the linked list ready.   
  
 head   
 |   
 |   
 +---+---+ +---+---+ +----+------+   
 | 1 | o----->| 2 | o-----> | 3 | NULL |   
 +---+---+ +---+---+ +----+------+   
   
   
 Note that only the head is sufficient to represent   
 the whole list. We can traverse the complete   
 list by following the next pointers. \*/* **return** 0;  
}  
  
*// This code is contributed by rathbhupendra*

**3.Traversal**

void printList(Node\* n) //n is head

{

    while (n != NULL) {

        cout << n->data << " ";

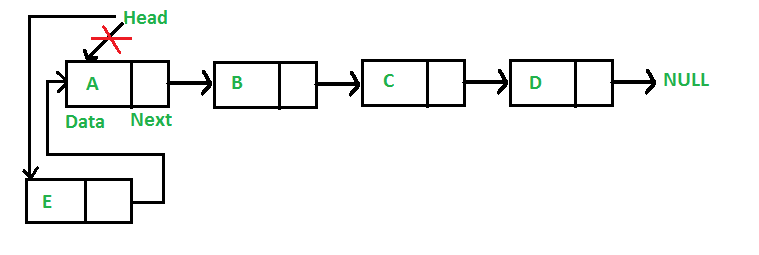
        n = n->next;

    }

}

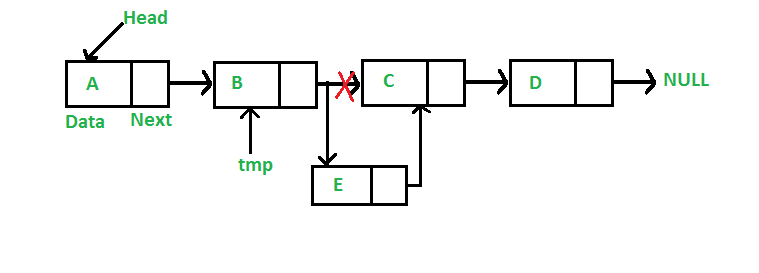
**4.Node insertion**

**(a) in front**



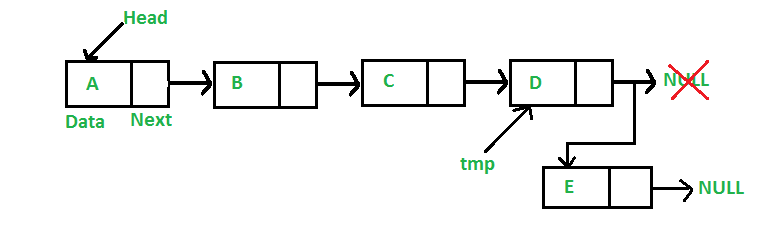
*/\* Given a reference (pointer to pointer)   
to the head of a list and an int,   
inserts a new node on the front of the list. \*/***void** push(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* Node\* new\_node = **new** Node();  
  
 */\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. Make next of new node as head \*/* new\_node->next = (\*head\_ref);  
  
 */\* 4. move the head to point to the new node \*/* (\*head\_ref) = new\_node;  
}  
  
*// This code is contributed by rathbhupendra*

(b) add node after a given node



*// Given a node prev\_node, insert a   
// new node after the given   
// prev\_node***void** insertAfter(Node\* prev\_node, **int** new\_data)  
{  
  
 *// 1. Check if the given prev\_node is NULL* **if** (prev\_node == NULL)  
 {  
 cout << **"the given previous node cannot be NULL"**;  
 **return**;  
 }  
  
 *// 2. Allocate new node* Node\* new\_node = **new** Node();  
  
 *// 3. Put in the data* new\_node->data = new\_data;  
  
 *// 4. Make next of new node as   
 // next of prev\_node* new\_node->next = prev\_node->next;  
  
 *// 5. move the next of prev\_node   
 // as new\_node* prev\_node->next = new\_node;  
}  
  
*// This code is contributed by anmolgautam818*

(c) at the end

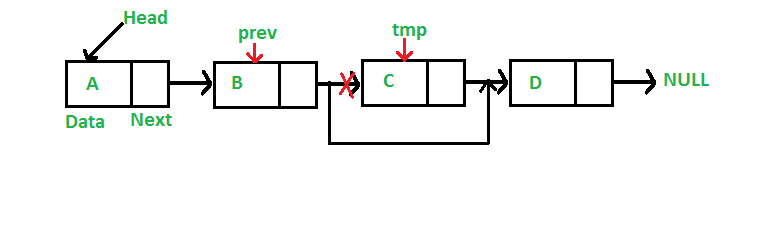


*// Given a reference (pointer to pointer) to the head   
// of a list and an int, appends a new node at the end***void** append(Node\*\* head\_ref, **int** new\_data)  
{  
  
 *// 1. allocate node* Node\* new\_node = **new** Node();  
  
 *// Used in step 5* Node \*last = \*head\_ref;  
  
 *// 2. Put in the data* new\_node->data = new\_data;  
  
 *// 3. This new node is going to be   
 // the last node, so make next of   
 // it as NULL* new\_node->next = NULL;  
  
 *// 4. If the Linked List is empty,   
 // then make the new node as head* **if** (\*head\_ref == NULL)  
 {  
 \*head\_ref = new\_node;  
 **return**;  
 }  
  
 *// 5. Else traverse till the last node* **while** (last->next != NULL)  
 last = last->next;  
  
 *// 6. Change the next of last node* last->next = new\_node;  
 **return**;  
}  
  
*// This code is contributed by anmolgautam818*

Clubbed insertion code

*// A complete working C++ program to demonstrate   
// all insertion methods on Linked List*#include **<bits/stdc++.h>  
using namespace** std;  
  
*// A linked list node***class** Node  
{  
**public**:  
 **int** data;  
 Node \*next;  
};  
  
*/\* Given a reference (pointer to pointer)   
to the head of a list and an int, inserts   
a new node on the front of the list. \*/***void** push(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* Node\* new\_node = **new** Node();  
  
 */\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. Make next of new node as head \*/* new\_node->next = (\*head\_ref);  
  
 */\* 4. move the head to point to the new node \*/* (\*head\_ref) = new\_node;  
}  
  
*/\* Given a node prev\_node, insert a new node after the given   
prev\_node \*/***void** insertAfter(Node\* prev\_node, **int** new\_data)  
{  
 */\*1. check if the given prev\_node is NULL \*/* **if** (prev\_node == NULL)  
 {  
 cout<<**"the given previous node cannot be NULL"**;  
 **return**;  
 }  
  
 */\* 2. allocate new node \*/* Node\* new\_node = **new** Node();  
  
 */\* 3. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 4. Make next of new node as next of prev\_node \*/* new\_node->next = prev\_node->next;  
  
 */\* 5. move the next of prev\_node as new\_node \*/* prev\_node->next = new\_node;  
}  
  
*/\* Given a reference (pointer to pointer) to the head   
of a list and an int, appends a new node at the end \*/***void** append(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* Node\* new\_node = **new** Node();  
  
 Node \*last = \*head\_ref; */\* used in step 5\*/  
  
 /\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. This new node is going to be   
 the last node, so make next of   
 it as NULL\*/* new\_node->next = NULL;  
  
 */\* 4. If the Linked List is empty,   
 then make the new node as head \*/* **if** (\*head\_ref == NULL)  
 {  
 \*head\_ref = new\_node;  
 **return**;  
 }  
  
 */\* 5. Else traverse till the last node \*/* **while** (last->next != NULL)  
 last = last->next;  
  
 */\* 6. Change the next of last node \*/* last->next = new\_node;  
 **return**;  
}  
  
*// This function prints contents of   
// linked list starting from head***void** printList(Node \*node)  
{  
 **while** (node != NULL)  
 {  
 cout<<**" "**<<node->data;  
 node = node->next;  
 }  
}  
  
*/\* Driver code\*/***int** main()  
{  
 */\* Start with the empty list \*/* Node\* head = NULL;  
  
 *// Insert 6. So linked list becomes 6->NULL* append(&head, 6);  
  
 *// Insert 7 at the beginning.   
 // So linked list becomes 7->6->NULL* push(&head, 7);  
  
 *// Insert 1 at the beginning.   
 // So linked list becomes 1->7->6->NULL* push(&head, 1);  
  
 *// Insert 4 at the end. So   
 // linked list becomes 1->7->6->4->NULL* append(&head, 4);  
  
 *// Insert 8, after 7. So linked   
 // list becomes 1->7->8->6->4->NULL* insertAfter(head->next, 8);  
  
 cout<<**"Created Linked list is: "**;  
 printList(head);  
  
 **return** 0;  
}

**Deletion of a particular node**



Code:

*// A complete working C program   
// to demonstrate deletion in   
// singly linked list*#include **<stdio.h>**#include **<stdlib.h>***// A linked list node***struct** Node  
{  
 **int** data;  
 **struct** Node \*next;  
};  
  
*/\* Given a reference (pointer to pointer) to the head of a list  
and an int, inserts a new node on the front of the list. \*/***void** push(**struct** Node\*\* head\_ref, **int** new\_data)  
{  
 **struct** Node\* new\_node = (**struct** Node\*) malloc(**sizeof**(**struct** Node));  
 new\_node->data = new\_data;  
 new\_node->next = (\*head\_ref);  
 (\*head\_ref) = new\_node;  
}  
  
*/\* Given a reference (pointer to pointer) to the head of a list  
and a key, deletes the first occurrence of key in linked list \*/***void** deleteNode(**struct** Node \*\*head\_ref, **int** key)  
{  
 *// Store head node* **struct** Node\* temp = \*head\_ref, \*prev;  
  
 *// If head node itself holds the key to be deleted* **if** (temp != **NULL** && temp->data == key)  
 {  
 \*head\_ref = temp->next; *// Changed head* free(temp); *// free old head* **return**;  
 }  
  
 *// Search for the key to be deleted, keep track of the  
 // previous node as we need to change 'prev->next'* **while** (temp != **NULL** && temp->data != key)  
 {  
 prev = temp;  
 temp = temp->next;  
 }  
  
 *// If key was not present in linked list* **if** (temp == **NULL**) **return**;  
  
 *// Unlink the node from linked list* prev->next = temp->next;  
  
 free(temp); *// Free memory*}  
  
*// This function prints contents of linked list starting from   
// the given node***void** printList(**struct** Node \*node)  
{  
 **while** (node != **NULL**)  
 {  
 printf(**" %d "**, node->data);  
 node = node->next;  
 }  
}  
  
*// Driver code***int** main()  
{  
 */\* Start with the empty list \*/* **struct** Node\* head = **NULL**;  
  
 push(&head, 7);  
 push(&head, 1);  
 push(&head, 3);  
 push(&head, 2);  
  
 puts(**"Created Linked List: "**);  
 printList(head);  
 deleteNode(&head, 1);  
 puts(**"\nLinked List after Deletion of 1: "**);  
 printList(head);  
 **return** 0;  
}

**Delete a node from given position**

*// A complete working C program to delete a node in a linked list   
// at a given position*#include **<stdio.h>**#include **<stdlib.h>***// A linked list node***struct** Node  
{  
 **int** data;  
 **struct** Node \*next;  
};  
  
*/\* Given a reference (pointer to pointer) to the head of a list   
and an int inserts a new node on the front of the list. \*/***void** push(**struct** Node\*\* head\_ref, **int** new\_data)  
{  
 **struct** Node\* new\_node = (**struct** Node\*) malloc(**sizeof**(**struct** Node));  
 new\_node->data = new\_data;  
 new\_node->next = (\*head\_ref);  
 (\*head\_ref) = new\_node;  
}  
  
*/\* Given a reference (pointer to pointer) to the head of a list   
and a position, deletes the node at the given position \*/***void** deleteNode(**struct** Node \*\*head\_ref, **int** position)  
{  
*// If linked list is empty* **if** (\*head\_ref == **NULL**)  
 **return**;  
  
*// Store head node* **struct** Node\* temp = \*head\_ref;  
  
 *// If head needs to be removed* **if** (position == 0)  
 {  
 \*head\_ref = temp->next; *// Change head* free(temp); *// free old head* **return**;  
 }  
  
 *// Find previous node of the node to be deleted* **for** (**int** i=0; temp!=**NULL** && i<position-1; i++)  
 temp = temp->next;  
  
 *// If position is more than number of nodes* **if** (temp == **NULL** || temp->next == **NULL**)  
 **return**;  
  
 *// Node temp->next is the node to be deleted   
 // Store pointer to the next of node to be deleted* **struct** Node \*next = temp->next->next;  
  
 *// Unlink the node from linked list* free(temp->next); *// Free memory* temp->next = next; *// Unlink the deleted node from list*}  
  
*// This function prints contents of linked list starting from   
// the given node***void** printList(**struct** Node \*node)  
{  
 **while** (node != **NULL**)  
 {  
 printf(**" %d "**, node->data);  
 node = node->next;  
 }  
}  
  
*/\* Driver program to test above functions\*/***int** main()  
{  
 */\* Start with the empty list \*/* **struct** Node\* head = **NULL**;  
  
 push(&head, 7);  
 push(&head, 1);  
 push(&head, 3);  
 push(&head, 2);  
 push(&head, 8);  
  
 puts(**"Created Linked List: "**);  
 printList(head);  
 deleteNode(&head, 4);  
 puts(**"\nLinked List after Deletion at position 4: "**);  
 printList(head);  
 **return** 0;  
}

**T0 delete whole list**

*// C++ program to delete a linked list*#include **<bits/stdc++.h>  
using namespace** std;  
  
*/\* Link list node \*/***class** Node  
{  
**public**:  
 **int** data;  
 Node\* next;  
};  
  
*/\* Function to delete the entire linked list \*/***void** deleteList(Node\*\* head\_ref)  
{  
  
*/\* deref head\_ref to get the real head \*/* Node\* current = \*head\_ref;  
 Node\* next;  
  
 **while** (current != **NULL**)  
 {  
 next = current->next;  
 free(current);  
 current = next;  
 }  
  
*/\* deref head\_ref to affect the real head back   
 in the caller. \*/* \*head\_ref = **NULL**;  
}  
  
*/\* Given a reference (pointer to pointer) to the head   
of a list and an int, push a new node on the front   
of the list. \*/***void** push(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* allocate node \*/* Node\* new\_node = **new** Node();  
  
 */\* put in the data \*/* new\_node->data = new\_data;  
  
 */\* link the old list off the new node \*/* new\_node->next = (\*head\_ref);  
  
 */\* move the head to point to the new node \*/* (\*head\_ref) = new\_node;  
}  
  
*/\* Driver code\*/***int** main()  
{  
 */\* Start with the empty list \*/* Node\* head = **NULL**;  
  
 */\* Use push() to construct below list   
 1->12->1->4->1 \*/* push(&head, 1);  
 push(&head, 4);  
 push(&head, 1);  
 push(&head, 12);  
 push(&head, 1);  
  
 cout << **"Deleting linked list"**;  
 deleteList(&head);  
  
 cout << **"\nLinked list deleted"**;  
}

**CIRCULAR LINKED LIST**



**Code for all types of insertions**

\#include<bits/stdc++.h>  
**using namespace** std;  
  
**struct** Node  
{  
 **int** data;  
 **struct** Node \*next;  
};  
  
**struct** Node \*addToEmpty(**struct** Node \*last, **int** data)  
{  
 *// This function is only for empty list* **if** (last != NULL)  
 **return** last;  
  
 *// Creating a node dynamically.* **struct** Node \*temp =  
 (**struct** Node\*)malloc(**sizeof**(**struct** Node));  
  
 *// Assigning the data.* temp -> data = data;  
 last = temp;  
  
 *// Creating the link.* last -> next = last;  
  
 **return** last;  
}  
  
**struct** Node \*addBegin(**struct** Node \*last, **int** data)  
{  
 **if** (last == NULL)  
 **return** addToEmpty(last, data);  
  
 **struct** Node \*temp =  
 (**struct** Node \*)malloc(**sizeof**(**struct** Node));  
  
 temp -> data = data;  
 temp -> next = last -> next;  
 last -> next = temp;  
  
 **return** last;  
}  
  
**struct** Node \*addEnd(**struct** Node \*last, **int** data)  
{  
 **if** (last == NULL)  
 **return** addToEmpty(last, data);  
  
 **struct** Node \*temp =  
 (**struct** Node \*)malloc(**sizeof**(**struct** Node));  
  
 temp -> data = data;  
 temp -> next = last -> next;  
 last -> next = temp;  
 last = temp;  
  
 **return** last;  
}  
  
**struct** Node \*addAfter(**struct** Node \*last, **int** data, **int** item)  
{  
 **if** (last == NULL)  
 **return** NULL;  
  
 **struct** Node \*temp, \*p;  
 p = last -> next;  
 **do** {  
 **if** (p ->data == item)  
 {  
 temp = (**struct** Node \*)malloc(**sizeof**(**struct** Node));  
 temp -> data = data;  
 temp -> next = p -> next;  
 p -> next = temp;  
  
 **if** (p == last)  
 last = temp;  
 **return** last;  
 }  
 p = p -> next;  
 } **while**(p != last -> next);  
  
 cout << item << **" not present in the list."** << endl;  
 **return** last;  
  
}  
  
**void** traverse(**struct** Node \*last)  
{  
 **struct** Node \*p;  
  
 *// If list is empty, return.* **if** (last == NULL)  
 {  
 cout << **"List is empty."** << endl;  
 **return**;  
 }  
  
 *// Pointing to first Node of the list.* p = last -> next;  
  
 *// Traversing the list.* **do** {  
 cout << p -> data << **" "**;  
 p = p -> next;  
  
 }  
 **while**(p != last->next);  
  
}  
  
*// Driven Program***int** main()  
{  
 **struct** Node \*last = NULL;  
  
 last = addToEmpty(last, 6);  
 last = addBegin(last, 4);  
 last = addBegin(last, 2);  
 last = addEnd(last, 8);  
 last = addEnd(last, 12);  
 last = addAfter(last, 10, 8);  
  
 traverse(last);  
  
 **return** 0;  
}

**DOUBLE LINKED LIST**

* Advantages are-

We can traverse in both the directions

Easy to delete or insert node

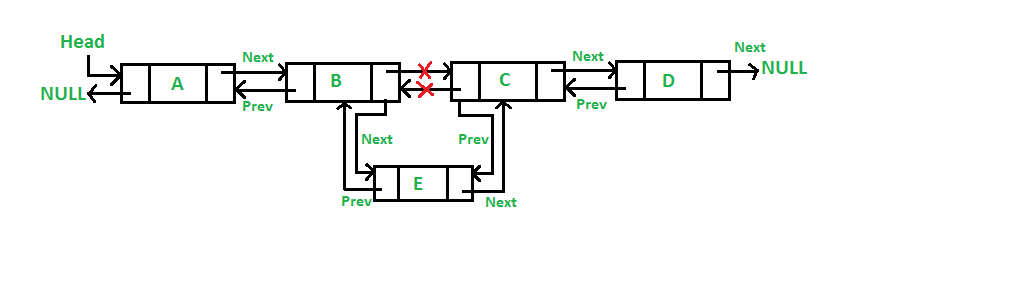


**1)inserting node at front**



*/\* Given a reference (pointer to pointer) to the head of a list   
and an int, inserts a new node on the front of the list. \*/***void** push(**struct** Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* **struct** Node\* new\_node = (**struct** Node\*)malloc(**sizeof**(**struct** Node));  
  
 */\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. Make next of new node as head and previous as NULL \*/* new\_node->next = (\*head\_ref);  
 new\_node->prev = NULL;  
  
 */\* 4. change prev of head node to new node \*/* **if** ((\*head\_ref) != NULL)  
 (\*head\_ref)->prev = new\_node;  
  
 */\* 5. move the head to point to the new node \*/* (\*head\_ref) = new\_node;  
}

**2)adding a node after a given node**



**Code for all approaches**

*// A complete working C++ program to   
// demonstrate all insertion methods*#include **<bits/stdc++.h>  
using namespace** std;  
  
*// A linked list node***class** Node  
{  
**public**:  
 **int** data;  
 Node\* next;  
 Node\* prev;  
};  
  
*/\* Given a reference (pointer to pointer) to the head of a list   
and an int, inserts a new node on the front of the list. \*/***void** push(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* Node\* new\_node = **new** Node();  
  
 */\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. Make next of new node as head and previous as NULL \*/* new\_node->next = (\*head\_ref);  
 new\_node->prev = **NULL**;  
  
 */\* 4. change prev of head node to new node \*/* **if** ((\*head\_ref) != **NULL**)  
 (\*head\_ref)->prev = new\_node;  
  
 */\* 5. move the head to point to the new node \*/* (\*head\_ref) = new\_node;  
}  
  
*/\* Given a node as prev\_node, insert a new node after the given node \*/***void** insertAfter(Node\* prev\_node, **int** new\_data)  
{  
 */\*1. check if the given prev\_node is NULL \*/* **if** (prev\_node == **NULL**)  
 {  
 cout<<**"the given previous node cannot be NULL"**;  
 **return**;  
 }  
  
 */\* 2. allocate new node \*/* Node\* new\_node = **new** Node();  
  
 */\* 3. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 4. Make next of new node as next of prev\_node \*/* new\_node->next = prev\_node->next;  
  
 */\* 5. Make the next of prev\_node as new\_node \*/* prev\_node->next = new\_node;  
  
 */\* 6. Make prev\_node as previous of new\_node \*/* new\_node->prev = prev\_node;  
  
 */\* 7. Change previous of new\_node's next node \*/* **if** (new\_node->next != **NULL**)  
 new\_node->next->prev = new\_node;  
}  
  
*/\* Given a reference (pointer to pointer) to the head   
of a DLL and an int, appends a new node at the end \*/***void** append(Node\*\* head\_ref, **int** new\_data)  
{  
 */\* 1. allocate node \*/* Node\* new\_node = **new** Node();  
  
 Node\* last = \*head\_ref; */\* used in step 5\*/  
  
 /\* 2. put in the data \*/* new\_node->data = new\_data;  
  
 */\* 3. This new node is going to be the last node, so   
 make next of it as NULL\*/* new\_node->next = **NULL**;  
  
 */\* 4. If the Linked List is empty, then make the new   
 node as head \*/* **if** (\*head\_ref == **NULL**)  
 {  
 new\_node->prev = **NULL**;  
 \*head\_ref = new\_node;  
 **return**;  
 }  
  
 */\* 5. Else traverse till the last node \*/* **while** (last->next != **NULL**)  
 last = last->next;  
  
 */\* 6. Change the next of last node \*/* last->next = new\_node;  
  
 */\* 7. Make last node as previous of new node \*/* new\_node->prev = last;  
  
 **return**;  
}  
  
*// This function prints contents of   
// linked list starting from the given node***void** printList(Node\* node)  
{  
 Node\* last;  
 cout<<**"\nTraversal in forward direction \n"**;  
 **while** (node != **NULL**)  
 {  
 cout<<**" "**<<node->data<<**" "**;  
 last = node;  
 node = node->next;  
 }  
  
 cout<<**"\nTraversal in reverse direction \n"**;  
 **while** (last != **NULL**)  
 {  
 cout<<**" "**<<last->data<<**" "**;  
 last = last->prev;  
 }  
}  
  
*/\* Driver program to test above functions\*/***int** main()  
{  
 */\* Start with the empty list \*/* Node\* head = **NULL**;  
  
 *// Insert 6. So linked list becomes 6->NULL* append(&head, 6);  
  
 *// Insert 7 at the beginning. So   
 // linked list becomes 7->6->NULL* push(&head, 7);  
  
 *// Insert 1 at the beginning. So   
 // linked list becomes 1->7->6->NULL* push(&head, 1);  
  
 *// Insert 4 at the end. So linked   
 // list becomes 1->7->6->4->NULL* append(&head, 4);  
  
 *// Insert 8, after 7. So linked   
 // list becomes 1->7->8->6->4->NULL* insertAfter(head->next, 8);  
  
 cout << **"Created DLL is: "**;  
 printList(head);  
  
 **return** 0;  
}

Reverse a linked list

<https://www.geeksforgeeks.org/reverse-a-linked-list/>

**1.HASH TABLE**

1.What is hashing and why do we do hashing

[**https://www.youtube.com/watch?v=wWgIAphfn2U&feature=youtu.be**](https://www.youtube.com/watch?v=wWgIAphfn2U&feature=youtu.be)

2.A simple example of hashing.

In this we create a big 2 dimensional matrix with big size(max).then we keep the index in that matrix as 1 which has same value as given array to make it as present.

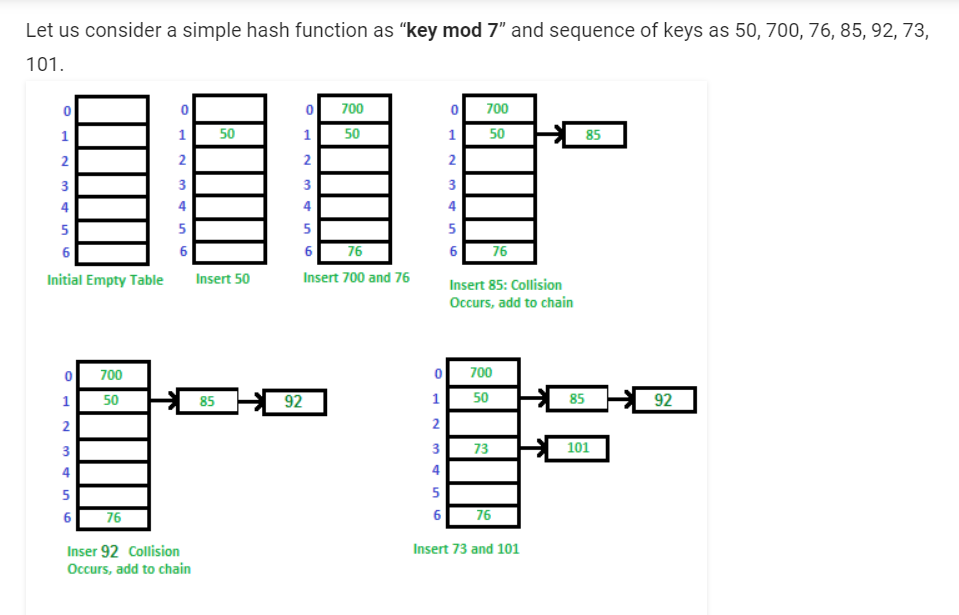
<https://drive.google.com/file/d/1AhJWjhnI5beY4TN80e-oqfhjiDqGysFx/view?usp=sharing>

3.Collision handling-

(a)Chaining-The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

(b)Open addressing-In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

(a) **Separate Chaining:**  
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.



Code- <https://www.geeksforgeeks.org/c-program-hashing-chaining/>

**Advantages:**  
1) Simple to implement.  
2) Hash table never fills up, we can always add more elements to the chain.  
3) Less sensitive to the hash function or load factors.  
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

**Disadvantages:**  
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.  
2) Wastage of Space (Some Parts of hash table are never used)  
3) If the chain becomes long, then search time can become O(n) in the worst case.  
4) Uses extra space for links.

(b) **Open addressing-**

(i) linear probing-If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

For eg-Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Challenges in Linear Probing :

1.Primary Clustering: One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element. 

2.Secondary Clustering*:*Secondary clustering is less severe, two records do only have the same collision chain(Probe Sequence) if their initial position is the same.

(ii) quadratic probing-let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S

If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S

If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S

..................................................

..................................................

(iii) double hashing-We use another hash function hash2(x) and look for i\*hash2(x) slot in i’th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S

If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S

If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S

..................................................

..................................................

**Comparison of above three:**   
Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.   
Quadratic probing lies between the two in terms of cache performance and clustering.   
Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

**STACKS**

Mainly the following three basic operations are performed in the stack:

* **Push:**Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
* **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
* **Peek or Top:** Returns top element of stack.
* **isEmpty:**Returns true if stack is empty, else false.



There are two ways to implement a stack:

* Using array
* Using linked list

**CODE:**

**Array**- <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

**Pros:** Easy to implement. Memory is saved as pointers are not involved.   
**Cons:** It is not dynamic. It doesn’t grow and shrink depending on needs at runtime.

**Linked List**-https://www.geeksforgeeks.org/stack-data-structure-introduction-program/

**Pros:** The linked list implementation of stack can grow and shrink according to the needs at runtime.   
**Cons:** Requires extra memory due to involvement of pointers.

STL CODE:

The functions associated with stack are:  
[empty()](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/) – Returns whether the stack is empty – Time Complexity : O(1)  
[size()](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/) – Returns the size of the stack – Time Complexity : O(1)  
[top()](https://www.geeksforgeeks.org/stack-top-c-stl/) – Returns a reference to the top most element of the stack – Time Complexity : O(1)  
[push(g)](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)  
[pop()](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) – Deletes the top most element of the stack – Time Complexity : O(1)

|  |
| --- |
| // CPP program to demonstrate working of STL stack  #include <bits/stdc++.h>  using namespace std;    void showstack(stack <int> s)  {      while (!s.empty())      {          cout << '\t' << s.top();          s.pop();      }      cout << '\n';  }    int main ()  {      stack <int> s;      s.push(10);      s.push(30);      s.push(20);      s.push(5);      s.push(1);        cout << "The stack is : ";      showstack(s);        cout << "\ns.size() : " << s.size();      cout << "\ns.top() : " << s.top();          cout << "\ns.pop() : ";      s.pop();      showstack(s);        return 0;  } |

**Output:**

The stack is : 1 5 20 30 10

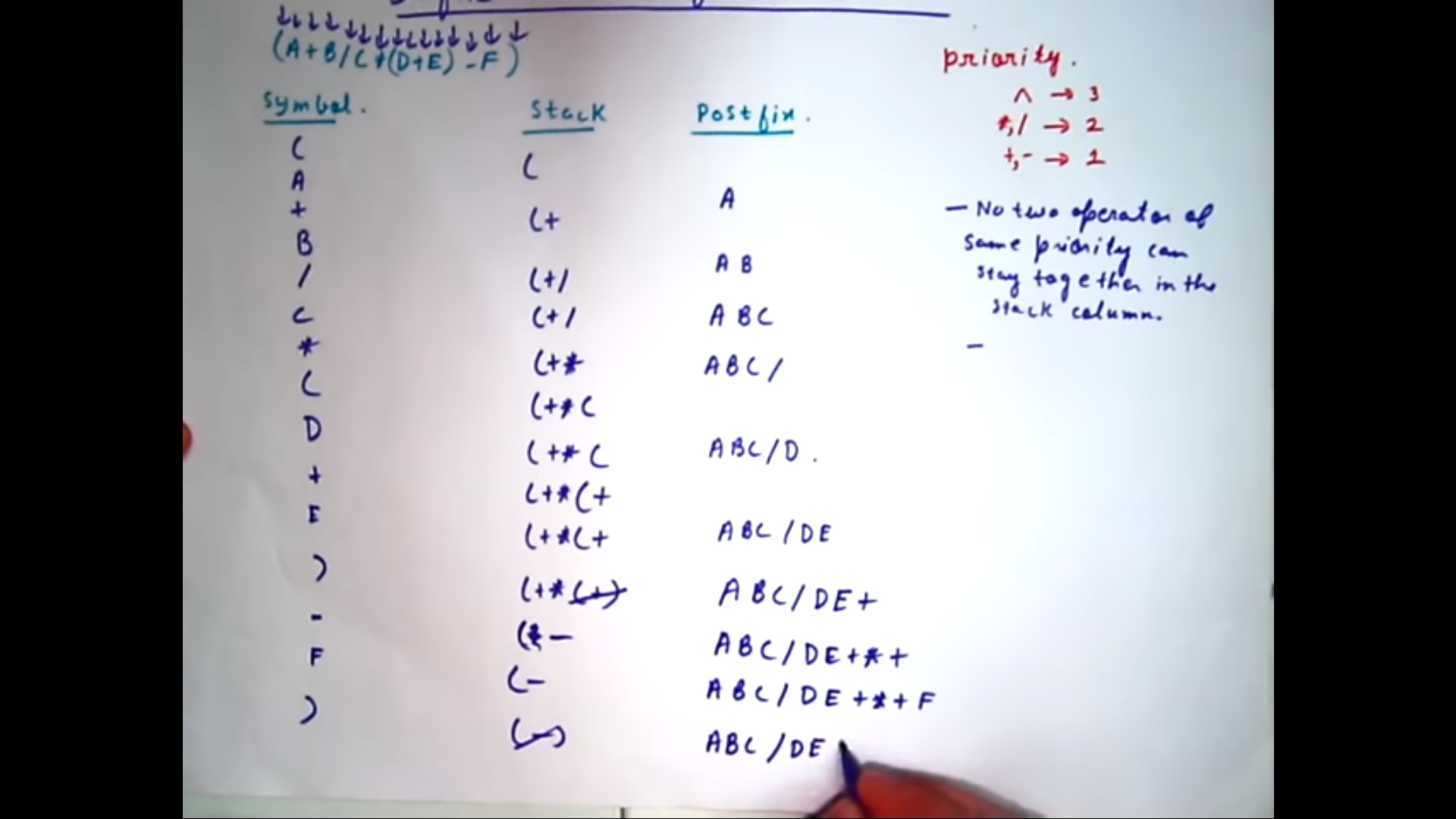
s.size() : 5

s.top() : 1

s.pop() : 5 20 30 10

Infix to Postflix

Algoritthm:



<https://www.youtube.com/watch?v=vXPL6UavUeA>

CODE:

<https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/?ref=lbp>

QUEUE

**Enqueue:**Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.  
**Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.  
**Front:**Get the front item from queue.  
**Rear:** Get the last item from queue.



CODE:

Array

<https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>

Linked List

<https://www.geeksforgeeks.org/queue-linked-list-implementation/>

QUEUE STL

**The functions supported by queue are :**

1. [empty()](https://www.geeksforgeeks.org/queueempty-queuesize-c-stl/) – Returns whether the queue is empty.
2. [size()](https://www.geeksforgeeks.org/queueempty-queuesize-c-stl/) – Returns the size of the queue.
3. [queue::swap() in C++ STL](https://www.geeksforgeeks.org/queue-swap-cpp-stl/): Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
4. [queue::emplace() in C++ STL](https://www.geeksforgeeks.org/queueemplace-c-stl/): Insert a new element into the queue container, the new element is added to the end of the queue.
5. [queue::front() and queue::back() in C++ STL](https://www.geeksforgeeks.org/queuefront-queueback-c-stl/)– **front()** function returns a reference to the first element of the queue. **back()** function returns a reference to the last element of the queue.
6. [push(g) and pop()](https://www.geeksforgeeks.org/queuepush-and-queuepop-in-cpp-stl/) – **push()** function adds the element ‘g’ at the end of the queue. **pop()** function deletes the first element of the queue.

CODE:

<https://www.geeksforgeeks.org/queue-cpp-stl/>

Priority Queue

Priority Queue is an extension of [queue](http://quiz.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/)with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.
4. A typical priority queue supports following operations.  
   **insert(item, priority):**Inserts an item with given priority.  
   **getHighestPriority():** Returns the highest priority item.  
   **deleteHighestPriority():**Removes the highest priority item.
5. **How to implement priority queue?**  
   ***Using Array:***A simple implementation is to use array of following structure.
6. struct item {
7. int item;
8. int priority;
9. }
10. insert() operation can be implemented by adding an item at end of array in O(1) time.
11. getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
12. deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.
13. We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don’t have to move items.

Dequeue

allows insert and delete at both ends.

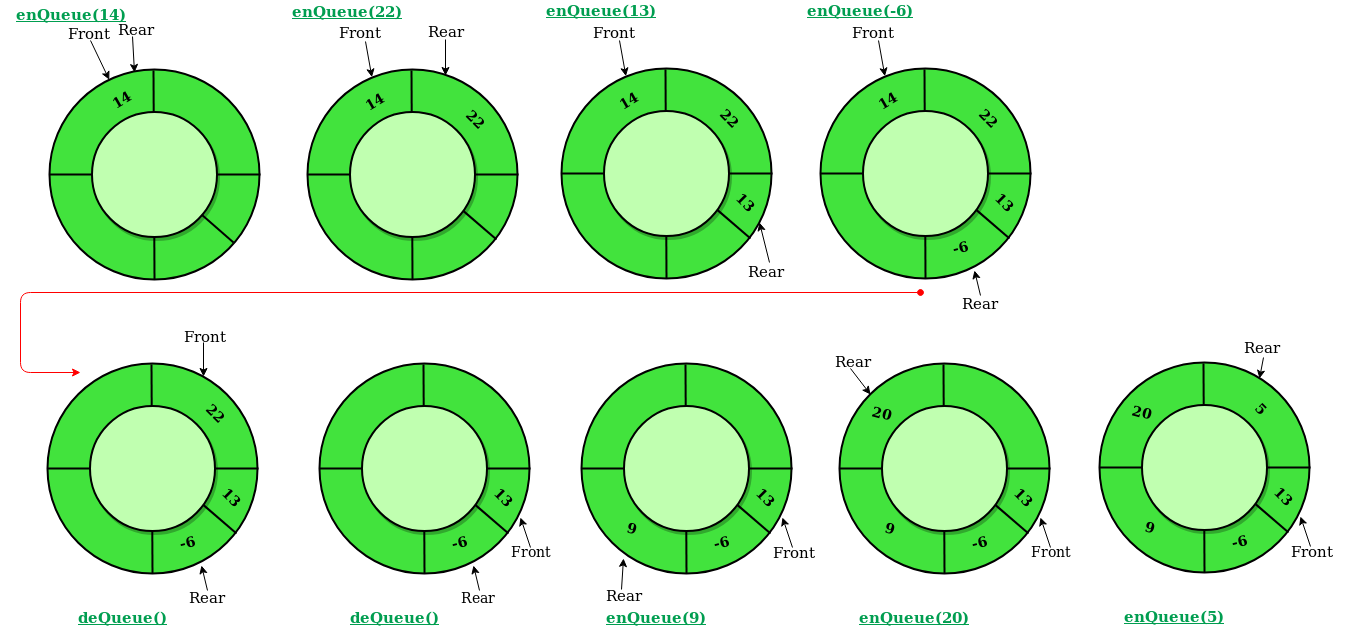
**Operations on Deque:**  
Mainly the following four basic operations are performed on queue:

***insertFront()***: Adds an item at the front of Deque.  
***insertLast()***: Adds an item at the rear of Deque.  
***deleteFront()***: Deletes an item from front of Deque.  
***deleteLast()***: Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported  
***getFront()***: Gets the front item from queue.  
***getRear()***: Gets the last item from queue.  
***isEmpty()***: Checks whether Deque is empty or not.  
***isFull()***: Checks whether Deque is full or not.

**Applications of Deque:**  
Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.  
Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque.

CIRCULAR QUEUE



Operations on Circular Queue:

* **Front:** Get the front item from queue.
* **Rear:** Get the last item from queue.
* **enQueue(value)**This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

**Steps:**

* 1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
* **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

**Steps:**

* 1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

**CODE:**

[**https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/**](https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/)