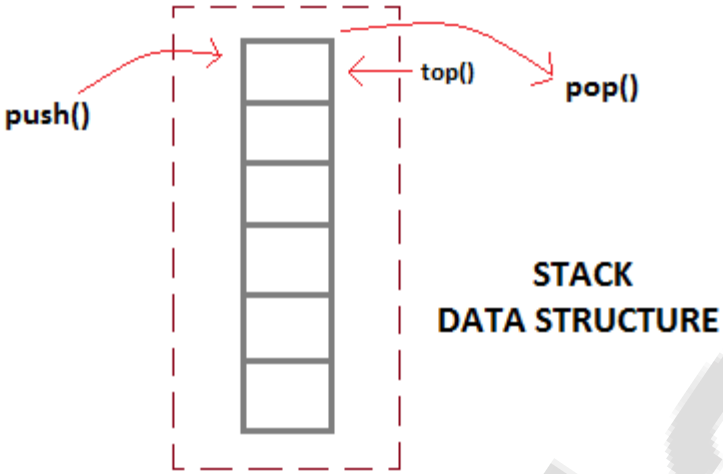
Stacks

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



Basic features of Stack

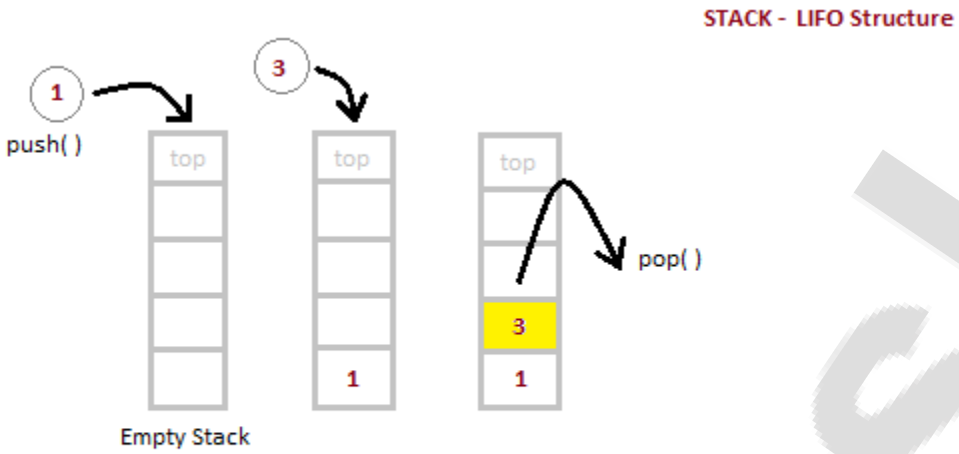
1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
There are other uses also like : **Parsing**, **Expression Conversion**(Infix to Postfix, Postfix to Prefix etc) and many more.

Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.
The "**pop**" operation removes the item on top of the stack.

```
/*  Below program is written in C++ language  */

Class Stack
{
  int top;
  public:
  int a[10];    //Maximum size of Stack
  Stack()
  {
    top = -1;
  }
};

void Stack::push(int x)
{
  if( top >= 10)
  {
    cout << "Stack Overflow";
  }
  else
  {
    a[++top] = x;
    cout << "Element Inserted";
  }
}

int Stack::pop()
{
  if(top < 0)
  {
    cout << "Stack Underflow";
    return 0;
  }
  else
  {
    int d = a[top--];
    return d;
  }
}
```

```
void Stack::isEmpty()
{
  if(top < 0)
  {
    cout << "Stack is empty";
  }
  else
  {
    cout << "Stack is not empty";
  }
}
```

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

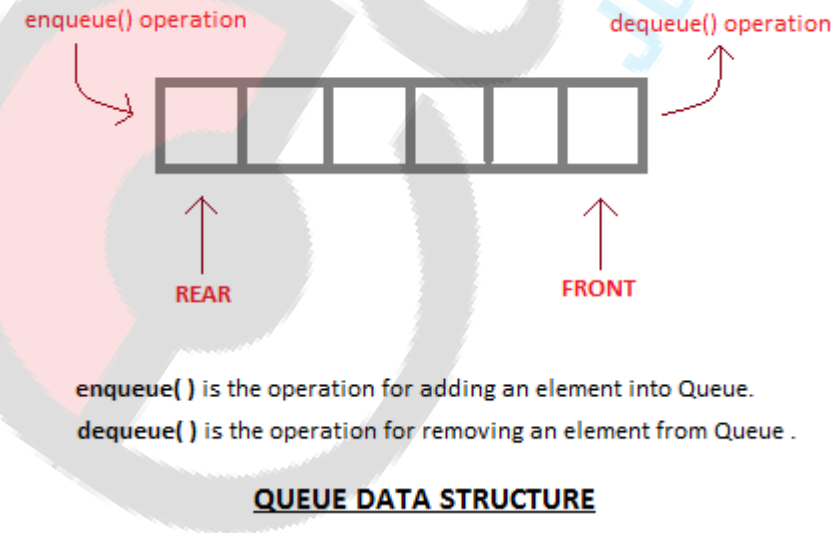
Analysis of Stacks

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

Queue Data Structures

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.
The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

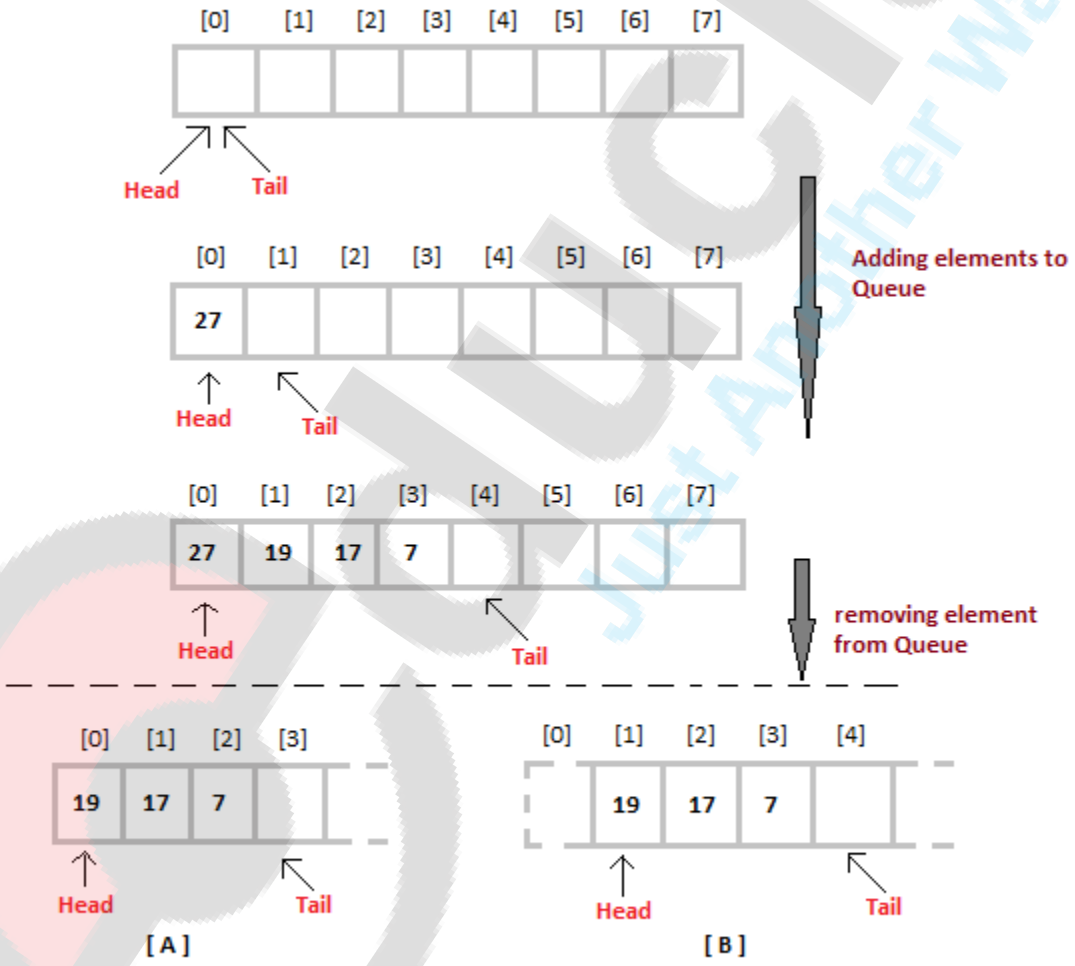
Applications of Queue

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

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Implementation of Queue

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



When we remove 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

move all the other elements on position forward. In approach [B] we remove the element from **head** position and then move **head** to the next position.
In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whener we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

```
/* Below program is wtitten in C++ language */

#define SIZE 100
class Queue
{
  int a[100];
  int rear;     //same as tail
  int front;    //same as head

  public:
  Queue()
  {
    rear = front = -1;
  }
  void enqueue(int x);     //declaring enqueue, dequeue and display functions
  int dequeue();
  void display();
}

void Queue :: enqueue(int x)
{
  if( rear = SIZE-1)
  {
    cout << "Queue is full";
  }
  else
  {
    a[++rear] = x;
  }
}

int queue :: dequeue()
{
  return a[++front];     //following approach [B], explained above
}

void queue :: display()
{
  int i;
  for( i = front; i <= rear; i++)
  {
    cout << a[i];
  }
}
```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements one position.

```
return a[0];      //returning first element
for (i = 0; i < tail-1; i++)      //shifting all other elements
{
  a[i]= a[i+1];
  tail--;
}
```

Analysis of Queue

- Enqueue : **O(1)**
- Dequeue : **O(1)**

- Size : **O(1)**

Queue Data Structure using Stack

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. Hence we can implement a Queue using Stack for storage instead of array.

For performing **enqueue** we require only one stack as we can directly **push** data into 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, we will call them InStack and OutStack.

```
class Queue {
  public:
  Stack S1, S2;
  //defining methods

  void enqueue(int x);

  int dequeue();
}
```

We know that, Stack is a data structure, in which data can be added using **push()** method and data can be deleted using **pop()** method. To learn about Stack, follow the link : Stack Data Structure

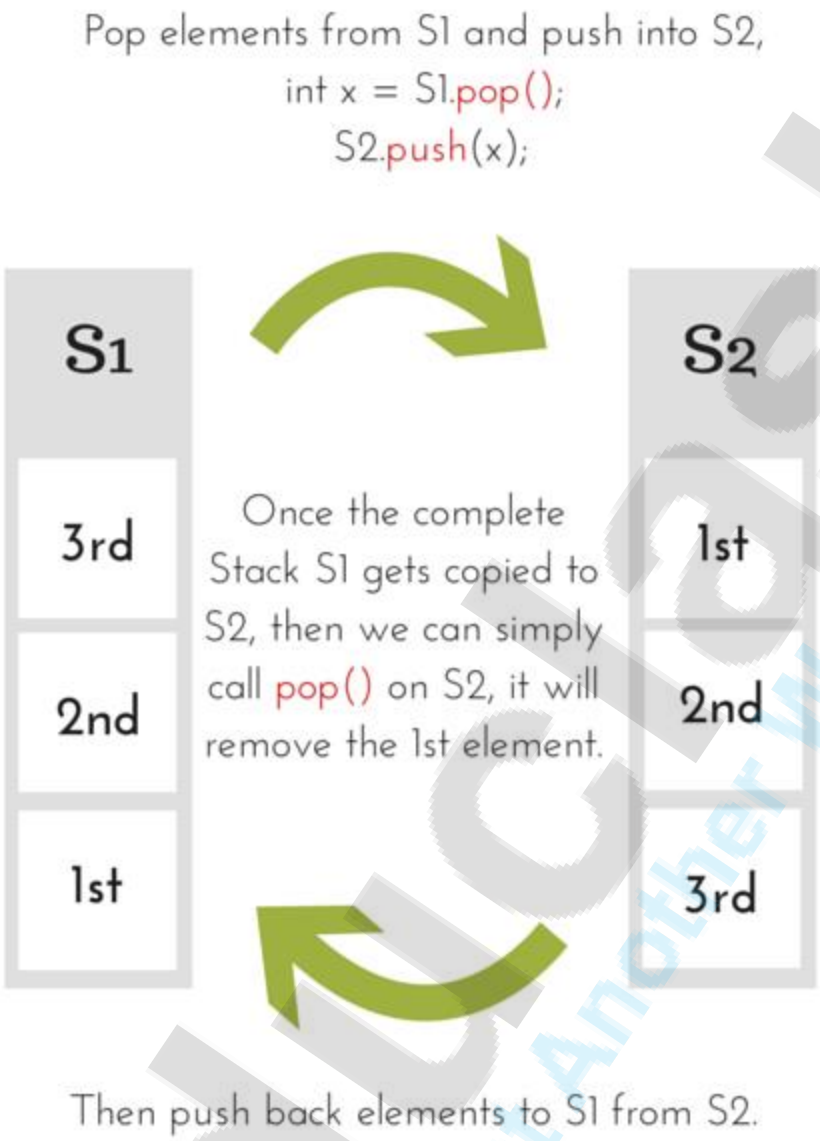
Adding Data to Queue

As our Queue has Stack for data storage in place of arrays, hence we will be adding data to Stack, which can be done using the push() method, hence :

```
void Queue :: enqueue(int x) {
  S1.push(x);
}
```

Removing Data from Queue

When we say remove data from Queue, it always means taking out the First element first and so on, as we have to follow the FIFO approach. But if we simply perform S1.pop() in our **dequeue** method, then it will remove the Last element first. So what to do now?




```
int Queue :: dequeue() {
  while(S1.isEmpty()) {
    x = S1.pop();
    S2.push();
  }

  //removing the element
  x = S2.pop();

  while(!S2.isEmpty()) {
    x = S2.pop();
    S1.push(x);
  }

  return x;
}
```

Introduction to Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

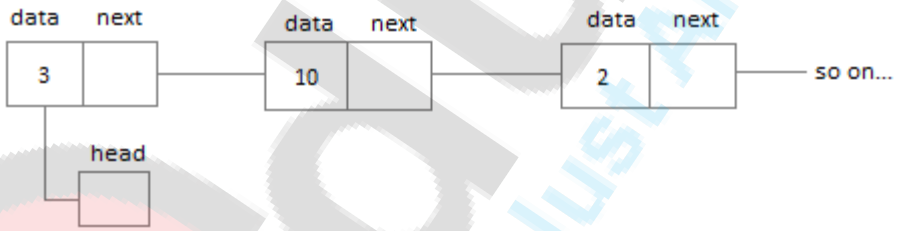
- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

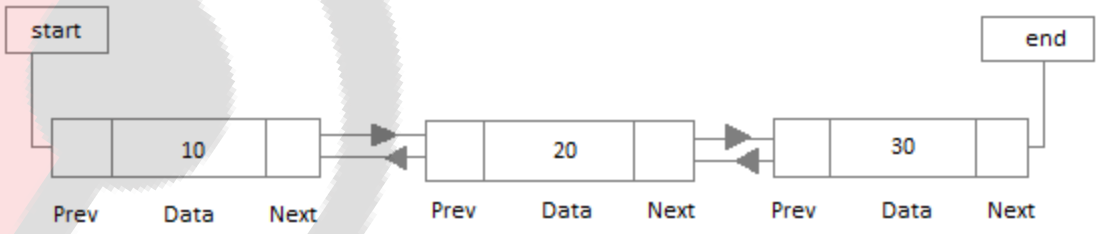
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

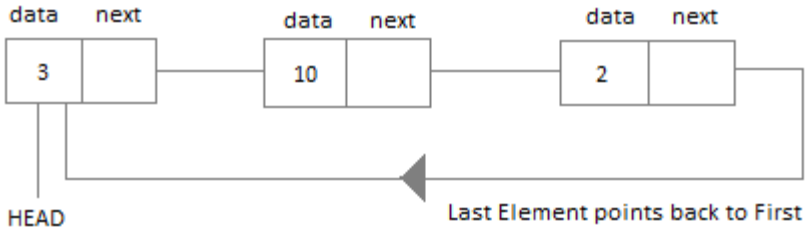
- **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



- **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



- **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



Linear Linked List

The element can be inserted in linked list in 2 ways :

- Insertion at beginning of the list.
- Insertion at the end of the list.

We will also be adding some more useful methods like :

- Checking whether Linked List is empty or not.
- Searching any element in the Linked List
- Deleting a particular Node from the List

Before inserting the node in the list we will create a class **Node**. Like shown below :

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;

  node() {
    data = 0;
    next = NULL;
  }

  node(int x) {
    data = x;
    next = NULL;
  }
}
```

We can also make the properties data and next as private, in that case we will need to add the getter and setter methods to access them. You can add the getters and setter like this :

```
int getData() {
  return data;
}

void setData(int x) {
  this.data = x;
}

node* getNext() {
  return next;
}

void setNext(node *n) {
  this.next = n;
}
```

Node class basically creates a node for the data which you enter to be included into Linked List. Once the node is created, we use various functions to fit in that node into the Linked List.

Linked List class

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all its methods. Following will be the Linked List class :

```
class LinkedList {
  public:
  node *head;
```

```
 //declaring the functions

 //function to add Node at front
 int addAtFront(node *n);
 //function to check whether Linked list is empty
 int isEmpty();
 //function to add Node at the End of list
 int addAtEnd(node *n);
 //function to search a value
 node* search(int k);
 //function to delete any Node
 node* deleteNode(int x);

 LinkedList() {
  head = NULL;
 }
}
```

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int LinkedList :: addAtFront(node *n) {
 int i = 0;
 //making the next of the new Node point to Head
 n->next = head;
 //making the new Node as Head
 head = n;
 i++;
 //returning the position where Node is added
 return i;
}
```

Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd(node *n) {
 //If list is empty
 if(head == NULL) {
  //making the new Node as Head
  head = n;
  //making the next pointe of the new Node as Null
  n->next = NULL;
 }
 else {
  //getting the last node
  node *n2 = getLastNode();
  n2->next = n;
 }
}

node* LinkedList :: getLastNode() {
 //creating a pointer pointing to Head
```

```
 node* ptr = head;
 //Iterating over the list till the node whose Next pointer points to null
 //Return that node, because that will be the last node.
 while(ptr->next!=NULL) {
  //if Next is not Null, take the pointer one step forward
  ptr = ptr->next;
 }
 return ptr;
}
```

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* LinkedList :: search(int x) {
 node *ptr = head;
 while(ptr != NULL && ptr->data != x) {
  //until we reach the end or we find a Node with data x, we keep moving
  ptr = ptr->next;
 }
 return ptr;
}
```

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {
 //searching the Node with data x
 node *n = search(x);
 node *ptr = head;
 if(ptr == n) {
  ptr->next = n->next;
  return n;
 }
 else {
  while(ptr->next != n) {
   ptr = ptr->next;
  }
  ptr->next = n->next;
  return n;
 }
}
```

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is NULL or not.

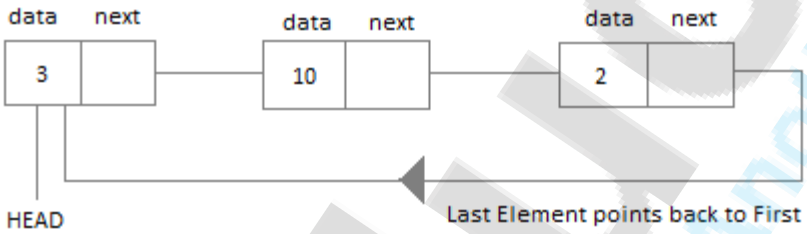
```
int LinkedList :: isEmpty() {
 if(head == NULL) {
  return 1;
 }
 else { return 0; }
}
```

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.
If you are still figuring out, how to call all these methods, then below is how your main() method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

```
int main() {
  LinkedList L;
  //We will ask value from user, read the value and add the value to our Node
  int x;
  cout << "Please enter an integer value : ";
  cin >> x;
  Node *n1;
  //Creating a new node with data as x
  n1 = new Node(x);
  //Adding the node to the list
  L.addAtFront(n1);
}
```

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.
So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;
```

```
  node() {
    data = 0;
    next = NULL;
  }

  node(int x) {
    data = x;
    next = NULL;
  }
}
```

Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

```
class CircularLinkedList {
  public:
  node *head;
  //declaring the functions

  //function to add Node at front
  int addAtFront(node *n);
  //function to check whether Linked list is empty
  int isEmpty();
  //function to add Node at the End of list
  int addAtEnd(node *n);
  //function to search a value
  node* search(int k);
  //function to delete any Node
  node* deleteNode(int x);

  CircularLinkedList() {
    head = NULL;
  }
}
```

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int CircularLinkedList :: addAtFront(node *n) {
  int i = 0;
  /* If the list is empty */
  if(head == NULL) {
    n->next = head;
    //making the new Node as Head
    head = n;
    i++;
  }
  else {
    n->next = head;
    //get the Last Node and make its next point to new Node
    Node* last = getLastNode();
    last->next = n;
    //also make the head point to the new first Node
    head = n;
```

```
  i++;
 }
 //returning the position where Node is added
 return i;
}
```

Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```
int CircularLinkedList :: addAtEnd(node *n) {
 //If list is empty
 if(head == NULL) {
  //making the new Node as Head
  head = n;
  //making the next pointer of the new Node as Null
  n->next = NULL;
 }
 else {
  //getting the last node
  node *last = getLastNode();
  last->next = n;
  //making the next pointer of new node point to head
  n->next = head;
 }
}
```

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* CircularLinkedList :: search(int x) {
 node *ptr = head;
 while(ptr != NULL && ptr->data != x) {
  //until we reach the end or we find a Node with data x, we keep moving
  ptr = ptr->next;
 }
 return ptr;
}
```

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

```
node* CircularLinkedList :: deleteNode(int x) {
 //searching the Node with data x
 node *n = search(x);
 node *ptr = head;
 if(ptr == NULL) {
  cout << "List is empty";
  return NULL;
 }
```

```
  else if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```