Linear Linked List

Linear Linked list is the default linked list and a linear data structure in which data is not stored in contiguous memory locations but each data node is connected to the next data node via a pointer, hence forming a chain.

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

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

Hence while writing the code for Linked List we will include methods to insert or add new data elements to the linked list, both, at the beginning of the list and at the end of the list.

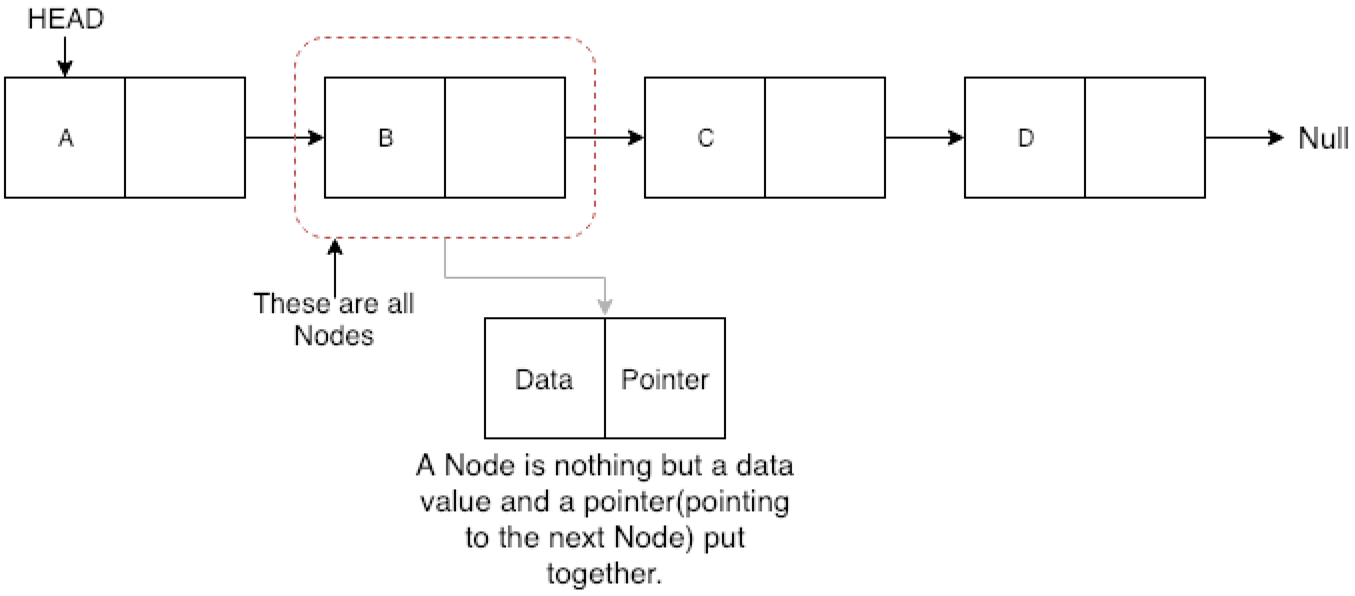
We will also be adding some other useful methods like:

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

Before learning how we insert data and create a linked list, we must understand the components forming a linked list, and the main component is the **Node**.

What is a Node?

A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.


HEAD
A
B
C
D
Null
These are all Nodes
Data
Pointer
A Node is nothing but a data value and a pointer(pointing to the next Node) put together.

In the picture above we have a linked list, containing 4 nodes, each node has some data(A, B, C and D) and a pointer which stores the location of the next node.

You must be wondering **why we need to store the location of the next node**. Well, because the memory locations allocated to these nodes are not contiguous hence each node should know where the next node is stored.

As the node is a combination of multiple information, hence we will be defining a class for `Node` which will have a variable to store **data** and another variable to store the **pointer**. In C language, we create a structure using the `struct` keyword.

```
class Node
```

```
{
    public:
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;

    // default constructor
    Node()
    {
        data = 0;
        next = NULL;
    }

    // parameterised constructor
    Node(int x)
    {
        data = x;
        next = NULL;
    }
}
```

Copy

We can also make the `Node` class properties `data` and `next` as **private**, in that case we will need to add the getter and setter methods to access them(don't know what getter and setter methods are: Inline Functions in C++). You can add the getter and setter functions to the `Node` class like this:

```
class Node
{
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;
    // default constructor same as above

    // parameterised constructor same as above

    /* getters and setters */
    // get the value of data
    int getData()
    {
        return data;
    }

    // to set the value for data
    void setData(int x)
    {
        this.data = x;
    }
    // get the value of next pointer
    node* getNext()
    {
        return next;
    }
    // to set the value for pointer
    void setNext(node *n)
    {
        this.next = n;
    }
}
```

Copy

The `Node` class basically creates a node for the data to be included into the Linked List. Once the object for the class `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 the methods like insertion, search, deletion etc. Also, the linked list class will have a pointer called `head` to store the location of the first node which will be added to the linked list.

```
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;
    }
}
```

Copy

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;

}
```

Copy

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;
}
```

Copy

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;

}
```

Copy

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;
  }
}
```

Copy

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; }
}
```

Copy

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);
}
```

Copy

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