Linear Data Structure

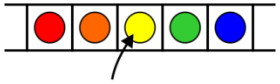
- Sequential Data Structure
- Linked Data Structure

Acknowlege: www.tutorialspoint.com and course materials from Prof. Ming Zhang at Tsinghua

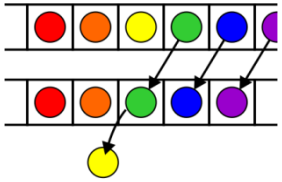
Operations

Operations at the k^{th} entry of the list include:

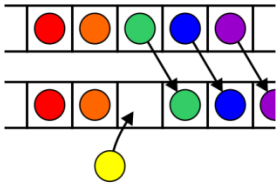
Access to the object



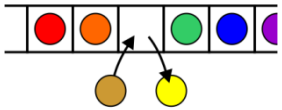
Erasing an object



Insertion of a new object

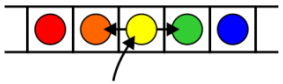


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



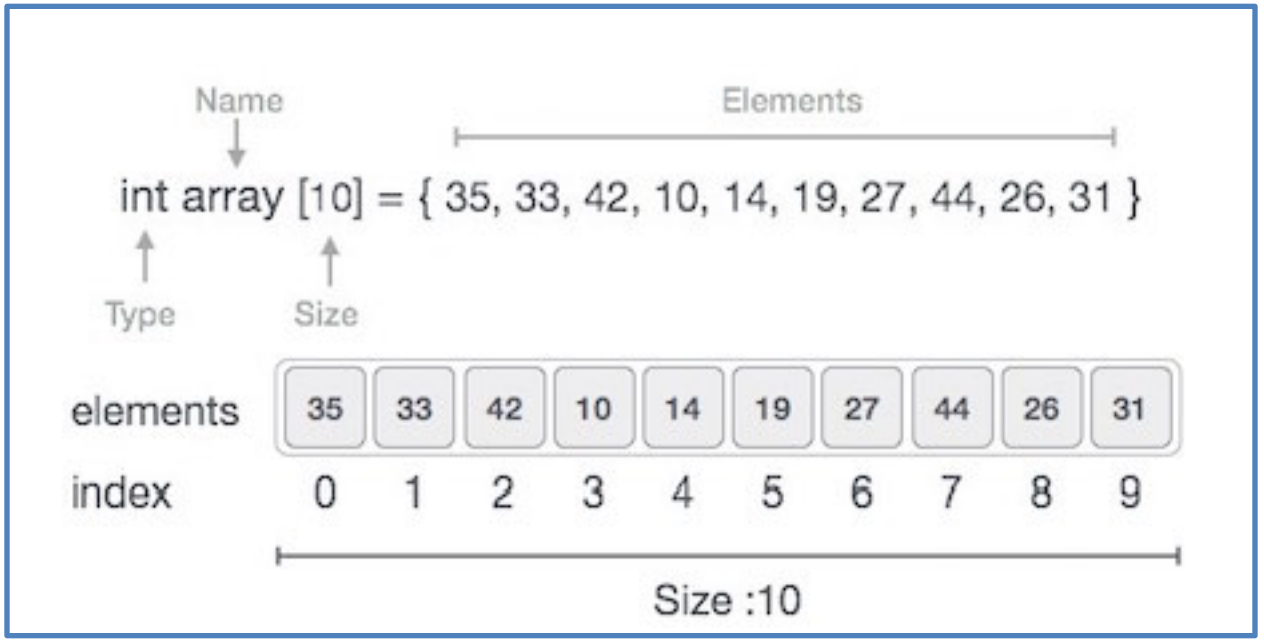
Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

Sequential Data Structure

Characteristics:

- Sequential data structure (such as array) holds elements that have the same data type
- Sequential data structure elements are stored in subsequent memory location
- Variable name represents the address of the starting element
- The size should be mentioned in the declaration. Array size must be a constant expression and not a variable.



Cite from: www.tutorialspoint.com

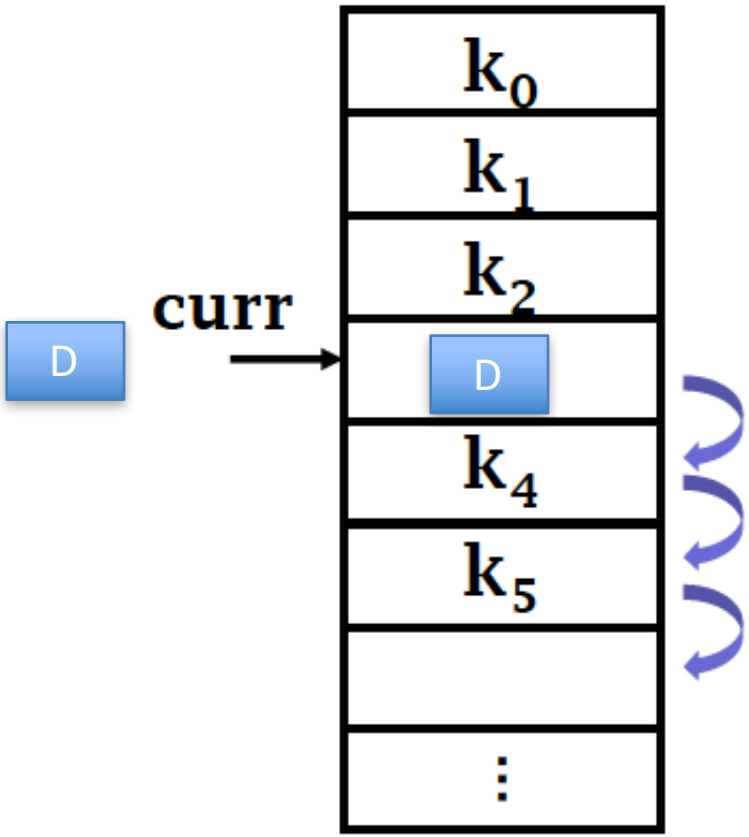
Definition:

```
template <class T>                                  // Sequential List
class seqList
{
    private:
        T *aList;                                   // Pointer to the data
        int maxSize;                                // Max size of the list
        int curLen;                                 // Current size of the list
        int curPos;                                 // Current position
    public:
        seqList(const int size)                     // Create a new list
            {
                maxSize = size;
                aList = new T[maxSize];
                curLen = curPos =0;
            }
        ~seqList()                                  // Deconstruct function
            {
                delete [] seqList;
            }
}
```

List of operations:

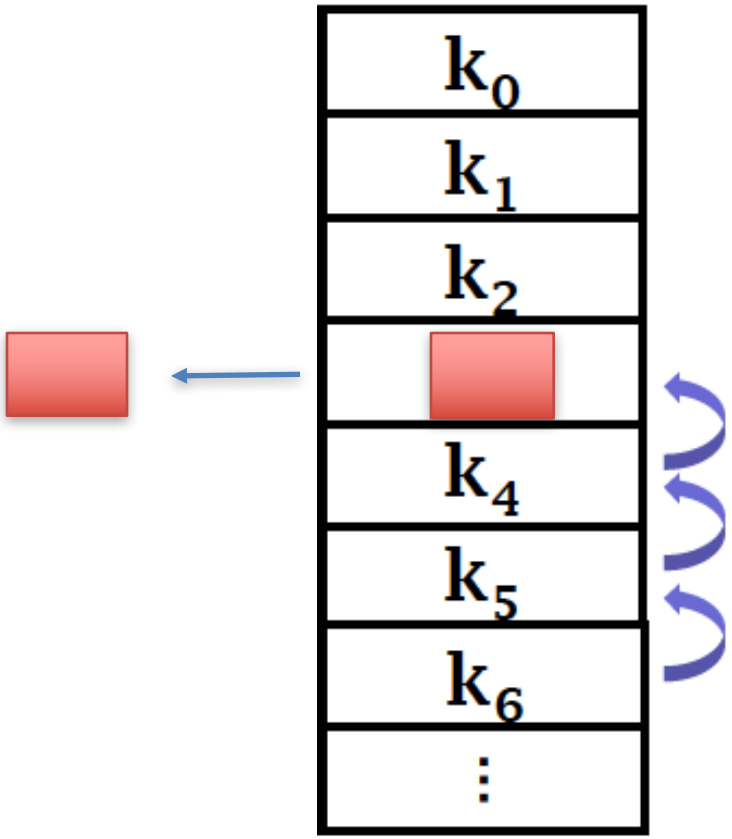
```
/* Selected List of operation functions*/
void clear() {};    // Clear the content of the list
bool insert(const int p, const T value);  //insert an element at position p
bool append(const T value); // append an element at the end of the list
bool delete(const int p);   // delete an element at position p
bool setValue(const int p, const T& value) // set the value of the element at position
bool getValue(const int p, T& value); // get the value of the element
bool getPos(int &p, const T value); //return the position of an element
```

Insertion:



```
// Insertion function
template <class T> bool seqList<T>::insert(const int p, const T value)
{
    int k;
    if (curLen >= maxSize) // check if there is a room to insert an element
    {
        cout << "Overflow found!"<< endl; return false;
    }

    if(p <0 || p> curLen)  // check if the inseration position is legal
    {
        cout << "Illegal inseration position"<<endl; return false;
    }

    for (k = curLen; k > p; k--)
        aList[k] = aList[k-1];  // move elements from p to curLen one position to right
    aList[p] = value;           // assign new value to p position
    curLen++;                   // increase the current size by 1
    return true;

}
```

Deletion:

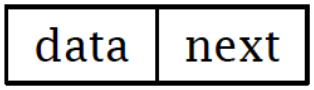


```
// Deletion function
template <class T> bool seqList<T>::delete(const int p)
{
    int k;
    if (curLen <= 0) // check if the list is empty
    {
        cout << "Empty list, no element to be deleted"<< endl; return false;
    }

    if(p <0 || p> curLen -1)  // check if the inseration position is legal
    {
        cout << "Illegal deletion position"<<endl; return false;
    }

    for (k = p; k < curLen; k++)
        aList[k] = aList[k+1];  // move elements from p to curLen one position to right
    aList[p] = value;           // assign new value to p position
    curLen--;                   // decrease the current size by 1
    return true;

}
```

Q1: What would be the run time complexity of inserting and deleting an element?

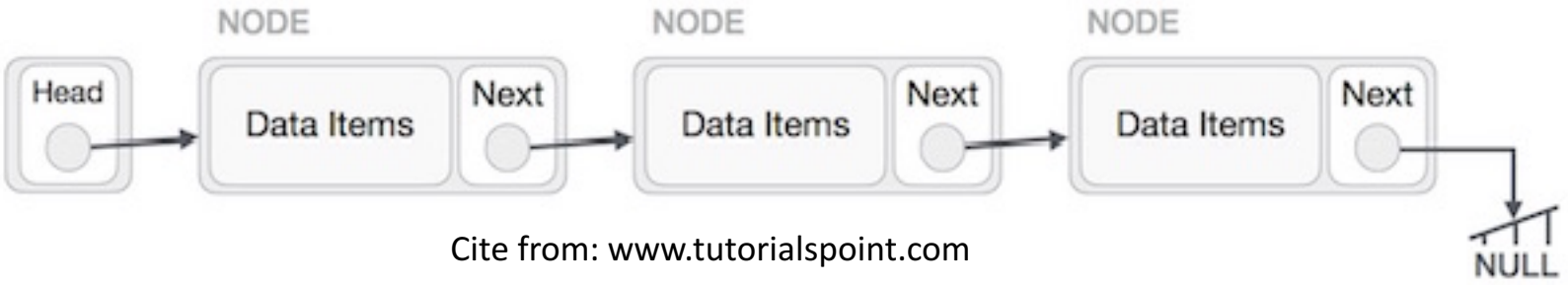
Linked Data Structure

A linked list is a sequence of data structures, which are connected together via links.

Node: 

(Data and Pointer)

Single Linked List



Cite from: www.tutorialspoint.com

As per the above illustration, following are the important points to be considered.

- Head: Only contains the pointer called Next.
- Node: Each link carries a data field(s) and a pointer called Next.
- Each node is linked with its Next pointer.
- Tail node: carries a pointer Next as null

Define a node structure:

```
template <class T>
class node                                  //Define a node structure
{
    public:
        T data;
        node<T> *next;
    node(const T info, const node<T>* nextValue =NULL)
        {
            data = info;                    //node assignment
            next = nextValue;
        }
    node(const node<T> * nextValue)
        {
            next = nextValue;               // head assignment
        }
}
```

Define a lnkList structure:

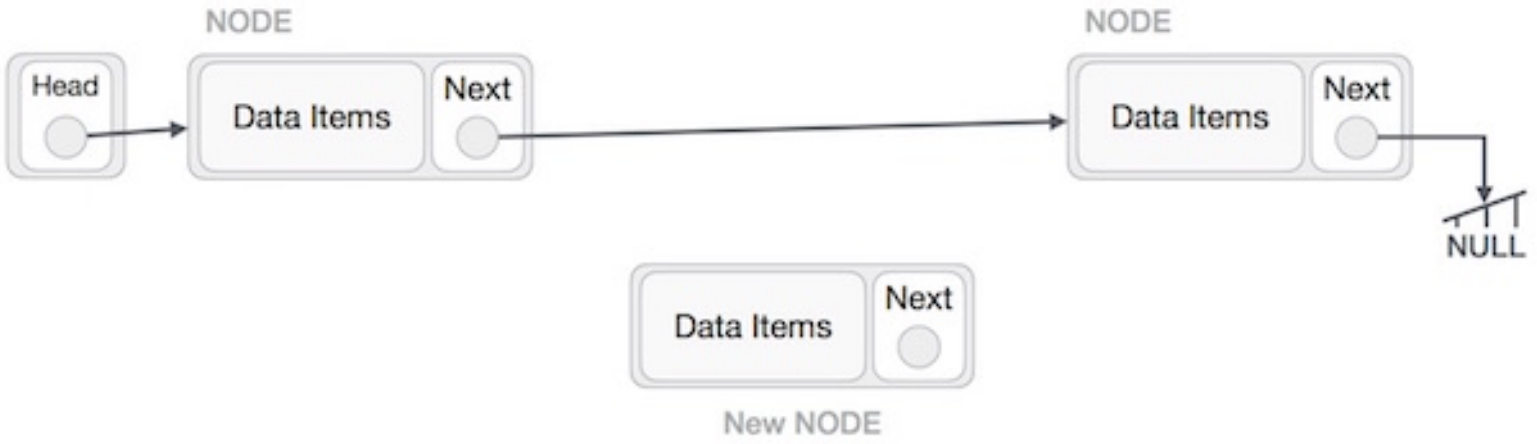
```
template <class T>
class lnkList
{
    private:
        node<T> *head, *tail; // Head and Tail Pointer
    public:
        lnkList();  //construct function
        ~lnkList(); //deconstruct function
        node<T> *setPos(const int p); //return the pointer of pth  node
        bool insert(const int p, const T value); //insert a node at position p
        bool delete(const int p); //delete the node at position p
        bool getValue(const int p, T& value); //return the data value of p node
        bool getPos(int &p, const T value); // return the position of the node (value)
}
```

Return pointer:

```
//Return the pointer to a node

template <class T>
node<T> *lnkList<T>::setPos(int i)
    {
        int count = 0;
        if(i == -1)                 // return pointer of head node
            return head;
        node<T> *p = head->next; // i == 0 point the first node
        while(p !=NULL && count <i)
            {
                p = p->next;
                count++;
            };
        return p;
    }
```

Insertion Operation

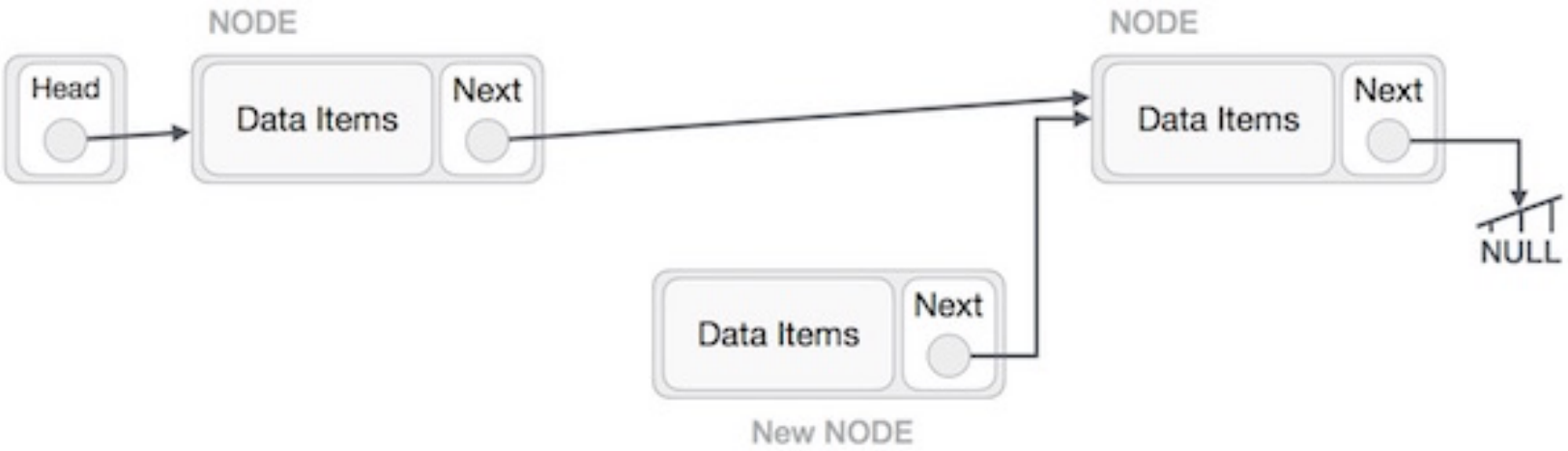
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

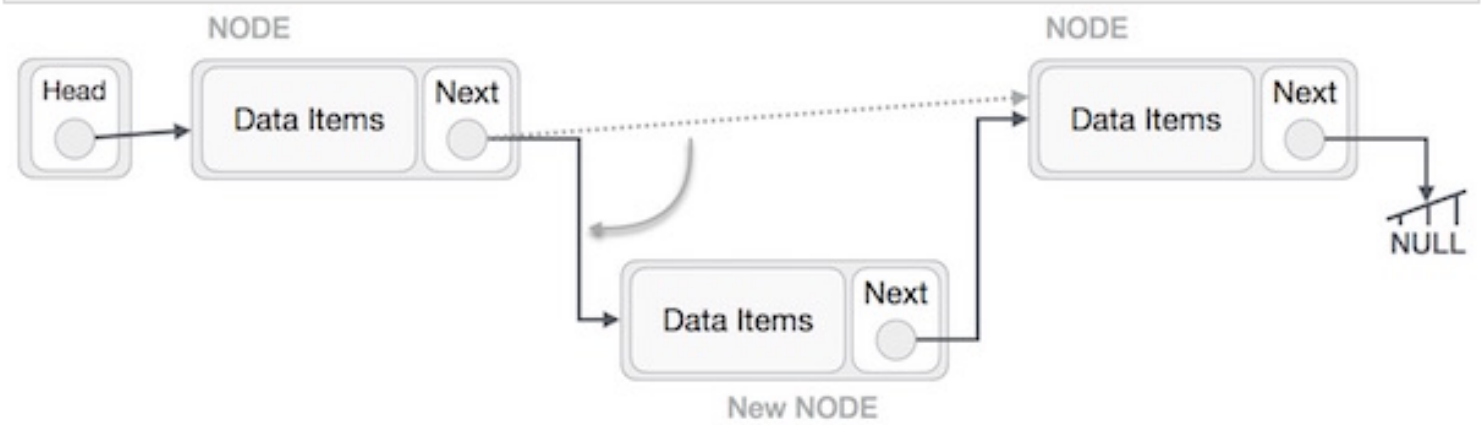
```
NewNode.next -> RightNode;
```

It should look like this –

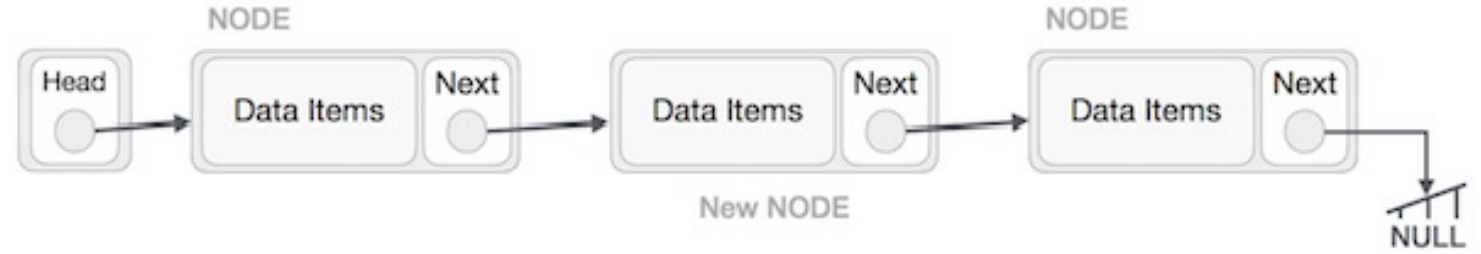


Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```



This will put the new node in the middle of the two. The new list should look like this –



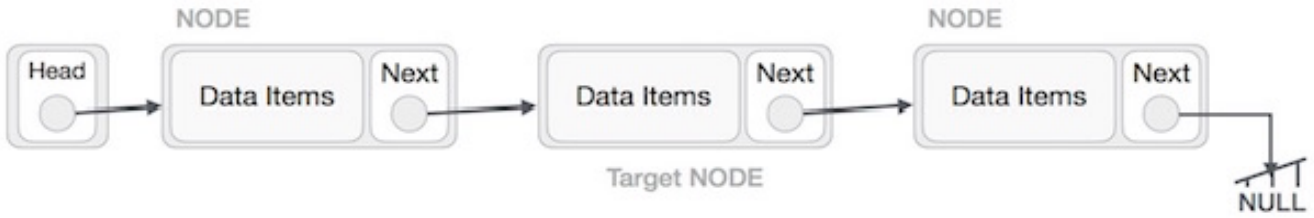
Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Insertion Function:

```
template <class T>
bool lnkList<T>:: insert(const int i, const T value)
    {
        node<T> *p, *q;
        if((p = setPos(i-1)) == NULL) //check if the inseration is valid
            {
                cout << "invalid inseration point"<<endl;
                return false;
            }
        q = new node<T>(value, p->next);
        p->next = q;
        if(p == tail)                    //insert at tail
            tail = q;
        return true;
    }
```

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



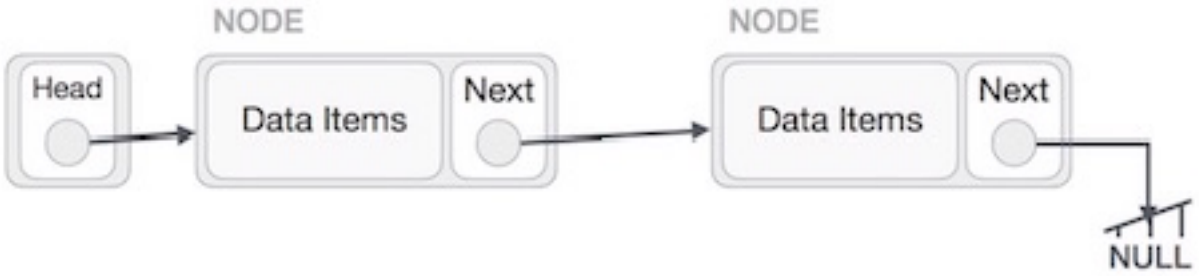
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

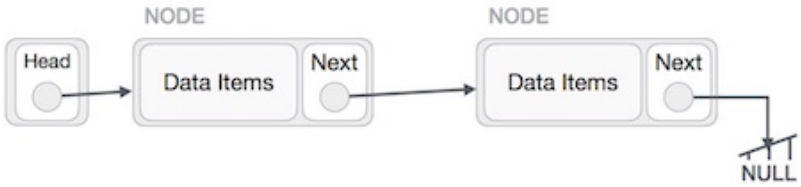
We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



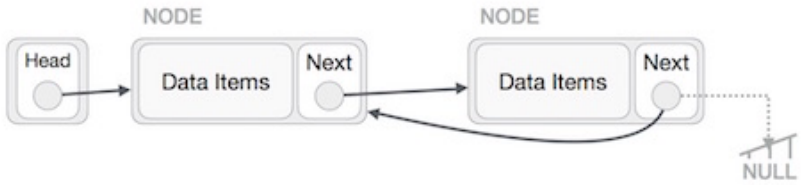
```
template <class T>
bool lnkList<T>::delete(const int i)
    {
        node<T> *p,*q; //create new nodes
        if ((p = setPos(i-1))==NULL || p ==tail) //check if the deletion node is
                                                 // valid
            {
                cout << "Invalid deletion" <<endl;
                return false;
            }
        q = p->next; // q is the node to be deleted
        if(q == tail)
        {
            tail = p;
            p->next = NULL;
        }
        else
            p->next = q->next;
        delete q;
        return true;
    }
```

Reverse Operation

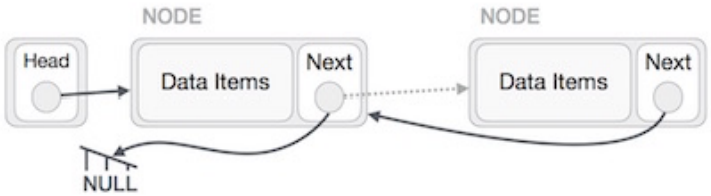
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



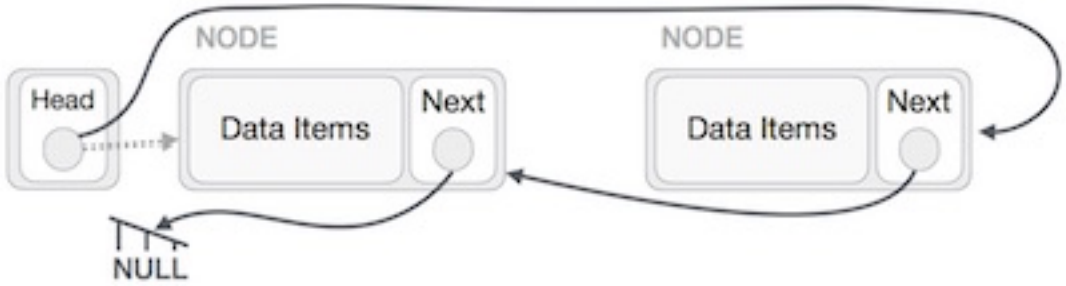
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



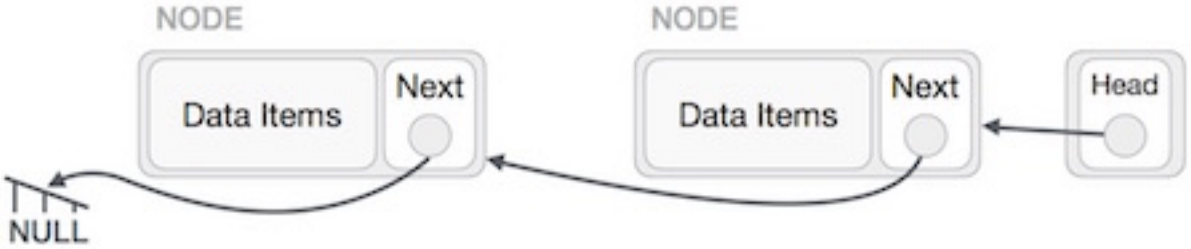
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.

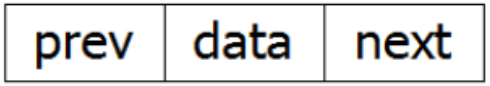


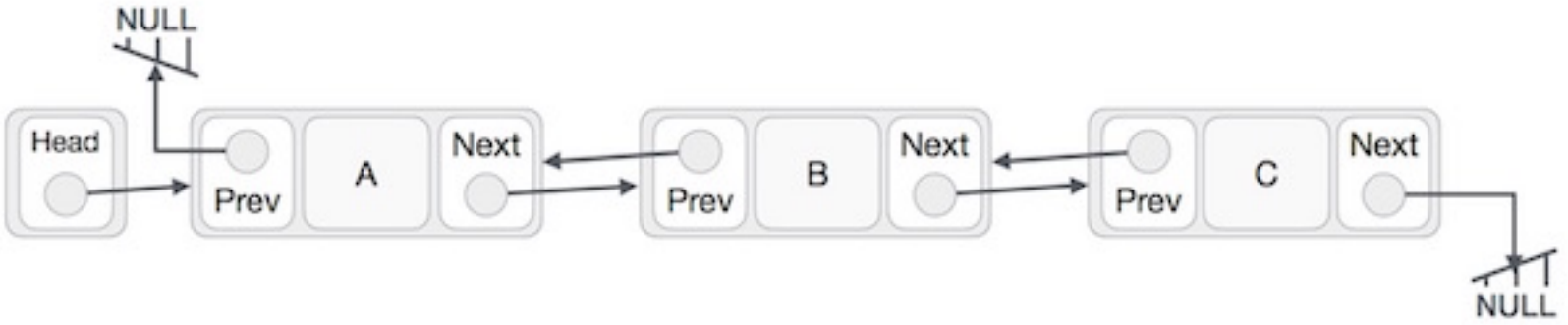
Implementation:

```
class node
{
    int data;
    node* next;
}

void reverse()
{
    node *cur, *prev, *next;
    cur = head;
    prev = NULL;
    while(cur)
    {
        next = cur->next;   //preserve the current node location
        cur->next = prev;   //point current node to previous node (reverse )
        prev = cur;         //change current to previous node for next iternary
        cur = next;         //change "next" node to "cur" node for next iternary
    }
}
```

Double Linked List

Node:

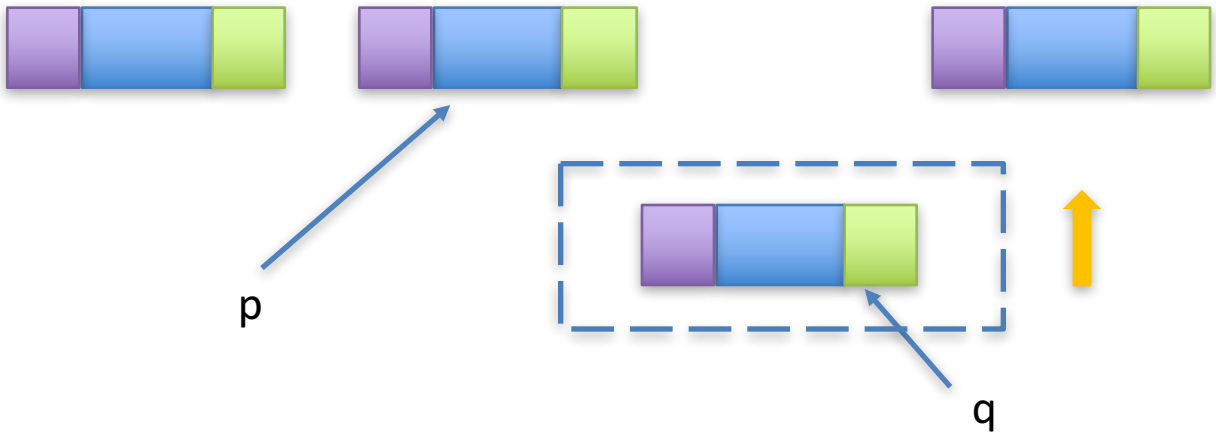




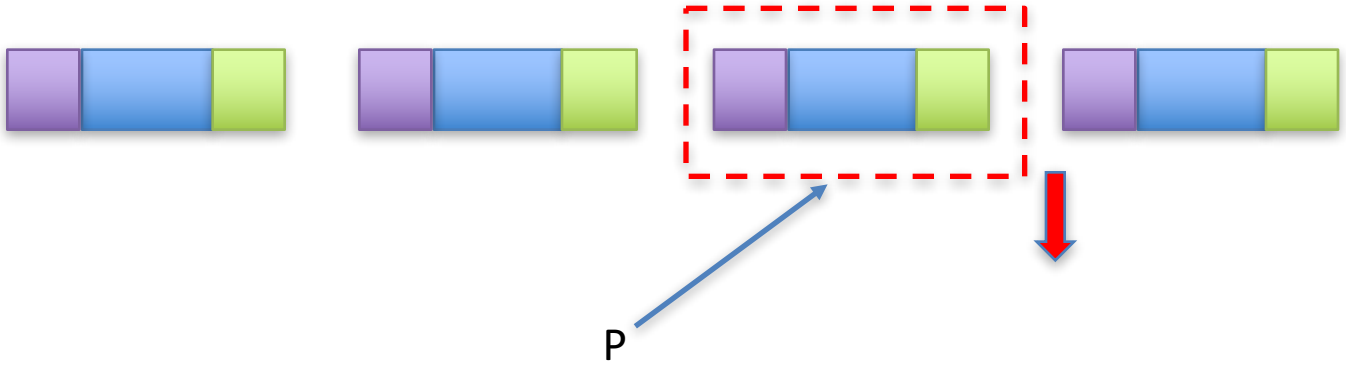
Definition of Node:

```
template <class T>
class node
{
    public:
        T data;
        node<T> *next;
        node<T> *prev;
}
```

Insertion




```
new q;
q->next = p->next;
q->prev = p;
p->next =q;
q->next->prev = q;
```

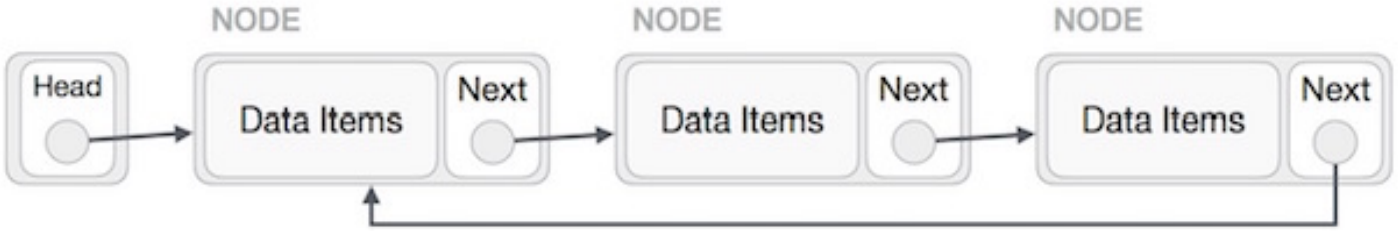



```
p->prev->next = p->next;
p->next->prev = p->prev;
p->next = NULL;
p->prev = NULL;
```

Circular Linked List

Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.