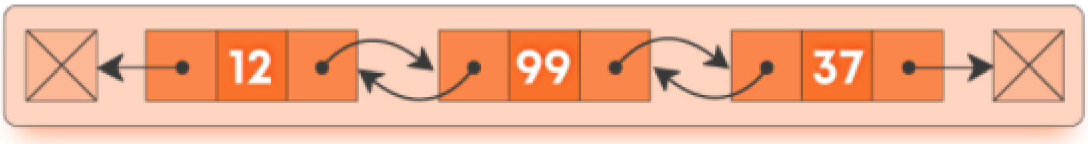
Introduction

Doubly Linked Lists contain an extra pointer pointing towards the previous node (called as the previous pointer) in addition to the pointer pointing to the next node (called the next pointer). The advantage of using doubly linked lists is that we can navigate in both the directions.
A node in a singly linked list can not be removed unless we have the predecessor node. But in a doubly linked list we don't need the access of the predecessor node.



Operations on doubly linked lists

Insertion Operations

- **insertAtBeginning(data):** Inserting a node in front of the head of a linked list.
- **insertAtEnd(data):** Inserting a node at the tail of a linked list.
- **insertAtIdx(idx, data)**: Inserting a node at a given index.

Deletion Operations

- **deleteFromBeginning:** Deleting a node from the front of a linked list.
- **deleteFromEnd:** Deleting a node from the end of a linked list.
- **deleteFromIdx(idx)**: Deleting a node at a given index.

Implementation of doubly LinkedList

Doubly Linked Lists contain a **head pointer** that points to the first node in the list (head is null of the list is empty)
Each node in a doubly linked list has three properties : **data, previous(pointer to the previous node), next(pointer to the next node).**

- **Insert at beginning**

```
function insertAtBeginning(data)
        /*
                create a new node : newNode
                set newNode's data to data
        */
```

```
        newNode.data =  data

        //   If list is empty, set head as newNode
        if head is null
                head = newNode
                return head

        newNode.next = head
        head.previous = newNode
        head = newNode
        return head
```

- **Insert at end**

```
function insertAtEnd(data)
        /*
                create a new node : newNode
                set newNode's data to data
        */

        newNode.data =  data

        //   If list is empty, set head as newNode
        if head is null
                head = newNode
                return head


        /*
                Otherwise create a cur node pointer and keep moving it
                until it reaches the last node of the list
        */

        cur = head
        while cur.next is not null
        cur = cur.next

        /*
                Now cur points to the last node of linked list, set the next
```

```
            pointer of this node to the newNode
    */

    cur.next = newNode
    newNode.previous = cur
    return head
```

- **Insert at given index (idx will be 0 indexed)**

```
function insertAtGivenIdx(idx, data)
    /*
            create a new node : newNode
            set newNode's data to data
    */

    newNode.data =  data

    //   call insertAtBeginning if idx = 0

    if idx == 0
            insertAtBeginning(data)
            return head

    count = 0
    cur = head

    while count < idx - 1 and cur.next is not null
            count += 1
            cur = cur.next

    /*
            If count does not reach (idx - 1), then the given index
            is greater than the size of the list
    */

    if count < idx - 1
            print "invalid index"
            return head
```

```
/*
        Otherwise setting the newNode next field as the address of the
        node present at position idx
*/

nextNode = cur.next
cur.next = newNode
newNode.prev = cur

if nextNode is not null
        nextNode.prev = newNode
        newNode.next = nextNode

return head
```

- **Delete from beginning**

```
function deleteFromBeginning()

        //   if head is null, return

        if head is null
                print "Linked List is Empty"
                return head


        temp  = head
        head = head.next
        head.prev = null
        delete temp
        return head
```

- **Delete from end.**

```
function deleteFromEnd()

        //   list is empty, if head is null
        if head is null
```

```
        print  “list is Empty”
        return head

    /*
        Keep a cur pointer and let it point to head move the cur
        pointer till cur.next is not equal to  null
    */

    cur = head ;
    while cur.next is not equal to null
        cur = cur.next

    prevNode = cur.prev
    prevNode.next = null
    delete cur
    return head
```

- Delete from given index (idx will be 0 indexed)

```
function deleteFromGivenIdx(idx)

    //   call deleteFromBeginning if idx = 0
    if idx == 0
         deleteFromBeginning()
         return head


    temp = head
    count = 0

    while count < idx - 1 and temp is not equal to null
          temp = temp.next
          count++

    if temp = null or temp.next is equal to null
          print “Invalid Index”
          return head

     nextNode = temp.next
     prevNode = temp.prev
```

```
prevNode.next = nextNode
nextNode.previous = prevNode
delete temp
return head
```

Time Complexity of various operations

Let 'n' be the number of elements in the linked lists. The complexities of linked list operations with this representation can be given as:

Operations	Time Complexity
insertAtBeginning(data)	O(1)
insertAtEnd(data)	O(n)
insertAtGivenIdx(idx, data)	O(n)
deleteFromBeginning()	O(1)
deleteFromEnd()	O(n)
deleteFromGivenIdx(idx)	O(n)

Applications of Doubly Linked Lists

- It is used by web browsers for backward and forward navigation of web pages
- LRU (Least Recently Used) / MRU (Most Recently Used) Cache are constructed using Doubly Linked Lists.
- Used by various applications to maintain undo and redo functionalities.
- In Operating Systems, a doubly linked list is maintained by thread scheduler to keep track of processes that are being executed at that time.