Add last

1. Update the next link of the current tail node, to point to the new node.

2. Update tail link to point to the new node.

```java
public void addLast(LinkedListNode newNode) {
    if (newNode == null)
        return;
    else {
        newNode.next = null;
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
    }
}
```
Inserted Between Two Nodes

Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the **new node**.

2. Update link of the **new node**, to point to the "next" node.

```java
public void insertAfter(LinkedListNode previous, LinkedListNode newNode) {
    if (newNode == null)
        return;
    else {
        if (previous == null)
            addFirst(newNode);
        else if (previous == tail)
            addLast(newNode);
        else {
            LinkedListNode.next = previous.next;
            previous.next = newNode;
            newNode.next = next;
        }
    }
}
```
**Singly-linked list, Removal (deletion) operation.**
There are four cases, which can occur while removing the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite. Notice, that removal algorithm includes the disposal of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

**List has only one node**
When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.

**Remove first**

In this case, first node (current head node) is removed from the list. It can be done in two steps:

1. Update **head link** to point to the node, next to the head.

2. Dispose removed node.

```java
public void removeFirst() {
    if (head == null)
        return;
    else {
        if (head == tail) {
            head = null;
            tail = null;
        } else {
            head = head.next;
            head = null;
        }
    }
}
```
Remove last

In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.

It can be done in three steps:

1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.

2. Set next link of the new tail to NULL.

3. Dispose removed node.
public void removeLast() {
    if (tail == null)
        return;
    else {
        if (head == tail) {
            head = null;
            tail = null;
        } else {
            LinkedListNode previousToTail = head;
            while (previousToTail.next != tail)
                previousToTail = previousToTail.next;
            tail = previousToTail;
            tail.next = null;
        }
    }
}

Remove Next
Such a removal can be done in two steps:

1. Update next link of the previous node, to point to the next node, relative to the removed node.

2. Dispose removed node.
public void removeNext(LinkedListNode previous) {
    if (previous == null)
        removeFirst();
    else if (previous.next == tail) {
        tail = previous; // remove last
        tail.next = null;
    } else if (previous == tail)
        return;
    else {
        previous.next = previous.next.next;
    }
}

Traversal algorithm
Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

Example

As for example, let us see an example of summing up values in a singly-linked list.

```
5 → -1 → 16 → 2
```

initial state
sum = 0

step 1
sum = 5

step 2
sum = 4
For some algorithms tracking the previous node is essential, but for some, like an example, it's unnecessary. We show a common case here and concrete algorithm can be adjusted to meet its individual requirements.

```java
public class LinkedList {
    ...

    public int traverse() {
        int sum = 0;
        LinkedListNode current = head;
        LinkedListNode previous = null;
        while (current != null) {
            sum += current.value;
            previous = current;
            current = current.next;
        }
        return sum;
    }
}
```

**H.W : Finding and Deleting Specified Links**

This example program search a linked list for a data item with a specified key value and to delete an item with a specified key value.
A Stack Implemented by a Linked List

When we created a stack we used an ordinary Java array to hold the stack’s data. The stack’s push() and pop() operations were actually carried out by array operations such as

arr[++top] = data;
and
data = arr[top--];
which insert data into, and take it out of, an array.

We can also use a linked list to hold a stack’s data. In this case the push() and pop() operations would be carried out by operations like

theList.insertFirst(data)
and
data = theList.deleteFirst()

The user of the stack class calls push() and pop() to insert and delete items without knowing, or needing to know, whether the stack is implemented as an array or as a linked list.

class Link
{
    public long dData;    // data item
    public Link next;     // next link in list
    // constructor
    // ------------------------------------
    // Display Link()
    /////////////////////////////////////////////////////////////
    class LinkList
    {
        private Link first;  // ref to first item on list
        // -----------------------------------------------
        public LinkList()    // constructor
        { first = null; }    // no items on list yet
        // ------------------------------------------------
        isEmpty()  // true if list is empty
        // ------------------------------------------------
        insertFirst(long dd) // insert at start of list
        // ------------------------------------------------
        deleteFirst()      // delete first item
        // ------------------------------------------------
    } // class LinkList
} // class Link
displayList()

// ---------------------------------------------
} // end class LinkList

/////////////////////////////////////////////////////////////////////////
class LinkStack
{
private LinkList theList;
//-------------------------------------------------------------
public LinkStack() // constructor
{
theList = new LinkList();
}
//--------------------------------------------------------------
public void push(long j) // put item on top of stack
{
theList.insertFirst(j);
}
//--------------------------------------------------------------
public long pop() // take item from top of stack
{
return theList.deleteFirst();
}
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{
return ( theList.isEmpty() );
}
//--------------------------------------------------------------
public void displayStack()
{
System.out.print(“Stack (top--->bottom): “);
theList.displayList();
}
//--------------------------------------------------------------
} // end class LinkStack

A Queue Implemented by a Linked List

class Link
{
public long dData; // data item
public Link next; // next link in list
//-------------------------------------------------------------
// constructor
class FirstLastList
{
    private Link first;       // ref to first item
    private Link last;        // ref to last item

    public FirstLastList()   // constructor
    {
        first = null;        // no items on list yet
        last = null;
    }

    isEmpty()       // true if no links

    insertLast(long dd) // insert at end of list

    deleteFirst()    // delete first link

    displayList()    // display this link

}  // end class FirstLastList

class LinkQueue
{
    private FirstLastList theList;

    public LinkQueue()    // constructor
    {
        theList = new FirstLastList();
    }

    isEmpty()       // true if queue is empty

    insert(long j)  // insert, rear of queue

    remove()        // remove, front of queue

    displayQueue()  // display this link

}  // end class LinkQueue