Pointers

Java does not have an explicit pointer type. Instead of pointers, all references to objects—including variable assignments, arguments passed into methods and array elements—are accomplished by using implicit references. References and pointers are essentially the same thing except that you can't do pointer arithmetic on references (nor do you need to).
Reference semantics also enable structures such as linked lists to be created easily in Java without explicit pointers; merely create a linked list node with variables that point to the next and the previous node. Then, to insert items in the list, assign those variables to other node objects.

How do I implement a Linked List in Java?

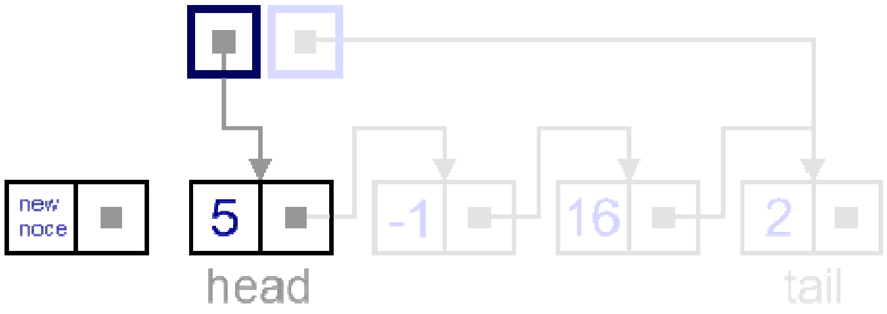
The linkList.java Program

```
// linkList.java
// demonstrates linked list
////////////////////////////////////////////////////////////////
class Link
{
public int iData; // data item (key)
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------

public Link(int id, double dd) // constructor
{
iData = id; // initialize data
dData = dd; // ('next' is automatically
} // set to null)
// -------------------------------------------------------------
public void displayLink() // display our self
{
System.out.print("{" + iData + ", " + dData + "} ");
}
} // end class Link
////////////////////////////////////////////////////////////////


class LinkList
{
private Link first; // ref to first link on list
// -------------------------------------------------------------
public LinkList() // constructor
{
first = null; // no items on list yet
}
// -------------------------------------------------------------
```

```
public boolean isEmpty() // true if list is empty
{
return (first==null);
}
// ------------------------------------------------------------
```

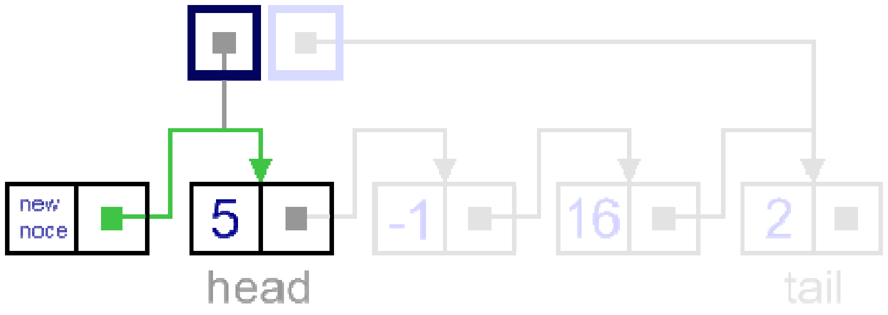
Ex: ***Add first*** **(linked list for one item)**

In this case, new node is inserted right before the current head node.

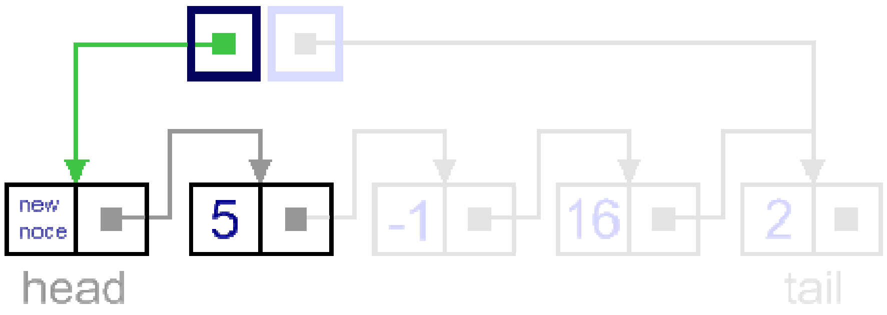


It can be done in two steps:

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



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



// insert at start of list

```
public void insertFirst(int id, double dd)
{ // make new link
Link newLink = new Link(id, dd);
newLink.next = first; // newLink --> old first
first = newLink; // first --> newLink
}
// -------------------------------------------------------------
public Link deleteFirst() // delete first item
{ // (assumes list not empty)
Link temp = first; // save reference to link
first = first.next; // delete it: first-->old next
return temp; // return deleted link
}
// -------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class LinkList
////////////////////////////////////////////////////////////////
```

Finding and Deleting Specified Links
Our next example program adds methods to search a linked list for a data item with a specified key value and to delete an item with a specified key value. These, along with insertion at the start of the list, are the same operations carried out by the LinkList Workshop applet.

```
// linkList2.java
// demonstrates linked list
////////////////////////////////////////////////////////////////
class Link
{
public int iData; // data item (key)
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(int id, double dd) // constructor
```

```
{
iData = id;
dData = dd;
}
// -------------------------------------------------------------
public void displayLink() // display ourself
{
System.out.print("{" + iData + ", " + dData + "} ");
}
} // end class Link
////////////////////////////////////////////////////////////////
class LinkList
{
private Link first; // ref to first link on list
// -------------------------------------------------------------
public LinkList() // constructor
{
first = null; // no links on list yet
}
// -------------------------------------------------------------
public void insertFirst(int id, double dd)
{ // make new link
Link newLink = new Link(id, dd);
newLink.next = first; // it points to old first link
first = newLink; // now first points to this
}
// -------------------------------------------------------------
public Link find(int key) // find link with given key
{ // (assumes non-empty list)
Link current = first; // start at 'first'
while(current.iData != key) // while no match,
{
if(current.next == null) // if end of list,
return null; // didn't find it
else // not end of list,
current = current.next; // go to next link
}
return current; // found it
}
// -------------------------------------------------------------
public Link delete(int key) // delete link with given key
{ // (assumes non-empty list)
Link current = first; // search for link
Link previous = first;
while(current.iData != key)
{
```

```
if(current.next == null)
return null; // didn't find it
else
{
previous = current; // go to next link

current = current.next;
}
} // found it
if(current == first) // if first link,
first = first.next; // change first
else // otherwise,
previous.next = current.next; // bypass it
return current;
}
// -------------------------------------------------------------
public void displayList() // display the list
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class LinkList
////////////////////////////////////////////////////////////////
class LinkList2App
{
public static void main(String[] args)
{
LinkList theList = new LinkList(); // make list
theList.insertFirst(22, 2.99); // insert 4 items
theList.insertFirst(44, 4.99);
theList.insertFirst(66, 6.99);
theList.insertFirst(88, 8.99);
theList.displayList(); // display list
Link f = theList.find(44); // find item
if( f != null)
System.out.println("Found link with key " + f.iData);
else
```

Finding and Deleting Specified Links 195

```
System.out.println("Can't find link");
Link d = theList.delete(66); // delete item
if( d != null )
System.out.println("Deleted link with key " + d.iData);
else
System.out.println("Can't delete link");
theList.displayList(); // display list
} // end main()
} // end class LinkList2App
////////////////////////////////////////////////////////////////
```

The main() routine makes a list, inserts four items, and displays the resulting list. It then searches for the item with key 44, deletes the item with key 66, and displays the list again. Here's the output:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Found link with key 44
Deleted link with key 66
List (first-->last): {88, 8.99} {44, 4.99} {22, 2.99}
```

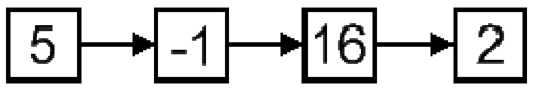
Traversal algorithm

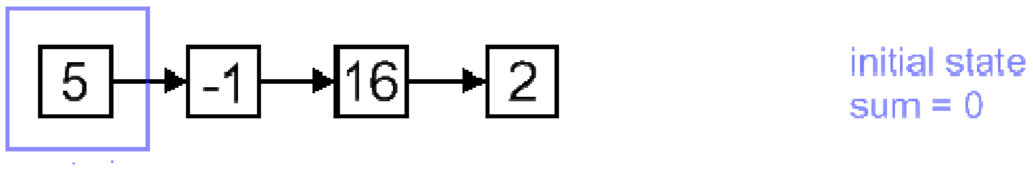
Beginning from the head,

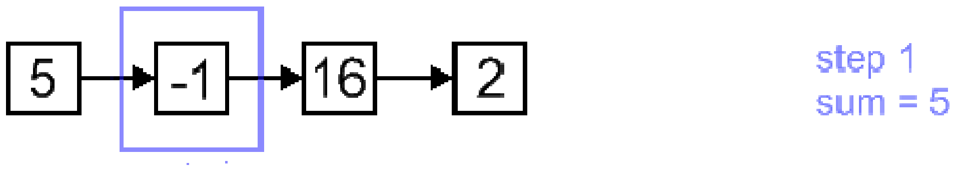
1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

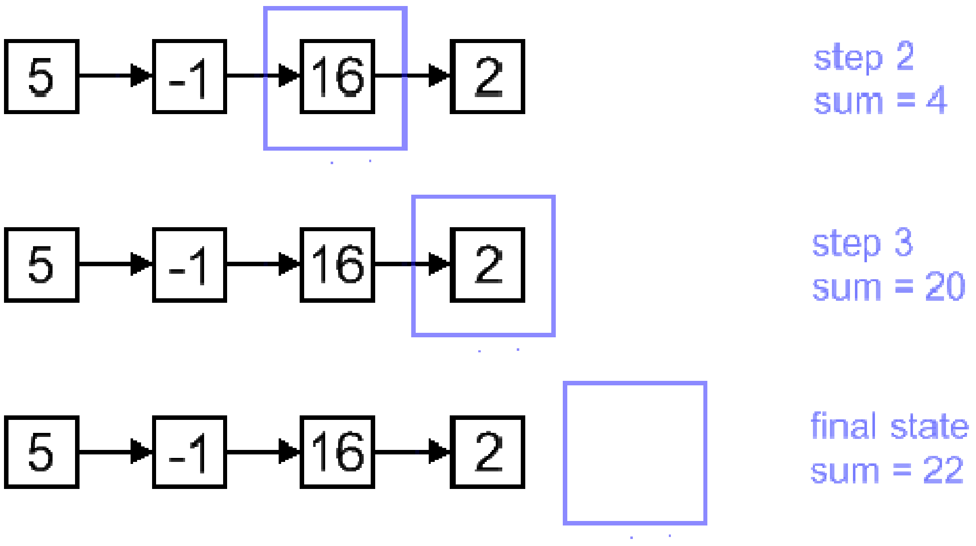
Example

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









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.

Code snippets

Although we have two classes for singly-linked list, *SinglyLinkedListNode* class is used as storage only. Whole algorithm is implemented in the *SinglyLinkedList* class.

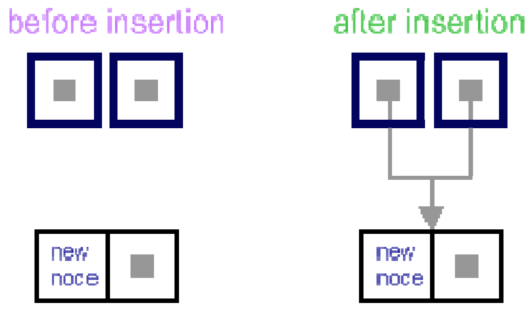
```
public class SinglyLinkedList {

    …



    public int traverse() {

        int sum = 0;

        SinglyLinkedListNode current = head;

        SinglyLinkedListNode previous = null;

        while (current != null) {

            sum += current.value;

            previous = current;

            current = current.next;
            }

            return sum;

      }

}
```

Singly-linked list. Addition (insertion) operation.

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

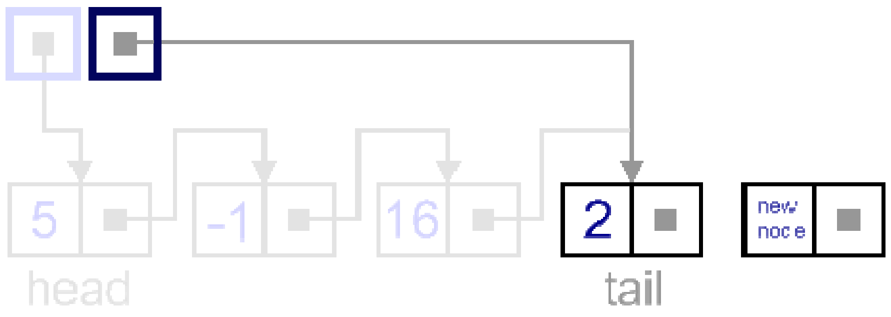
Empty list case

When list is empty, which is indicated by (head == NULL) condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.



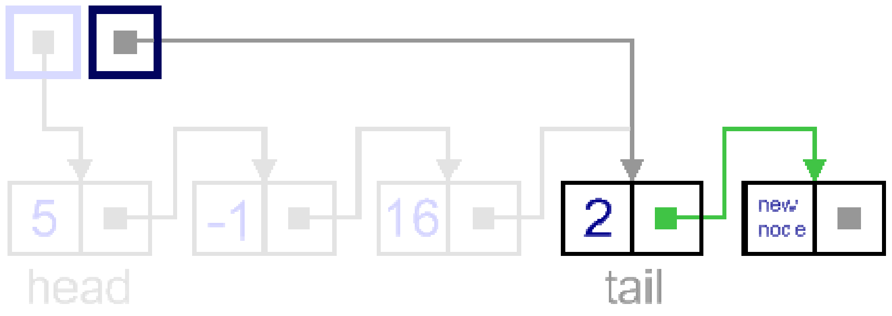
Add last

In this case, new node is inserted right after the current tail node.

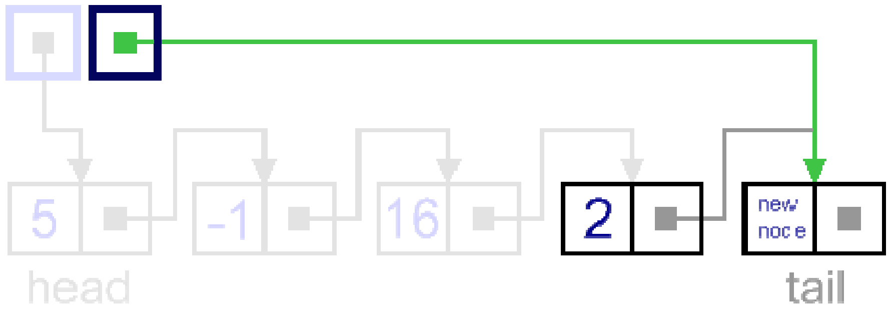


It can be done in two steps:

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

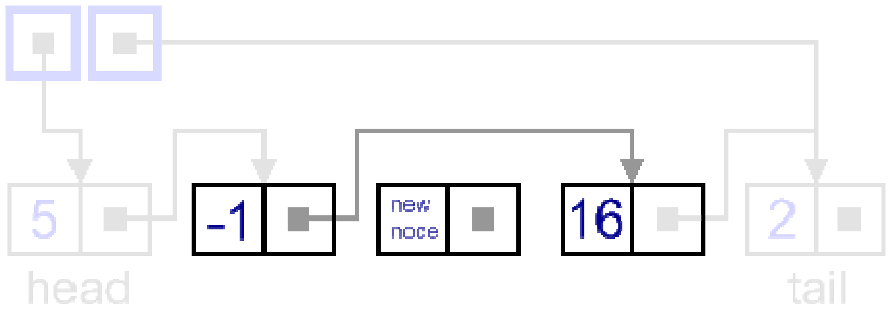


2. Update tail link to point to the new node.



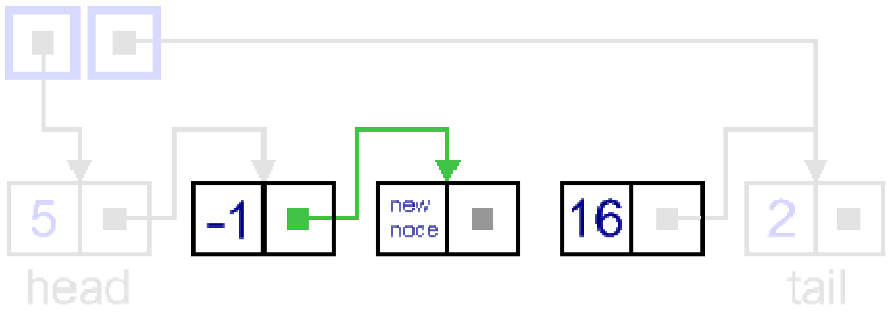
General case

In general case, new node is **always inserted between** two nodes, which are already in the list. Head and tail links are not updated in this case.

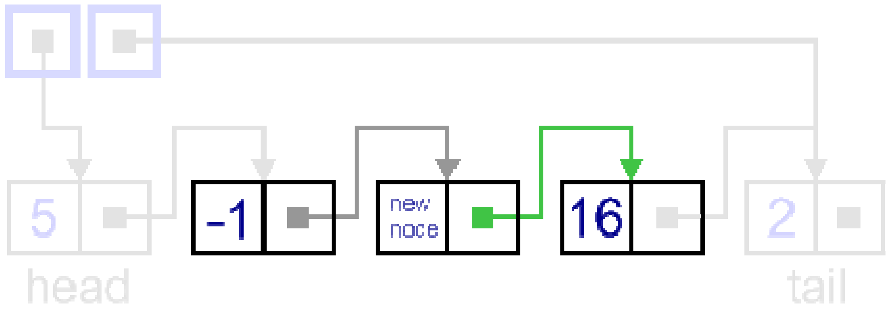


Such an insert can be done in two steps:

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.



Code snippets

All cases, shown above, can be implemented in one function with two arguments, which are node to insert after and a new node. For *add first* operation, the arguments are *(NULL, newNode)*. For *add last* operation, the arguments are *(tail, newNode)*. Though, this specific operations (add first and add last) can be implemented separately, in order to avoid unnecessary checks.

Java implementation

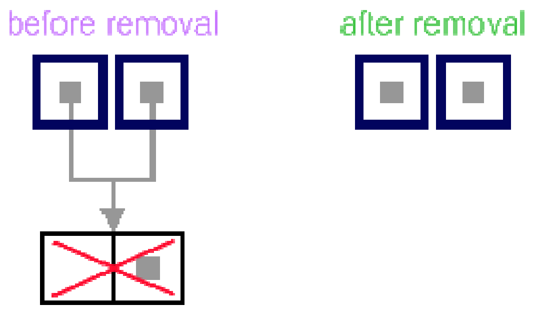
```
public class SinglyLinkedList {

    …



    public void addLast(SinglyLinkedListNode newNode) {

        if (newNode == null)

            return;

        else {
            newNode.next = null;
            if (head == null) {
                head = newNode;
                tail = newNode;
            } else {
                tail.next = newNode;
                tail = newNode;
            }
        }
    }

    public void addFirst(SinglyLinkedListNode newNode) {
        if (newNode == null)
            return;
        else {
            if (head == null) {
                newNode.next = null;
                head = newNode;
                tail = newNode;
            } else {
                newNode.next = head;
                head = newNode;
            }
        }
    }

    public void insertAfter(SinglyLinkedListNode previous,
            SinglyLinkedListNode newNode) {
        if (newNode == null)
            return;
        else {
            if (previous == null)
                addFirst(newNode);
            else if (previous == tail)
                addLast(newNode);
            else {
                SinglyLinkedListNode 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).

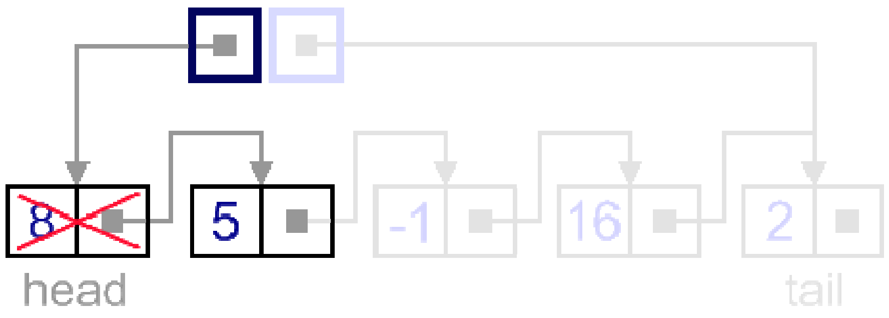
List has only one node

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*.



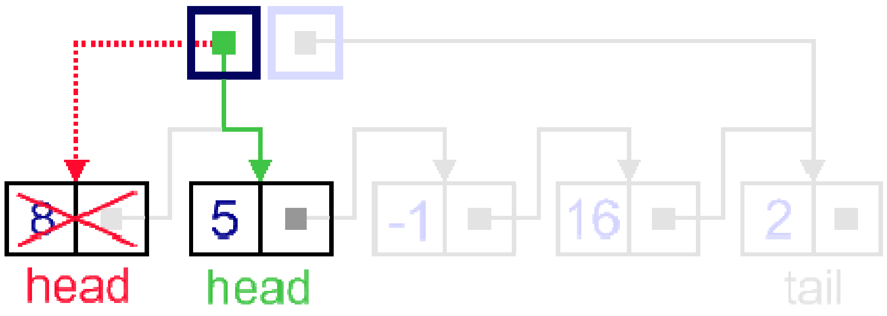
Remove first

In this case, first node (current head node) is removed from the list.

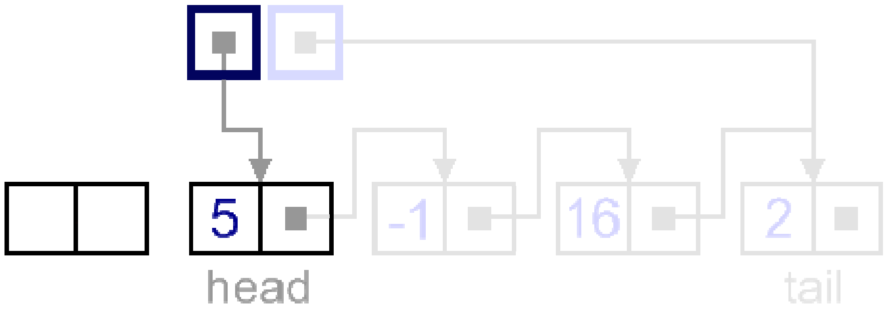


It can be done in two steps:

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

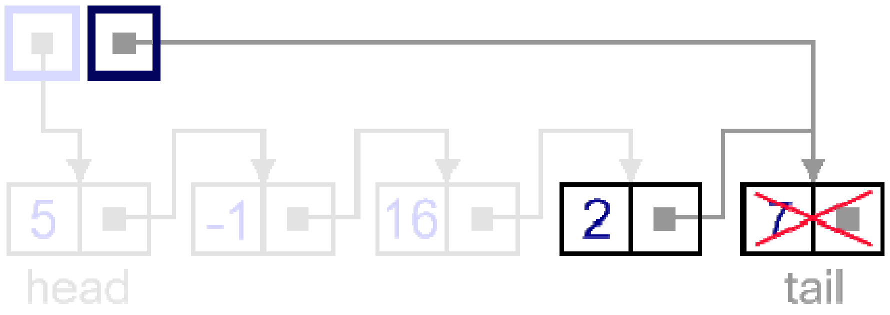


2. Dispose removed node.



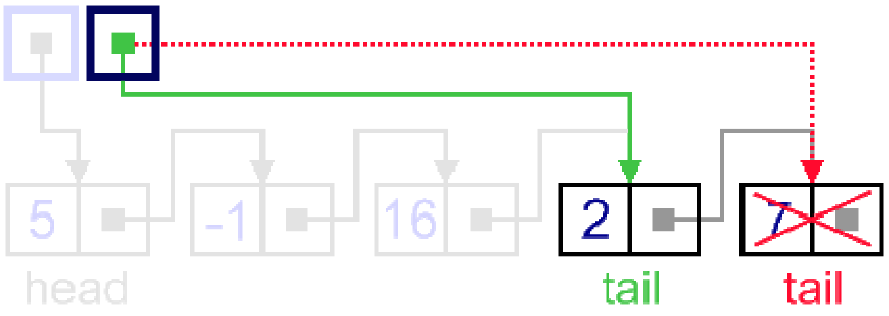
Remove last

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.

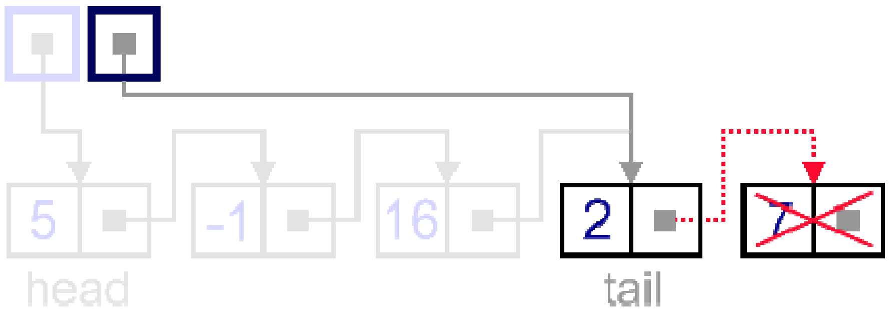


It can be done in three steps:

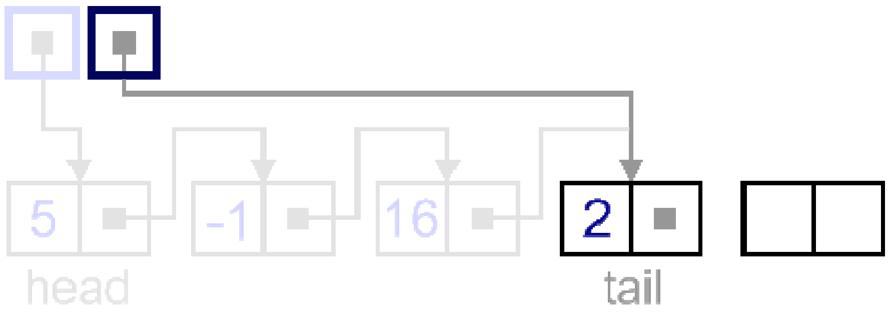
1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



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

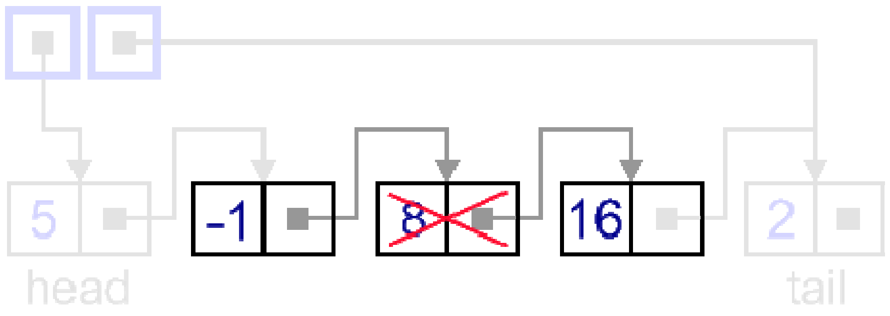


3. Dispose removed node.



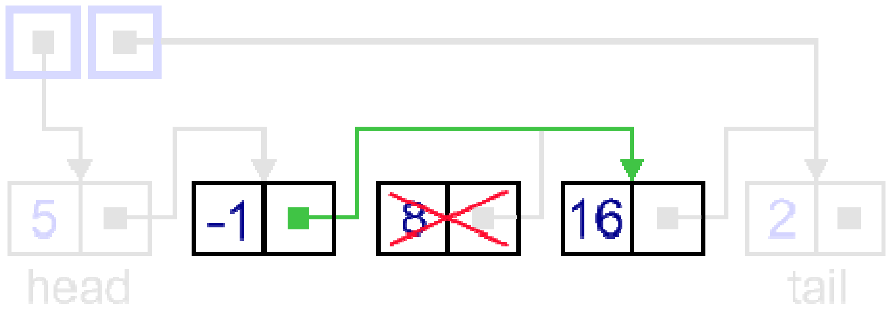
General case

In general case, node to be removed is **always located between** two list nodes. Head and tail links are not updated in this case.

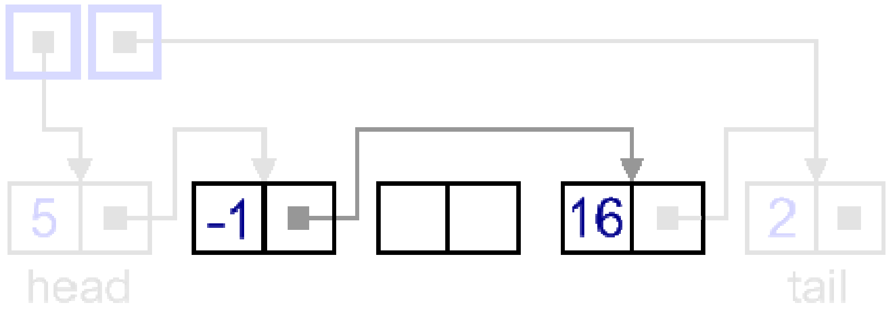


Such a removal can be done in two steps:

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



2. Dispose removed node.



Code snippets

All cases, shown above, can be implemented in one function with a single argument, which is node previous to the node to be removed. For *remove first* operation, the argument is *NULL*. For *remove last* operation, the argument is the node, previous to tail. Though, it's better to implement this special cases (remove first and remove last) in separate functions. Notice, that removing first and last node have different complexity, because *remove last* needs to traverse through the whole list.

Java implementation

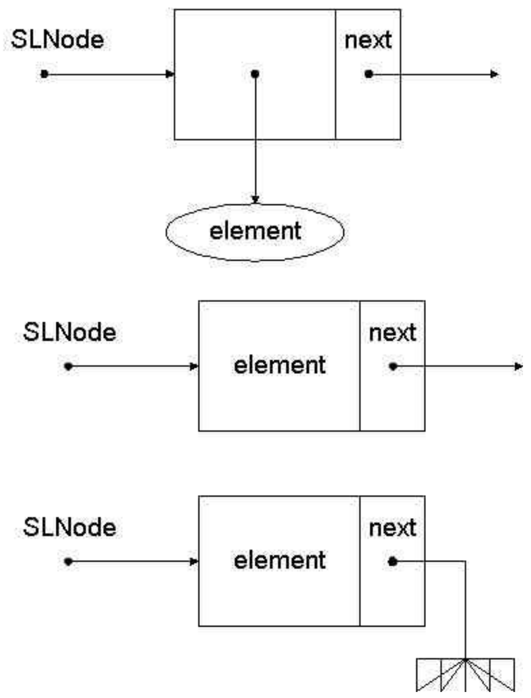
```
public class SinglyLinkedList {
    …

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

    public void removeLast() {
        if (tail == null)
            return;
        else {
            if (head == tail) {
                head = null;
                tail = null;
            } else {
                SinglyLinkedListNode previousToTail = head;
                while (previousToTail.next != tail)
                    previousToTail = previousToTail.next;
                tail = previousToTail;
                tail.next = null;
            }
        }
    }

    public void removeNext(SinglyLinkedListNode previous) {
        if (previous == null)
            removeFirst();
        else if (previous.next == tail) {
            tail = previous;
            tail.next = null;
        } else if (previous == tail)
            return;
        else {
            previous.next = previous.next.next;
        }
    }
```

}

Linked-List Node

A linked list is constructed from a collection of self-referential nodes which may be defined as follows:

```
class SLNode {

private Object element; // element stored in this node

private SLNode next; // reference to the next node in the list

:

}
```



SLNode constructors and methods

A SLNode object can be created to point to the next object in the list.

```
public SLNode(Object e, SLNode n) { //# create a node given element and next
```

```
        element = e;

        next = n;

}
```

An alternative constructor has only one argument for the item; the next field is set to null.

```
                public SLNode(Object element)

                {

                this.element = element;

                this.next = null; //optional

                }
```

- An update method allows modification of the next field.

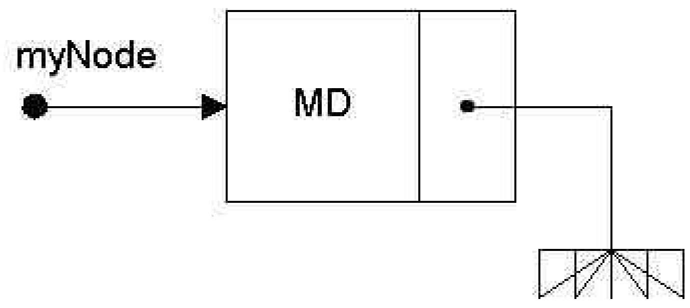
```
public void setNext(SLNode newNext)

{

next = newNext;

}
```

- Another update method allows modification of the element field.

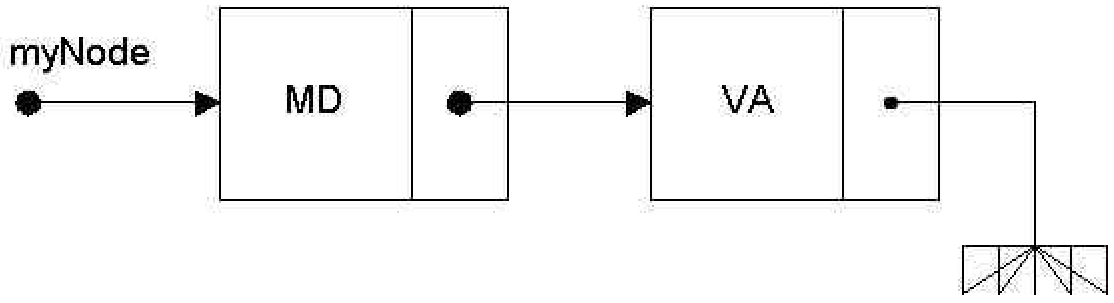
```
public void setElement(Object newElem) { element = newElem; }
```

Building a linked list from SLNode objects

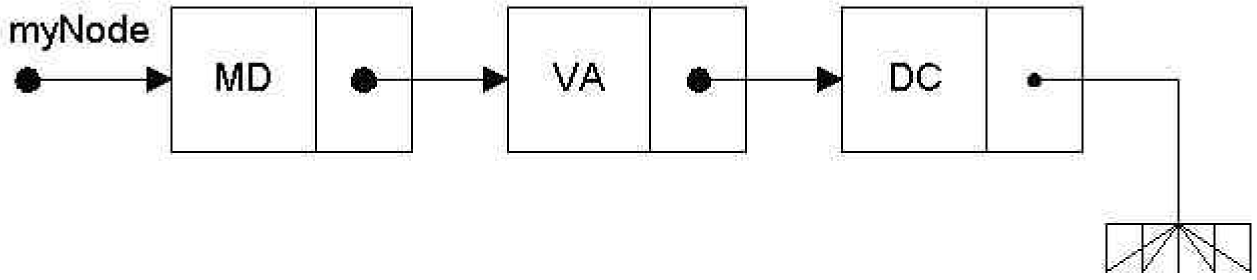
```
SLNode myNode = new SLNode("MD");
```



myNode.setNext(new ListNode("VA"));



myNode.getNext().setNext(new ListNode("DC"));



We could have built the same list with a single command.

ListNode myNode = new ListNode("MD", new ListNode("VA", new ListNode("DC")));

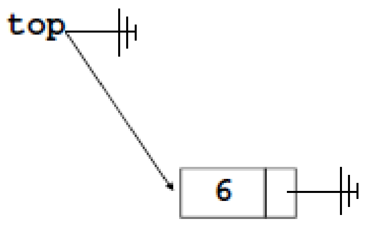
Stack: Linked List Implementation

- Push and pop at the head of the list
  - New nodes should be inserted at the front of the list, so that they become the top of the stack
  - Nodes are removed from the front (top) of the list
- Straight-forward linked list implementation
  - push and pop can be implemented fairly easily, e.g. assuming that head is a reference to the node at the front of the list

```
public void push(int x){
// Make a new node whose next reference is
// the existing list
Node newNode = new Node(x, top);
top = newNode; // top points to new node
```

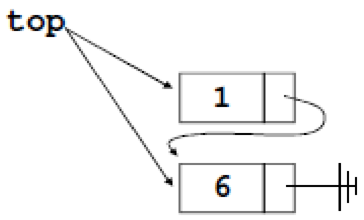
}

List Stack Example



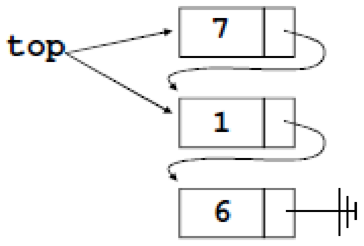
Java Code

```
Stack st = new Stack();
st.push(6);
```



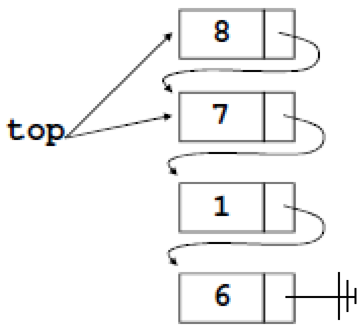
Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
```



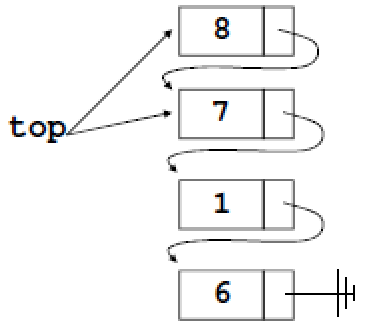
Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
```

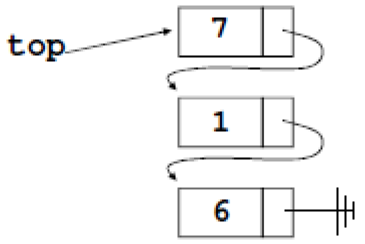


Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```



Java Code
Stack st = **new** Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();



Java Code
Stack st = **new** Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();

<u>Stack: ADT List Implementation</u>

- Push() and pop() either at the beginning or at the end of ADT List
    - at the beginning:

```
public void push(Object newItem) {
  list.add(1, newItem);
 }  // end push
        public Object pop() {
  Object temp = list.get(1);
  list.remove(1);
  return temp;
 }  // end pop
```

- Push() and pop() either at the beginning or at the end of ADT List
    - at the end:

```
                public void push(Object newItem) {
  list.add(list.size()+1, newItem);
 }  // end push
        public Object pop() {
  Object temp = list.get(list.size());
  list.remove(list.size());
  return temp;
```

```
}  // end pop
```

- Push() and pop() either at the beginning or at the end of ADT List
- Efficiency depends on implementation of ADT List (not guaranteed)
- On other hand: it was very fast to implement (code is easy, unlikely that errors were introduced when coding).