WCOM125/COMP125

Fundamentals of Computer Science

Linked Lists

Gaurav Gupta

July 10, 2018

Contents

List of exercises

Section 1.

So why are we learning this?

We have mentioned in earlier sections that **ArrayLists** generally offer better performance than **LinkedLists** . So why do we need to learn about **LinkedLists** ? Because:

1. **LinkedLists** require n small pockets of memory to hold n values rather than 1 large block of memory.
2. **LinkedLists** are a fantastic introduction to *recursive data structures*. We can extend these to binary trees, trees, and graphs.

Section 2.

*The **Node** class*

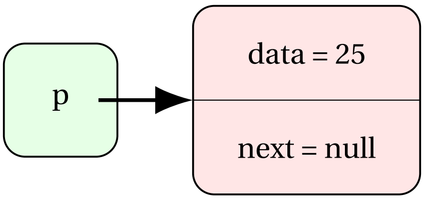
A **Node** is a class that has two instance variables: **next: Node** and **data: <dataType>**
.

A **Node** holding **int** data is:

```java
class Node {
        public int data;
        public Node next; //reference to next node
}
```

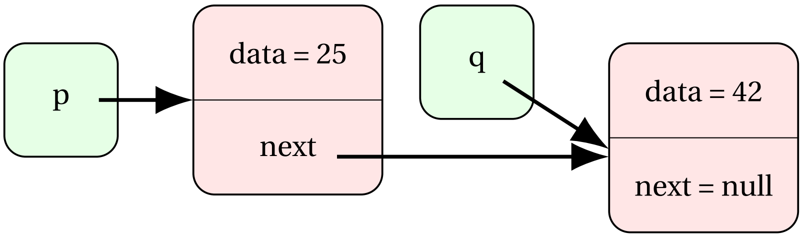
Code to create object **p** of class **Node** and resulting memory diagram are:

```java
Node p = new Node();
p.data = 25;
p.next = null;
```



Code to create objects **p, q** of class **Node** and resulting memory diagram are:

```java
Node q = new Node();
q.data = 42;
q.next = null;
Node p = new Node();
p.data = 25;
p.next = q;
```



Here, we can see that **p.next** (of type **Node**) is a shallow copy of **q** (of type **Node**).

The **data** instance variable in a **Node** class may be object of another class too. For example,

```java
class Rectangle {
        public double width, height;
}
```

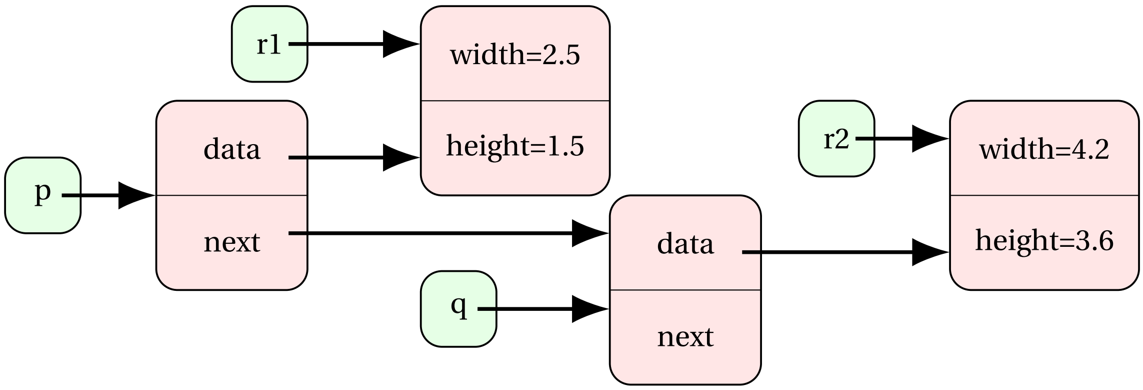
```java
class Node {
        public Rectangle data;
        public Node next;
}
```

```java
Rectangle r1 = new Rectangle();
r1.width = 2.5;
r1.height = 1.5;
Rectangle r2 = new Rectangle();
r2.width = 4.2;
r2.height = 3.6;
Node q = new Node();
q.data = r2;
q.next = null;
Node p = new Node();
p.data = r1;
p.next = q;
```

The memory diagram for this code is below. Notice they key points:

- **p.next (Node)** is a shallow copy of **q (Node)**
- **p.data (Rectangle)** is a shallow copy of **r1 (Rectangle)**

- **q.data (Rectangle)** is a shallow copy of **r2 (Rectangle)**



Exercise 1

Draw a memory diagram for the following code:

```
class Circle {
        private double radius;
        public Circle(double r) {
                radius = Math.abs(r);
        }
}
class Node {
        private Circle data;
        private Node next;
        public Node(Circle d, Node n) {
                data = d;
                next = n;
        }
}
public class Client {
        public static void main(String[] args) {
                Circle c1 = new Circle(2.8);
                Circle c2 = new Circle(1.6);
                Node p = new Node(c1, null);
                Node q = new Node(c2, p);
                Node r = new Node(c1, p);
        }
}
```

Write your answer here

(SOLUTION 1)

Section 3.

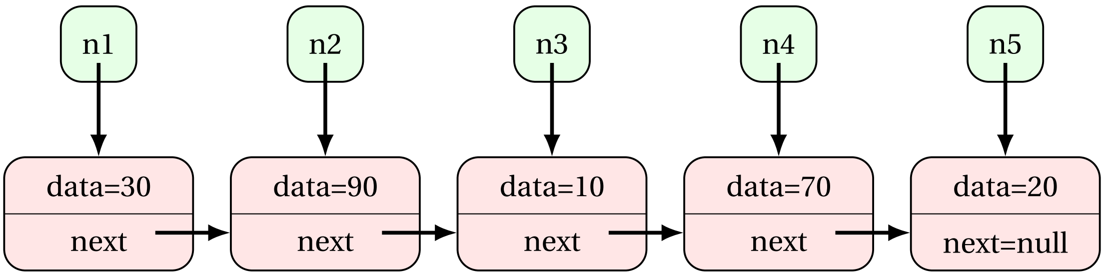
Creating a "linked list" *manually*

Once we understand the **Node** class, we can link objects of **Node** together to create a *"linked" list.*

```java
class Node {
        private int data; //primitive data type for simplicity
        private Node next;

        //getter, setter for data
        public int getData() { return data; }
        public void setData(int d) { data = d; }

        //getter, setter for next
        public Node getNext() { return next; }
        public void setNext(Node n) { next = n; }

        public Node(int d) {
                data = d;
                next = null;
        }
        public Node(int d, Node n) {
                data = d;
                next = n;
        }
}
```

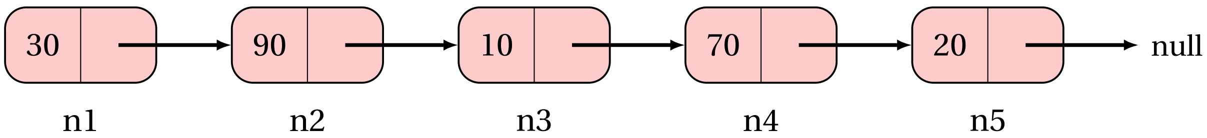
```java
class Client {
	public static void main(String[] args) {
		Node n5 = new Node(20, null);
		Node n4 = new Node(70, n5);
		Node n3 = new Node(10, n4);
		Node n2 = new Node(90, n3);
		Node n1 = new Node(30, n2);
	}
}
```

The memory diagram for the above code is given below:



3.1 Simplified representation of linked nodes

The above diagram, while thorough, is *too* detailed. We apply a little abstraction and represent the same diagram as:

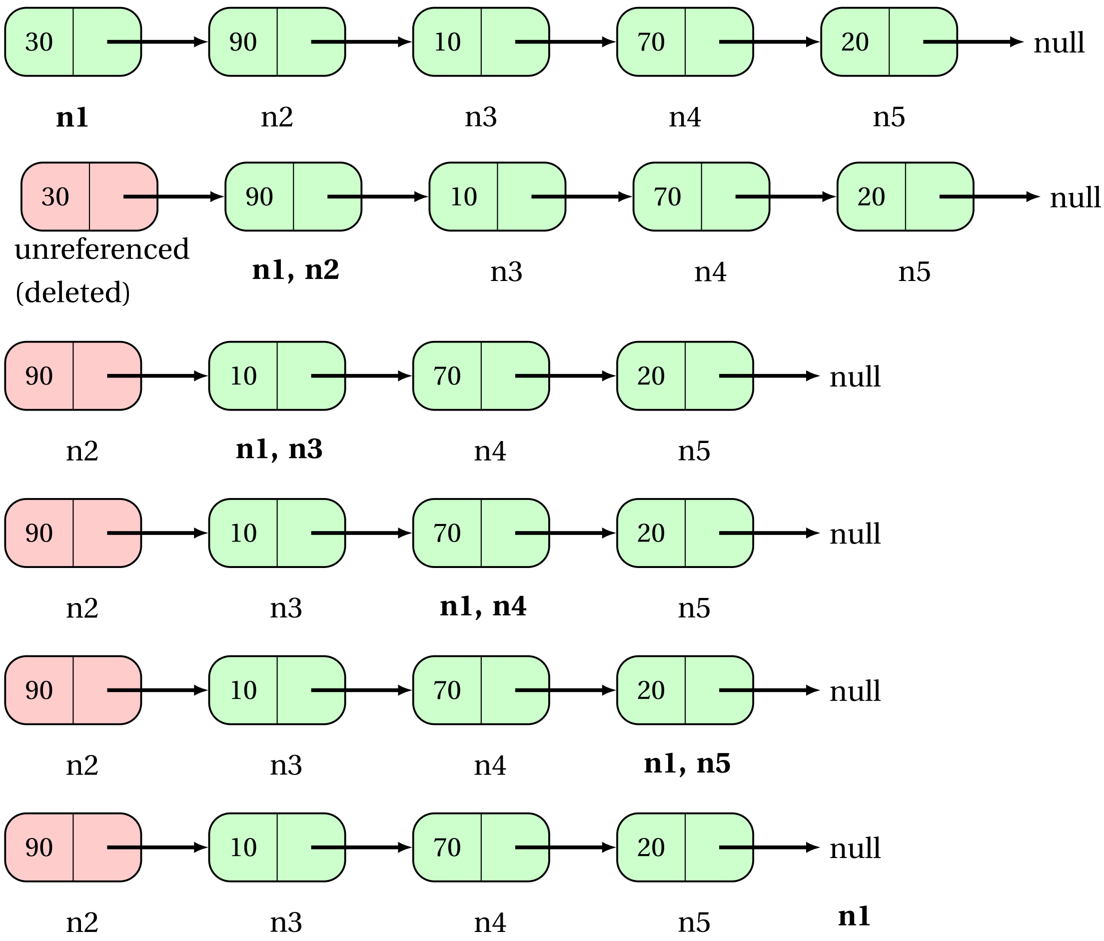


3.2 Traversing a linked list

3.2.1 Incorrect Approach

```
while(n1 != null) {
        n1 = n1.getNext();
}
```

You are modifying the reference **n1** . Each time you update **n1** , the instance referred by **n1** before the update is deleted from memory. Thereby, all the nodes will be erased from the memory at the end of execution. So yeah, not a good idea.

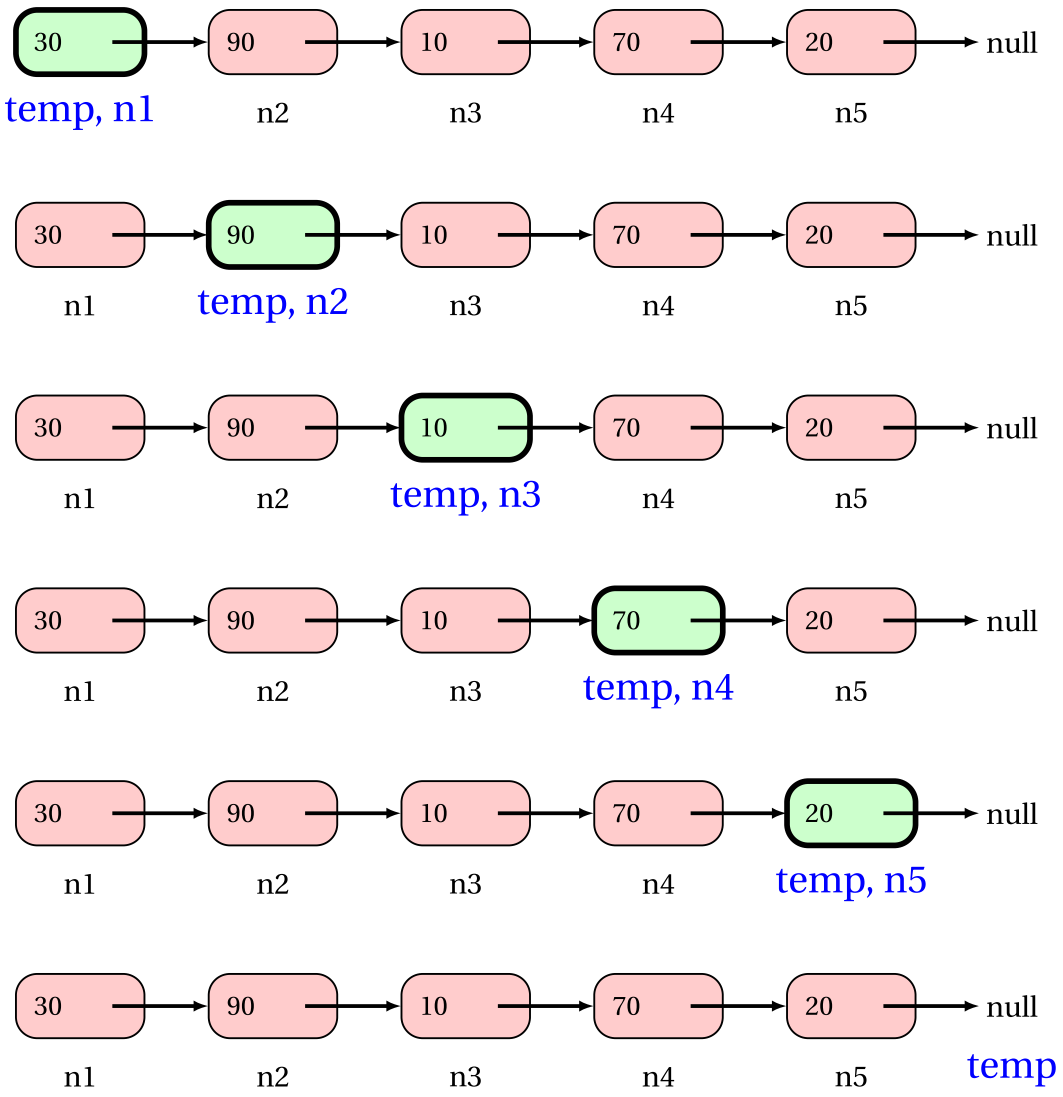


This results in us losing the reference to the first item (**n1**) and the object being deleted from the memory.

3.2.2 Correct Approach

We shallow copy the starting node into a *traversal* node. Then we shift it forward every time in the loop by shallow copying the **next** instance variable into it.

```
Node temp = n1;
while(temp != null) {
        temp = temp.getNext();
}
```



3.3 Some examples of traversal

1. Counting the number of linked nodes:

```java
Node temp = n1;
int counter = 0;
while(temp != null) {
        counter++;
        temp = temp.getNext();
}
```

2. Adding the data values in the linked nodes:

```java
Node temp = n1;
int total = 0;
while(temp != null) {
        total = total + temp.getData();
        temp = temp.getNext();
}
```

3. Adding values over 30 in the linked nodes:

```java
Node temp = n1;
int total = 0;
while(temp != null) {
        if(temp.getData() > 30) {
                total = total + temp.getData();
        }
        temp = temp.getNext();
}
```

4. Determining highest data value in the linked nodes (assuming list is not empty):

```
Node temp = n1;
int highest = temp.getData();
while(temp != null) {
        if(temp.getData() > highest) {
                highest = temp.getData();
        }
        temp = temp.getNext();
}
```

5. Determining if each item is different from the other:

```
Node temp = n1;
boolean foundDuplicate = false;
while(temp != null && !foundDuplicate) {
        Node temp2 = temp.getNext();
        while(temp2 != null && !foundDuplicate) {
                if(temp.getData() == temp2.getData()) {//DUPLICATE
                        foundDuplicate = true;
                }
                temp2 = temp2.getNext();
        }
        temp = temp.getNext();
}
```

Obviously, we can pass a **Node** object to a method just like any other object.

1. Example 1: Counting occurrences of an item

```java
public static int countOccurrences(Node node, int item) {
	int result = 0;
	/*
		note that node is a shallow copy of
		the actual object passed to the method
		call and hence can be modified without
		modifying the actual object.
	*/
	while(node != null) {
		if(node.getData() == item) {
			result++;
		}
		node = node.getNext();
	}
	return result;
}
```

2. Example 2: Checking if all values are positive

```java
public static boolean allPositives(Node node) {
	while(node != null) {
		if(node.getData() <= 0) {
			return false;
		}
		node = node.getNext();
	} //end loop
	return true;
}
```

3.3.1 Recursive methods on linked nodes

Advanced A brilliant example of how this is useful is given in the following method:

```java
/*
return true the sum of some items starting at node n
is total, false otherwise
*/
public static boolean addsTo(Node node, int total) {
	if(total == 0)
		return true;
	//now we know total is not 0
	if(node == null)
		return false;
	int remaining = total - node.getData();
	if(addsTo(node.getNext(), remaining)) {
		/*
		there is a combination after node for
		(total minus what node contains)
		*/
		return true;
	}
	else {
		/*
		there is no combination after node for
		(total minus what node contains) so check for a
		 combination after node for total
		*/
		return addsTo(node.getNext(), total);
	}
}
```

Section 4.

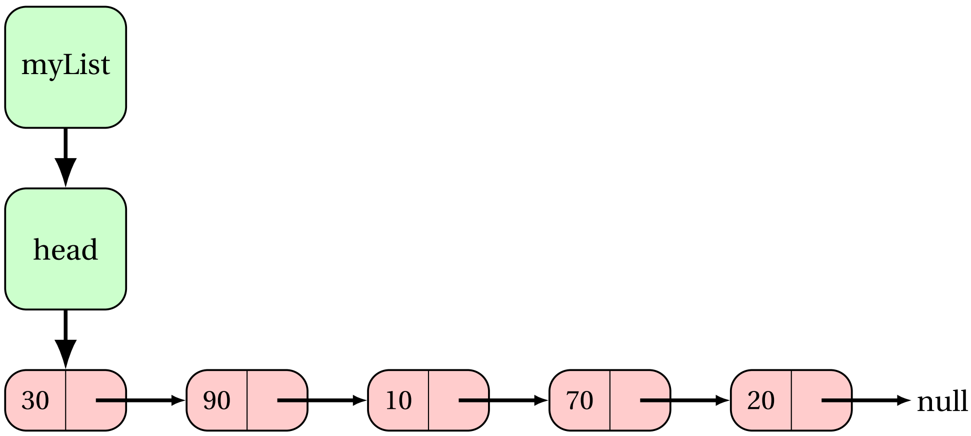
Custom-built LinkedList class

In this next and final step, we put the starting node in a class and operate on the list using the starting node.

```
class MyLinkedList {
        private Node head;

        public MyLinkedList() {
                head = null;
        }

        /**
        insert the passed node object at beginning of list
        */
        public void add(Node node) {
                if(head == null) { //empty list
                        head = node;
                }
                else { //not empty
                        node.setNext(head); //link so head follows node
                        head = node; //update  reference for starting node
                }
        }

        public String toString() {
                Node temp = head;
                String result = "[";
                while(temp != null) {
                        result = result + temp.getData() + ", ";
                        temp = temp.getNext();
                }
                result = result.substring(0, result.length()-2);
```

```
                //remove the last ", "
                return result + "]";
        }
}
```

A linked list **myList** where head holds a reference to a node with data 30, whose **next** instance variable holds a reference to a node with data 90, whose **next** instance variable holds a reference to a node with data 10, whose **next** instance variable holds a reference to a node with data 70, whose **next** instance variable holds a reference to a node with data 20, is given below.



Exercise 2

Define a method `total()` that returns the sum of data values of all nodes in a list based along the lines of `toString()` method.

Write your answer here

(SOLUTION 2)

Exercise 3

Define a method `highest()` that returns the highest value in the list (`null` if list is empty). Note that since `null` is required as error return status, return type should be `Integer` , not `int` .

Write your answer here

(SOLUTION 3)

4.1 Accessing an item at an arbitrary index

Assuming that indexing begins with 0, we can write a method **get(int idx)** that returns the value of an item at an arbitrary index **idx** .

First, we should write a method **size()** since valid indices would then be **[0, ..., size()-1]** .

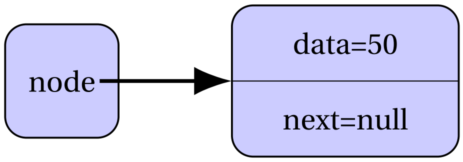
```
public int size() {
        int count = 0;
        Node temp = head;
        while(temp!=null) {
                count++;
                temp = temp.getNext();
        }
        return count;
}
```

Our method **get(int idx)** is:

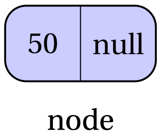
```
public Integer get(int idx) {
        if(idx < 0 || idx >= size()) {
                return null;
        }

        Node temp = head;
        /*
        move forward idx times.
        if idx = 0, don't move forward at all
        if idx = 4, move forward four times
        ...
        */
        for(int i=0; i < idx; i++) {
                temp = temp.getNext();
        }
        return temp.getData();
}
```

4.2 Inserting an item at an arbitrary index

We can either pass the item to be inserted (in our case, an integer), or a **Node** containing the item as instance variable **data** as shown below:

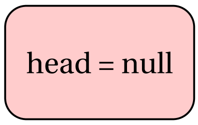


For simplicity, the diagram is reduced as follows,

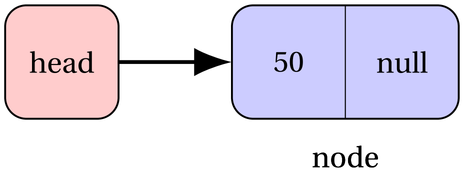


4.2.1 SCENARIO 1: Inserting in an empty list

We can only insert at index 0 in an empty list. When the list is empty, **head** is **null** .



In this case, all we have to do is re-refer **head** to **node** .



4.2.2 SCENARIO 2: Inserting in a non-empty list

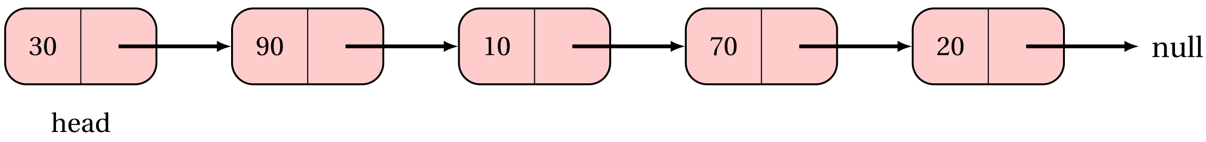
We can insert at any index from 0 (insert before the first item) to **size()** (insert after the last item).

If we are going to insert at index 0, **head** must be updated. In fact, this code also works when list is empty (**head** is **null**).

```java
if(index == 0) {
        node.setNext(head);
        head = node;
}
```

For any index more than 0, we follow the procedure described below:

Let's say we want to insert a node with data 50 at index 3 (after the 3rd item) in the following list.

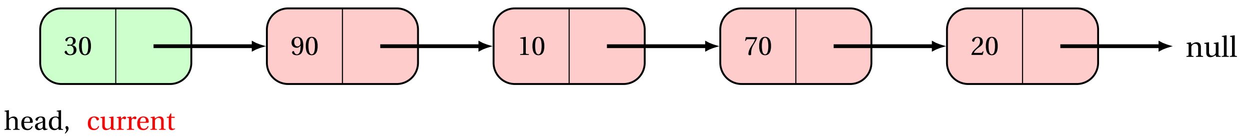


We must reach the 3rd item (at index 2) and manipulate it's **next** instance variable.

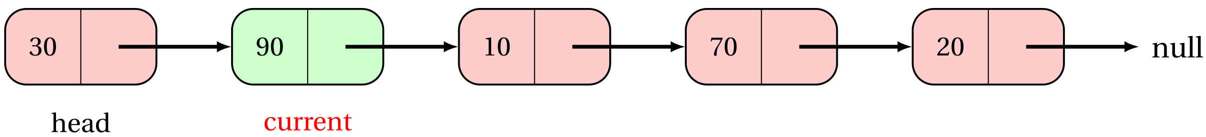
Starting with **head** , how many times must we *move forward* to reach the item at index 2? That's right - 2 times.

```java
Node current = head;
for(int i=1; i < 3; i++) {
        current = current.getNext();
}
```

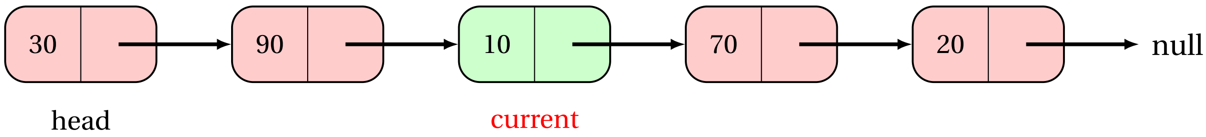
Initial state:



After iteration 1:

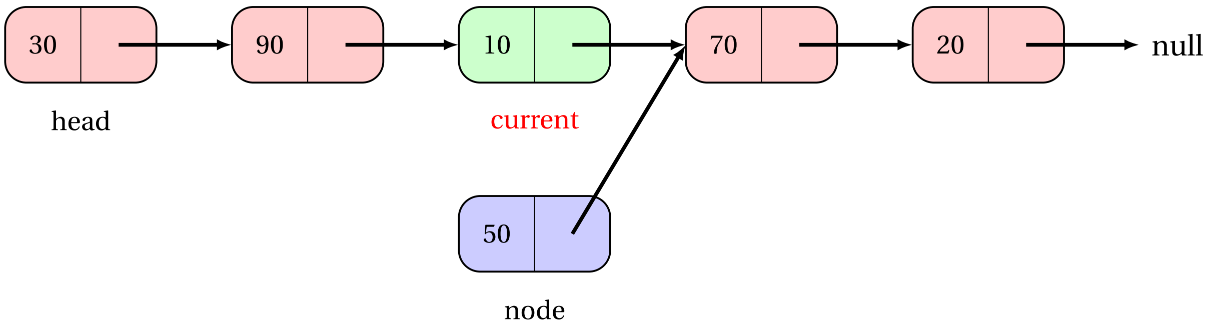


After iteration 2:



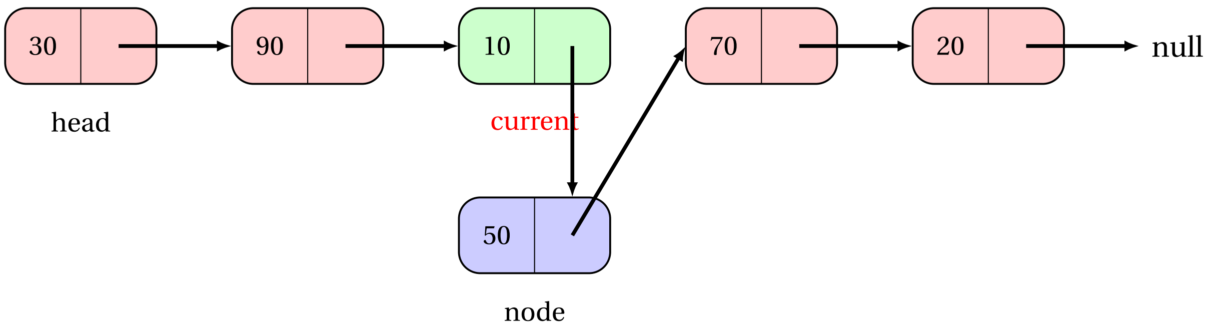
Then we have a reference to the 3rd item in **current** .
The node after **current** should be after the node to inserted.

```
node.setNext(current.getNext());
```

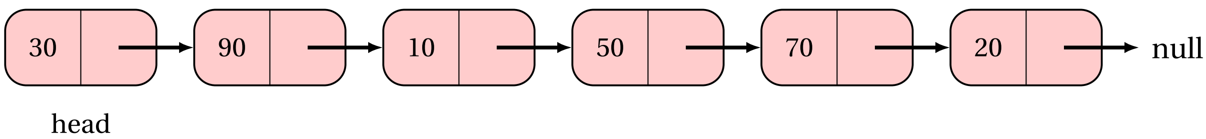


And the node to be inserted should be after **current** .

```
current.setNext(node);
```



Thus, the list becomes,



The overall code is below.

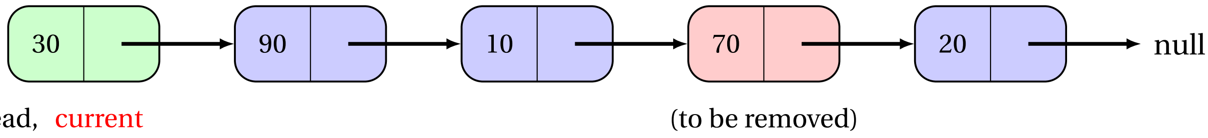
```
public void add(Node node, int idx) {
        if(idx < 0 || idx > size())
                return;

        // at the beginning of an empty or non-empty list
        if(index == 0) {
                node.setNext(head);
                head = node;
        }

        Node current = head;
        for(int i=1; i < idx; i++)
                current = current.getNext();
        node.setNext(current.getNext());
        current.setNext(node);
}
```

Note that **idx == size()** refers to adding an item at the end of the list and the above code works for the same.

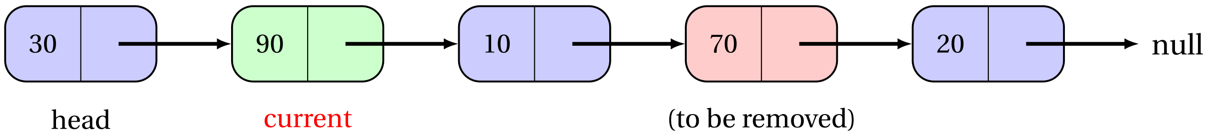
4.3 Removing an item from an arbitrary index

If we need to remove an item at a particular index **idx** (in this example **idx = 3**), we need to reach the node **before** the node to be removed. For this, we move forward **idx-1** times. Then, simply link the current node (at index **idx-1**) to the node at index **idx+1** .

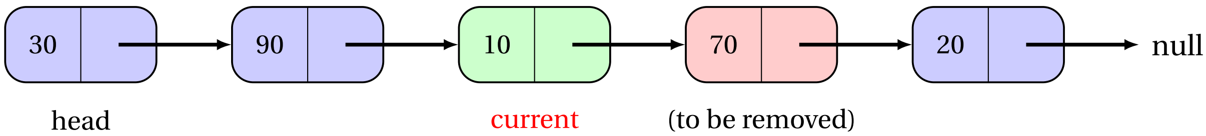
Initial state **(item to be removed in red)**:



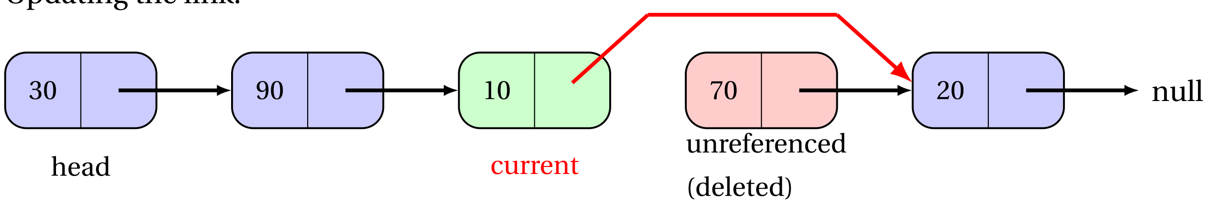
After iteration 1:



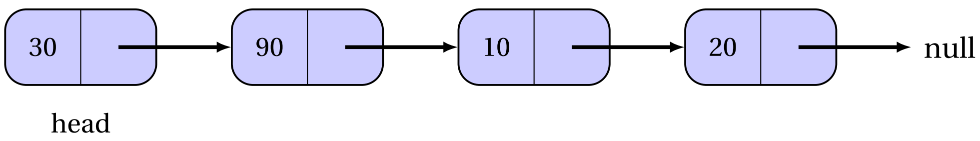
After iteration 2:



Updating the link:



End product:



```java
public Integer remove(int idx) {
        if(idx < 0 || idx >= size())
                return null;
        if(idx == 0) { //removing head
                double result = head.getData();
                head = head.getNext();
                return result;
        }
        Node current = head;
        for(int i=1; i < idx; i++)
                current = current.getNext();
        int result = current.getNext().getData();
        current.setNext(current.getNext().getNext());
        return result;
}
```

Section 5.

Sample solutions for exercises

Solution: Exercise 1

Solution not provided for this exercise

Solution: Exercise 2

```java
public int total() {
        Node temp = head;
        int result = 0;
        while(temp != null) {
                result = result + temp.getData();
                temp = temp.getNext();
        }
        return result;
}
```

Solution: Exercise 3

```java
public Integer highest() {
        if(head == null)
                return null;
        Node temp = head;
        int result = head.getData();
        while(temp != null) {
                if(temp.getData() > result) {
                        result = temp.getData();
                temp = temp.getNext();
        }
        return result;
}
```