# COMP 1020 - Linked Lists

UNIT 8

# Lists

- We covered ArrayLists in Java:
  - A combination of arrays, and lists… but what are lists exactly?

# Lists

- Previously, we covered ArrayLists in Java:
  - A combination of arrays, and lists… but what are lists exactly?
- In general, in Computer Science, a "list" is a sequence of data items where each has a position (1st, 2nd, etc.)
- Examples:
  - List of 5 integers: 34, -67, 21, 5, -13
  - List of 3 Strings: "Fred", "likes", "COMP1020"
  - List of 0 doubles:

# Lists

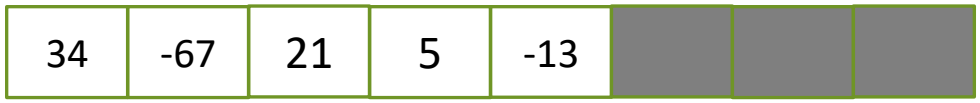
- We have used lists a lot, and have stored them in different ways:
  - As a full array:  new int[ ] {34,-67,21,5,-13}
  - As a partially-full array: just an array with a separate size variable
  - As an ArrayList
    - That's just a partially-full array written by someone else really

# Lists

- Lists typically have other properties in most programming languages:
  - Insertion and deletion of elements is easily doable without needing to restructure/reorganize the entire data structure (size can change easily)
  - We can access a specific element using its position, such as an index that counts from zero.

# Physical adjacency

- Arrays and ArrayLists have two things in common:
  - The elements are stored physically adjacent in an array, like this:



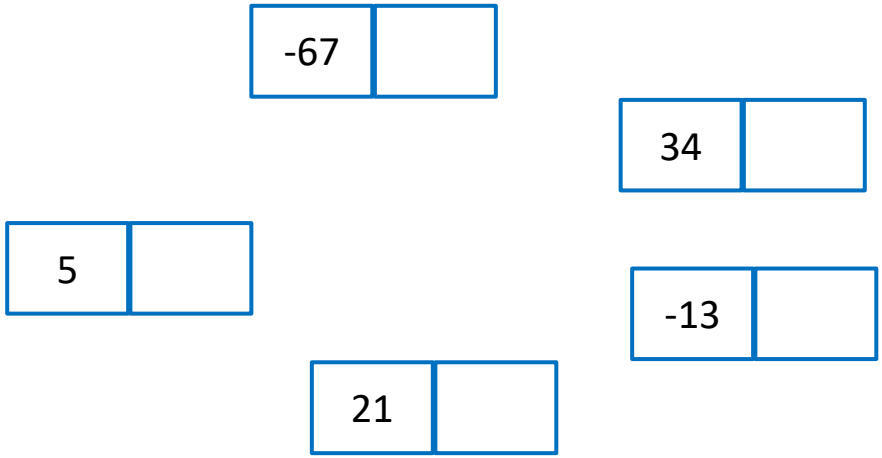

    - not like this:


      - To add or delete an element from the middle or the front requires other elements to be shifted
  - The array might become full (or be full all the time)
    - To add another element (when full) requires a complete re-build of the array into a new one

# A Linked list

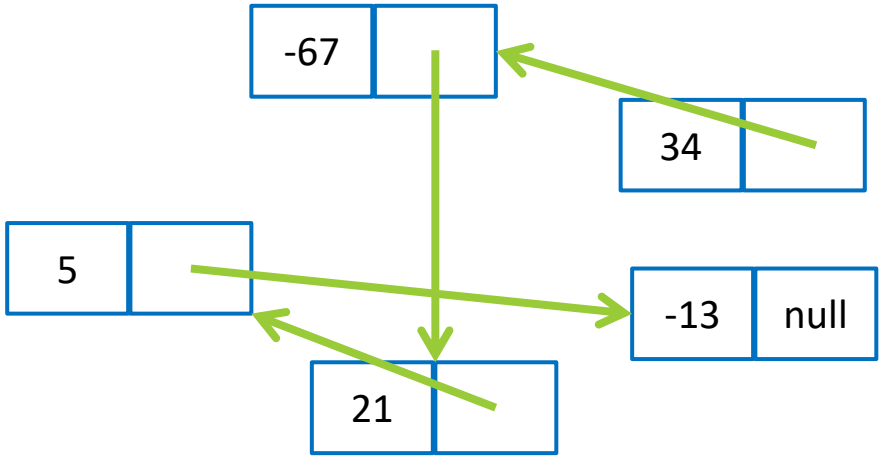
- A linked list is one specific type of a list (there exists others)
- A linked list solves both of these problems (but creates a few more of its own)

# A Linked list



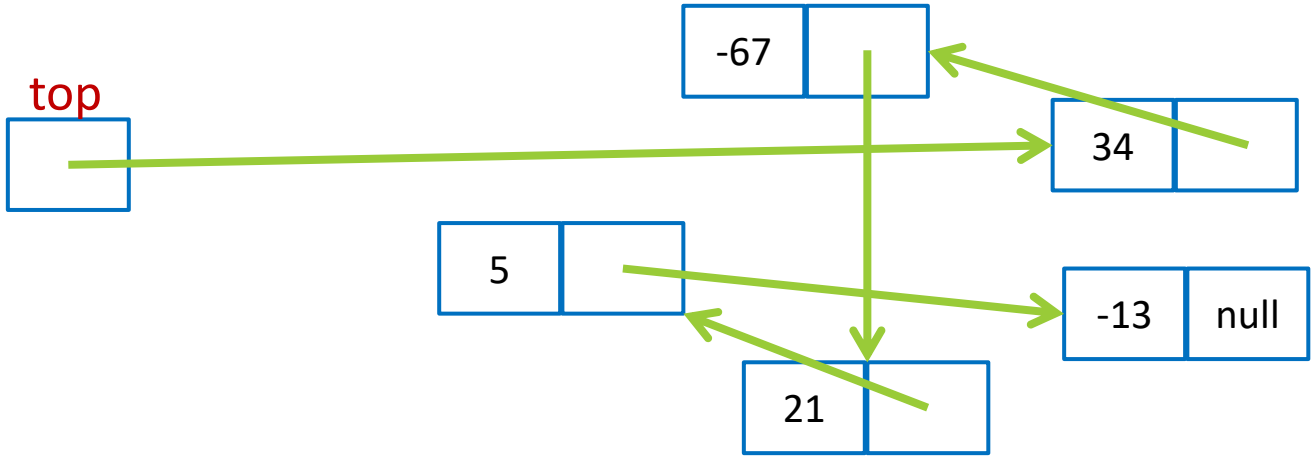

- The data is stored in a set of Node objects

# A Linked list




- The data is stored in a set of Node objects
- Each Node contains the data (or a reference to it), and a reference to the next Node in the list (or null if there isn't one)

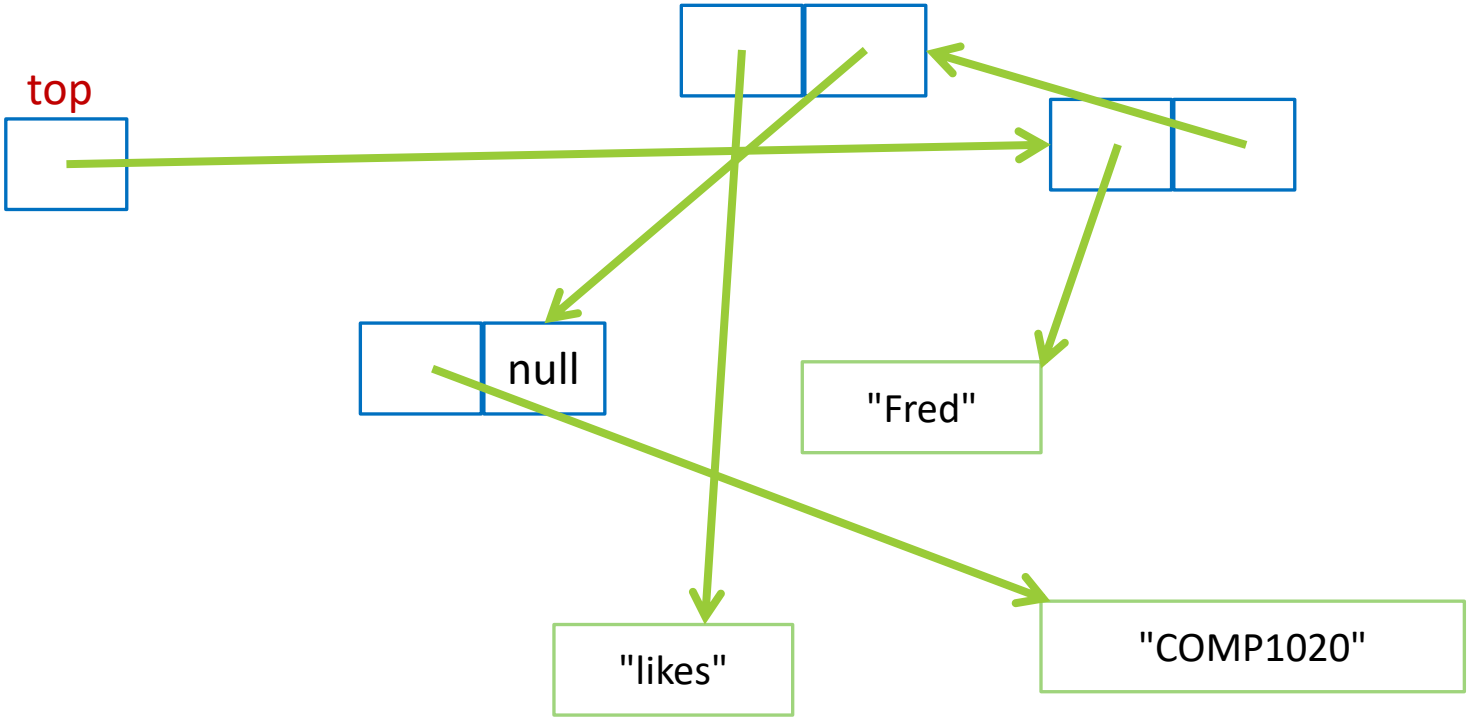
# A Linked list




- The data is stored in a set of Node objects
- Each Node contains the data (or a reference to it), and a reference to the next Node in the list (or null if there isn't one)
- A "top pointer" (a reference to the first Node) will give you access to all of them, through the first one

# A Linked list of objects

- If the data items are objects, then they're stored as references, too, as objects always are



# Code for a generic linked list

- Let's create a linked list of Objects (that way, it can handle any kind of data)
- It's best to define two classes:

```
public class LinkedList {
    private Node top; //The reference to the first Node
}

public class Node {
    public Object data; //The data in this Node
    public Node link; //The link to the next Node
}
```

# Wait… “public”?

- The instance variables in a Node are public. What about the rule that all instance variables must be private?

- This is a special case: because the Nodes are private in the LinkedList, only the linked list methods have access to them. Node is a “friend” to LinkedList.

- The consequence for LinkedLists is that the Nodes must stay private. That is, we can never return a Node from a (public) LinkedList method.

# Constructors

- We don't really need to write this, though we should:

```
public LinkedList( ) {
    top = null; //It's null by default anyway.
}
```

- But we'll need this:

```
public Node(Object initData, Node initLink){
    data = initData;
    link = initLink;
}
```

# An add method

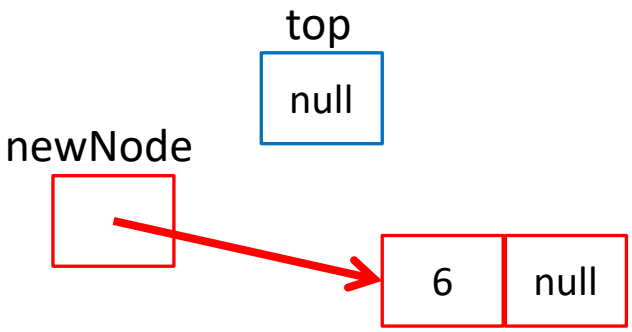
- The set methods on the previous slides only change the data (or link) of an existing Node
- Let's write a method to add a new piece of data to our list
  - It's much easier to add new elements at the beginning
  - Unlike arrays, where it's easy to add them at the end

# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list (if the list is empty):

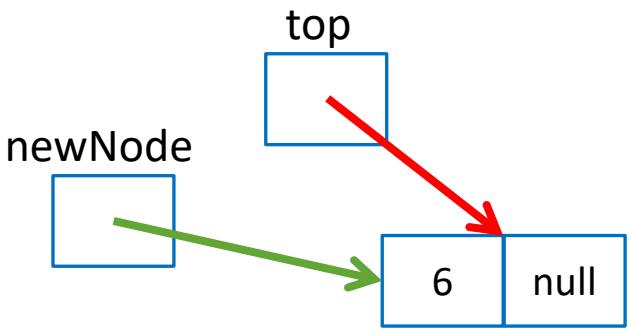


# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list (if the list is empty):

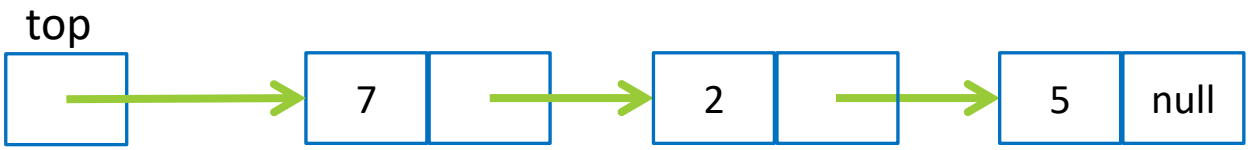


# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list: (assuming Integer data)

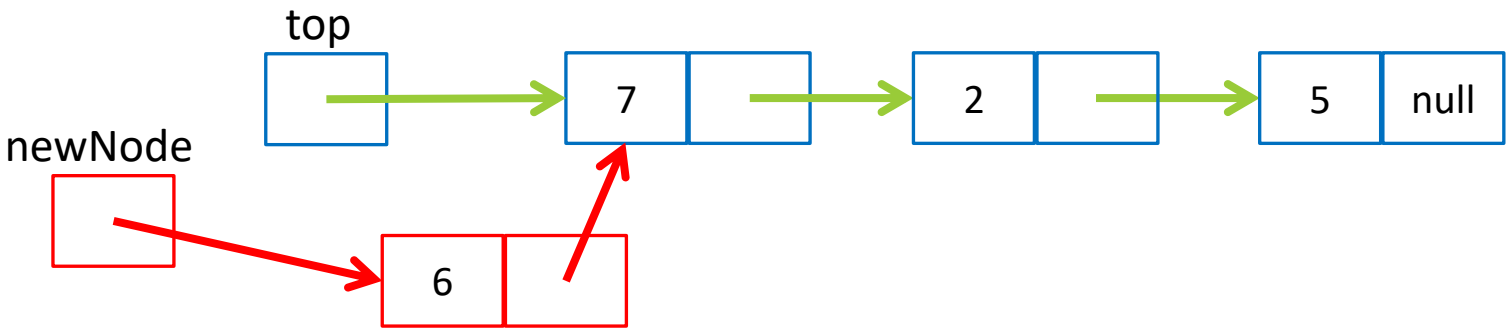


# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list:

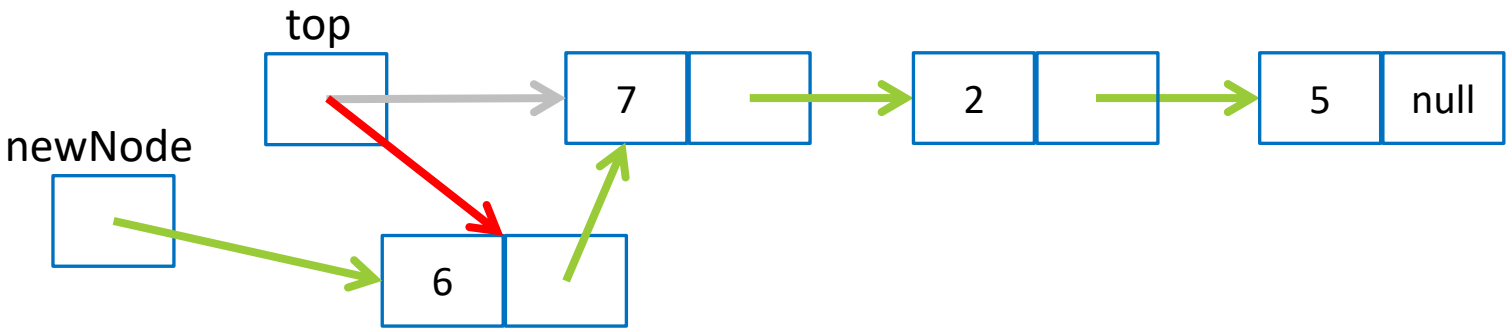


# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list:

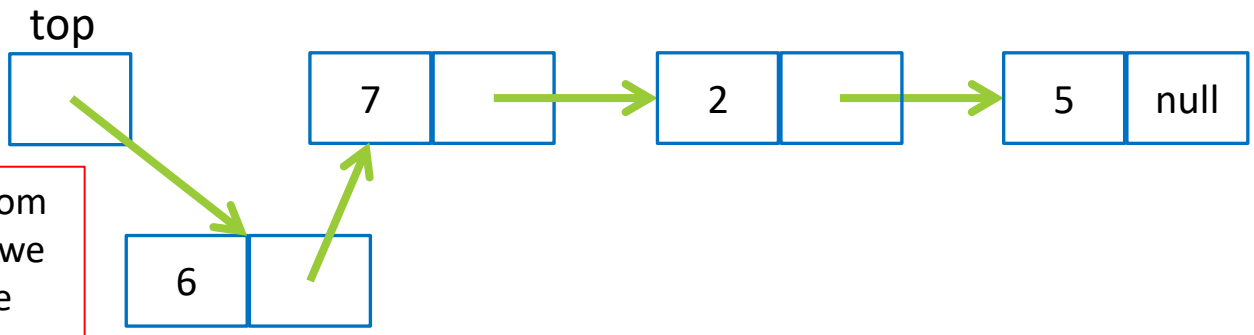


# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list:



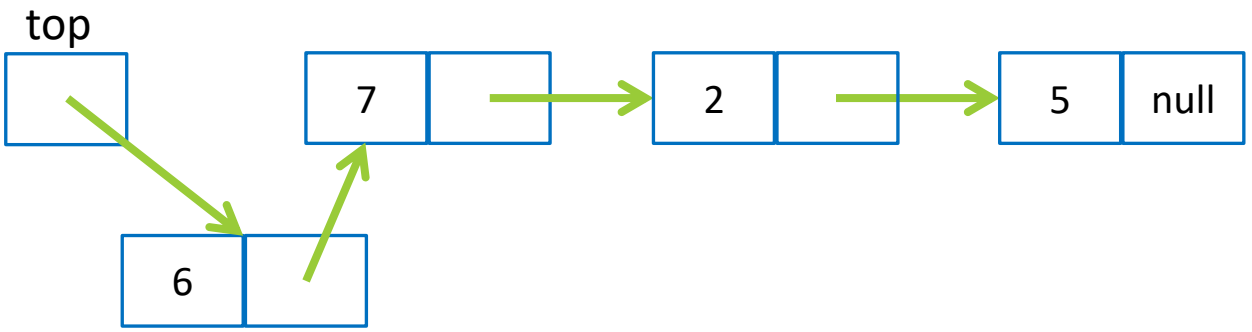

After returning from the add method, we lose the newNode reference

# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list:



# An add method

- Add a new element to the beginning of a LinkedList (This is in the LinkedList class)

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

- What is happening in the list (if the list is empty):

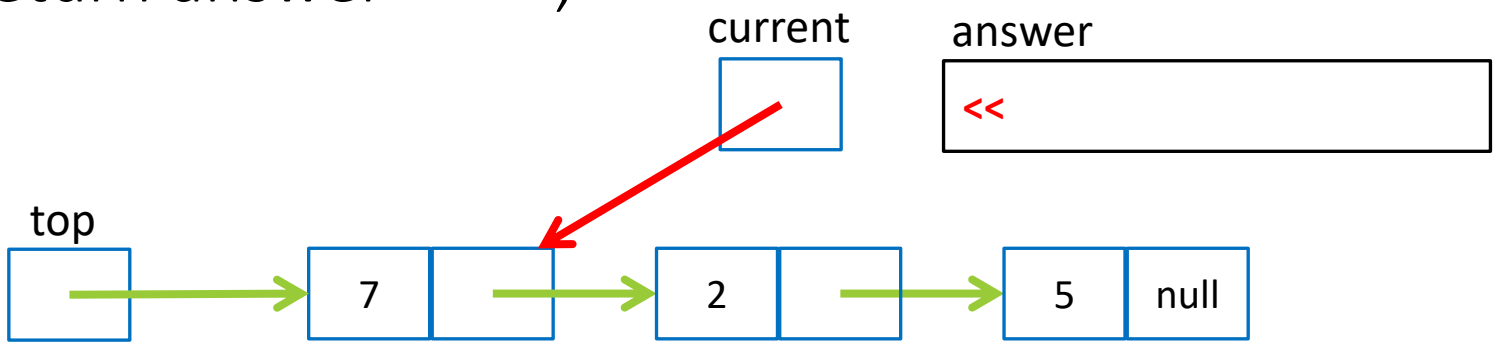
# A toString method

- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```

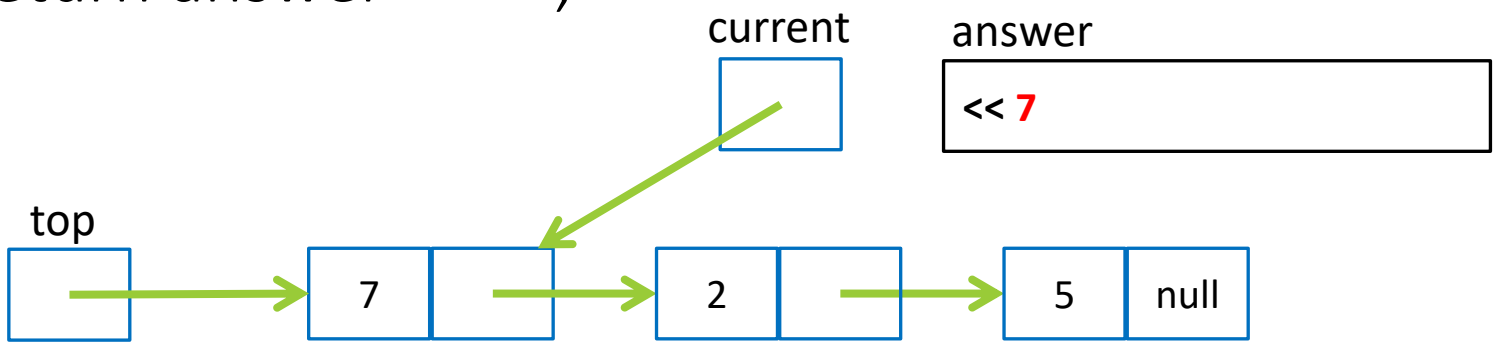
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



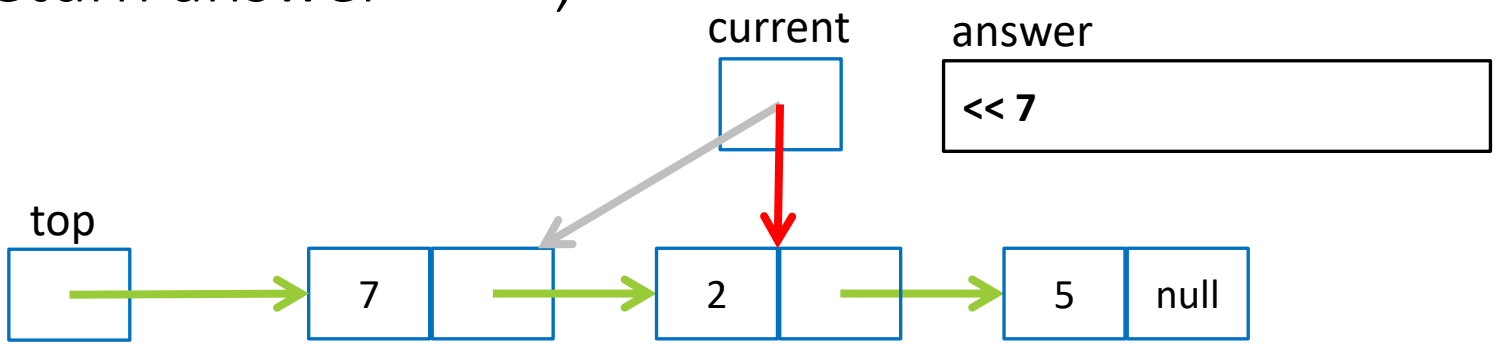
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



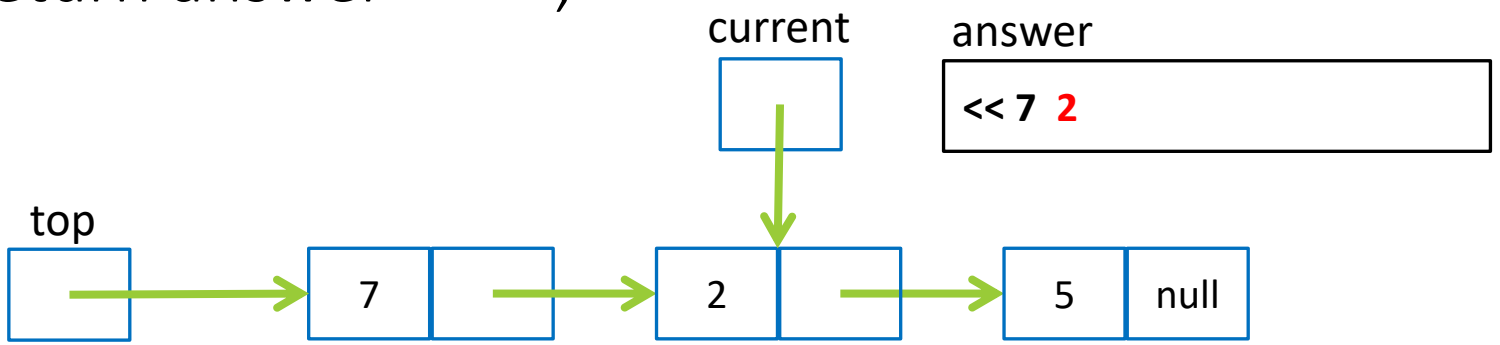
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



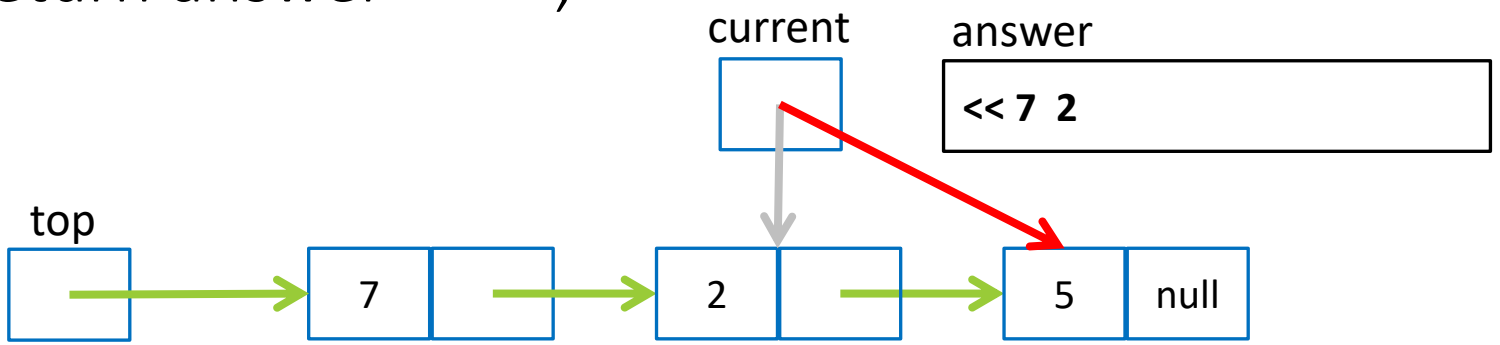
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



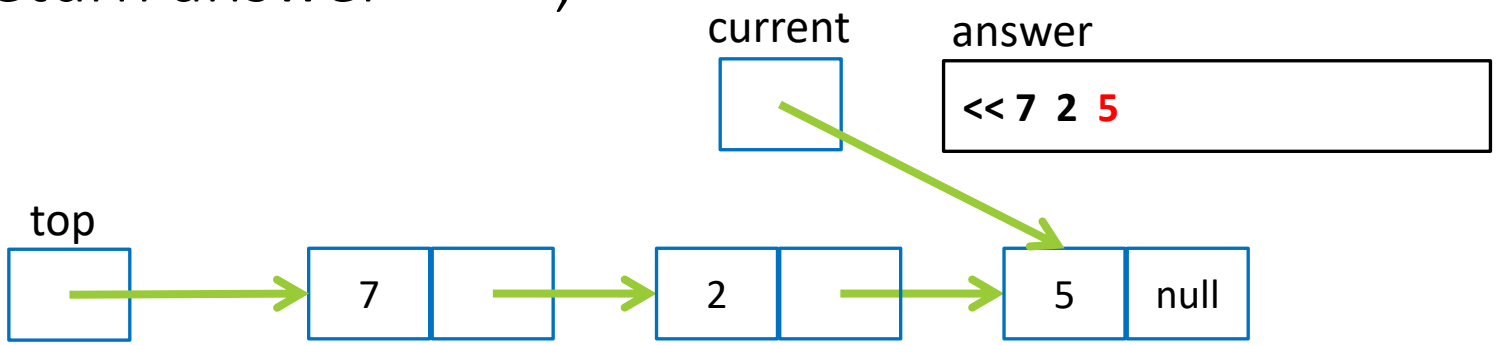
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



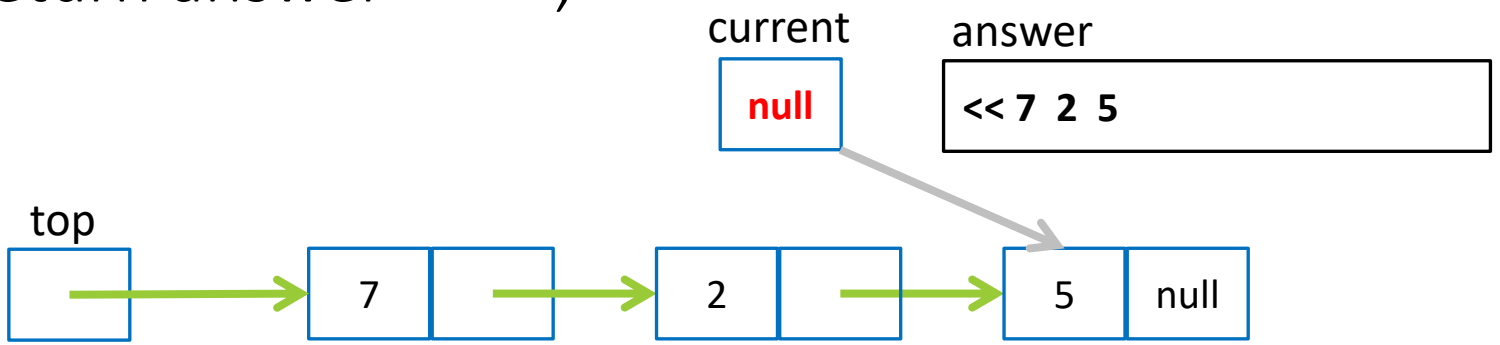
# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```



# Follow the execution

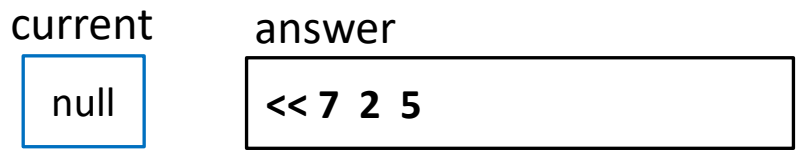
```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```

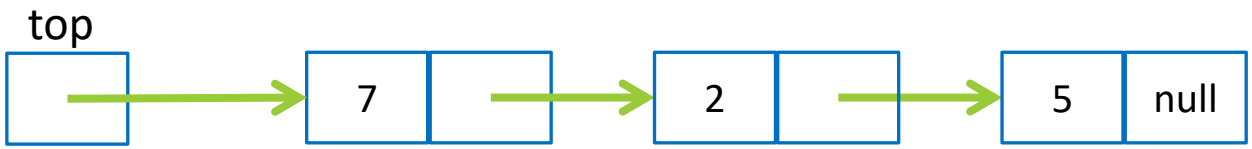


# Follow the execution

```
public String toString() {
    String answer = "<< "; //Why not? Looks like “LL”
    Node current = top; //Start at the first one
    while(current != null) {
        answer += current.data + "  ";
        current = current.link; //Advance to the next
    }
    return answer+">>";
}
```

<< 7 2 5 >>



# Try these in a main!

```
LinkedList ll = new LinkedList();
ll.add("Fred");
ll.add(345); //Mix them up.
ll.add("last");
System.out.println(ll.toString());
```

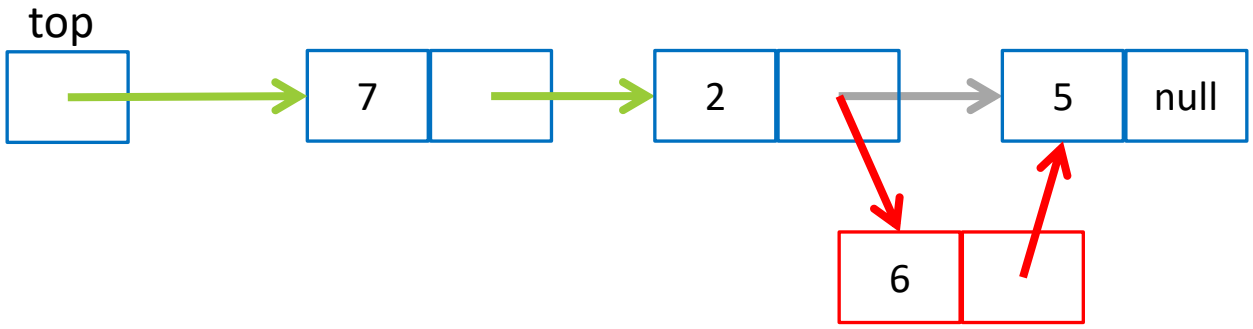
**you will get:**

<< last 345 Fred >>

Note that last is first!

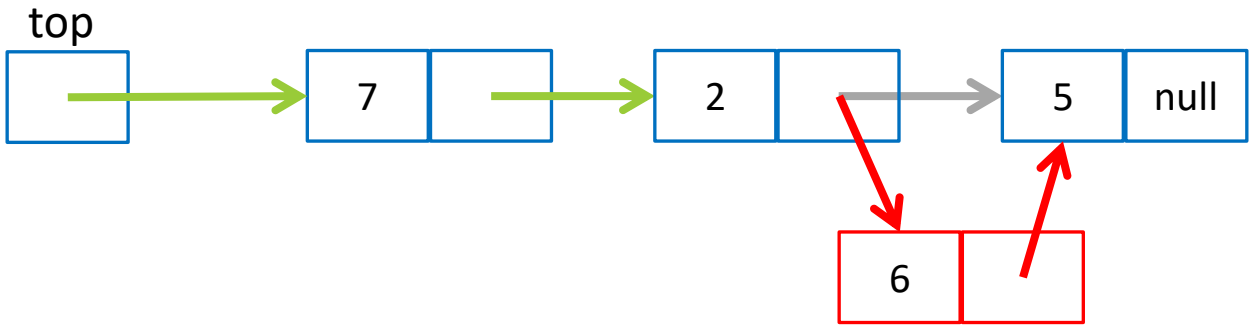
# Add to a particular position

- ArrayLists have add(n,data) to add data at index n of a list. Let's do the same!
- What parts of the list must change?



# Add to a particular position

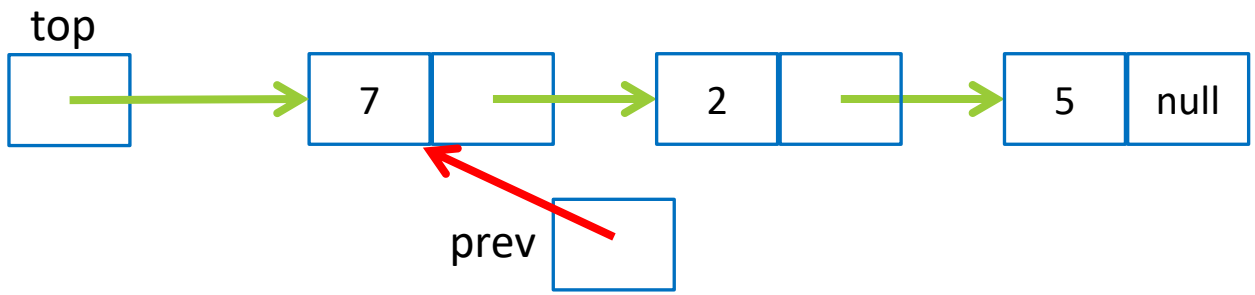
- ArrayLists have add(n,data) to add data at index n of a list. Let's do the same!
- What parts of the list must change?




- To put a new node at position 2, it's necessary to change the link in the node at position 1
  - We have to find the previous node!

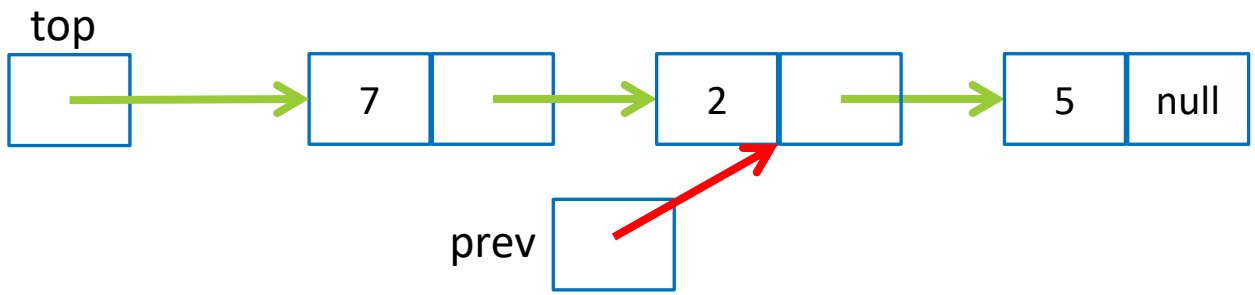
# Add to a particular position

```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```



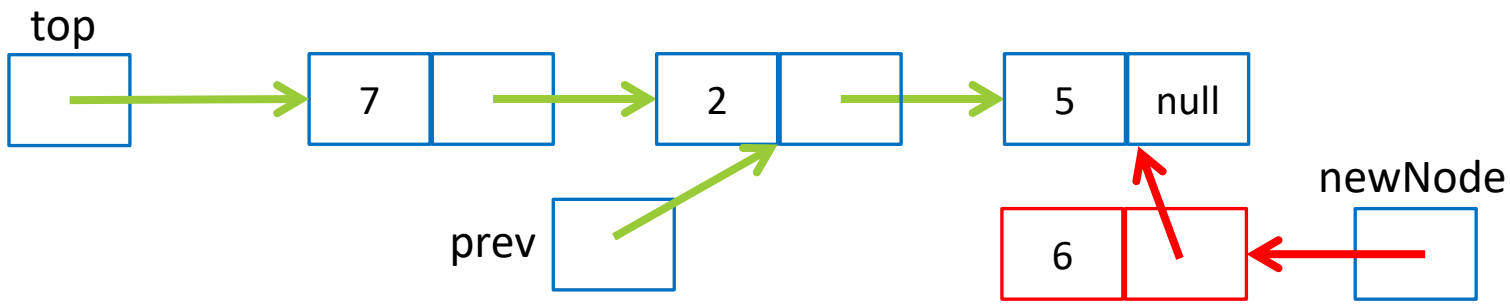
# Add to a particular position

```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```



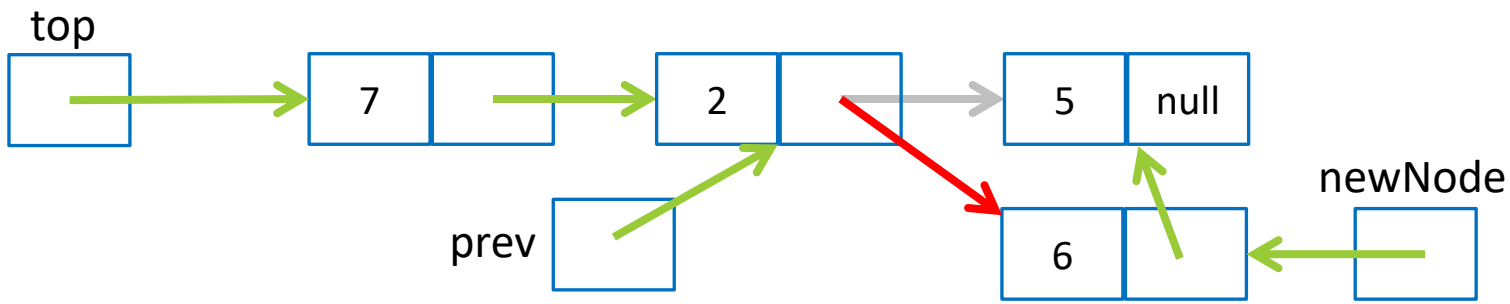
# Add to a particular position

```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```



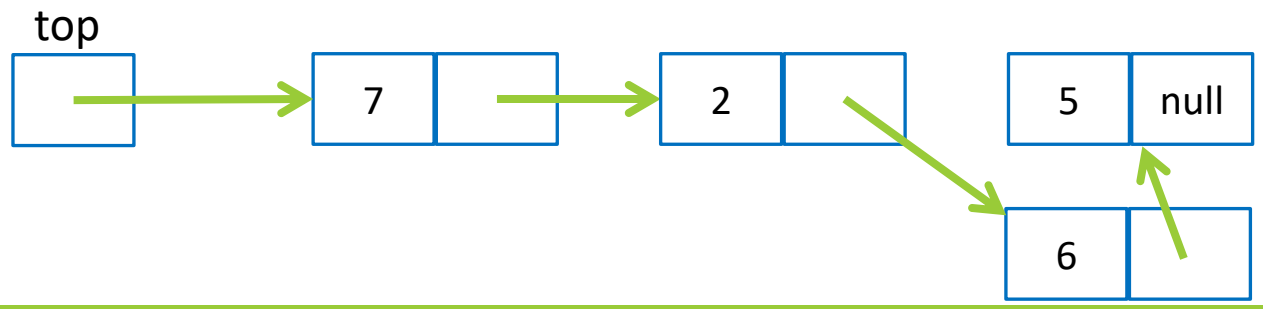
# Add to a particular position

```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```



# Add to a particular position

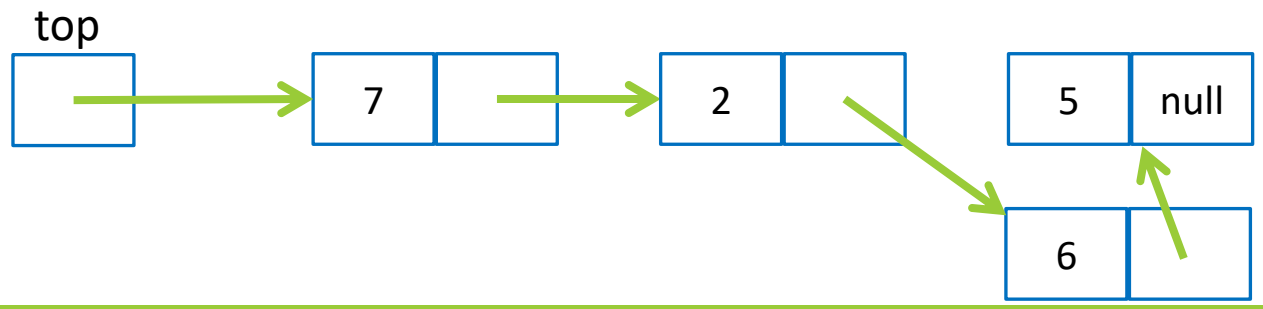
```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```



# Add to a particular position

```
public void add(int position, Object newItem) {
    //There will be no error checking on position.
    if(position==0)
        add(newItem);
    else {
        //Make prev point to node with index position-1
        Node prev = top; //Start at the first node
        //follow position-1 links to get there
        for(int i=0; i<position-1; i++)
            prev = prev.link;
        Node newNode = new Node(newItem,prev.link);
        prev.link = newNode;
    }
}
```

Note that the order of these two operations is important! It will not work if we update the previous link first, because then we'll lose the link to the next node!



# Add to a particular position

- The previous method is not safe… why?

# Add to a particular position

- The previous method is not safe… why?
  - It does not check if the position exists!

# Add to a particular position

- The previous method is not safe… why?
  - It does not check if the position exists!
- Let's make a getNode(int n) method to make our life easier
  - then we'll fix our
    add(int position, Object newItem)
    method using it

# getNode(int n)

```
//get Node at index n: note that the method is private because we
//   can't let a Node out of our LinkedList class!
//a get(int n) that returns the data at position n could be public.
private Node getNode(int n)
{
      Node current = top;
      while(n > 0)
      {
            if(current == null)
                  return null;
            current = current.link;
            n--;
      }
      return current;
}
```

# Add to a particular position (SAFE)

```
public void add(int position, Object newItem) {
    if(position == 0) {    //add to front
        add(newItem);
        return;
    }

    //Finding the previous node, if it exists (null otherwise)
    Node previous = getNode(position-1);

    if(previous == null)
        throw new IndexOutOfBoundsException("Cannot add at position" +
                position);  // Why not? We are experts in Exceptions now!

    else
    {
        Node newNode = new Node(newItem, previous.link);
        previous.link = newNode;
    }
}
```

# Add to a particular position (SAFE)

```
public void add(int position, Object newItem) {
    if(position == 0) {    //add to front
        add(newItem);
        return;
    }

    //Finding the previous node, if it exists (null otherwise)
    Node previous = getNode(position-1);

    if(previous == null)
        throw new IndexOutOfBoundsException("Cannot add at position" +
                position);  // Why not? We are experts in Exceptions now!

    else
    {
        Node newNode = new Node(newItem, previous.link);
        previous.link = newNode;
    }
}
```

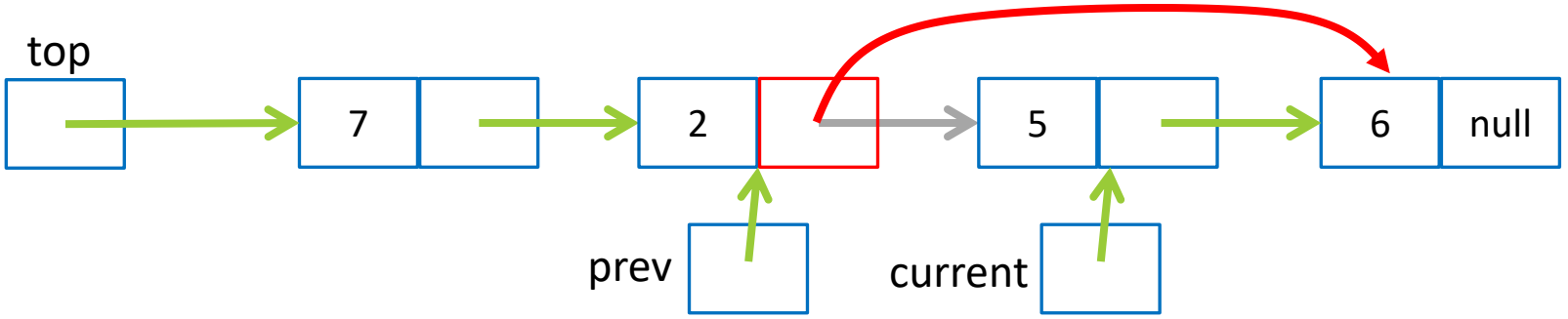
This makes our code much shorter and cleaner!

# add: array vs. linked list

- When adding to the middle of a list:
  - Array:
    - Must "shuffle" all elements after that (Loop needed)
    - Can access the desired position directly
    - Might get full and require expansion of the array
  - Linked list:
    - No need to shuffle anything. Very quick insertion.
    - Must follow the chain through all previous elements to find the right position (Loop needed)
    - Can't get full (Unless you completely run out of memory, which is unlikely)

# Deletion

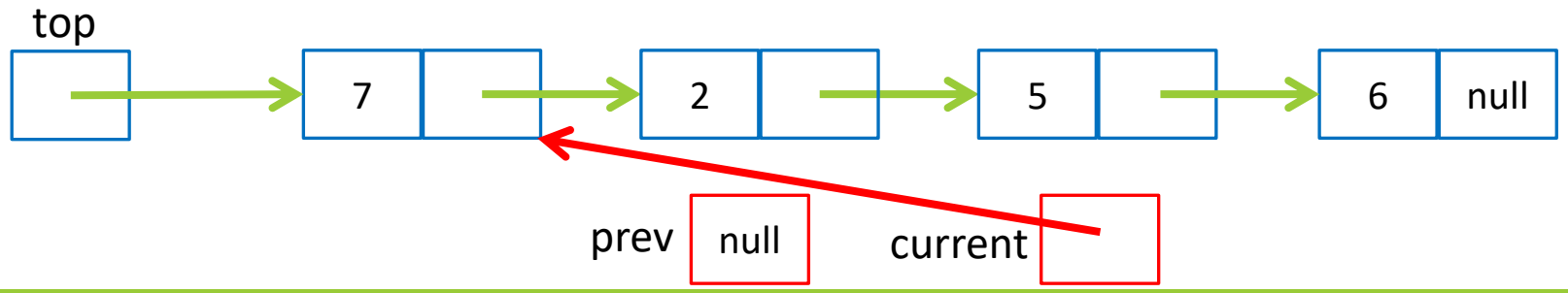
- Write the equivalent of ArrayList's remove(x) → where x is the element, not the index
- What we need to do to remove the 5:




- We need to move some Node variable (current) through the list, looking for the correct value
- But we need to keep track of the previous node (prev) because that's the only one that actually needs to change!

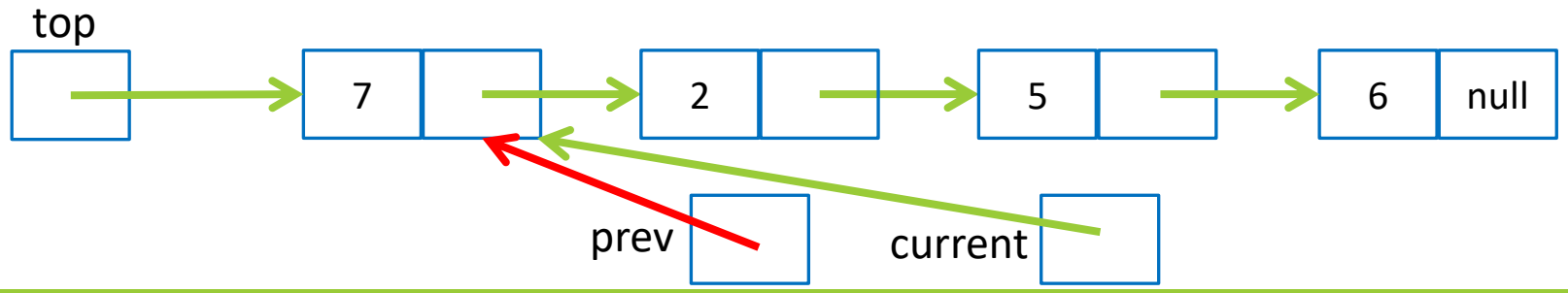
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



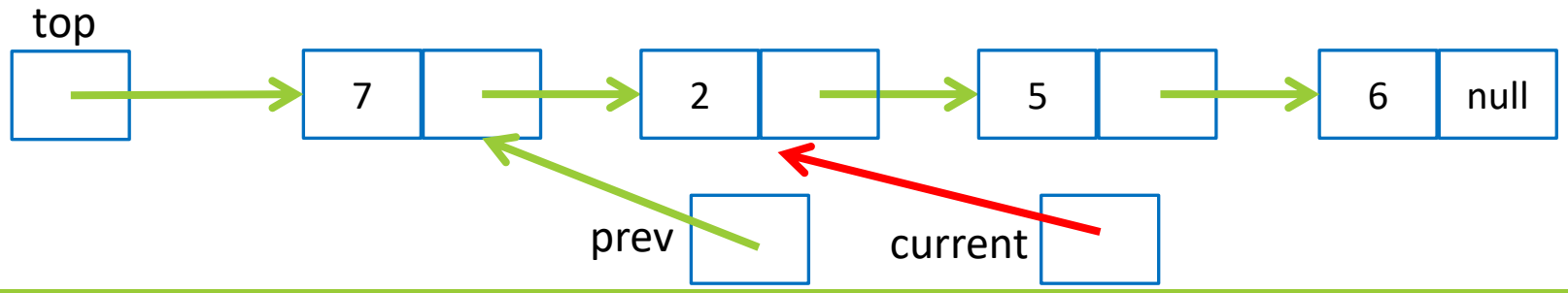
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



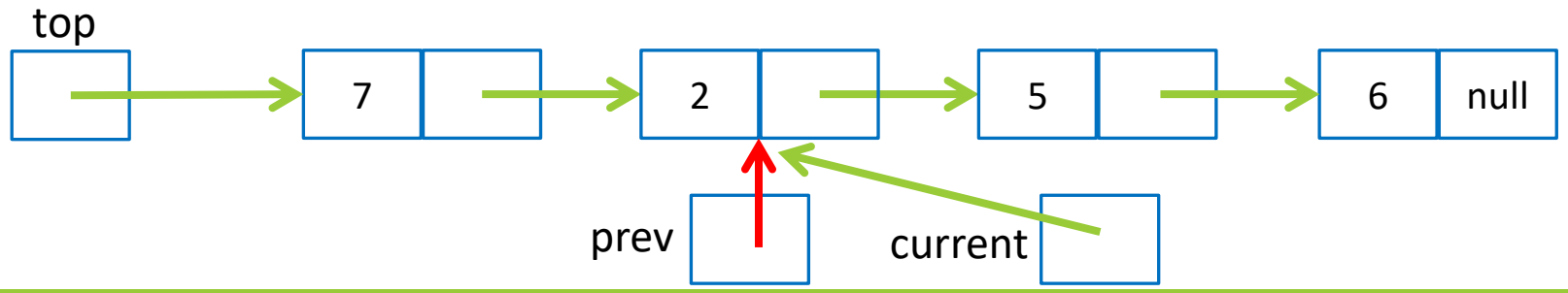
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



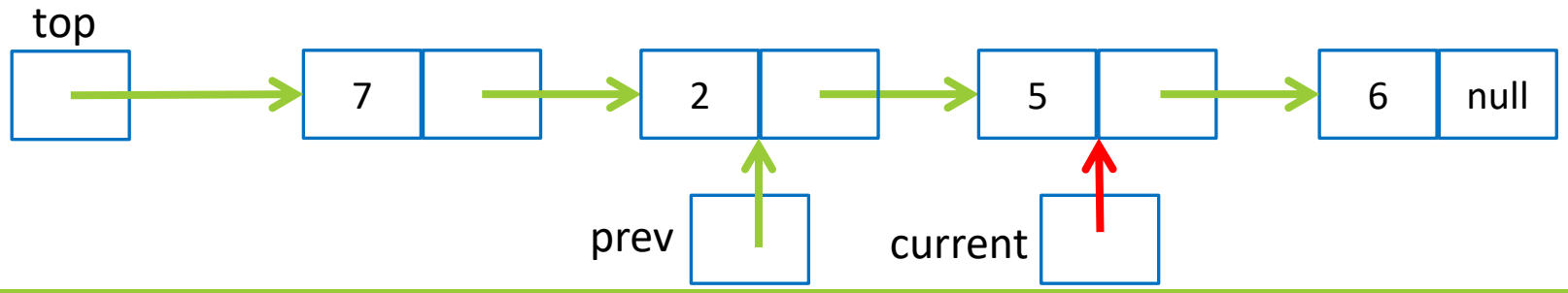
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



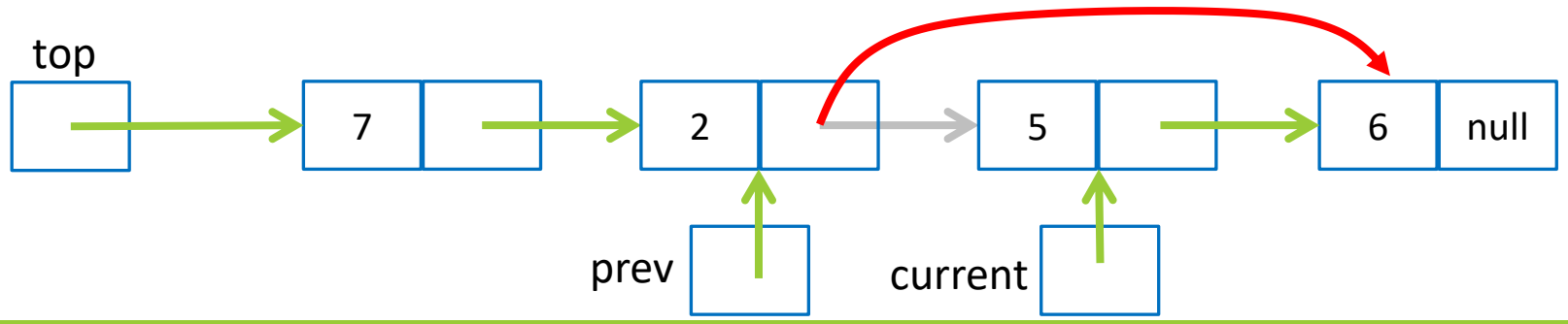
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



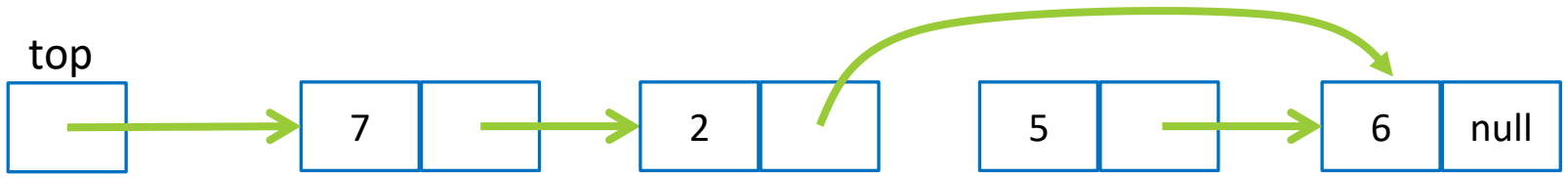
# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



# Deletion

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```

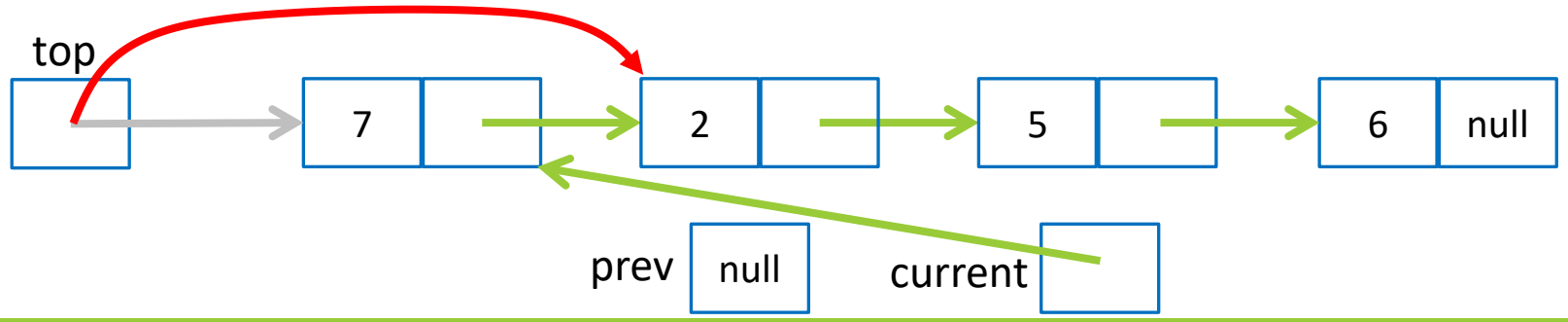


# If the node to delete is the first one?

- The while loop does no iteration at all
- The prev reference will still be null → that's how we are going to know that the first node needs to be deleted
- We only have to change top!

# Deletion of the first node

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



# One thing is missing…

- In its current state, the remove method is not completely safe… why?

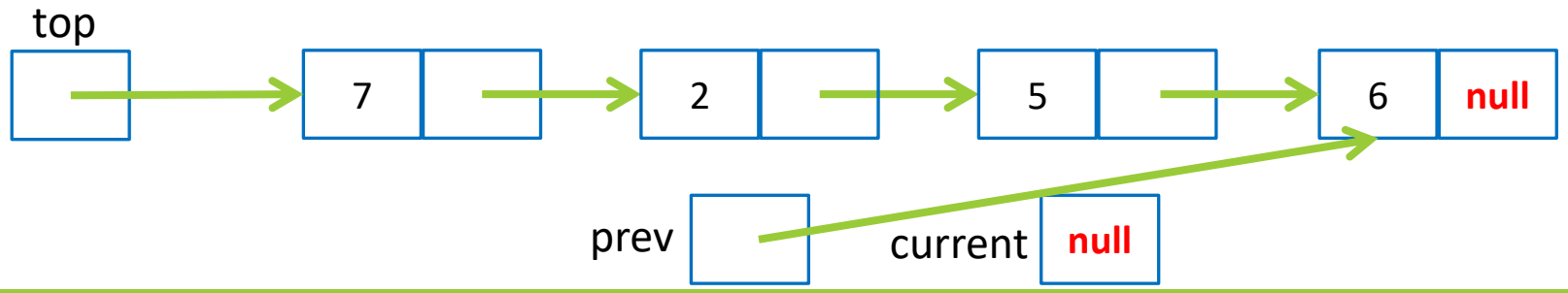
# One thing is missing…

- In its current state, the remove method is not completely safe… why?

- Calling remove for an element that is not in the list will result in a NullPointerException!

# Remove unexisting element

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(!current.data.equals(key)) { //throws a NullPointerException!
                prev = current;
                current = current.link;
        }
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```



# Remove unexisting element: Fix

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(current != null && !current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if (current == null)
                return;  //nothing to remove!
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```

# Lazy boolean evaluation

```
public void remove(Object key) {
        Node prev = null;
        Node current = top;
        while(current != null && !current.data.equals(key)) {
                prev = current;
                current = current.link;
        }
        if (current == null)
                return;  //nothing to remove!
        if(prev==null)
                top = current.link;
        else
                prev.link = current.link;
}
```

Why does this work? Why is it safe? → because of what we call "lazy boolean evaluation".

Since the two conditions on both sides of the && need to be true for the resulting expression to be true, programming languages only evaluate the second argument if the first was not enough to determine the value of the expression. In this case, when the first argument is false, we know the expression will be false.

# Remove at a specific position

- Try to do it yourself!
- Using the getNode method could be useful!

# Remove at a specific position

```
public void removeIndex(int n)
{
	if (n == 0) //removing the first node
	{
		if (top != null) //making it safe, in case the list is empty!
			top = top.link;
	}
	else
	{
		Node prev = getNode(n-1);
		if (prev != null && prev.link != null) //Otherwise, nothing to remove!
			prev.link = prev.link.link; //Skipping over
	}
}
```

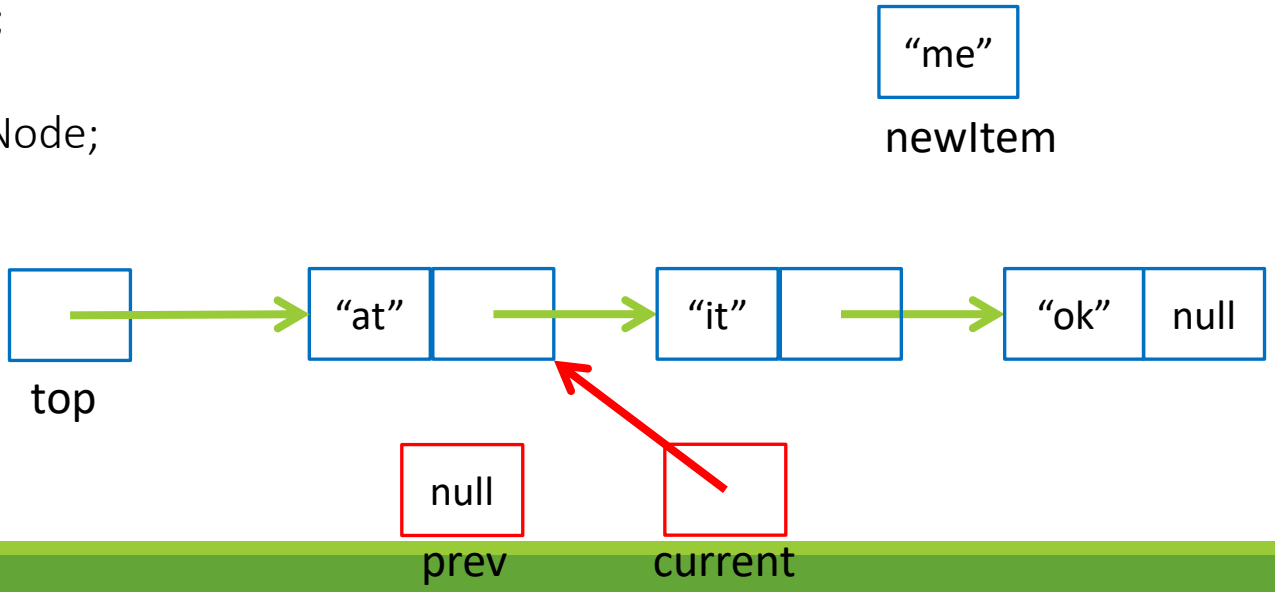
# One more

- We could keep the elements in the list sorted into ascending order
  - But not because we want to use a binary search
  - Binary search is impossible on a linked list!
    - On a list of 1000 items, it relies on having direct and quick access to a[500] (then 250 or 750, etc.)
    - But to access the 500th element, a linked list would have to go through all of the previous 499
    - This completely defeats the purpose of a binary search
  - There are other data structures that allow VERY quick searching (wait for COMP 2140)

# Ordered insertion

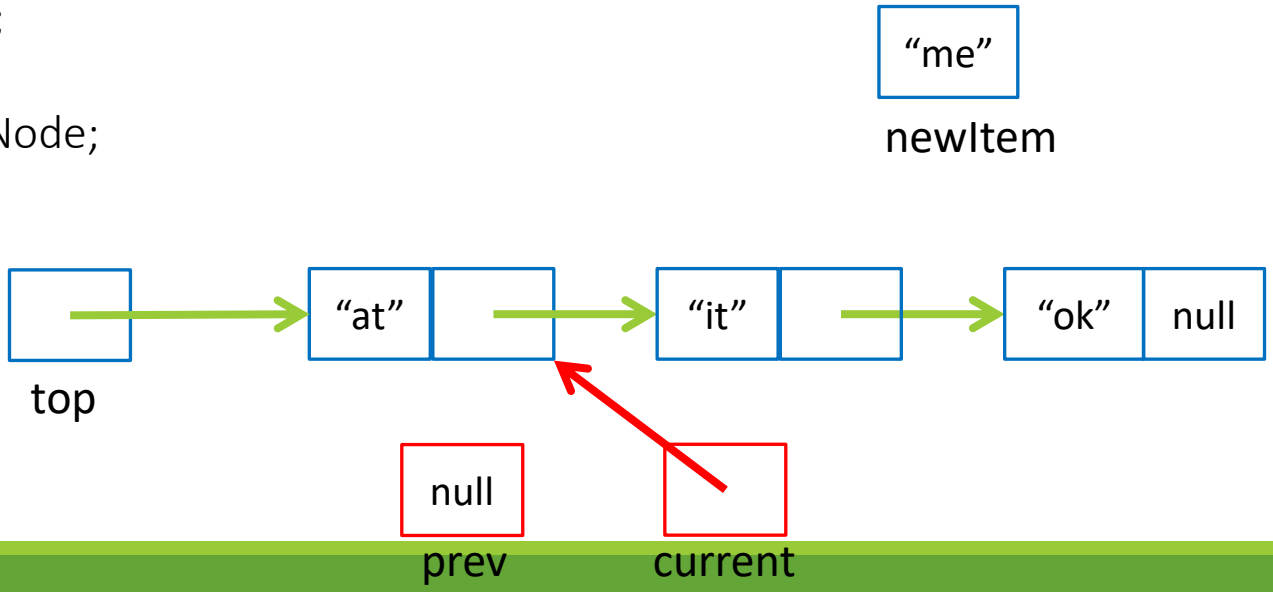
```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```

Object does not have the compareTo method, so we need to downcast



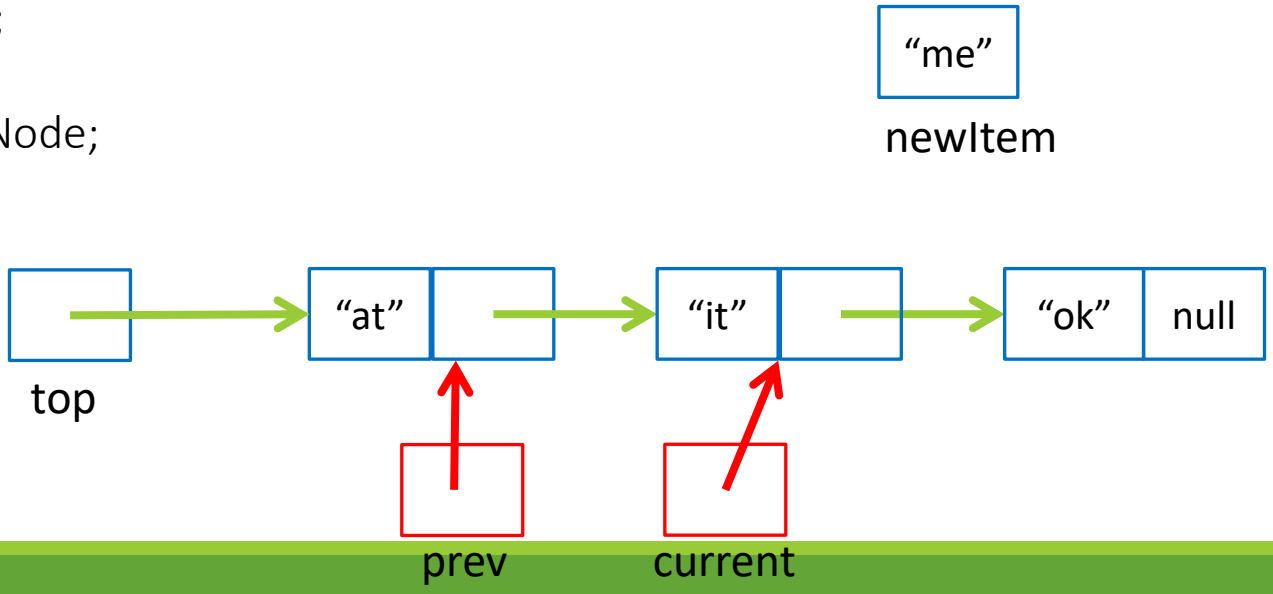
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



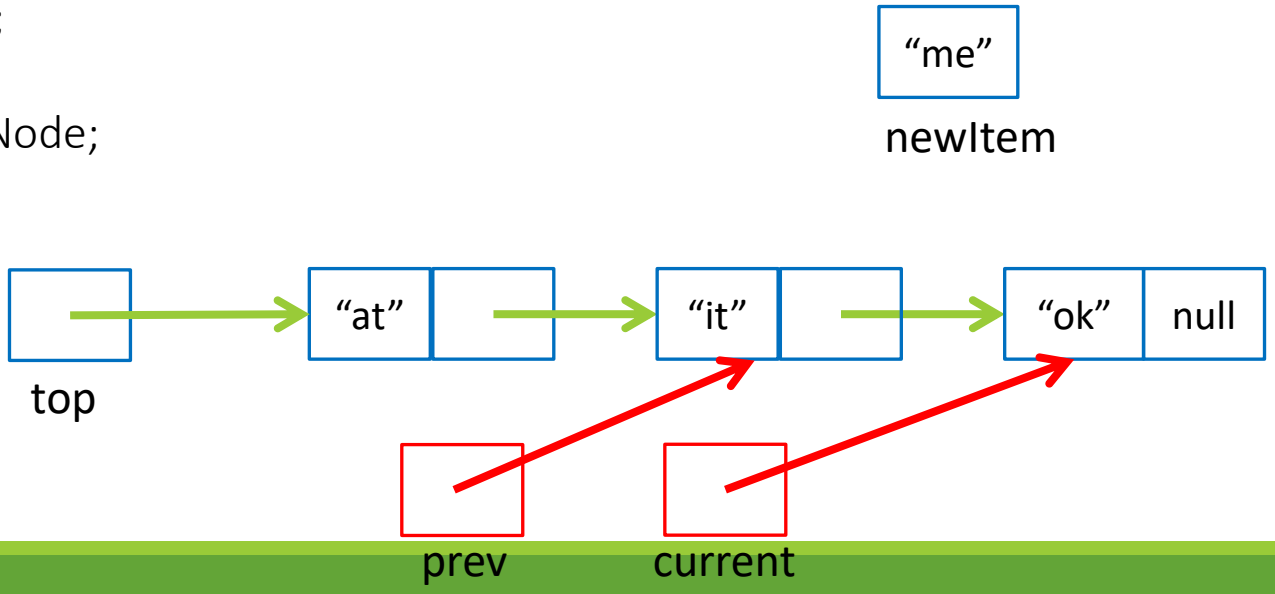
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



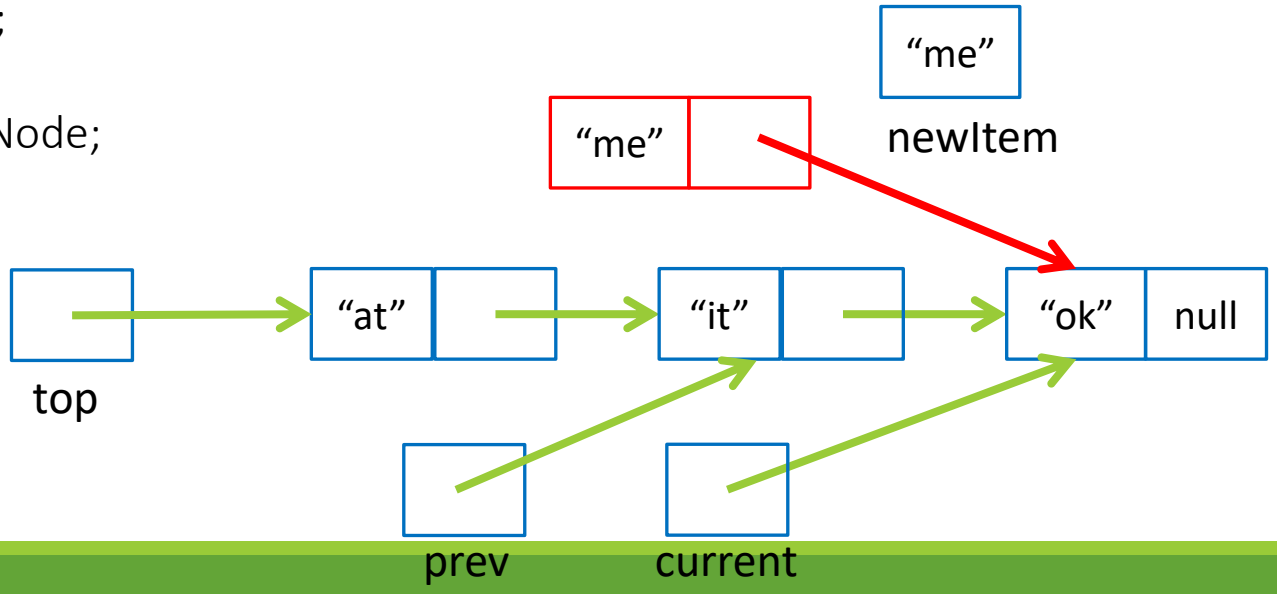
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



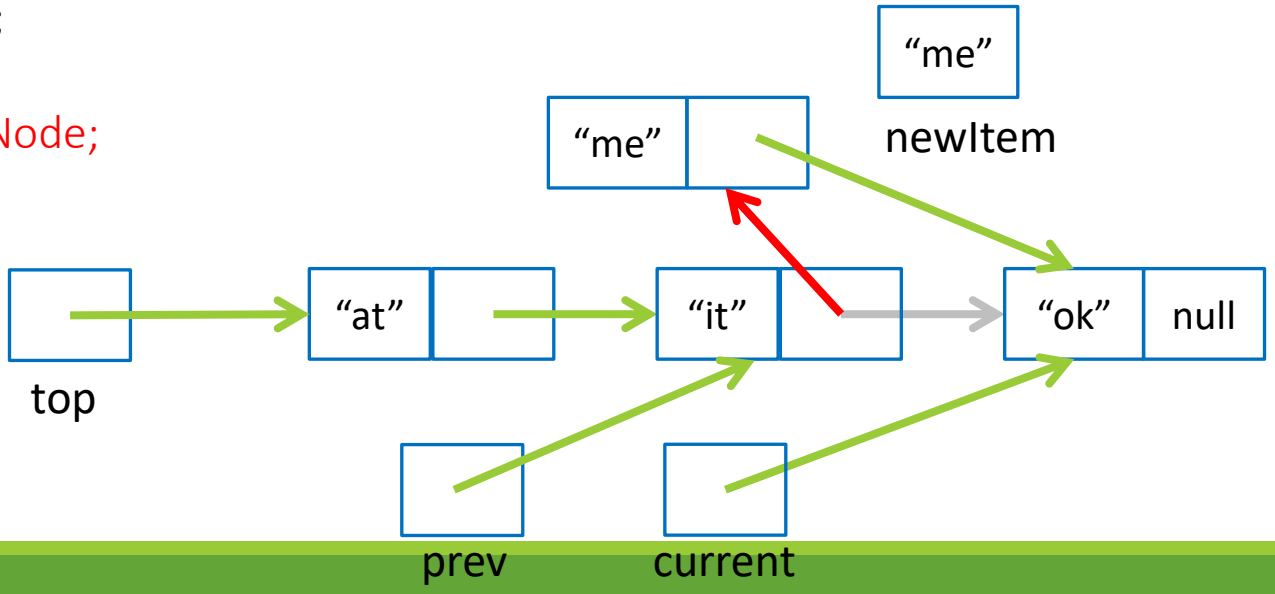
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



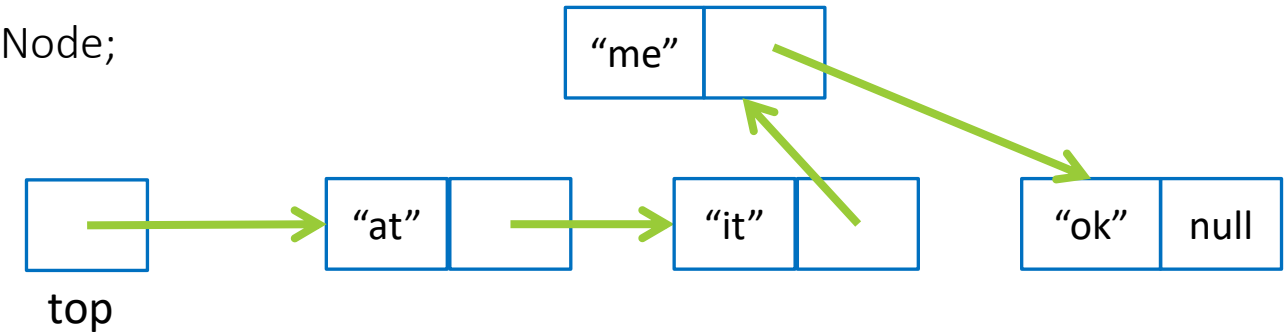
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



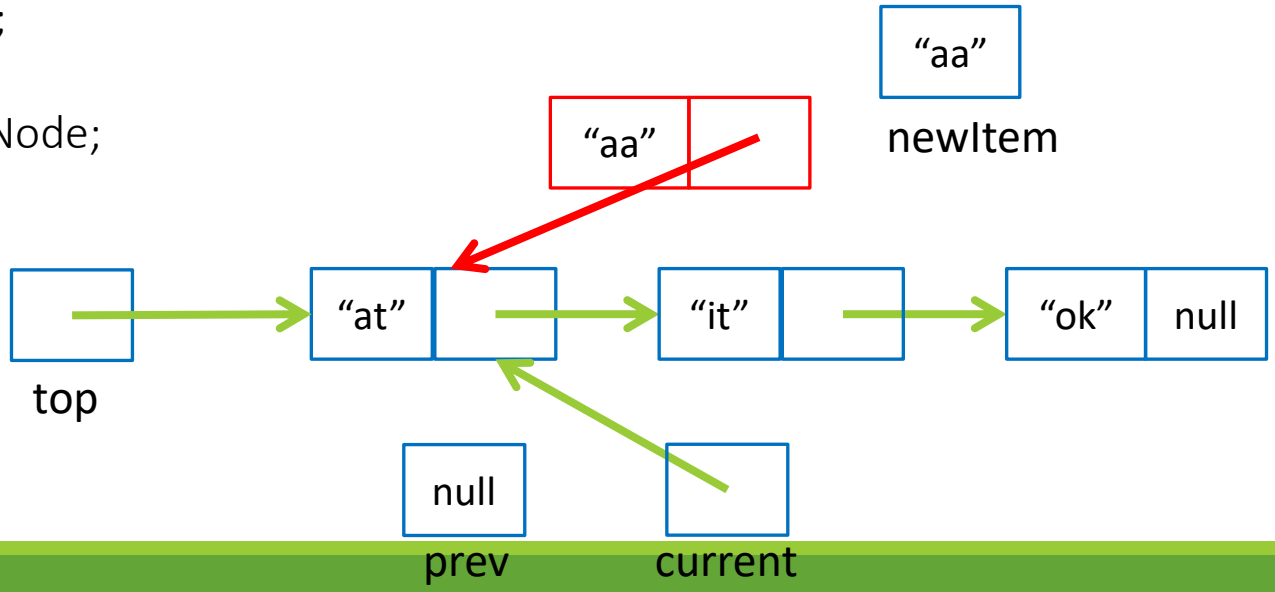
# Ordered insertion

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



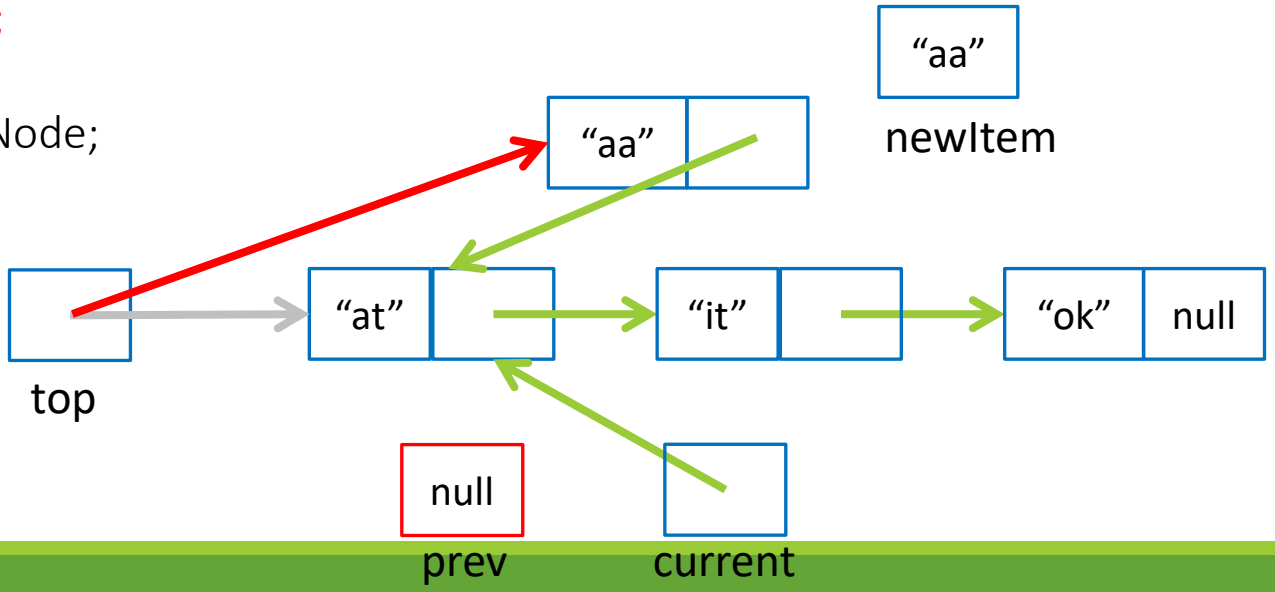
# Ordered insertion - first node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



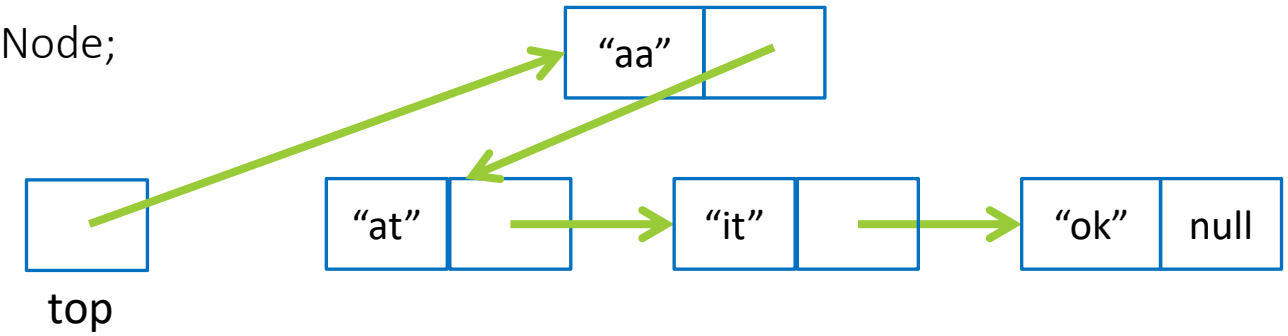
# Ordered insertion - first node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



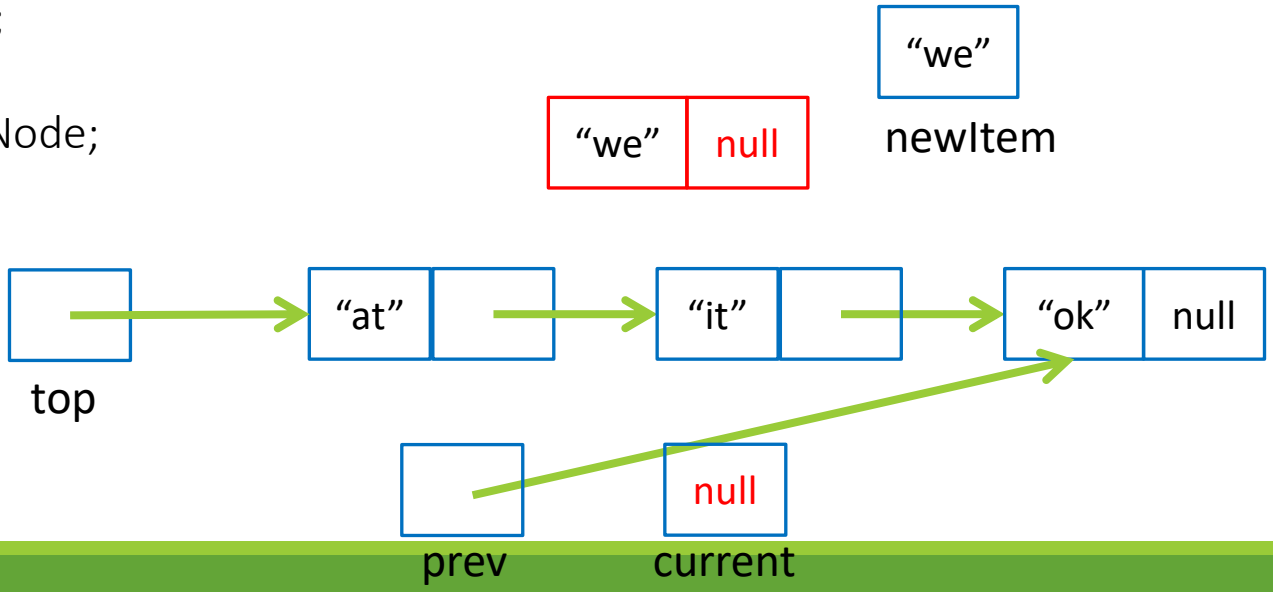
# Ordered insertion - first node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



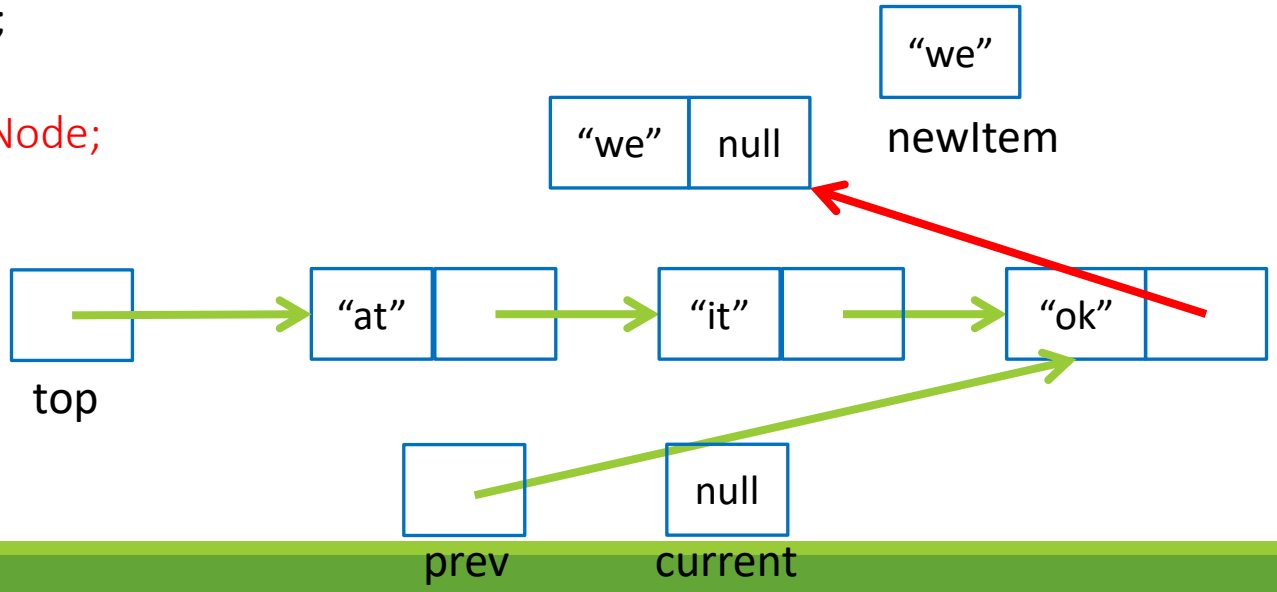
# Ordered insertion - last node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



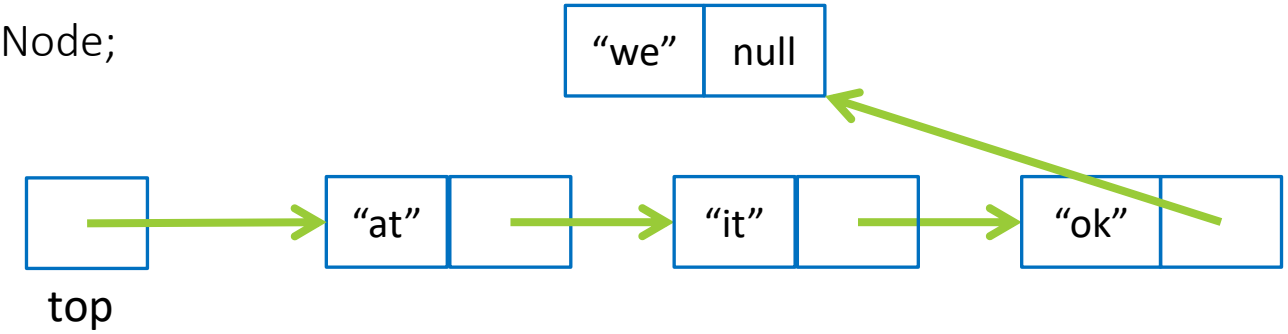
# Ordered insertion - last node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



# Ordered insertion - last node

```
public void ordInsert(String newItem) {
    //Make it work for Strings only this time.
    Node prev = null;
    Node current = top;
    while(current != null && ((String)(current.data)).compareTo(newItem) < 0){
        prev = current;
        current = current.link;
    }
    //This item belongs between prev and next
    Node newNode = new Node(newItem, current);
    if(prev == null)
        top = newNode;
    else
        prev.link = newNode;
}
```



# Take-home message

- We have learned how to build our very own data structure: the linked list!
  - it requires only 2 simple classes: LinkedList and Node
- The LinkedList class is where we put the methods that can access or modify the LinkedList (add, remove, toString, etc.)
- Whenever you implement a method:
  - Be careful of not breaking the list! (e.g. losing links)
  - Think about the special cases! Your method must handle all possible cases!