COSC 222 Data Structure

Collections, Lists and Linked List

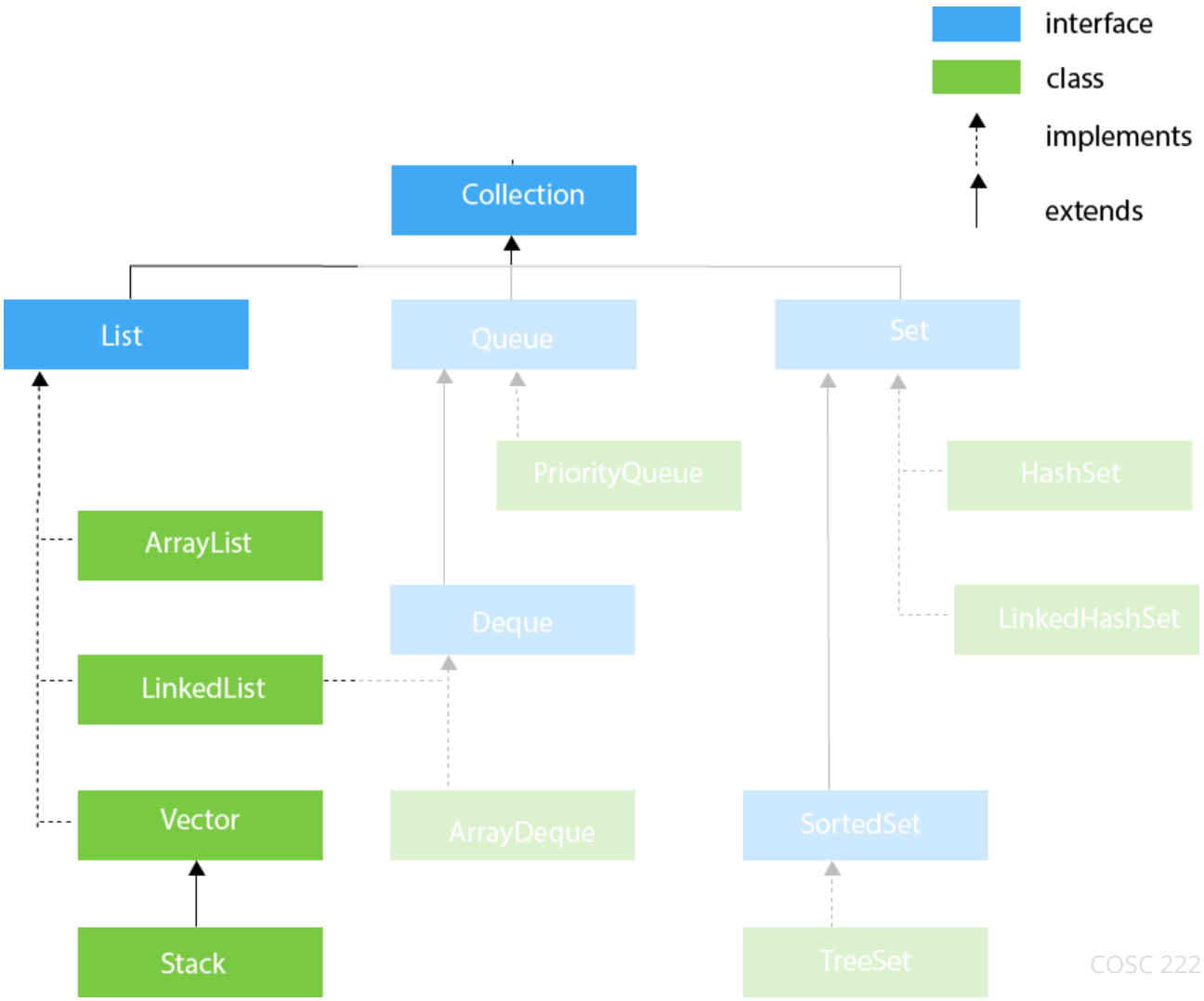
Collections

- **Collection**: an object that stores data inside it; a.k.a. a "data structure"
  - the objects stored are called **elements**
  - some maintain an ordering, some don't
  - some collections allow duplicates, some don't
  - an array is like a very crude "collection"

  - **Typical operations**: add element, remove element, clear all elements, contains or find element, get size
  - examples: java.util.ArrayList, java.util.HashMap, java.util.TreeSet

Hierarchy of Collection Framework

- The `java.util` package contains classes and interfaces for the Collection framework.



Java's `Collection` interface

- `java.util.Collection` has the following methods:

`public boolean add(Object o)`
Appends the specified element to this collection.

`public void clear()`
Removes all of the elements of this collection.

`public boolean contains(Object o)`
Returns true if this collection contains the specified element.

`public boolean containsAll(Collection coll)`
Returns true if this collection contains all of the elements in the specified collection.

Java's `Collection` interface

`public boolean isEmpty()`

Returns true if this list contains no elements.

`public boolean remove(Object o)`

Removes the first occurrence in this list of the specified element.

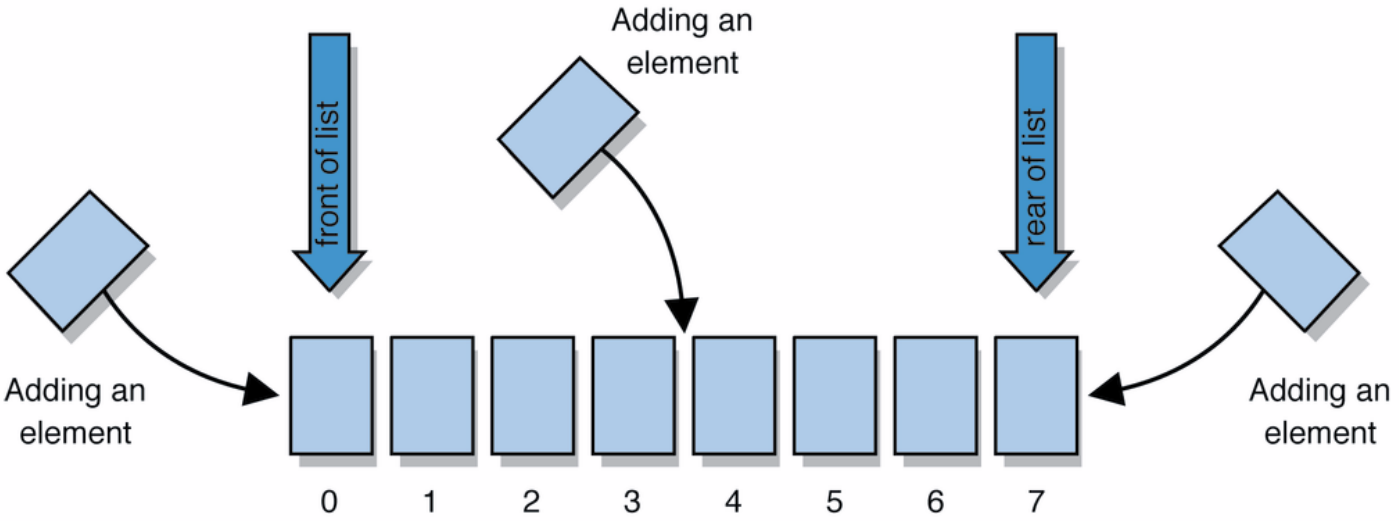
`public int size()`

Returns the number of elements in this list.

More methods:
https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Lists

- In general, an ordered sequence of elements, each accessible by a 0-based index
- Ordering: new elements are added to the end by default.
- Examples:
  - List of 5 integers: 34, -67, 21, 5, -13.
  - List of 3 Strings: "Fred", "likes", "COMP1020"



Java's `List` interface

- Java also has an interface `java.util.List` to represent a list of objects. It adds the following methods to those in `Collection`: (a partial list)

```
public void add(int index, Object o)
```

Inserts the specified element at the specified position in this list.

```
public Object get(int index)
```

Returns the element at the specified position in this list.

```
public int indexOf(Object o)
```

Returns the index in this list of the first occurrence of the specified element, or `-1` if the list does not contain it.

Java's `List` interface

`public int lastIndexOf(Object o)`

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain it.

`public Object remove(int index)`

Removes the object at the specified position in this list.

`public Object set(int index, Object o)`

Replaces the element at the specified position in this list with the specified element.

More methods:
https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Partially-full arrays

```
double[] temp = new double [20];
```

- We've used "partially full arrays" in many cases
  - An array with some maximum size
  - An integer to keep track of the current size

    The number of elements of the array currently in use.

  - They have some drawbacks:

    The maximum size is a limitation

    It can be increased in some cases by allocating an entirely new array, and copying all of the existing elements.

    Elements must be shifted left or right often

    When inserting, deleting, moving, sorting, etc.

ArrayList class

- Java has the built-in ArrayList class that provides partially-full arrays.
  - It works exactly as we would write it ourselves.
  - `import java.util.ArrayList` is required.

  - To create one for a specific type of data:

    ```
    ArrayList<String> myList = new ArrayList<String>( );
    ```

    This is some new syntax called a "generic".

Objects in ArrayLists

- One small disadvantage to ArrayLists:
  - They can only hold objects, not primitive types.

    ArrayList<int> is an error

    Can't use int, double, boolean, char, float, long, byte, short

    ArrayList<String> is OK

    Integer, Double, Boolean, Character, Long are classes that give Object versions of the primitive types, allowing them to be used, too.

- An ArrayList<Object>, or just ArrayList, is flexible and powerful.
  - It can store a list of any kind of data

Common ArrayList methods

- Let's set up:

```
ArrayList<String> a = new ArrayList<String>( );
```

- Adding/Removing objects to an ArrayList:

```
a.add("testing"); //adds to the end
a.add("fred");
a.add("hippo");
a.add(1, "second"); //add “second” to index 1
        // "fred" and “hippo” move up one place
a.add(10,"far"); //error – can't leave "gaps“
a.remove(0); //Removes the first element.
a.remove("hippo"); //removes that String
a.remove(10); //Error – no such element
a.clear(); //a is now empty. This one is void.
```

Output:

- Determining the size of the list

```
a.size()
```

Arrays vs. ArrayLists

Array	ArrayList
`String[ ] a = new String[10];`	`ArrayList<String> a = new ArrayList<String>( );`
`a.length  //cannot change`	`a.size( )  //changes`
`a[0]`	`a.get(0)`
`a[0] = "test"`	`a.set(0, "test")`
…n/a…	`a.add("new")`
…n/a…	`a.remove(0)`
Contains any type	Contains objects only

ArrayList of User Defined Objects

- You can an create an ArrayList of any type

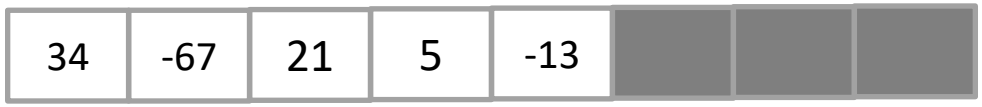
```
class User {
    private String name;
    private int age;

public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
…
}

ArrayList<User> users = new ArrayList<>();
users.add(new User("Adam", 19));
users.add(new User("David", 20));
```

Physical adjacency

- These styles have two things in common:
  - The elements are stored physically adjacent in an array.



Not

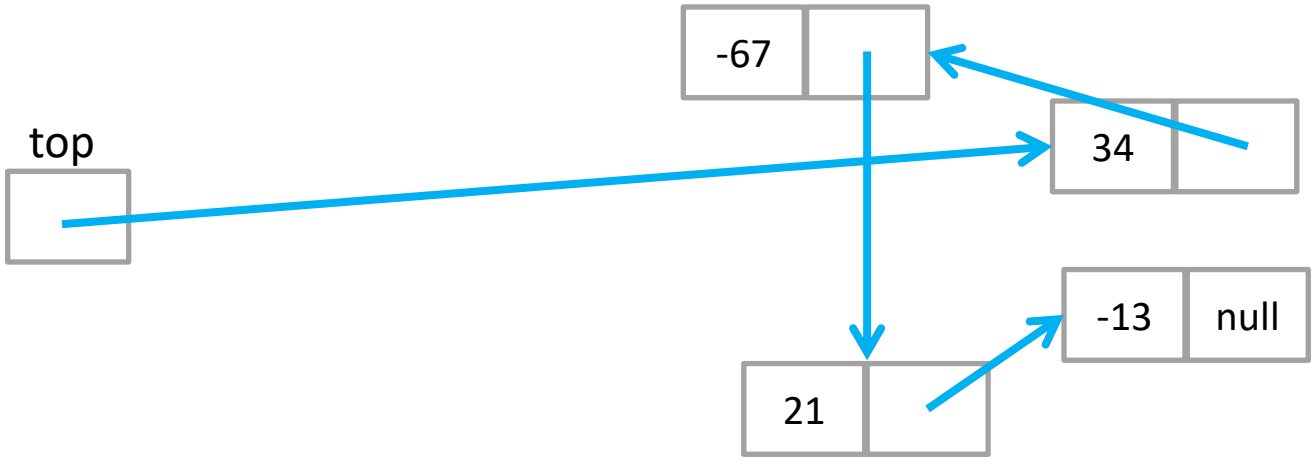
To add or delete an element from the middle or the front requires other elements to be shifted left or right.

  - The array might become full (or be full all the time).

To add another element requires a complete re-build of the array into a new one.

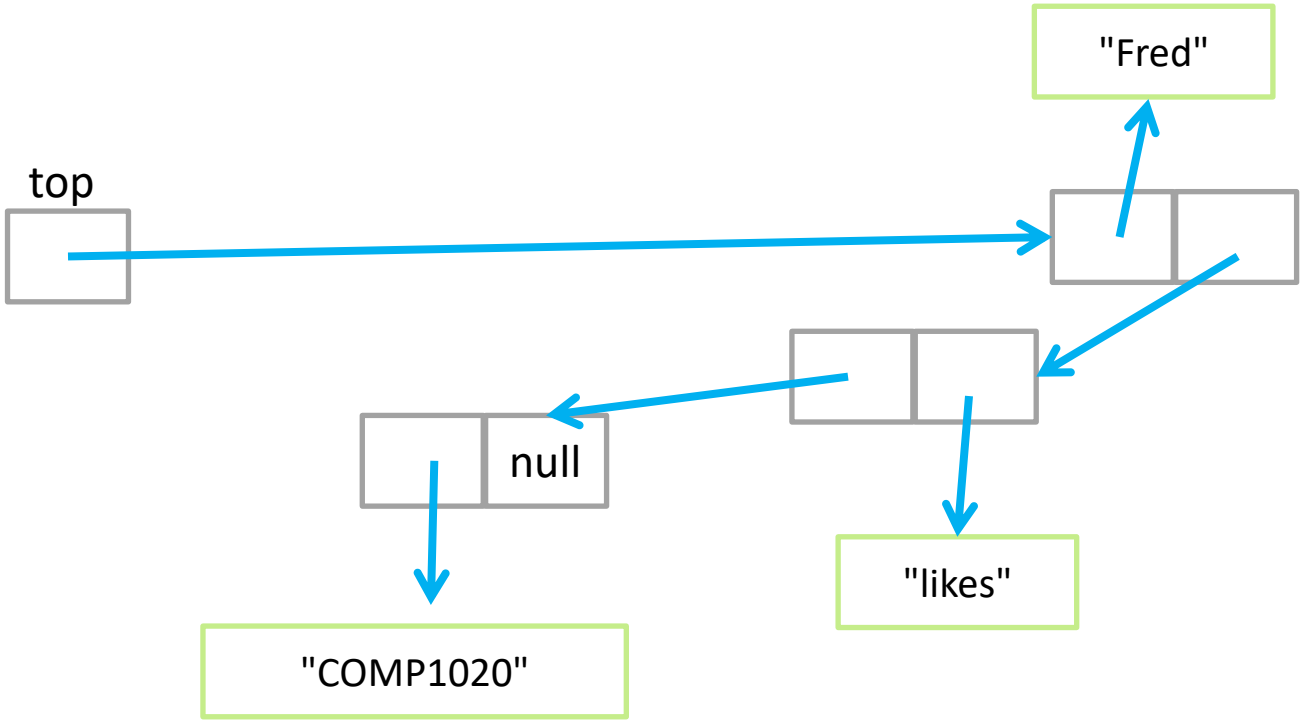
A linked list

- A linked list solves both of these problems
- The data is stored in a set of **Node** objects.
- Each 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.



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 Node {

}

public class LinkedList {

}
```

Code for a generic linked list

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

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

Get/Set methods

- We'll need get/set methods for Nodes:

```
public Object getData() {return data;}
public Node getLink() {return link;}

public void setData(Object o) {data = o;}
public void setLink(Node n) {link = n;}
```

Code for a generic linked list

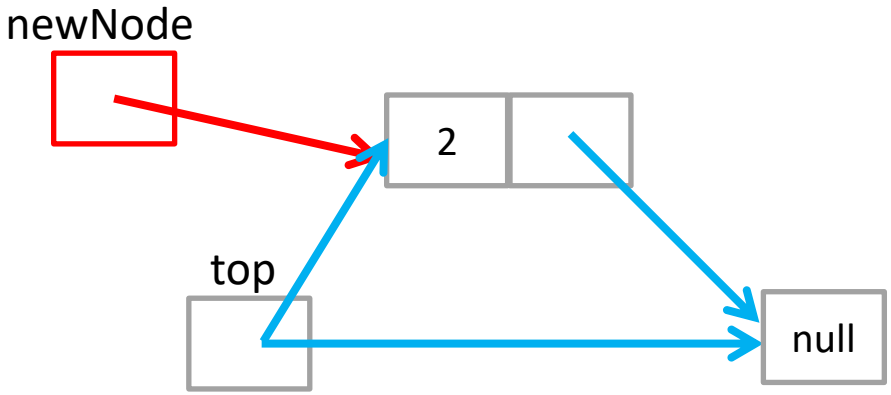
```
public class LinkedList {
  private Node top; //The reference to the first Node
}
```

- We don't really need to write this:

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

An add method

- 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 only easy to add them **at the end**.

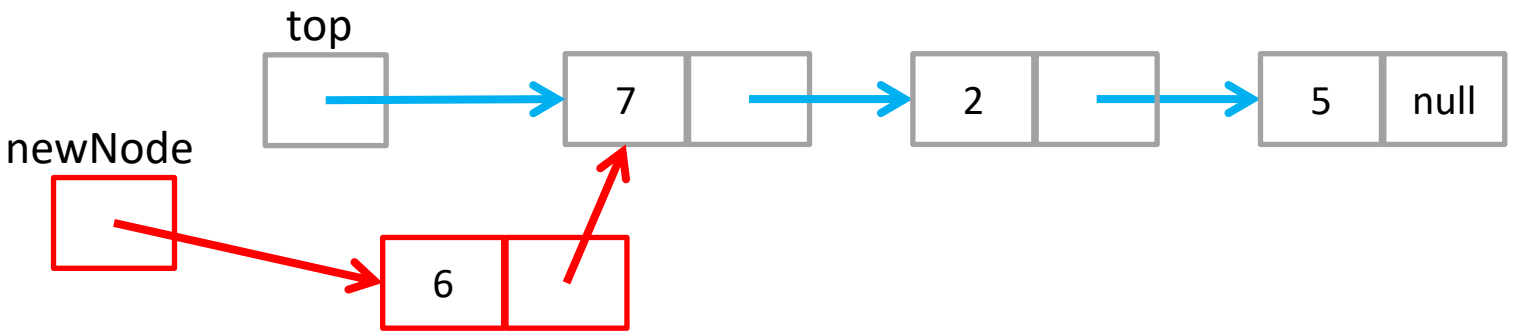


An add method

- 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;
}
```

- Diagram (simpler version):

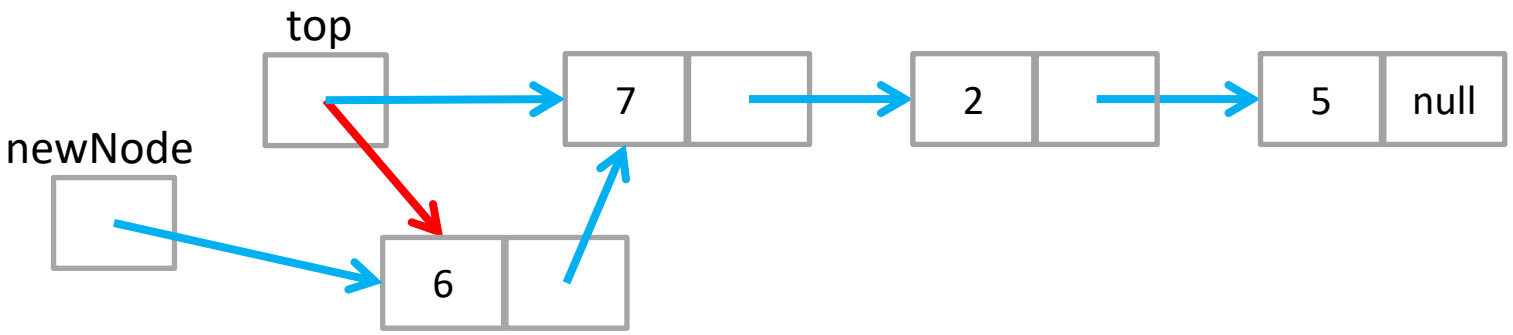


An add method

- 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;
}
```

- Diagram (simpler version):

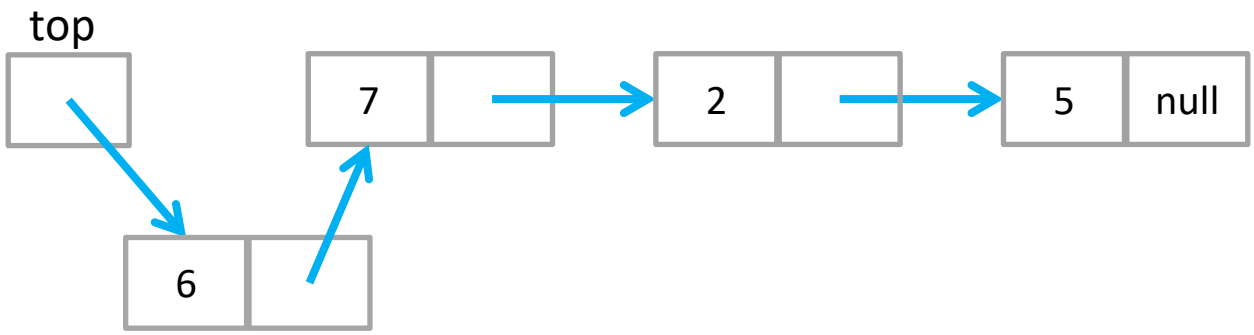


An add method

- 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;
}//add
```

- Diagram (simpler version):



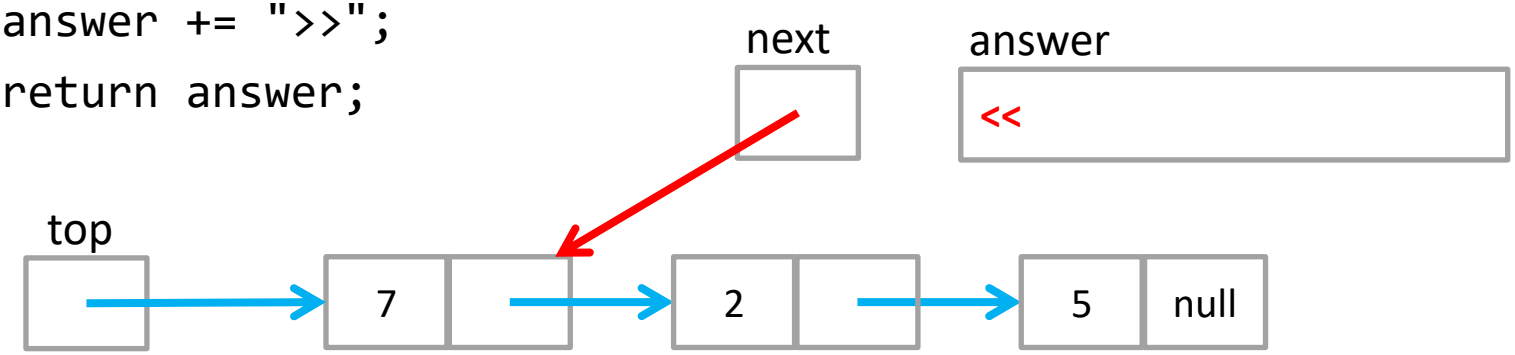
A toString method

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

```
public String toString() {
  String answer = "<<";
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```

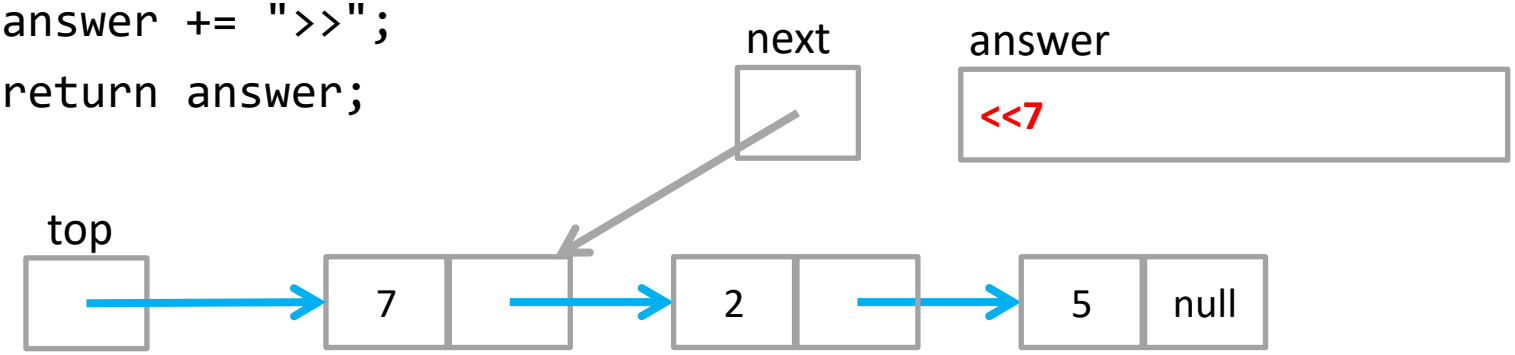
Follow the execution:

```
public String toString() {
  String answer = "<<";
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



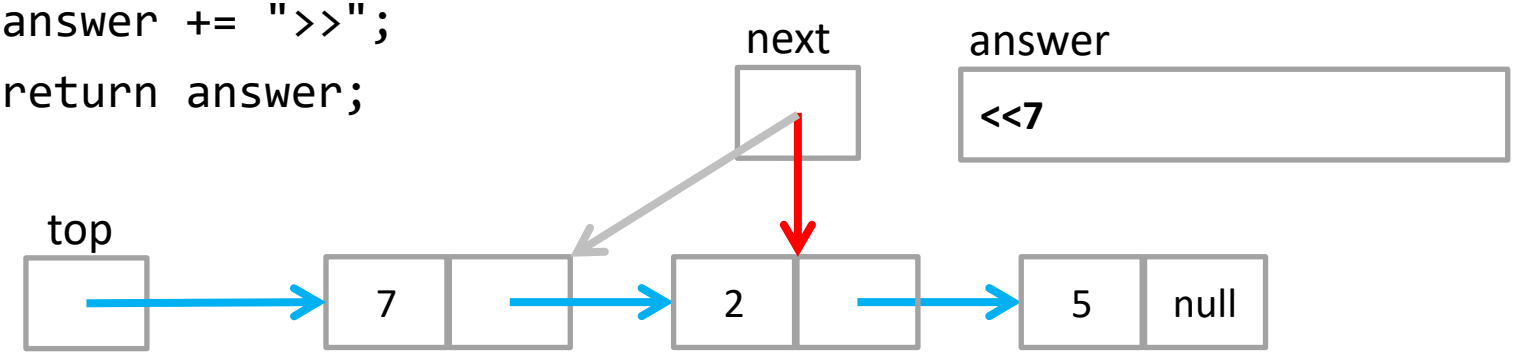
Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



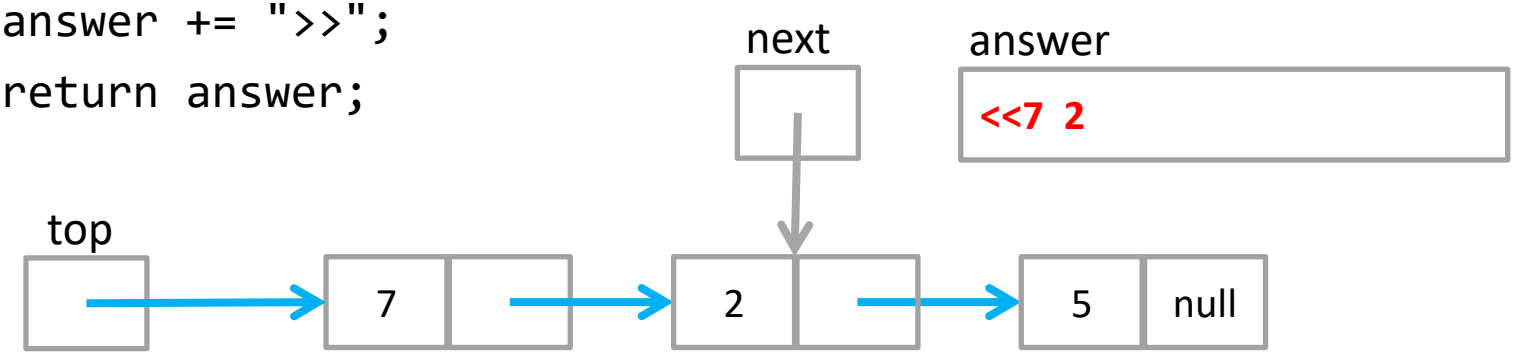
Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



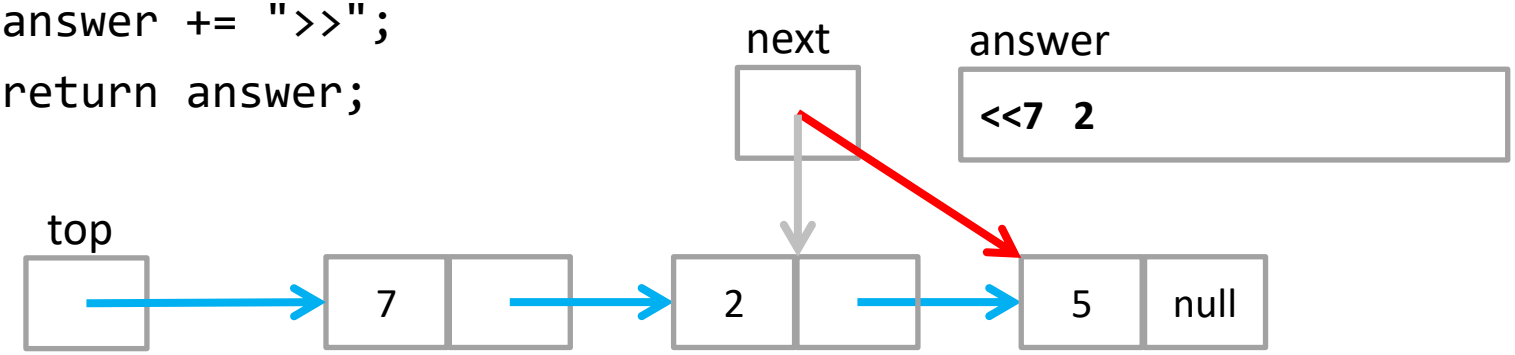
Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



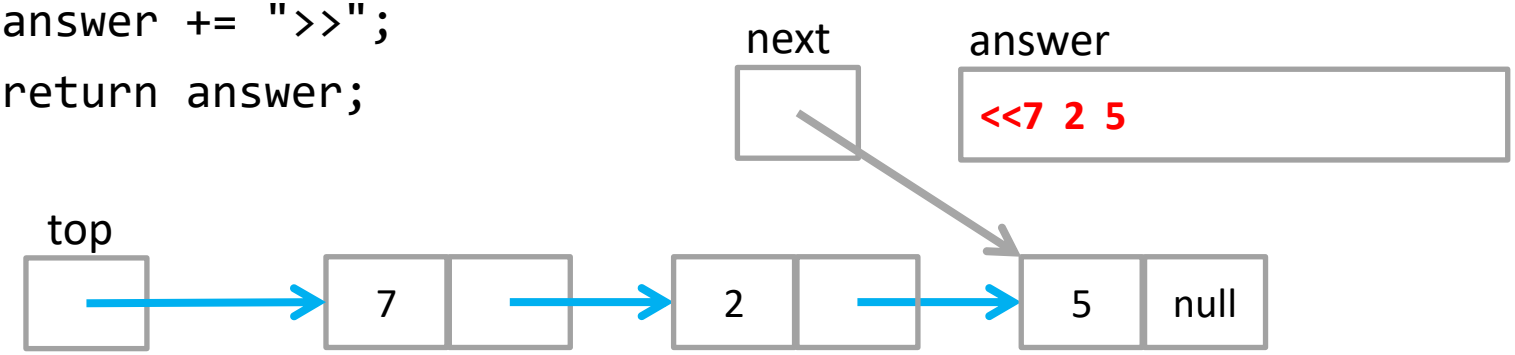
Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



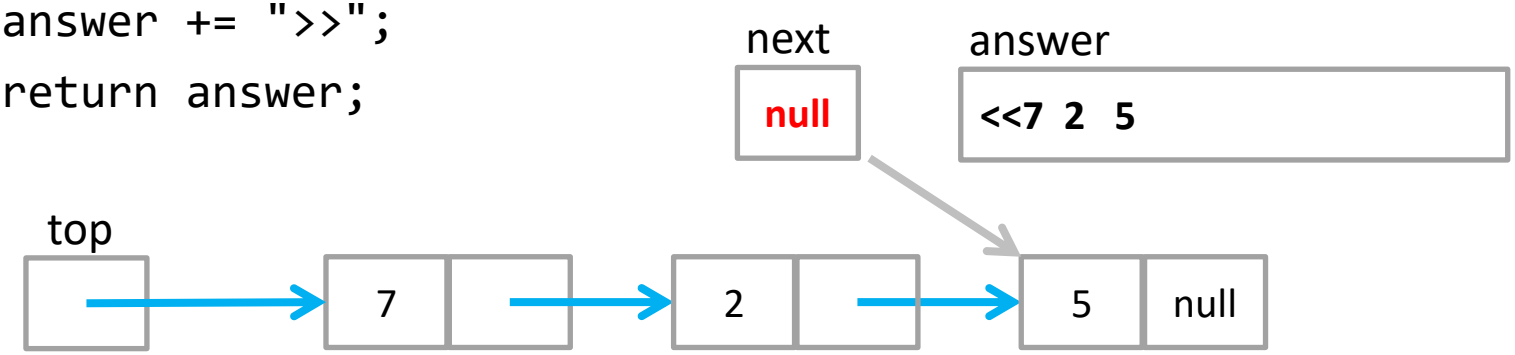
Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```



Follow the execution:

```
public String toString() {
  String answer = "<<"; //Why not?
  Node next = top; //Start at the first one
  while(next != null) {
    answer += next.getData().toString() + "  ";
    next = next.getLink(); //Advance to the next
  }
  answer += ">>";
  return answer;
}
```

Another Example

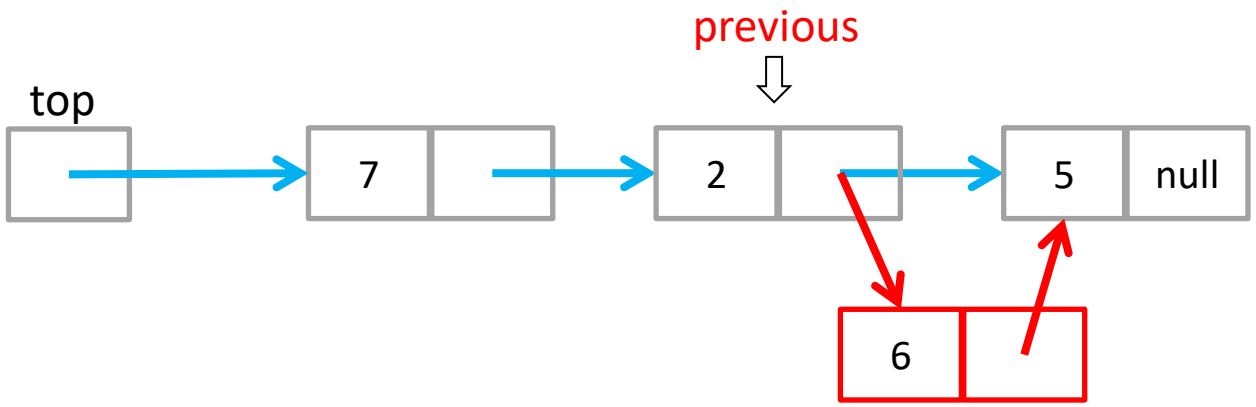
```
LinkedList ll = new LinkedList();
ll.add("Fred");
ll.add(345); //Mix them up.
ll.add("last");
System.out.println(ll.toString());
```

```
<<last  345  Fred  >>
```

Note last is first

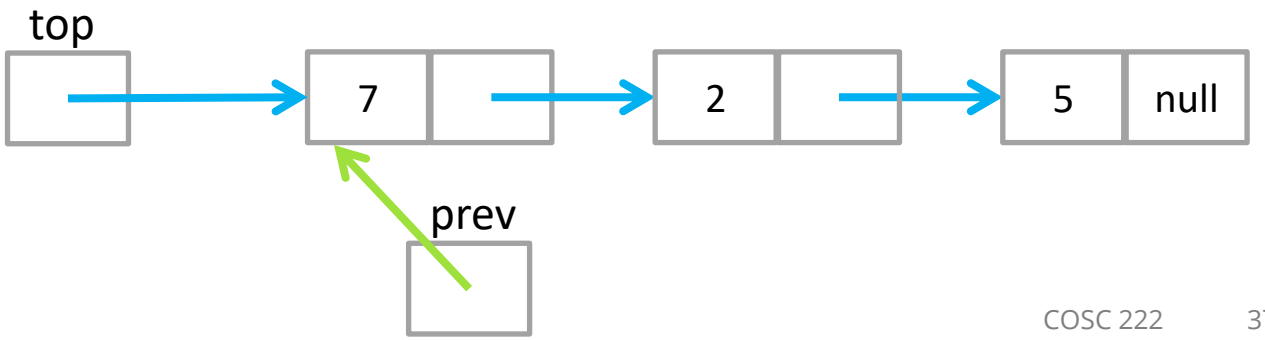
Add to a particular position

- 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?
- 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.



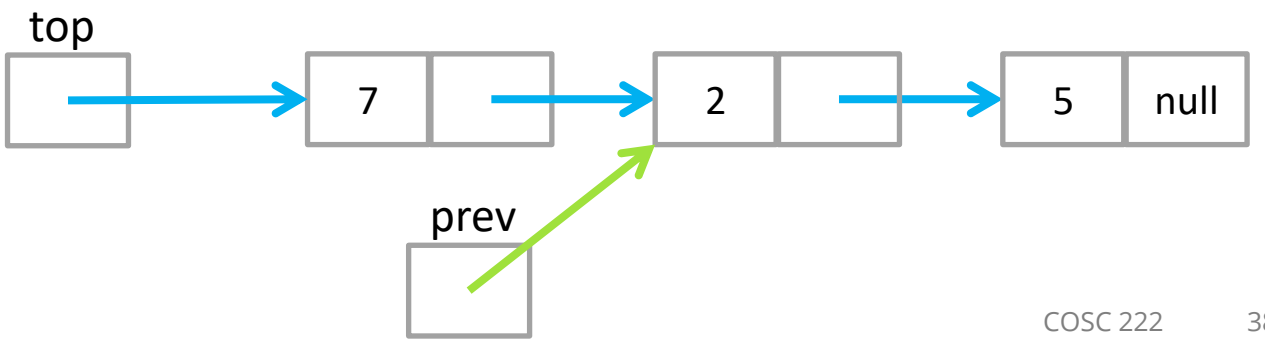
Add to a particular position

```
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.getLink();
    Node newNode = new Node(newItem,prev.getLink());
    prev.setLink(newNode);
  }
}
```



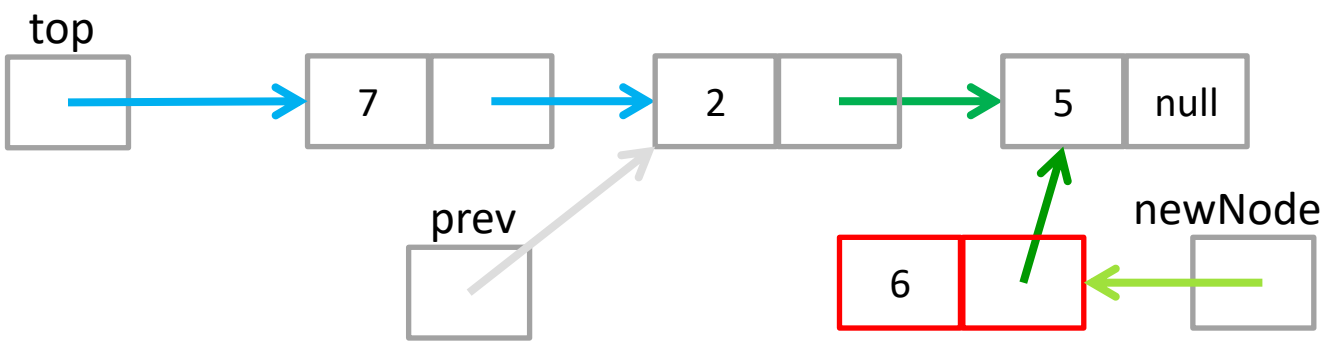
Add to a particular position

```
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.getLink();
    Node newNode = new Node(newItem,prev.getLink());
    prev.setLink(newNode);
  }
}
```



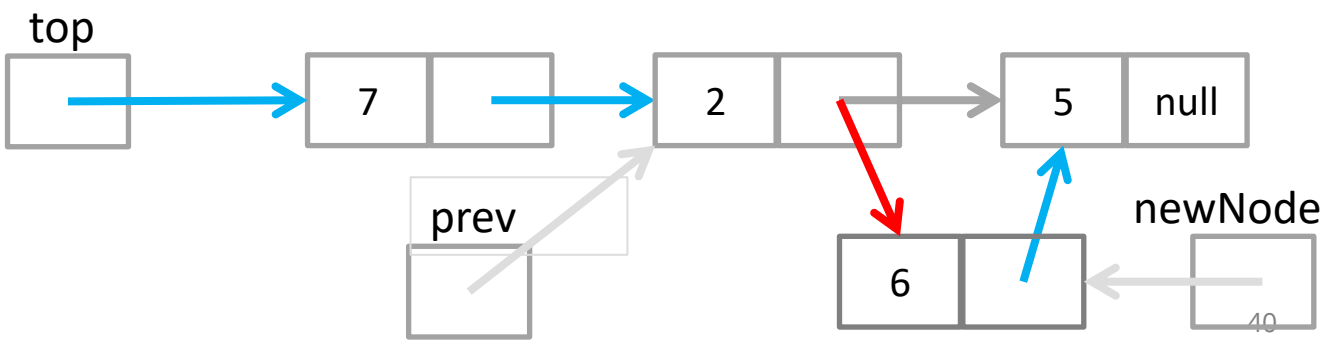
Add to a particular position

```
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.getLink();
    Node newNode = new Node(newItem,prev.getLink());
    prev.setLink(newNode);
  }
}
```



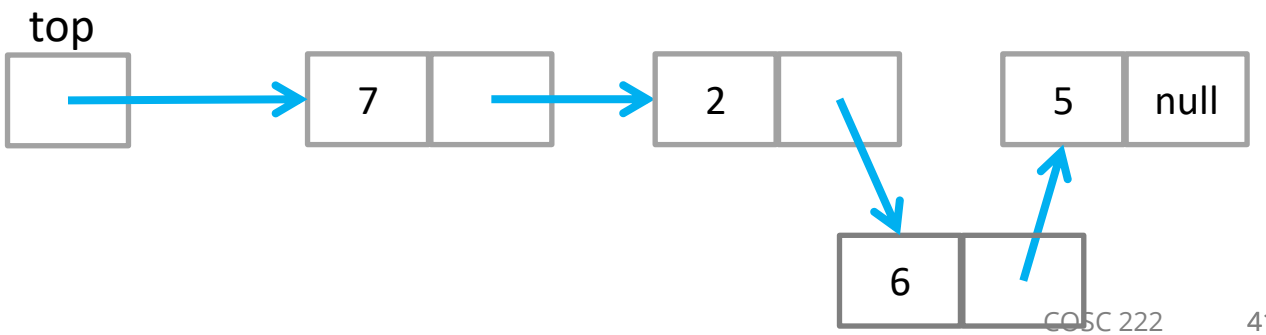
Add to a particular position

```
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.getLink();
    Node newNode = new Node(newItem,prev.getLink());
    prev.setLink(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.getLink();
    Node newNode = new Node(newItem,prev.getLink());
    prev.setLink(newNode);
  }
}//add
```



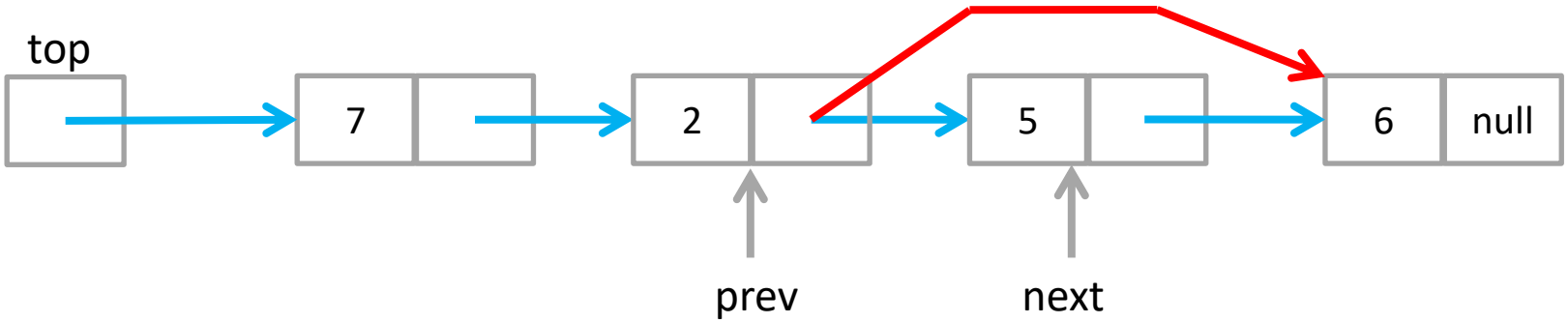
COSC 222

add: array vs. linked list

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

Deletion

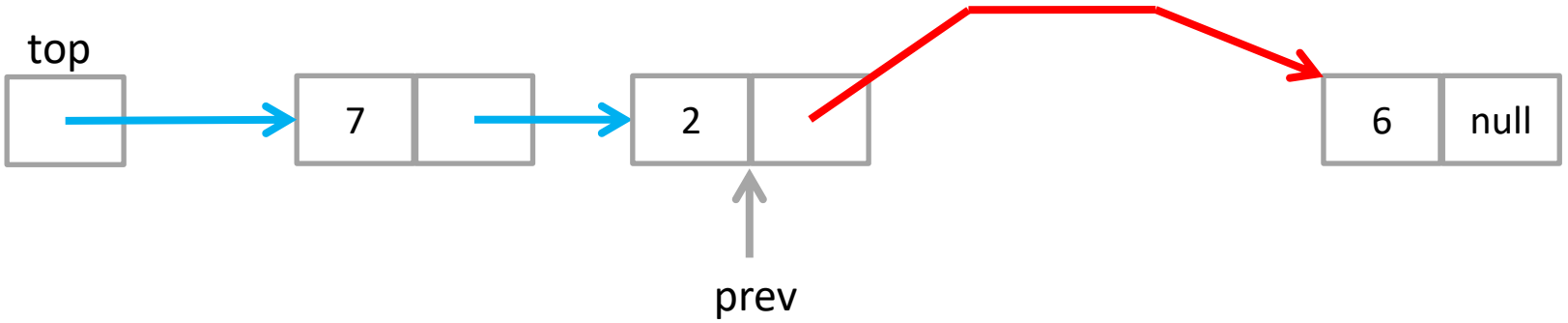
- Write the equivalent of ArrayList's remove(x).
- What we need to do to remove the 5:



- We need to move some Node variable (next) 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 needs to change.

Deletion

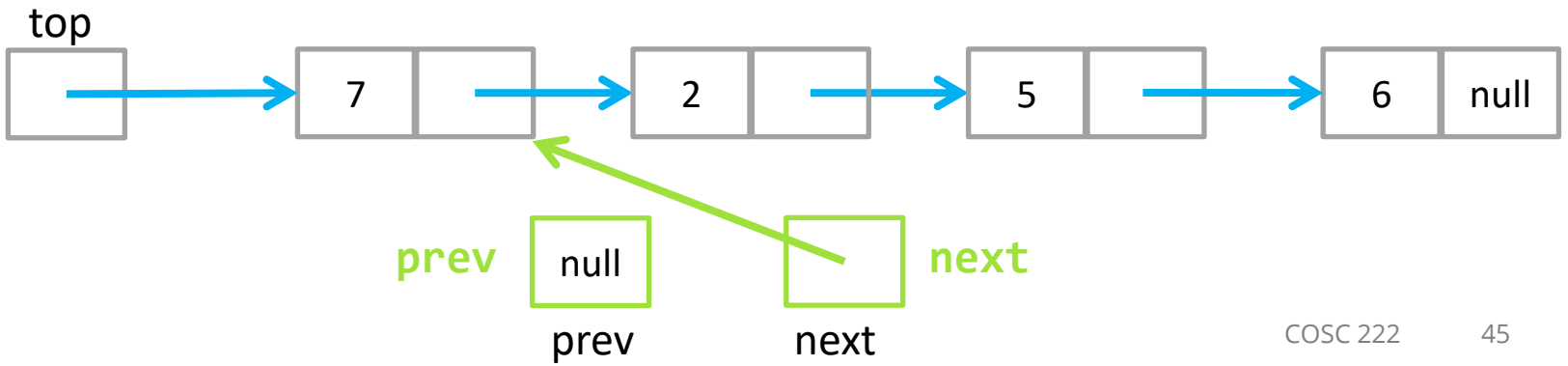
- Write the equivalent of ArrayList's remove(x).
- What we need to do to remove the 5:



- We need to move some Node variable (next) 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 needs to change.

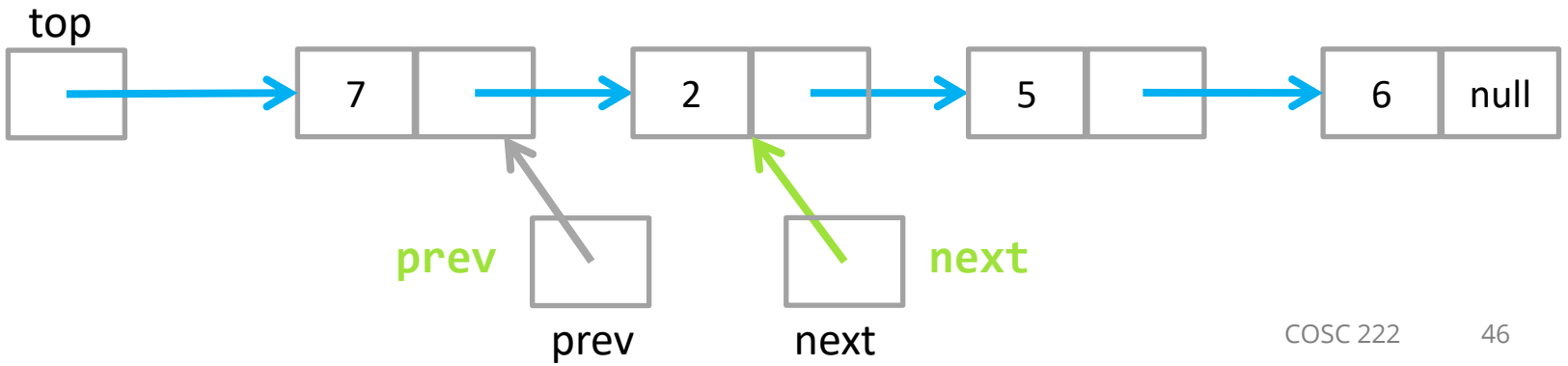
Deletion

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



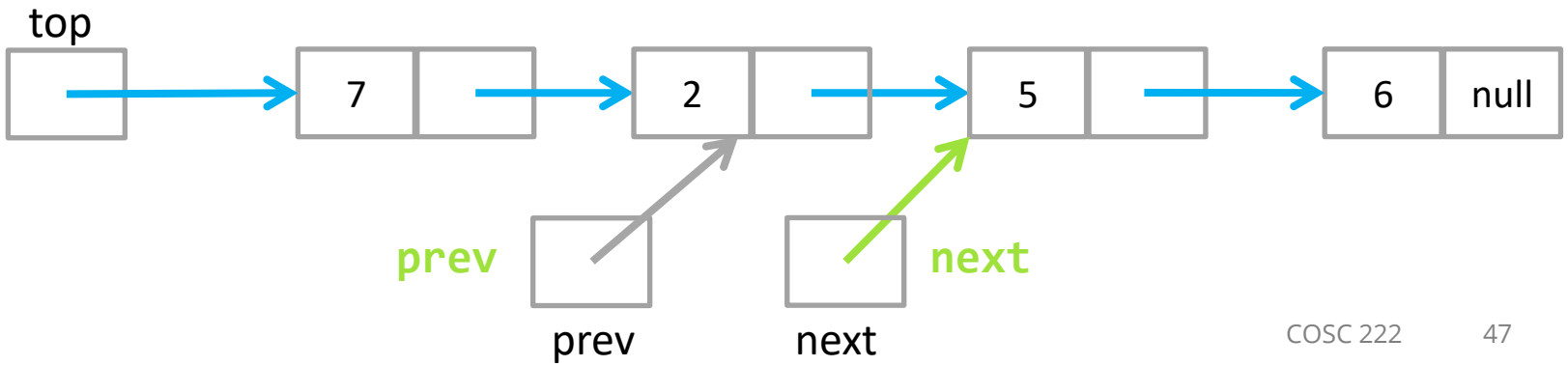
Deletion

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



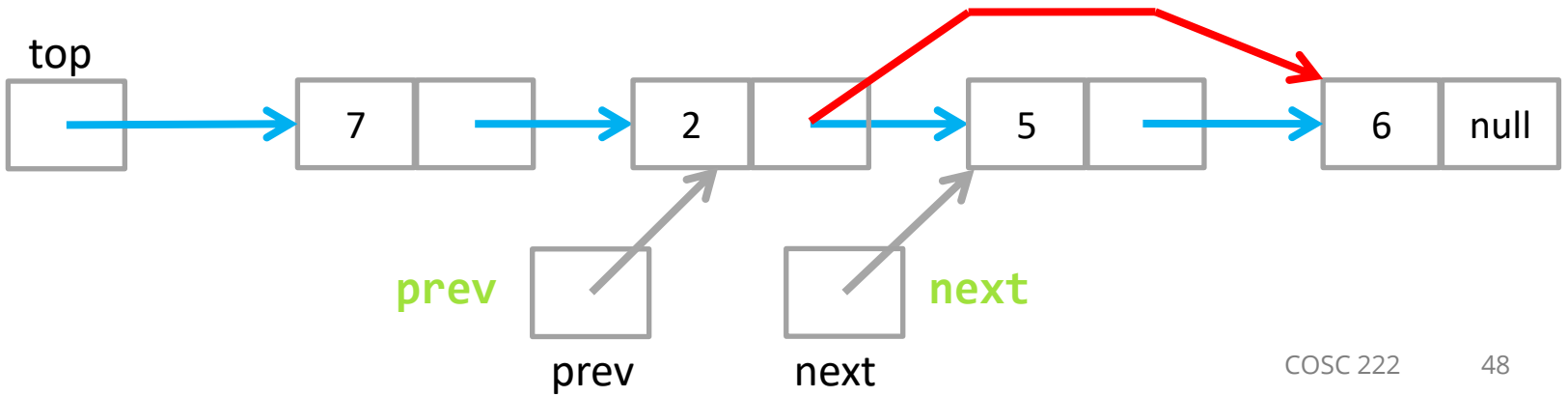
Deletion

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



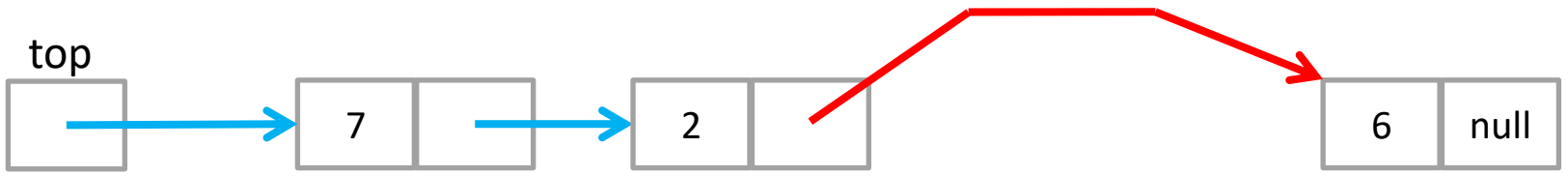
Deletion

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



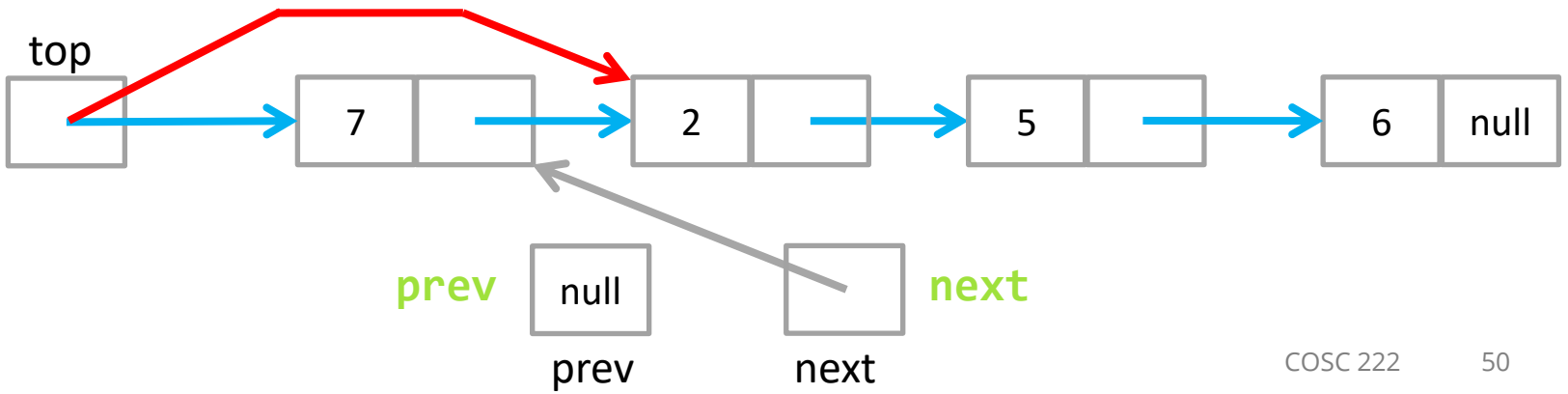
Deletion

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



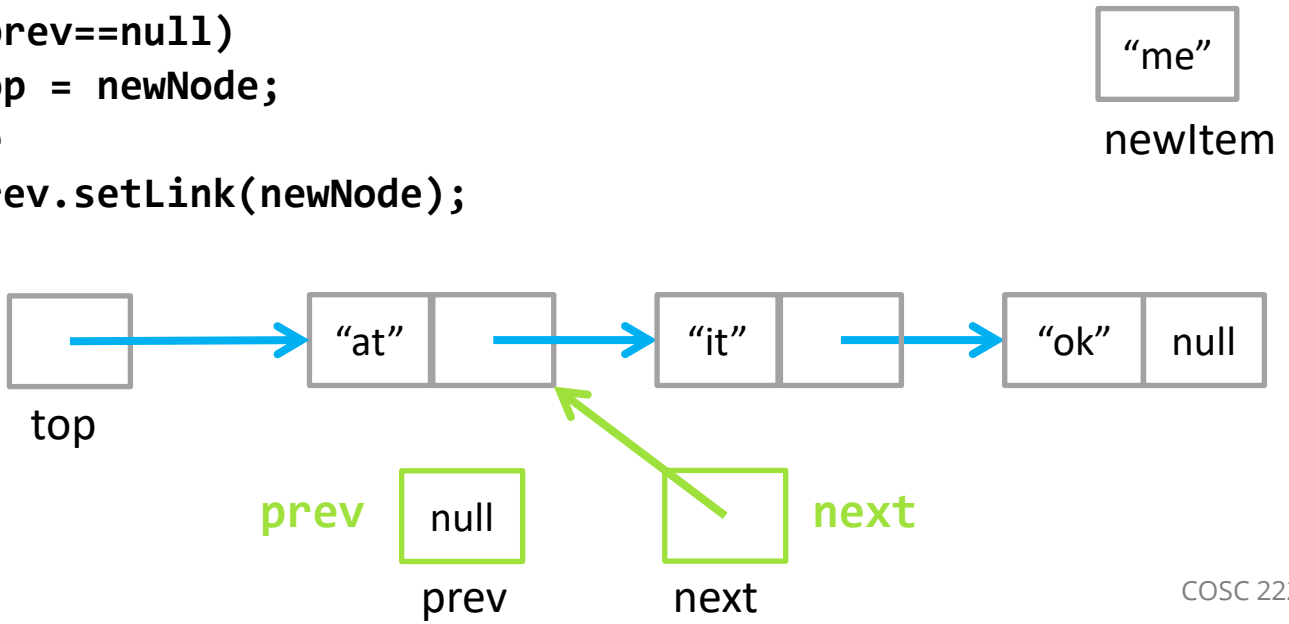
Deletion of first node

```
public void remove(Object key) {
  Node prev = null;
  Node next = top;
  while(!next.getData().equals(key)) {
    prev = next;
    next = next.getLink();
  }
  if(prev==null)
    top = next.getLink();
  else
    prev.setLink(next.getLink());
}
```



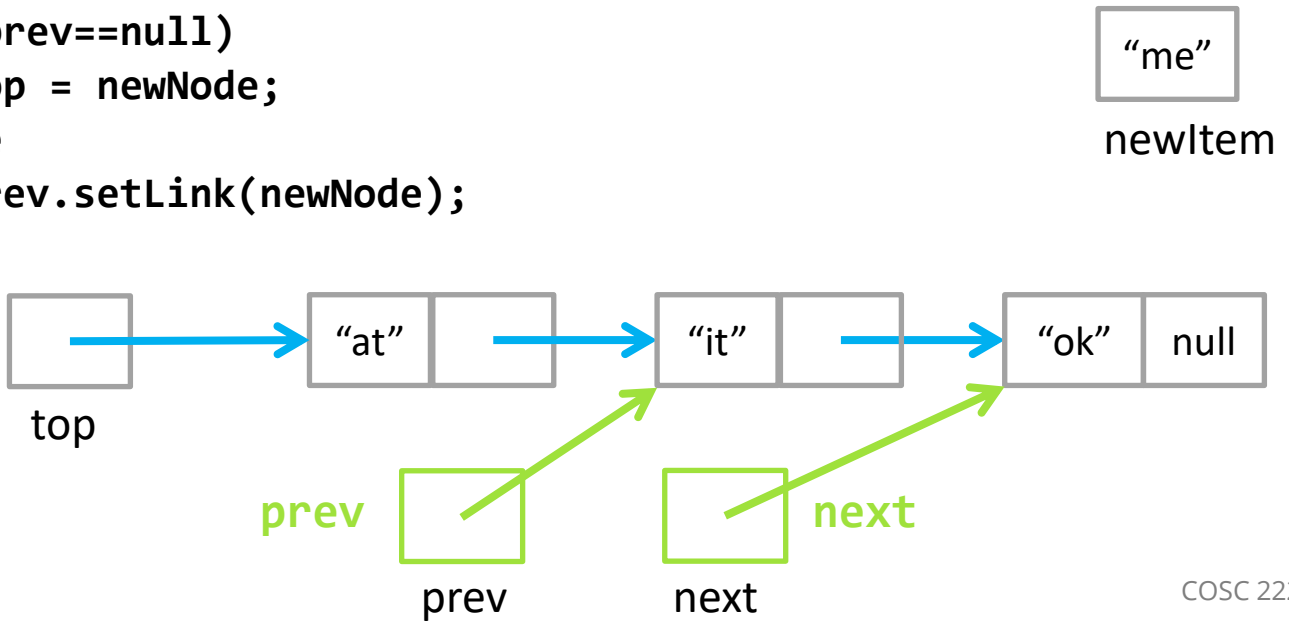
Ordered insertion

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



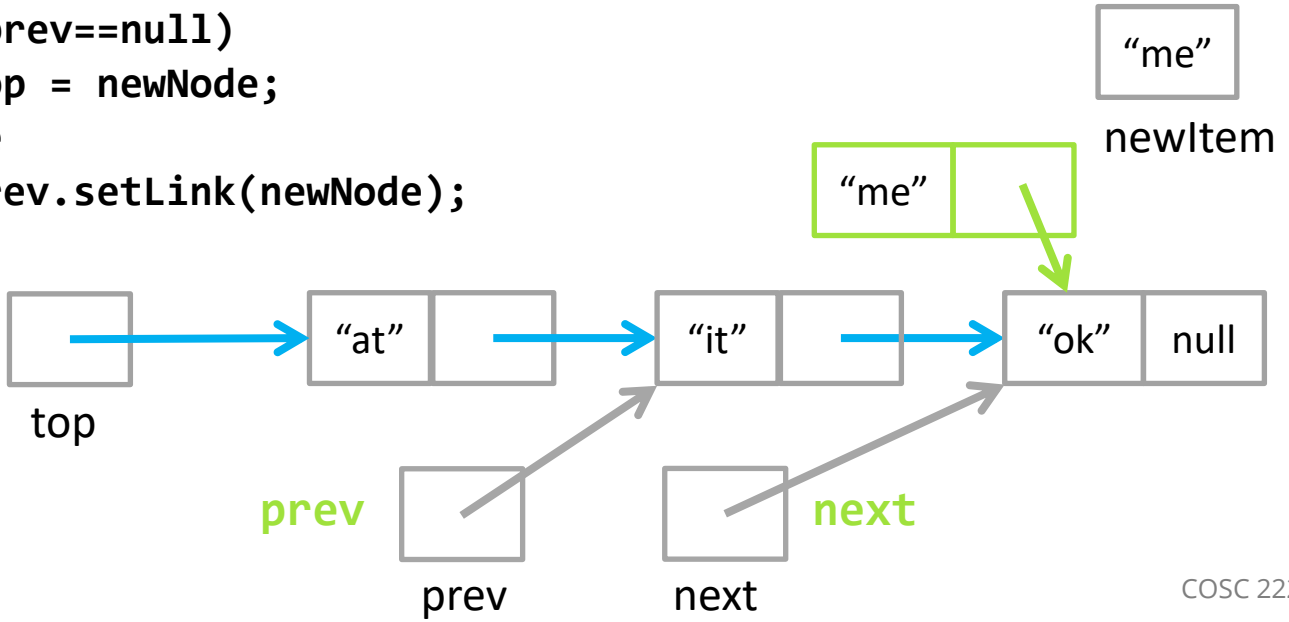
Ordered insertion

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



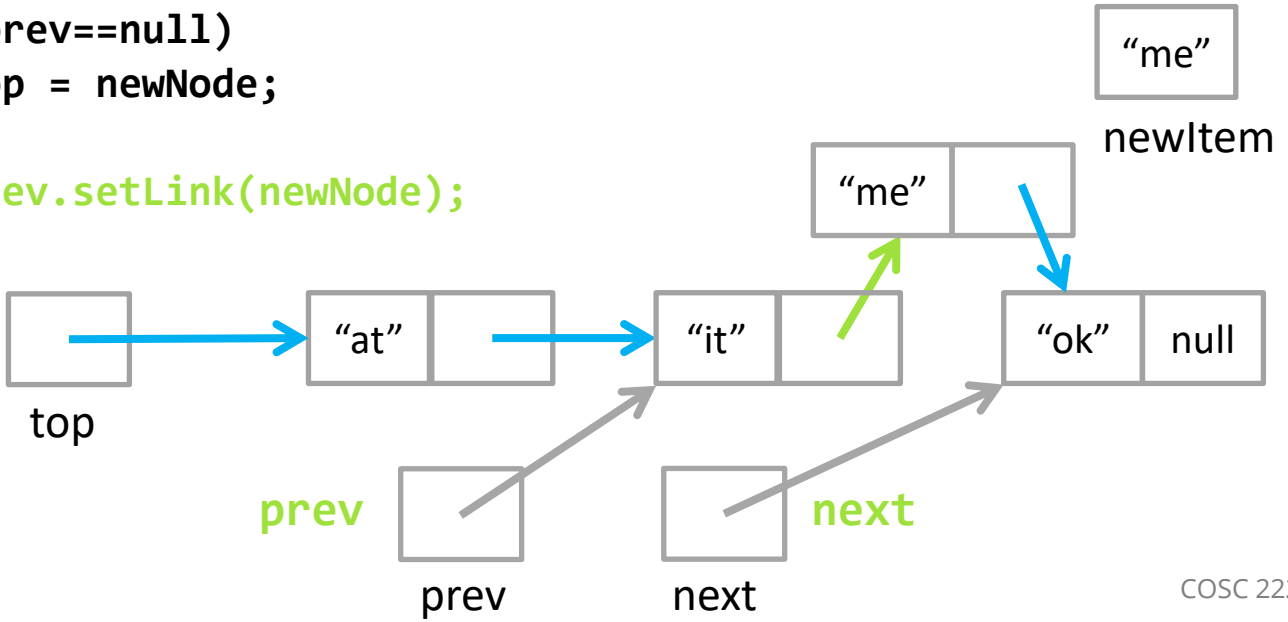
Ordered insertion

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



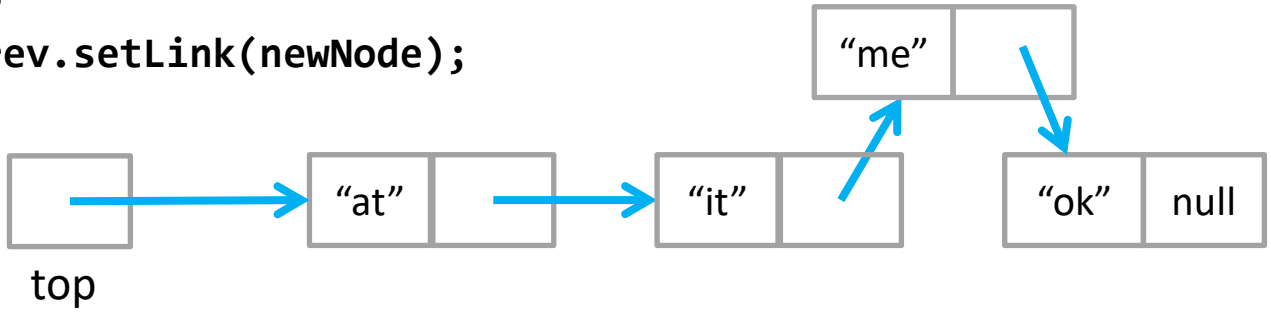
Ordered insertion

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



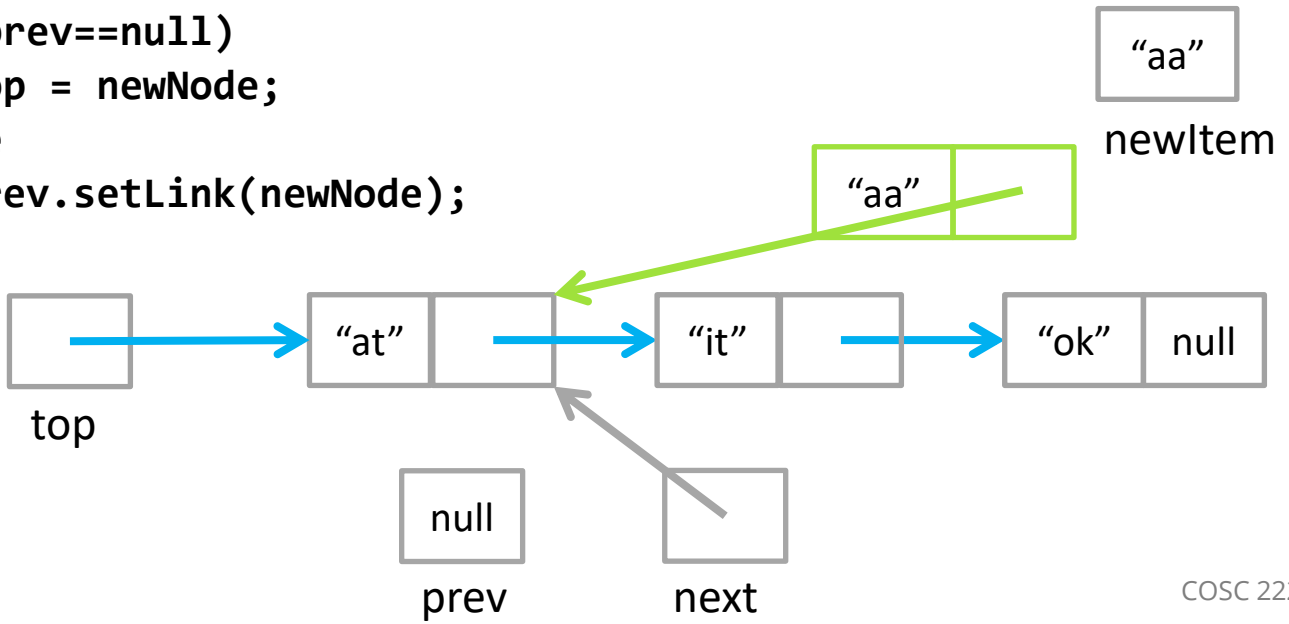
Ordered insertion

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



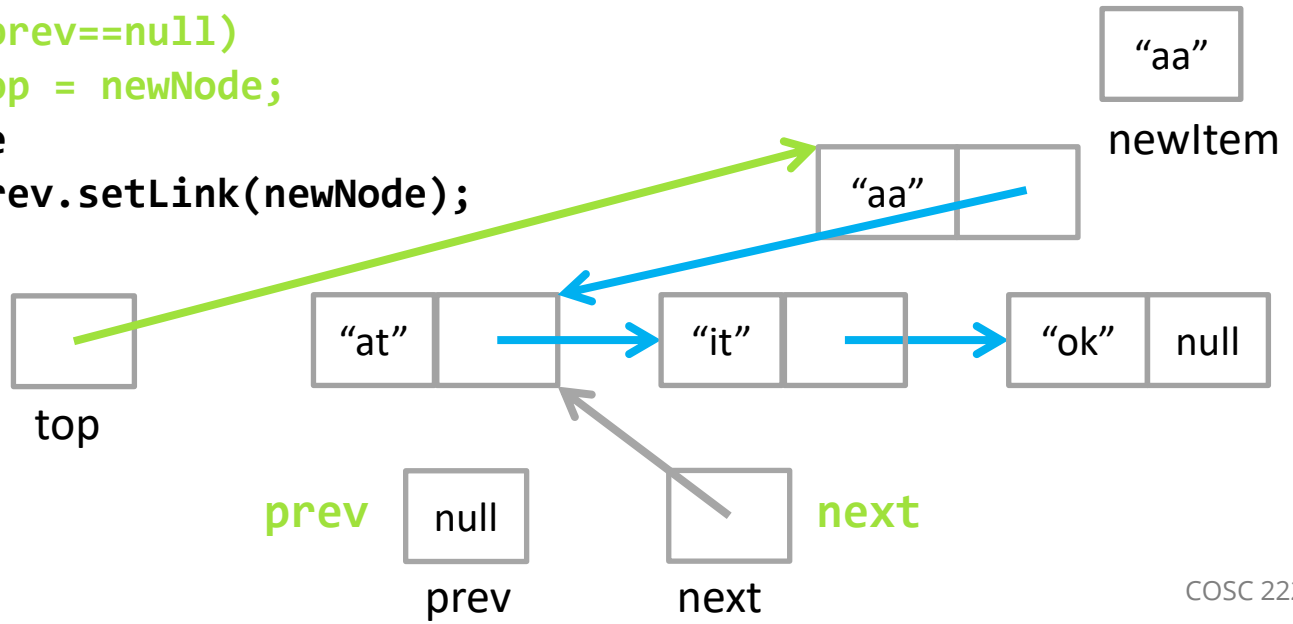
Ordered insertion – first node

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



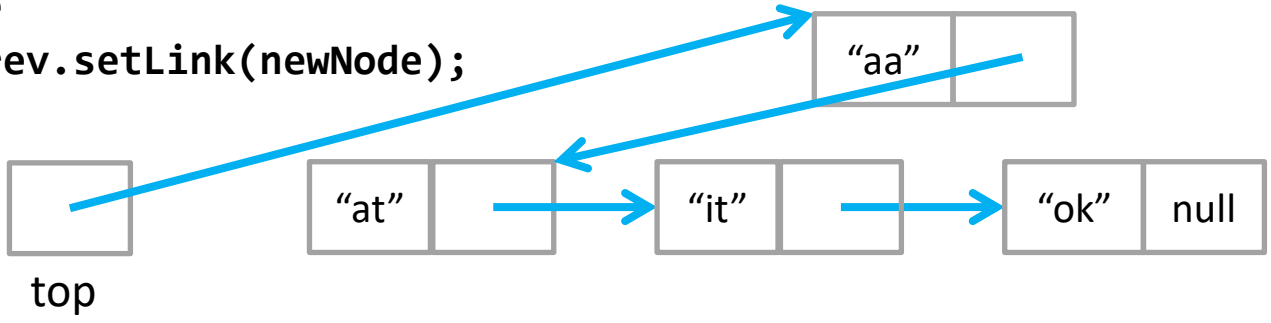
Ordered insertion – first node

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



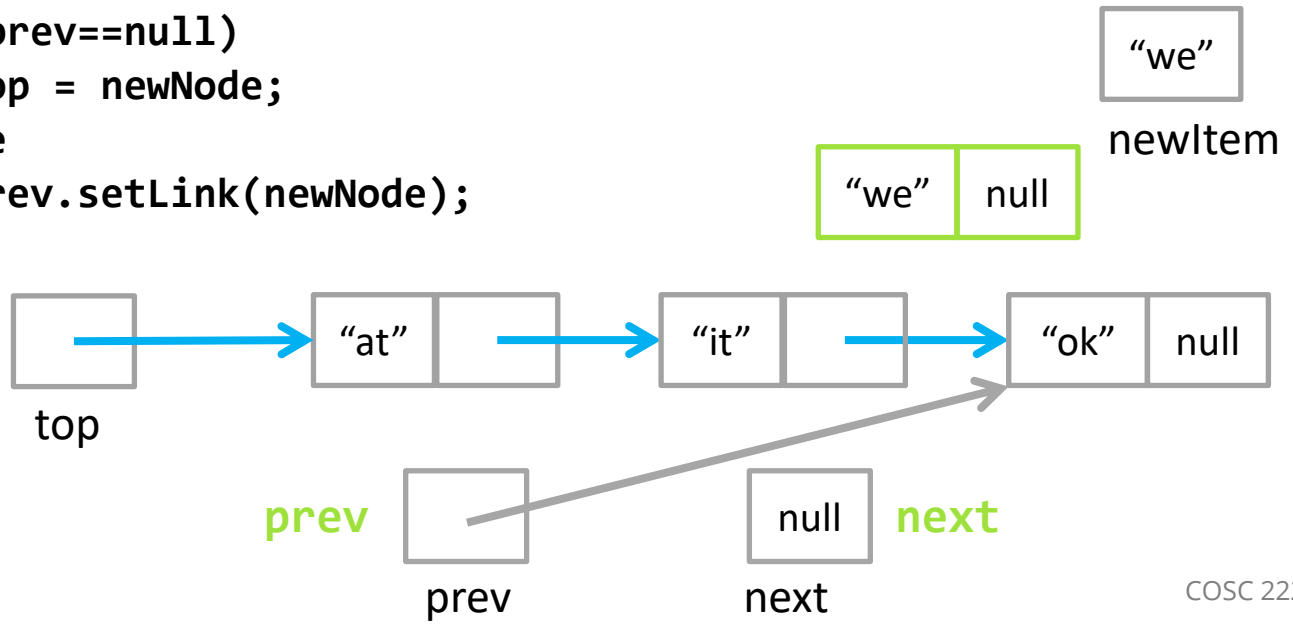
Ordered insertion – first node

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



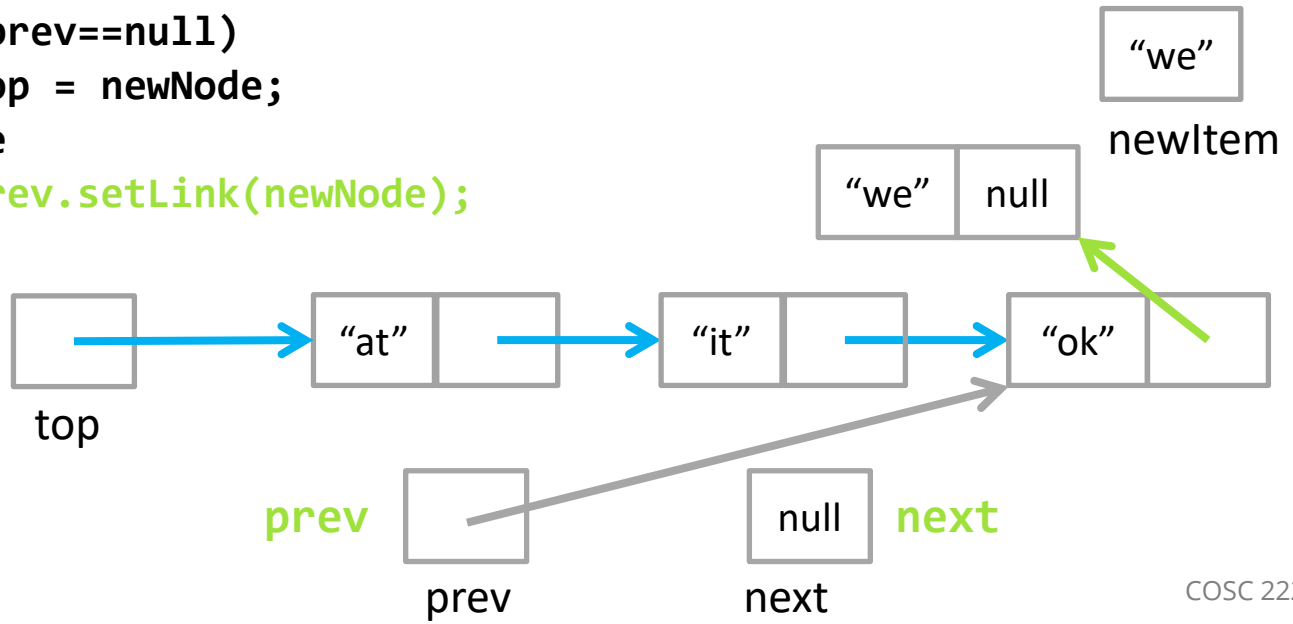
Ordered insertion – last node

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



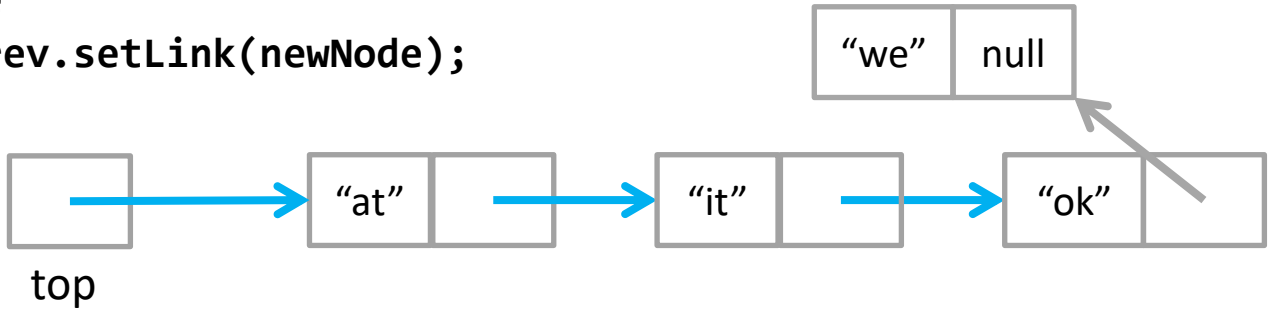
Ordered insertion – last node

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



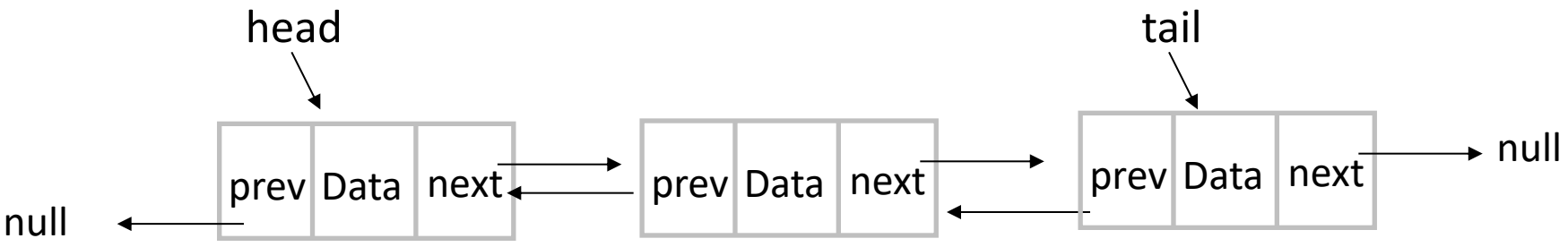
Ordered insertion – last node

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



Doubly Linked Lists

- Each node has a reference or pointer back to the previous nodes



- Most actual linked list implementations are doubly-linked and use a dummy **head** and dummy **tail**

Code for a doubly linked list

- Let's create a doubly linked list of Objects

- It's best to define two classes:

```
public class Node {

}

public class DoublyLinkedList {

}
```

Code for a doubly linked list

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

public Node(Object initData) {
    this.data = initData;
    previous = next = null;
}
```

Get/Set methods

- We'll need get/set methods for Nodes:

```
public Object getData() {return data;}

public Node getNext() {return next;}
public Node getPrevious() {return previous;}

public void setNext(Node next) {this.next = next;}
public void setPrevious(Node previous) {this.previous
= previous;}

public String toString() {return " "+data;}
```

Code for a doubly linked list

```
public class DoublyLinkedList {
  private Node head; //Reference to the first Node
  private Node tail; //Reference to the last Node
}

public DoublyLinkedList(){    //Constructor
      head = tail = null;
}

public boolean isEmpty() {
      return head == null;
}
```

toString and an add method

```
public String toString() {
        String str = "";
        Node next = head;
        while (next != null) {
            str += next.toString()+"\n";
            next = next.getNext();
        }
        return str;
    }
```

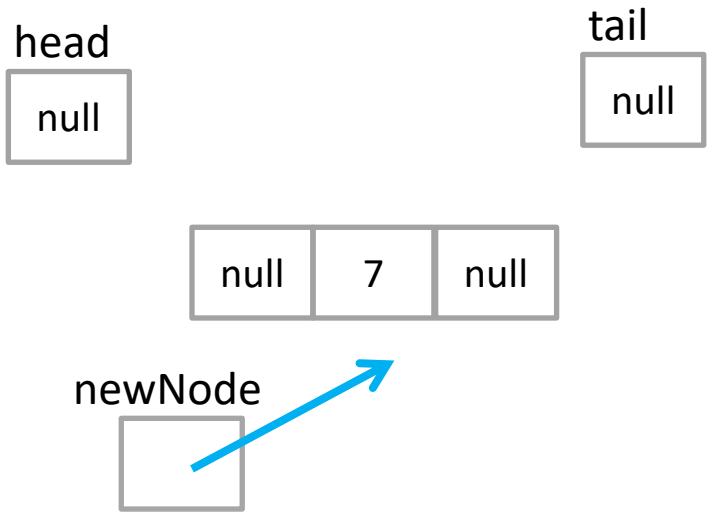
- Let's write a method to add a new piece of data to our list.
  - Data can be added at the **beginning** or at the **end**.

An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

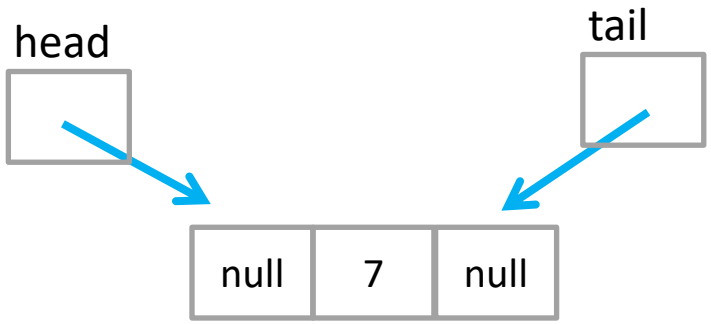


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

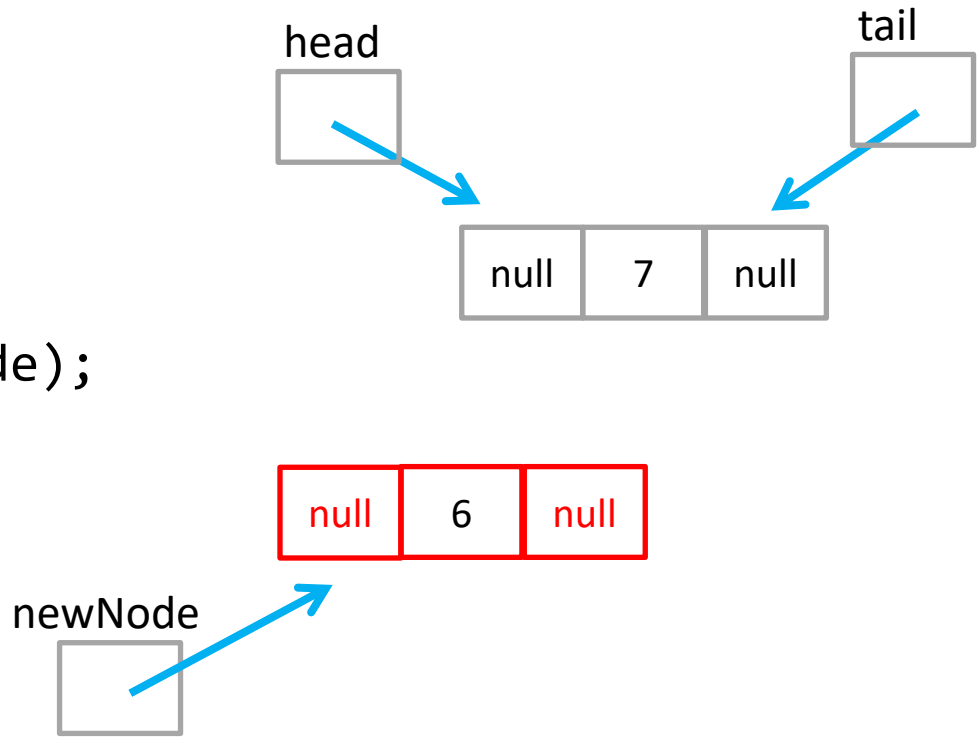


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

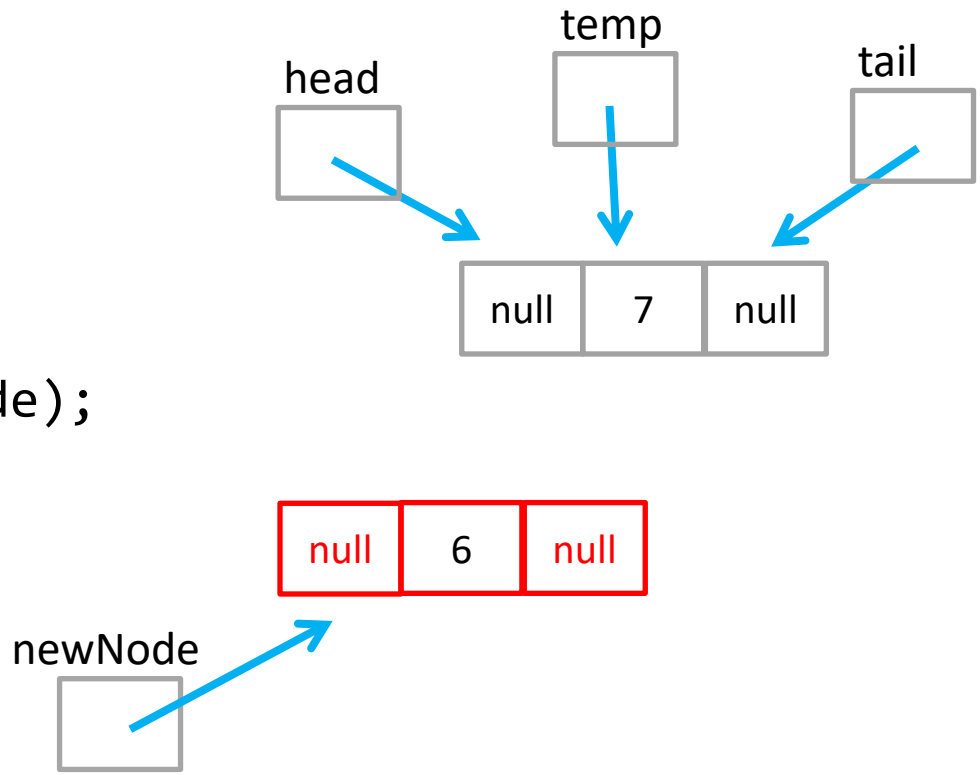


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

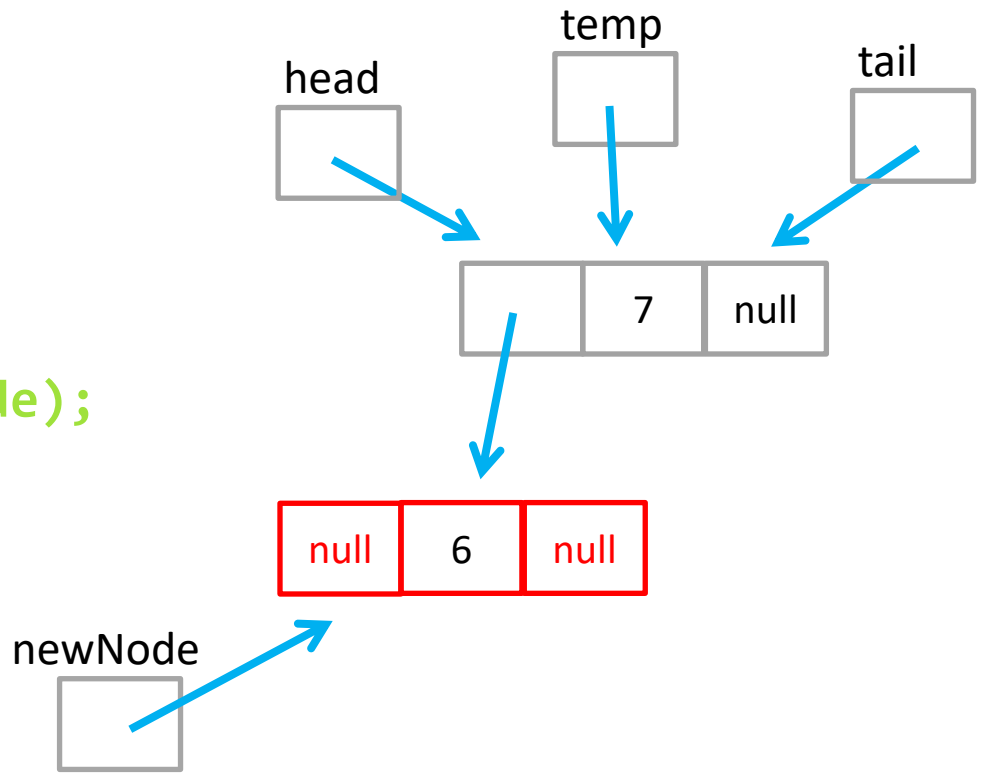


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

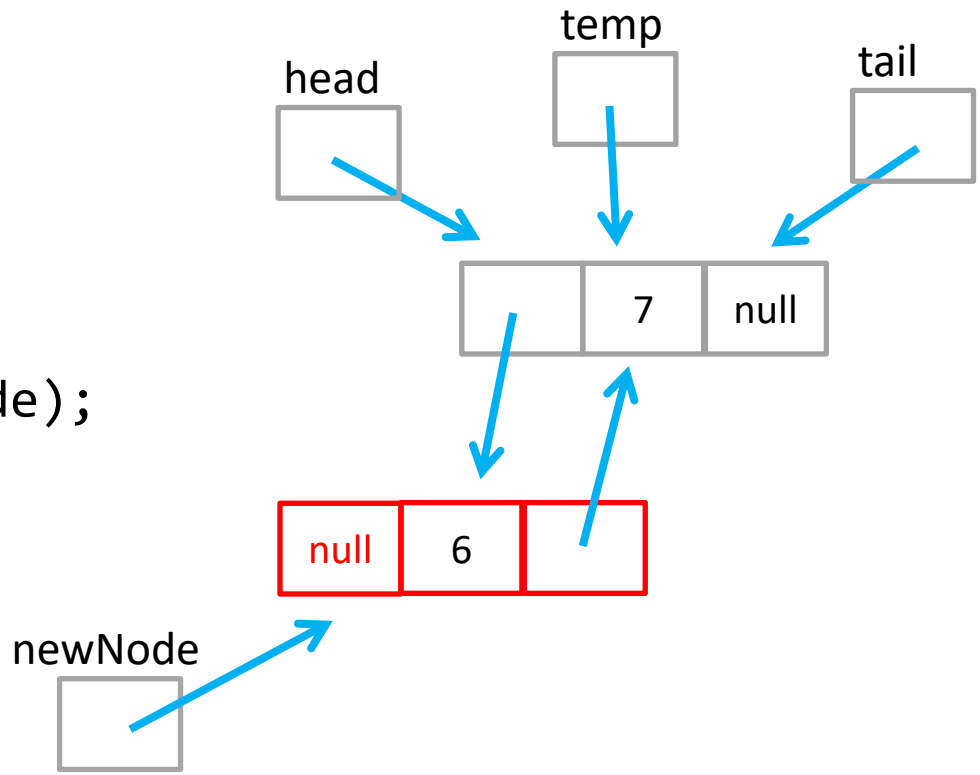


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

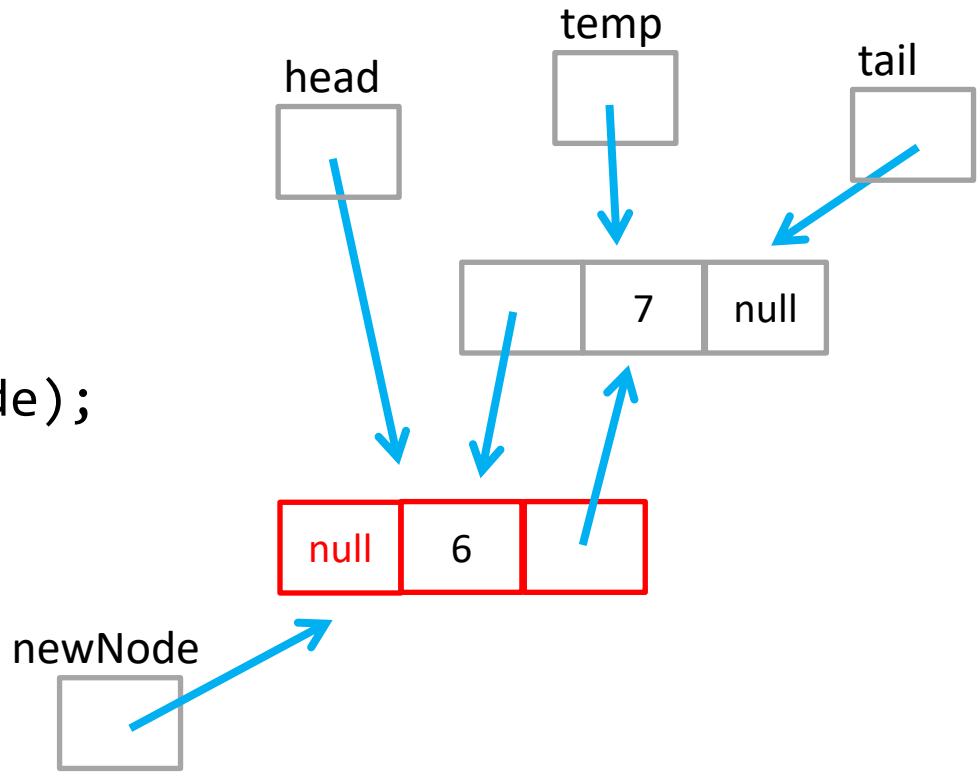


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```

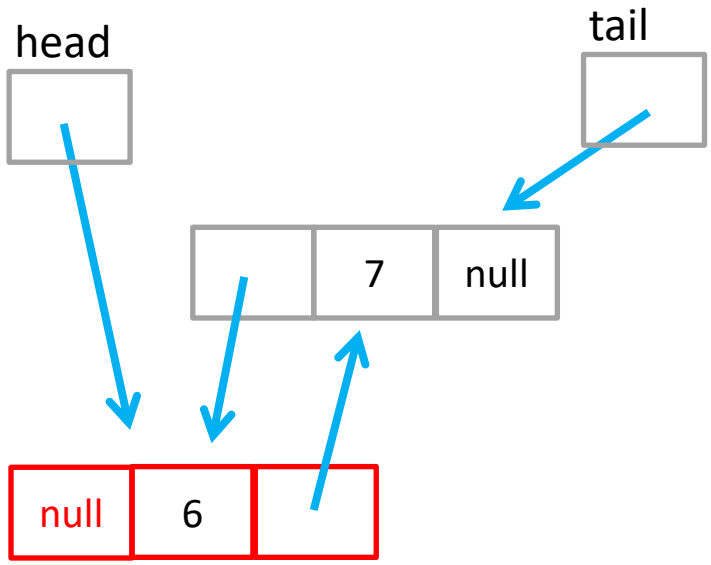


An add method

- Add a new element to the beginning of a Doubly LinkedList.

```
public void addToStart(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        head=tail=newNode;
        return;
    }

    Node temp = head;
    temp.setPrevious(newNode);
    newNode.setNext(temp);
    head = newNode;
}
```



An add method

- Add a new element to the end of a Doubly LinkedList.

```
public void addToEnd(Object newItem) {
        Node newNode = new Node(newItem);
        if (isEmpty()) {
            head=tail=newNode;
            return;
        }

        Node temp = tail;
        temp.setNext(newNode);
        newNode.setPrevious(temp);
        tail = newNode;
 }
```

Pros/Cons Of Doubly Linked Lists

- Pros
  - Traversing the list in reverse order is now possible.
  - It's more efficient for lists that require frequent additions and deletions near the front and back

- Cons
  - An extra reference is needed
  - Additions and deletions are more complex

Questions?