ICS 240: Introduction to Data Structures

Module 4 – Part 1

Linked Lists

Introduction to Linked Lists

Chapter 4

Reading

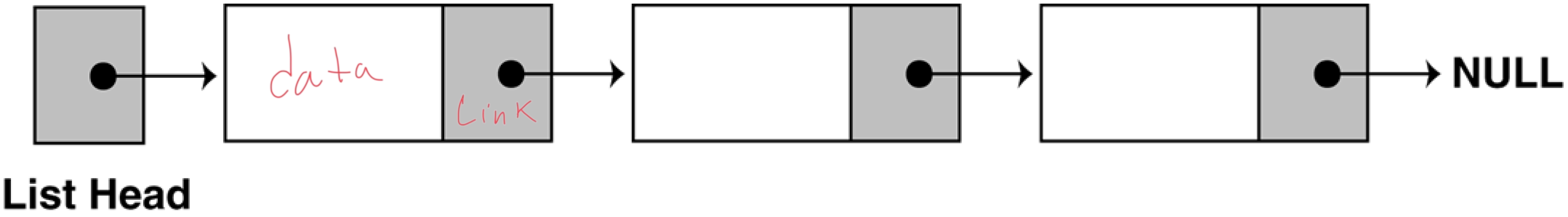
- Chapter 4:
    - Sections 4.1, 4.2, and 4.3.

What is a linked list?

A linked list is a series of connected *nodes*.

A linked list can grow or shrink in size as nodes are added or deleted

A linked list is called "linked" because each node in the list has a `reference` that `links` it to the next node in the list.



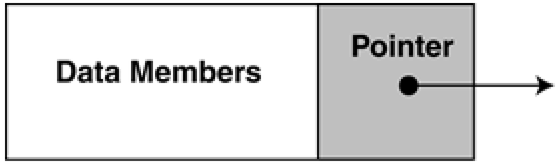
Why do we need linked lists?

- Linked lists can be used instead of arrays if we do not know how many data elements are to be stored

- To create an array, you have to specify the array capacity (i.e., the maximum number of elements to be stored in the array):
  - `int[] myArr = new int[10];`
  - If you specify a size that is **too big**, then you are wasting memory.
  - If you specify a size that is **too small**, then you may have to re-copy the elements to another bigger array which may slow your program.

A linked list consists of `nodes`

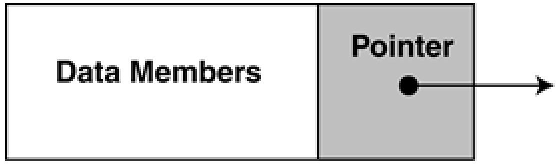
- Each `node` in a linked list contains two instance variables:
  - **`data`**:
    - Data element stored in this node
  - **`link`**:
    - a pointer (or a `reference`), that `links` this node it to the next node in the list.



```
public class IntNode{
    private int data;
    private IntNode link;

    public IntNode(int data, IntNode link){
        this.data = data;
        this.link = link;
    }

    public int getData(){}
    public void setData(int element){}

    public IntNode getLink(){}
    public void setLink(IntNode link){}

    }
```

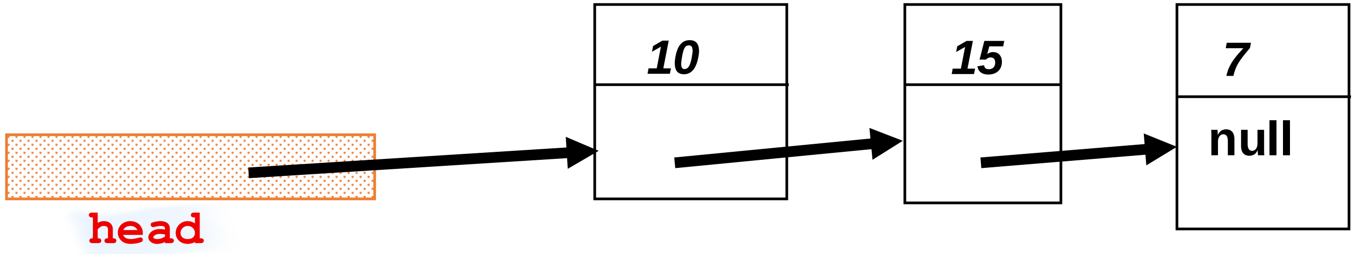
IntNode constructor

```
public class IntNode{
   private int data;
   private IntNode link;

   public IntNode(int data, IntNode link)     {
      this.data = data;
      this.link = link;
   }
}
```



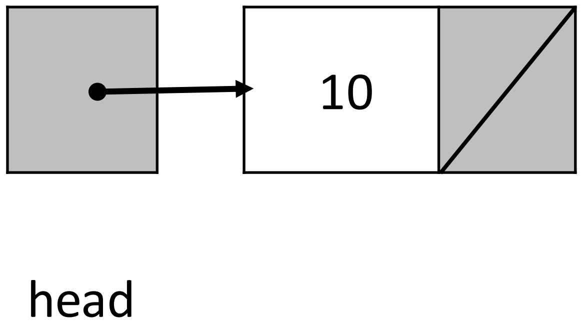
How to handle a linked list programmatically?

- A program usually handles the linked list by keeping a reference to the front node by using a variable called **`head`** which is a `reference` variable that refers to an `IntNode`.



Using the Constructor

```
IntNode head = new IntNode(10, null);
```



IntNode Class Methods

Modifier and Type	Method and Description
int	getData() Accessor method to get the data from this node.
IntNode	getLink() Accessor method to get a reference to the next node after this node.
void	setData(int newData) Modification method to set the data in this node.
void	setLink(IntNode newLink) Modification method to set the link to the next node after this node.

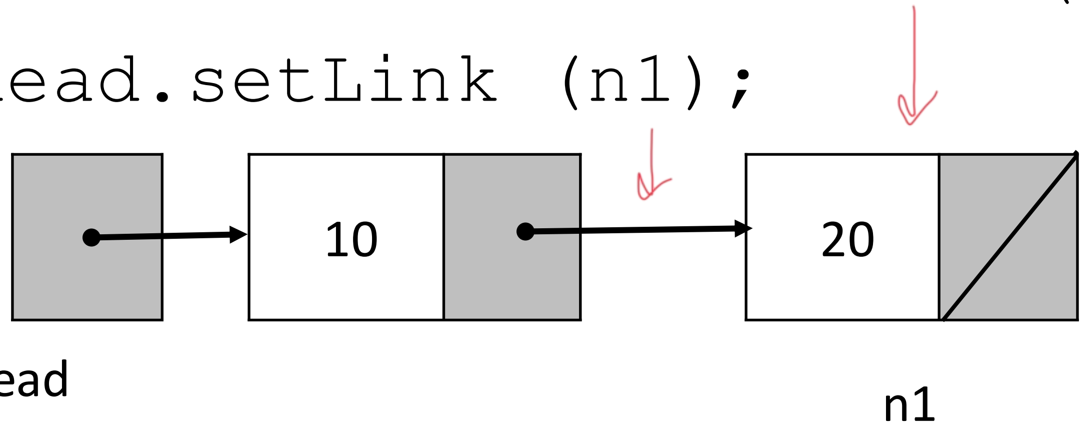
Getters and Setters

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


public void setData(int element){
      this.data = element;
}
```

```
public IntNode getLink(){
      return link;
}


public void setLink(IntNode link)
      {
      this.link = link;
}
```

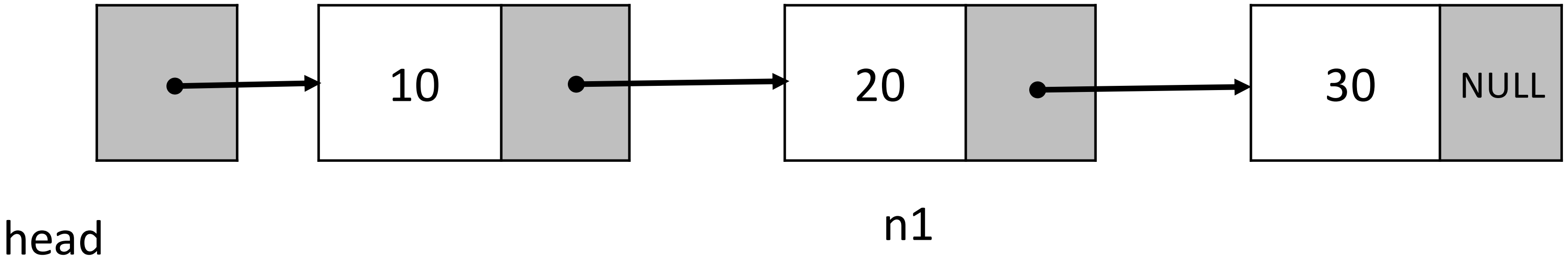
Connecting Nodes

```
IntNode n1 = new IntNode(20, null);
head.setLink (n1);
```



And Adding Another Node

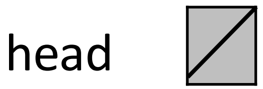
```
head.getLink().setLink(new IntNode(30, null));
```



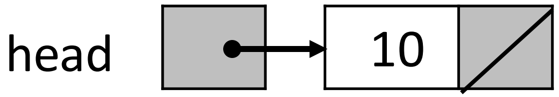
There’s An Easier Way

→ if we don't care about order

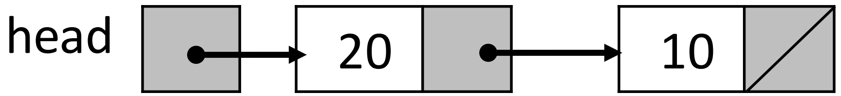
```
IntNode head = null;
```



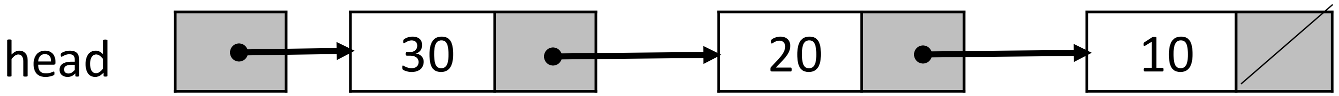
```
head = new IntNode(10, head);
```



```
head = new IntNode (20, head);
```



```
head = new IntNode (30, head);
```



Basic operations on a linked list

- **`create`**: to create a new linked list

- **`traverse`**: to traverse the list (and may be perform an operation on each node's data).

- **`insert`**: to insert a new node in the list

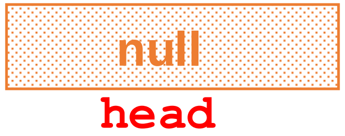
- **`delete`**: to delete a node from the list

create a new linked list

- A list is created by creating a node that represent the **head** of the list:

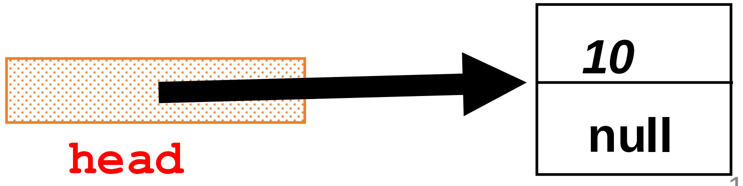
- To create an empty linked list:

```
IntNode head = null;
```



- To create a list with one node (with data value = 10):

```
IntNode head = new IntNode(10,null)
```



How to traverse a linked list?

- There is a pattern that can be used whenever you need to step through all the nodes of a linked list one at a time

- The steps are as follows:
  - Start a cursor to refer to the head of the list

    `IntNode cursor = head;`

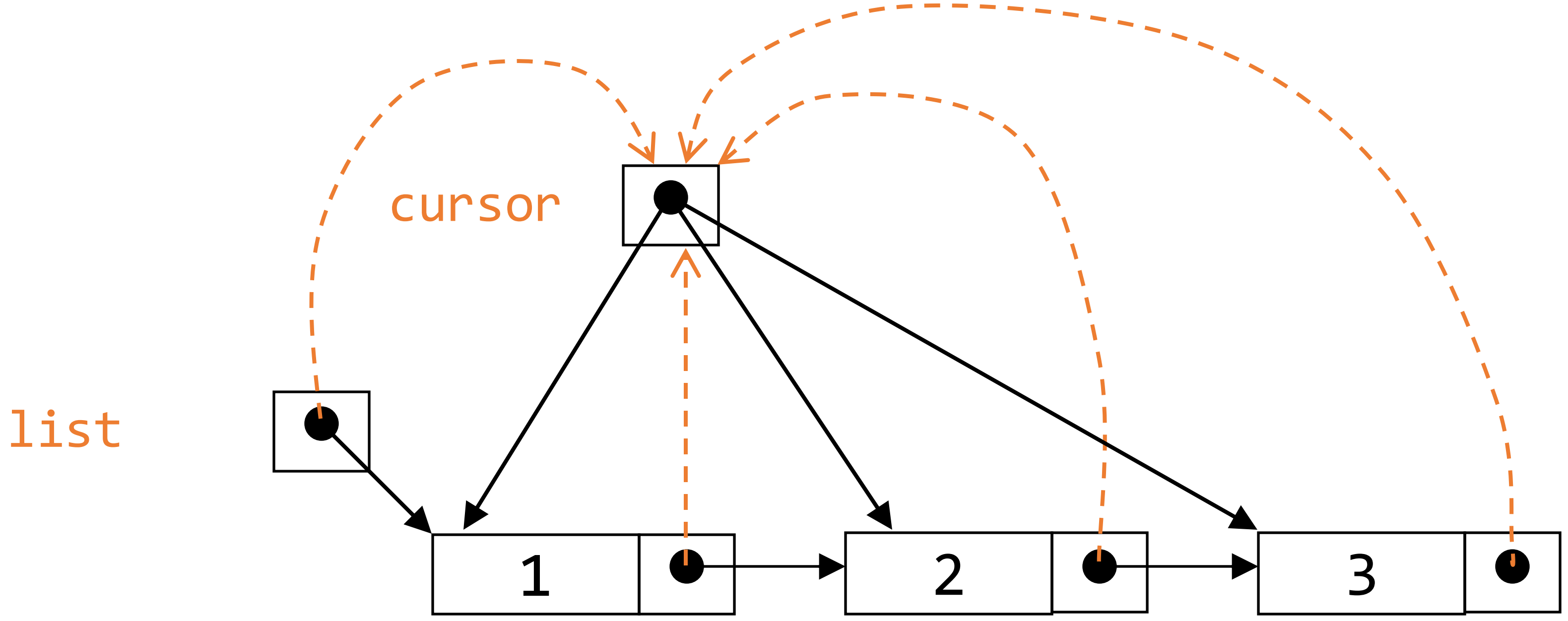
    alias

    both refer to same node
  - To move the cursor to the next node, we use

    `cursor = cursor.getLink()`
  - The loop should terminate when `cursor is null` because this means there are no more nodes in the list

```
IntNode cursor = head;
while (cursor != null){
        //do something with
        //the current node
        cusror = cursor.getLink()
}
```

traverse a linked list(animation)



display a linked list

```
public static void display(IntNode list){
    IntNode cursor = list;
    while (cursor != null){
        System.out.println(cursor.data);
        cursor = cursor.getLink();
    }
}
```

head

`insert` a node in a linked list

- There are different ways to insert a new node to a linked list:
  - At the front of the list
  - At the end of the list
  - Before a given node (specified by a *reference*)
  - After a given node (specified by a *reference*)
  - Before a given value
  - After a given value

- All are possible, but differ in difficulty

Example:

Insert at the front

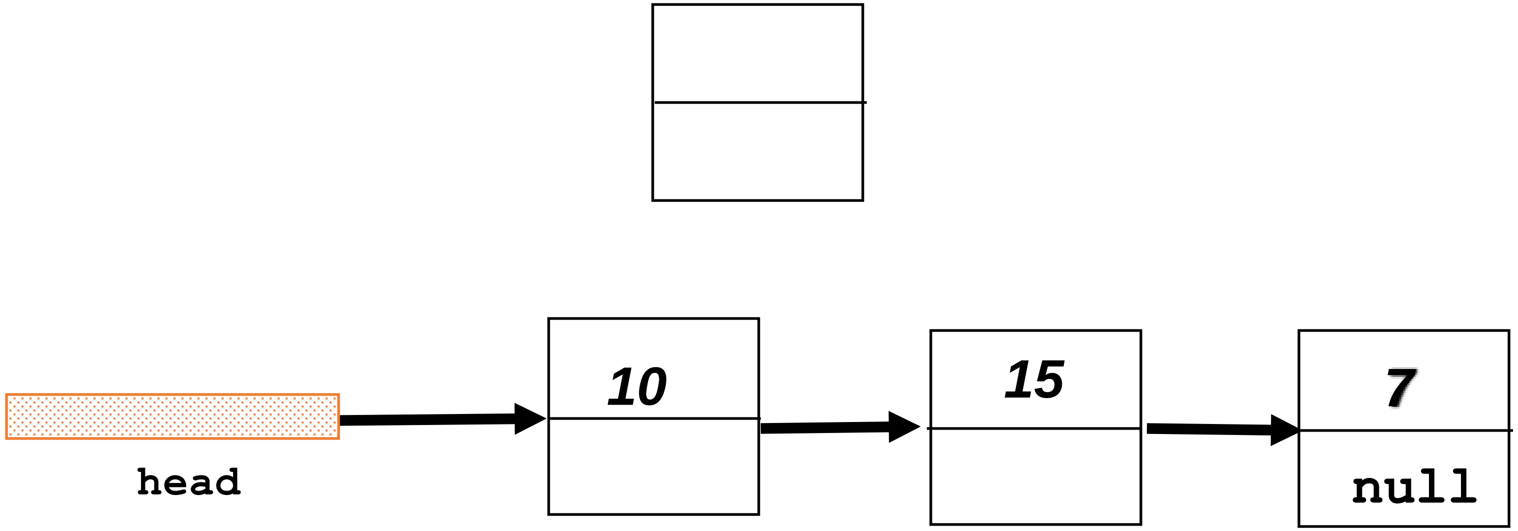
insert a node with data value 13 at the front of the linked list that is defined by **head**

- This is the easiest insertion operation to implement

```
head = new IntNode(13,head);
```

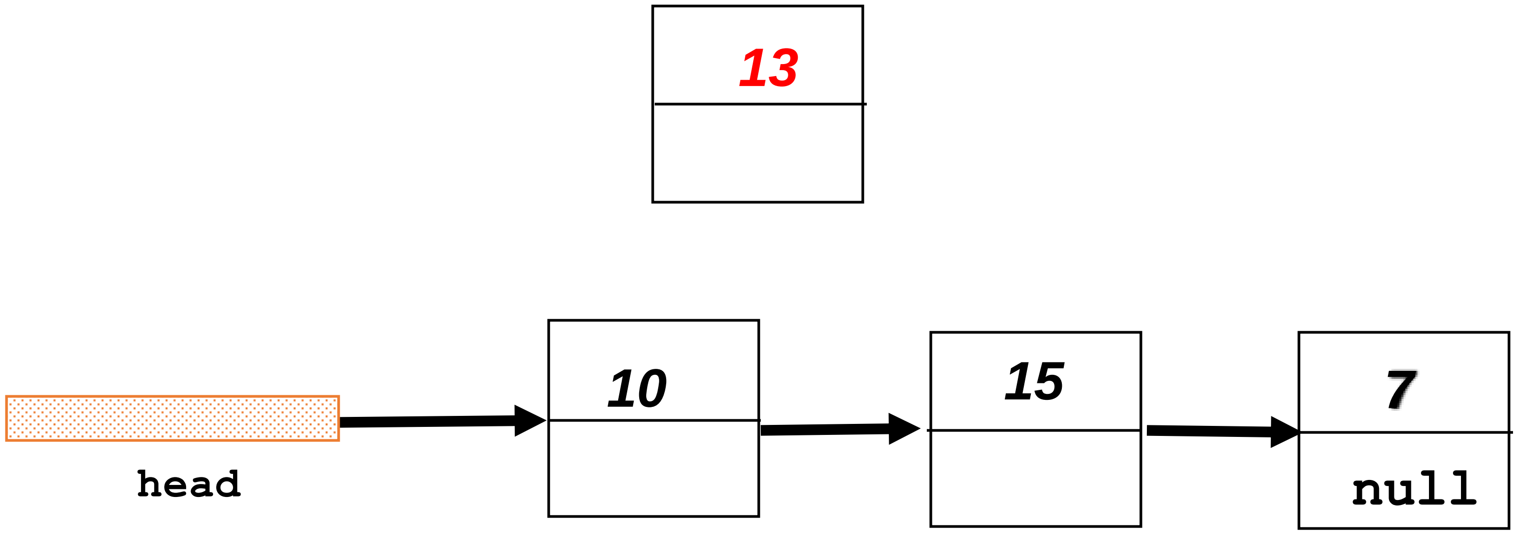
insert a node at the head of a linked list (animation) 1

action 1: create a new node



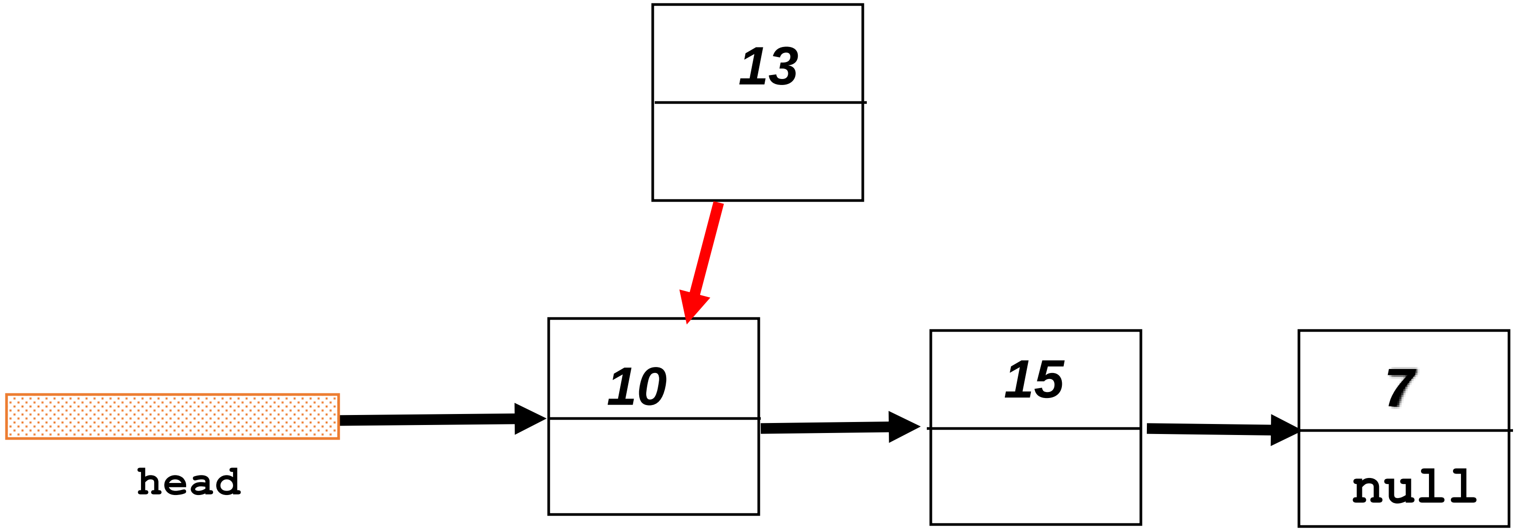
`insert` a node at the front of a linked list (animation) 2

action 2: place 13 in the new node's `data` field.



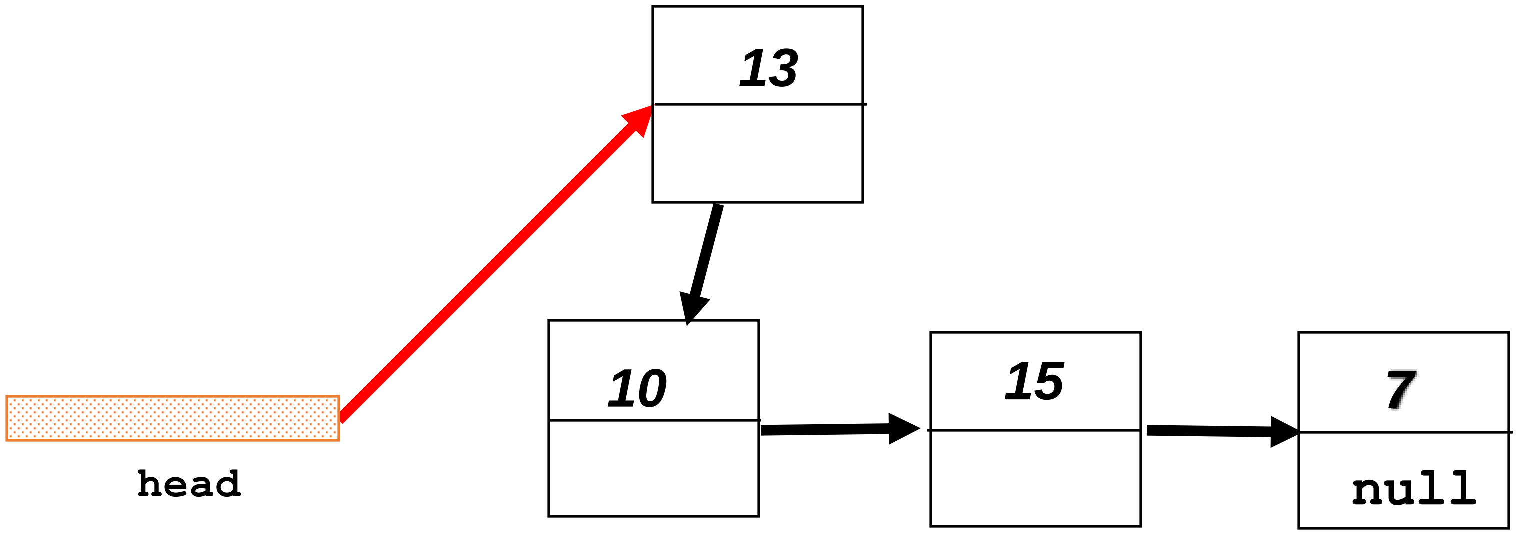
insert a node at the front of a linked list (animation) 3

action 3: make the new node's `link` points to the current **head**



insert a node at the front of a linked list (animation) 4

action 4: change **head** to refer to the new node as the new node becomes the new head of the linked list.



insert a node at the front of a linked list

You just need to use `IntNode`’s constructor

```
head = new IntNode(13,head)
```

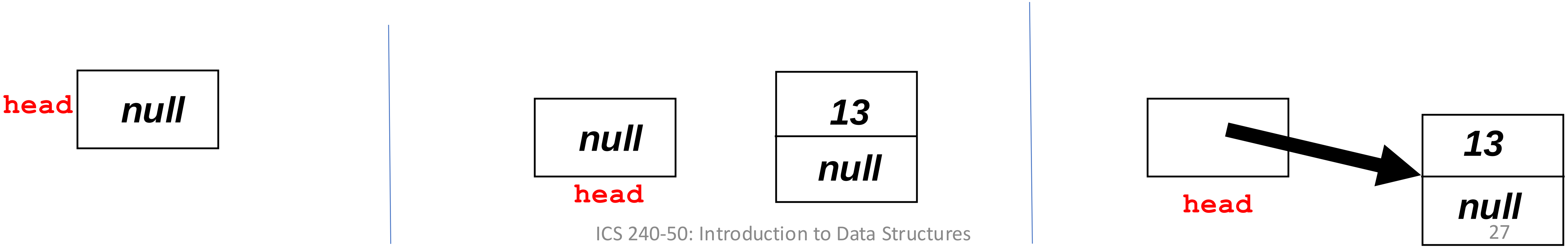
```
public IntNode(int data, IntNode link){
    this.data = data;
    this.link = link;
}
```

Does the constructor work correctly for the first node in a new list ? yes

```
public IntNode(int data, IntNode link){
    this.data = data;
    this.link = link;
}
```

Suppose **head** is `null` and we execute the following assignment:

```
head = new IntNode(13, head);
```

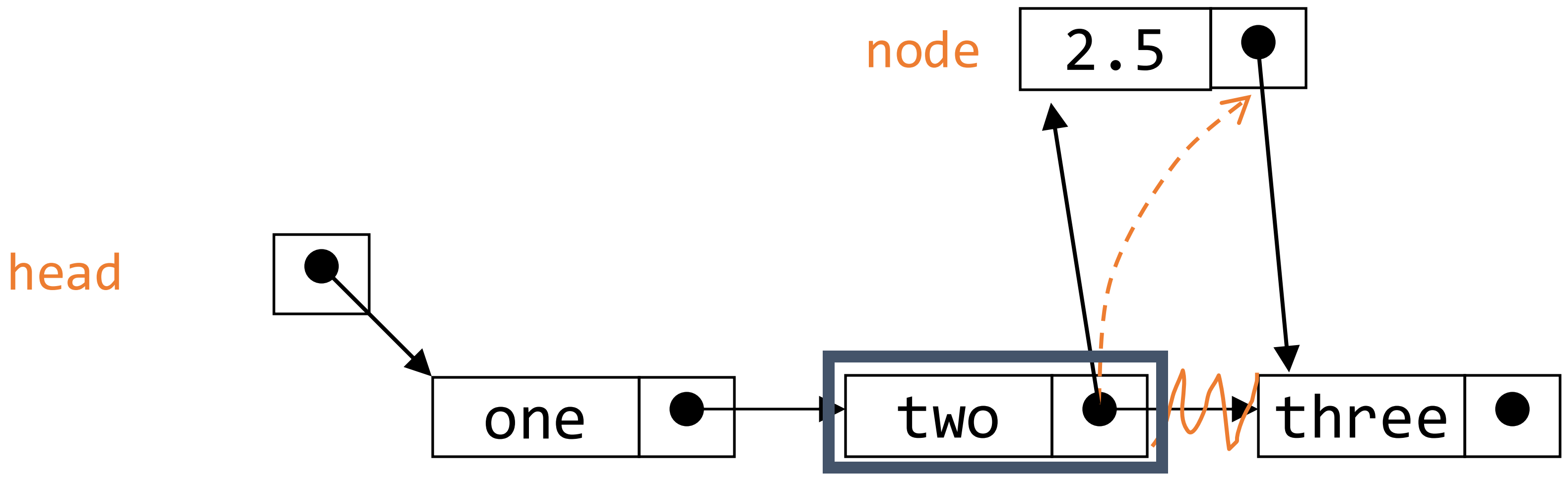


Pseudocode for Inserting a node at any position in the list

- Nodes are often inserted at places other than the front of a linked list.

- There is a general pseudocode that you can follow for any insertion in the linked list.

insert after a specific node (animation)



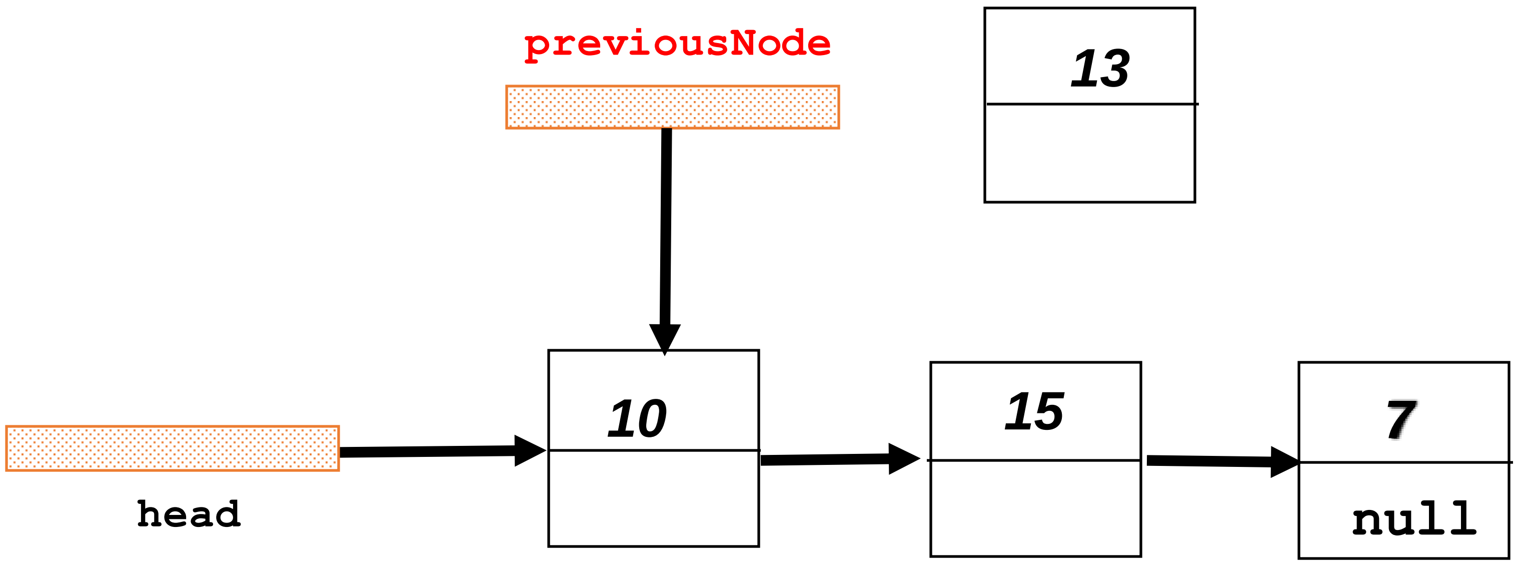
Find the node you want to insert after

First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

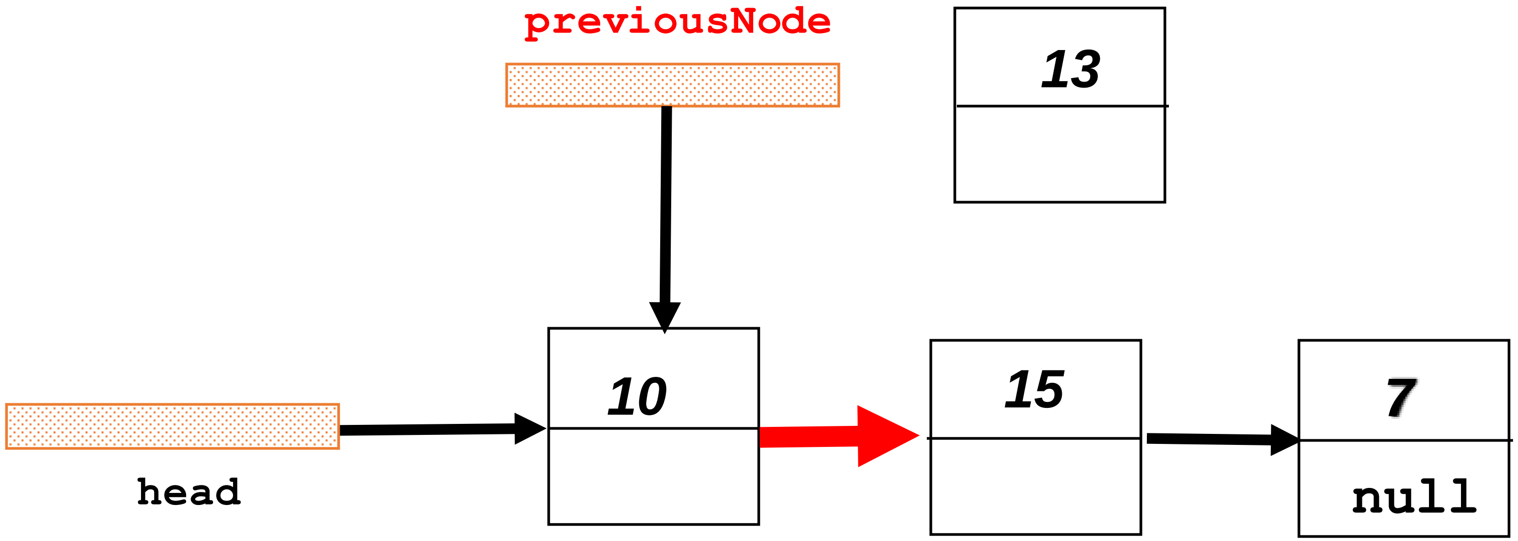
Assume you want to insert a node with value 13 as the second node in the linked list

Create a reference called **previousNode** that refers to the node that you are going to insert after.



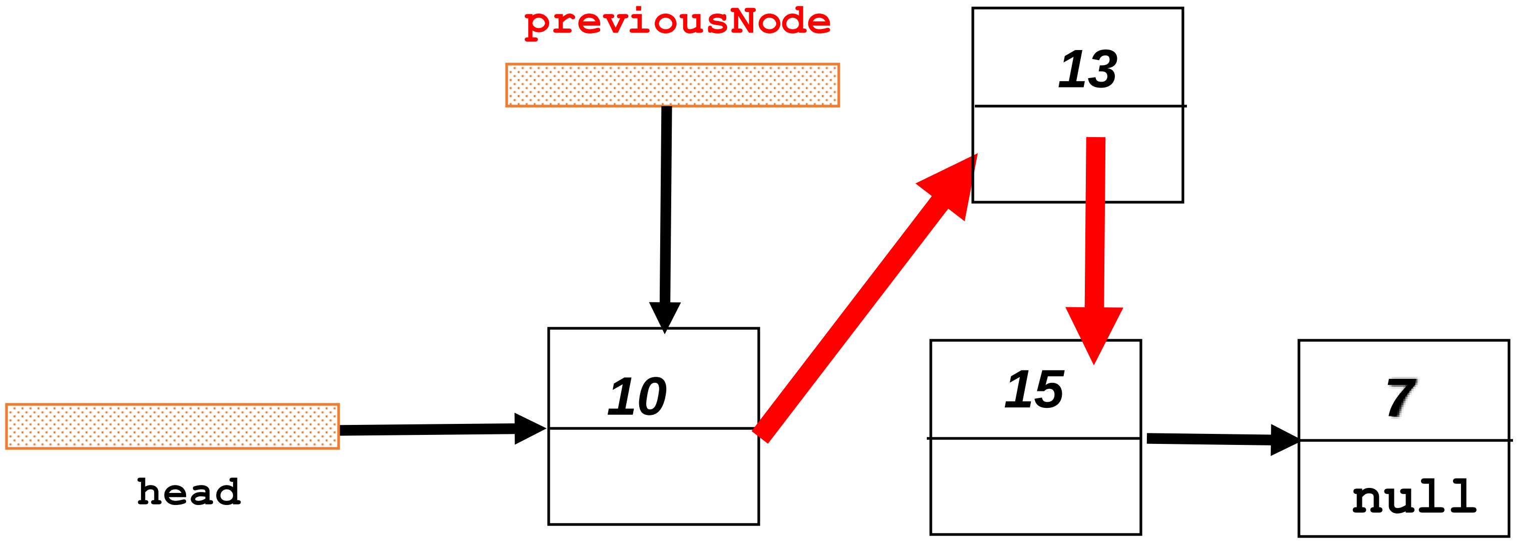
What is the value of `link` in `previousNode`?

This link is called
previousNode.link



Adjusting links

```
previousNode.link =  new IntNode(13, previousNode.link);
```



Summary for **insert** in a linked list

to insert at the head of the list:

```
head = new IntNode(newValue, head);
```

else:

- set a reference named previousNode to refer to the node which is just before the new node's position.
- Then perform the following:

```
previousNode.link =  new IntNode(newValue, previous.link);
```

addNodeAfter

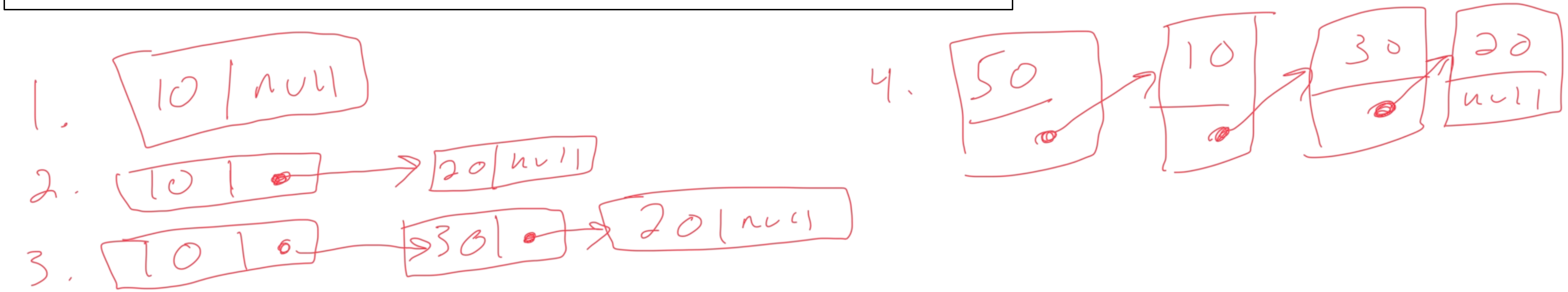
addNodeAfter is a method that is used to insert a node after a given node

This method is an **instance** method in the IntNode class which means it is called from an instance of type IntNode

```
public void addNodeAfter(int element){
   this.link = new IntNode(element,this.link);
 }
```

Draw the linked list the results from running the following code

```
IntNode myList = new IntNode(10,null);
myList.addNodeAfter(20);
myList.addNodeAfter(30);
myList = new IntNode(50,myList);
```



`deleting` a node from a Linked List

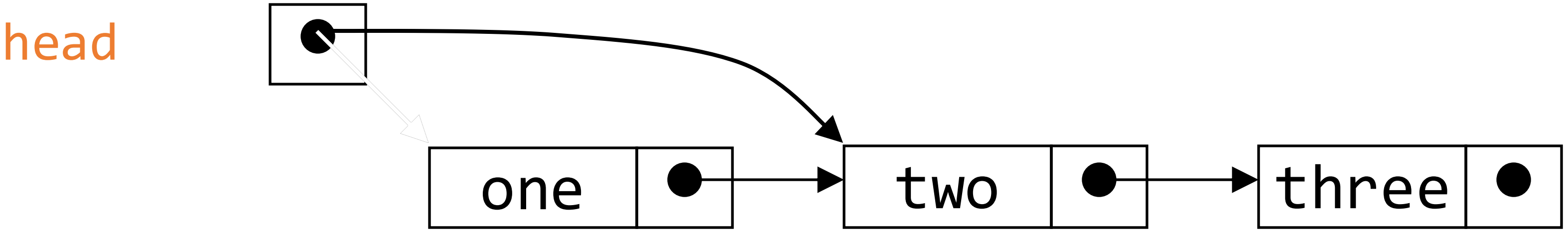
- In order to delete a node from a linked list, you need to change the `link` in its *predecessor* node.

- This is slightly tricky, because we can't follow a pointer backwards

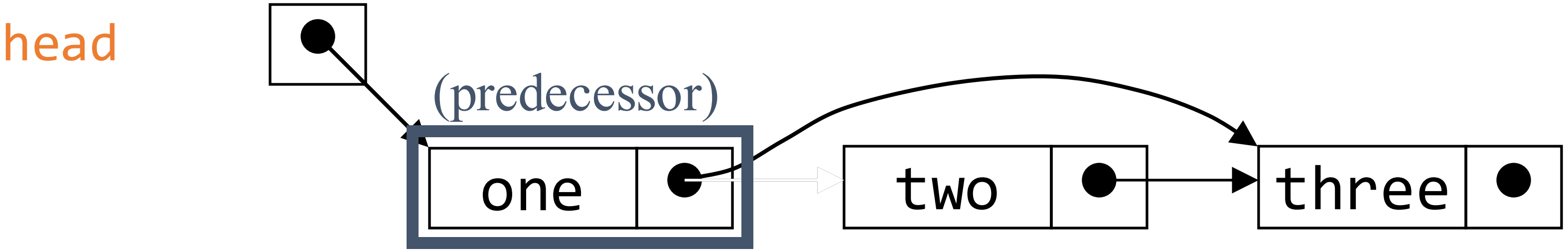
- Deleting the first node in a list is a special case, because it is the `head` of the list and it does not have a *predecessor* node.

deleting a node from a Linked List

- To delete the first node, just change the head

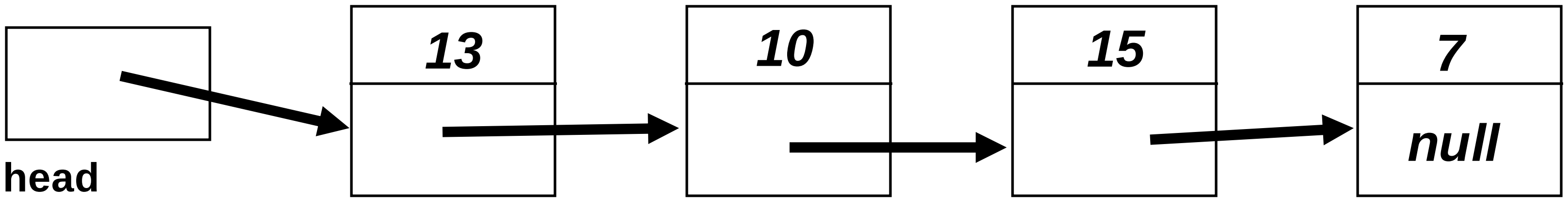


- To delete some other node, change the link in its *predecessor*

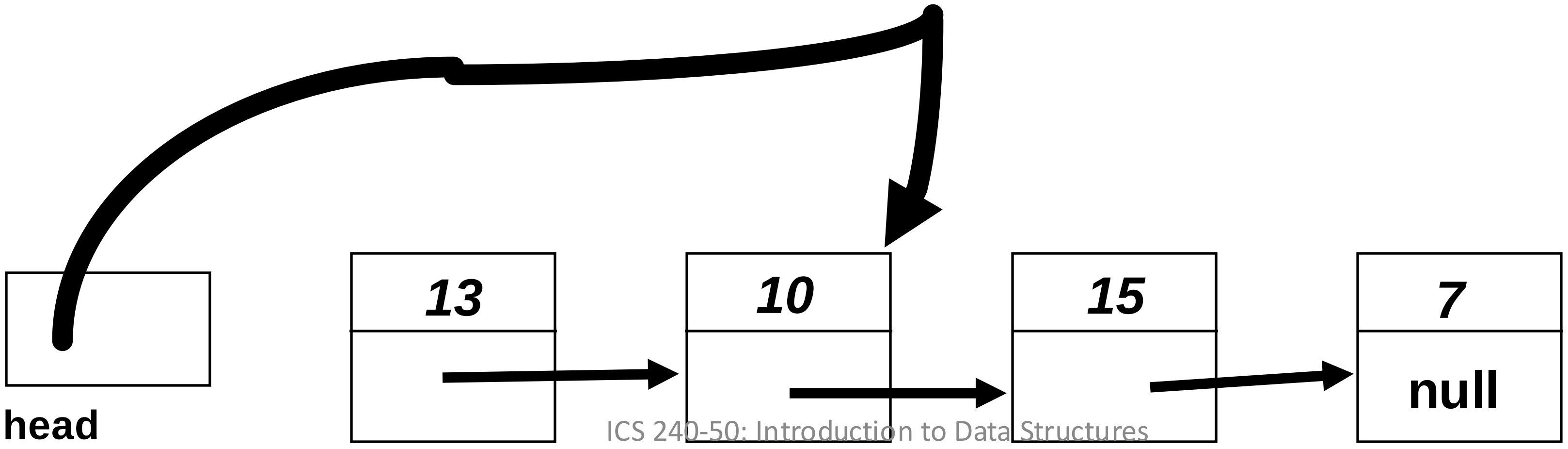


- Note that the deleted nodes will eventually be garbage collected

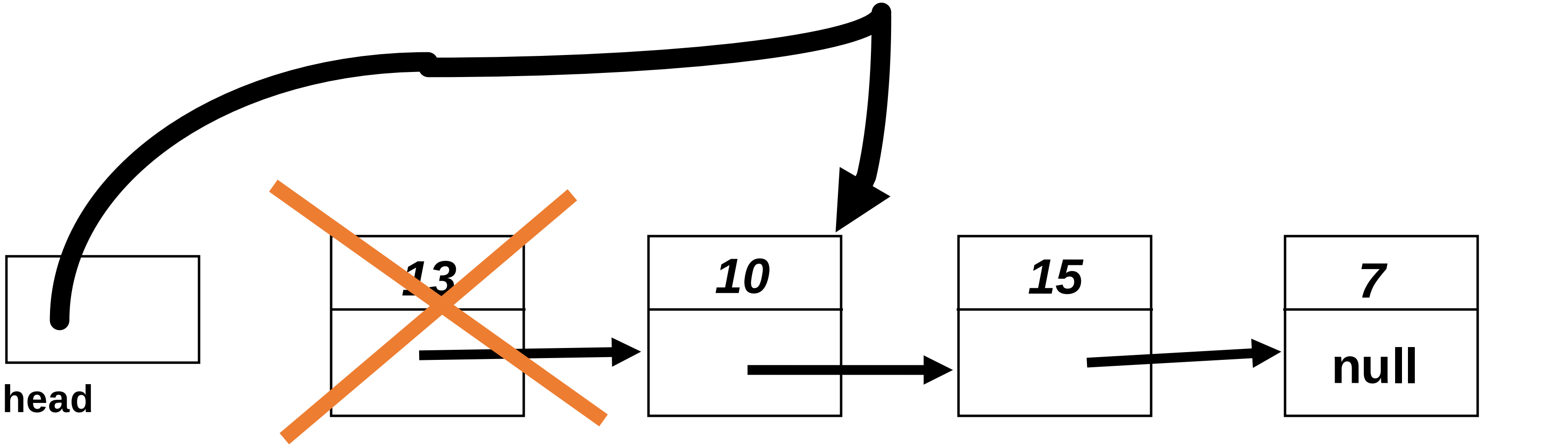
deleteing the head node



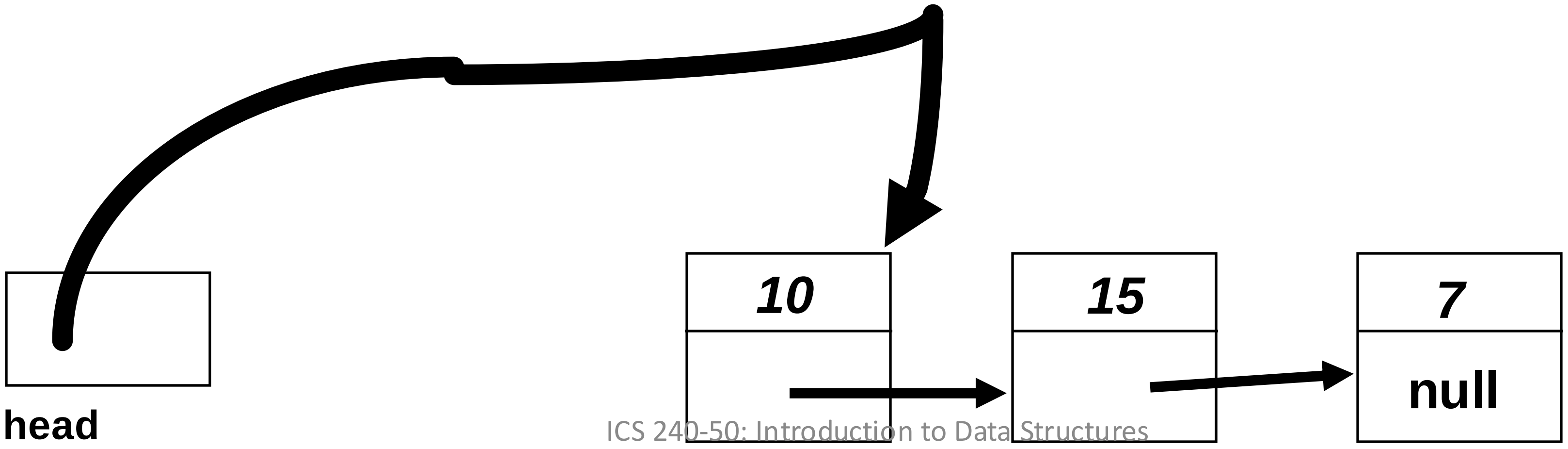
```
head  =  head.link;
```



deleting the head node (continued)

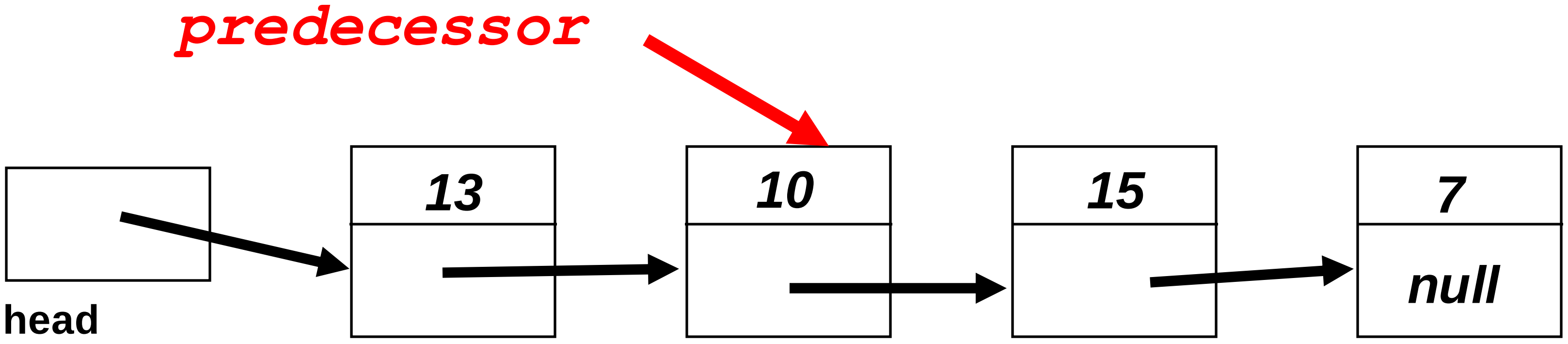


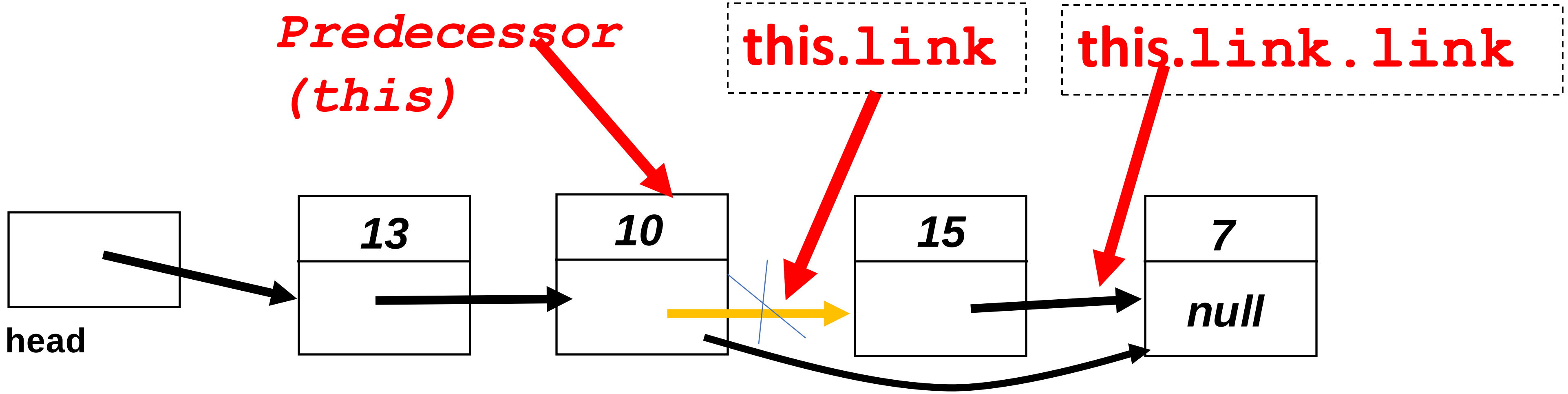
Here's what the linked list looks like after deleting the head node



deleting a node other than the head

- Similar to insertion, you need to first set up a reference to the node that is just before the node we are removing (*predecessor*).
- Assume you want to delete the value 15 from the following list:





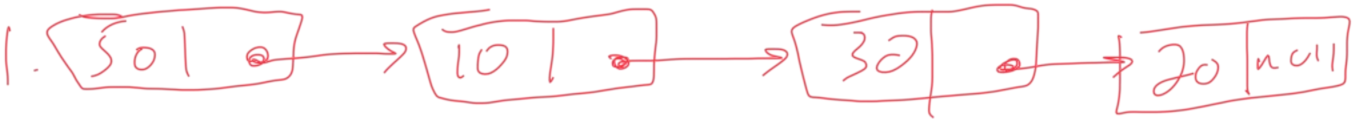
```
public void removeNodeAfter(){
   this.link = this.link.link
 }
```

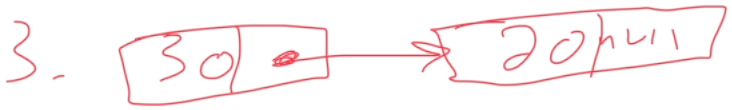
IntNode class: instance methods

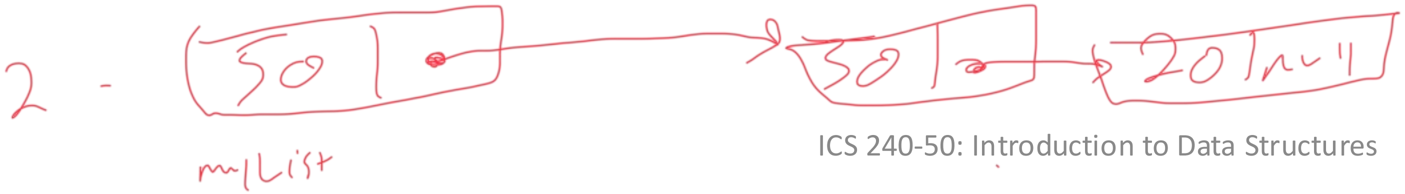
Modifier and Type	Method and Description
void	addNodeAfter(int item) Modification method to add a new node after this node.
int	getData() Accessor method to get the data from this node.
IntNode	getLink() Accessor method to get a reference to the next node after this node.
void	removeNodeAfter() Modification method to remove the node after this node.
void	setData(int newData) Modification method to set the data in this node.
void	setLink(IntNode newLink) Modification method to set the link to the next node after this node.

Example 1: show the linked list created by this code

```
IntNode myList = new IntNode(10,null); //{10}
myList.addNodeAfter(20);//{10,20}
myList.addNodeAfter(30);//{10,30,20}
//adding a node at the beginning of the list:
myList = new IntNode(50,myList); //{50,10,30,20}
myList.removeNodeAfter(); //{50,30,20}
//removing the head node
myList = myList.getLink(); //{30,20}
```

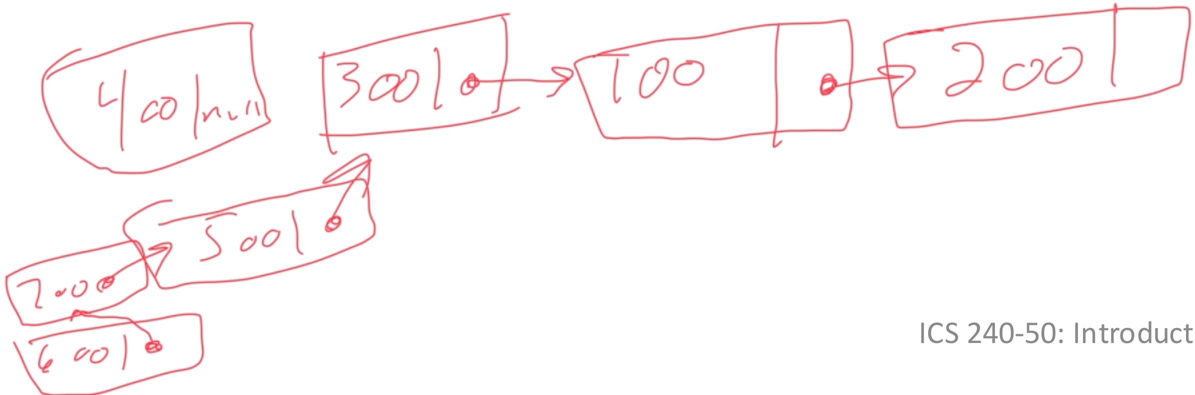






Example 2: show the linked list created by this code

```
IntNode myList = new IntNode(100,null);
myList.addNodeAfter(200);
myList = new IntNode(300,myList);
myList = new IntNode(400,myList);
myList.addNodeAfter(500);
myList = myList.getLink();
myList = new IntNode(600,myList);
myList.addNodeAfter(700);
```



Why do we need linked lists?

- To overcome the following disadvantages of using arrays:
  - **Fixed size:** to create an array you have to specify the size, however, a linked list can grow and shrink dynamically.

  - **Adding at random positions:** adding an element to the front of an array (or in the middle) is very hard since a lot elements need to be copied to other locations in order to make space for the new element to be inserted.
    - However, for a linked list, an element can be added at any location by performing few assignment statements to adjust the links.

Guidelines for choosing between an array and a linked list

Operation	Which data structure to use?
Frequent random-access operations	Array
Frequent Insertion and deletion at random location	Linked list (to avoid moving elements up and down)
Frequent capacity change	Linked list (to avoid the resizing inefficiency)

Linked List Animation

- https://visualgo.net/list

Manipulating an entire linked list

- So far, we discussed the `IntNode` class which is used to represent only one node in the linked list

- Next, we will discuss how to perform more actions on an entire linked list, for example:
  - Find how many nodes are there in the list
  - Print the list contents in a reverse order
  - Count occurrences of a certain value.

- Actions that works on an entire linked list are implemented as as static methods. Why?
  - static methods can be used even for an empty list

Difference between instance methods and static methods?

- Assume you want to implement a method to count how many nodes are there in the linked list (i.e., `size()`)

- There are two different approaches to implement this methods:
  - **Instance method:**
    - Method declaration: `public int size()` {}
    - Method use:

```
IntNode head = new IntNode(10,null);
head.addNodeAfter(20);
head.size() // will return 2
//however,if the list is empty and head is null then we cannot call this method
```

  - **Static method:**
    - Method declaration: `public static int size(IntNode head)` {}
    - Method use:

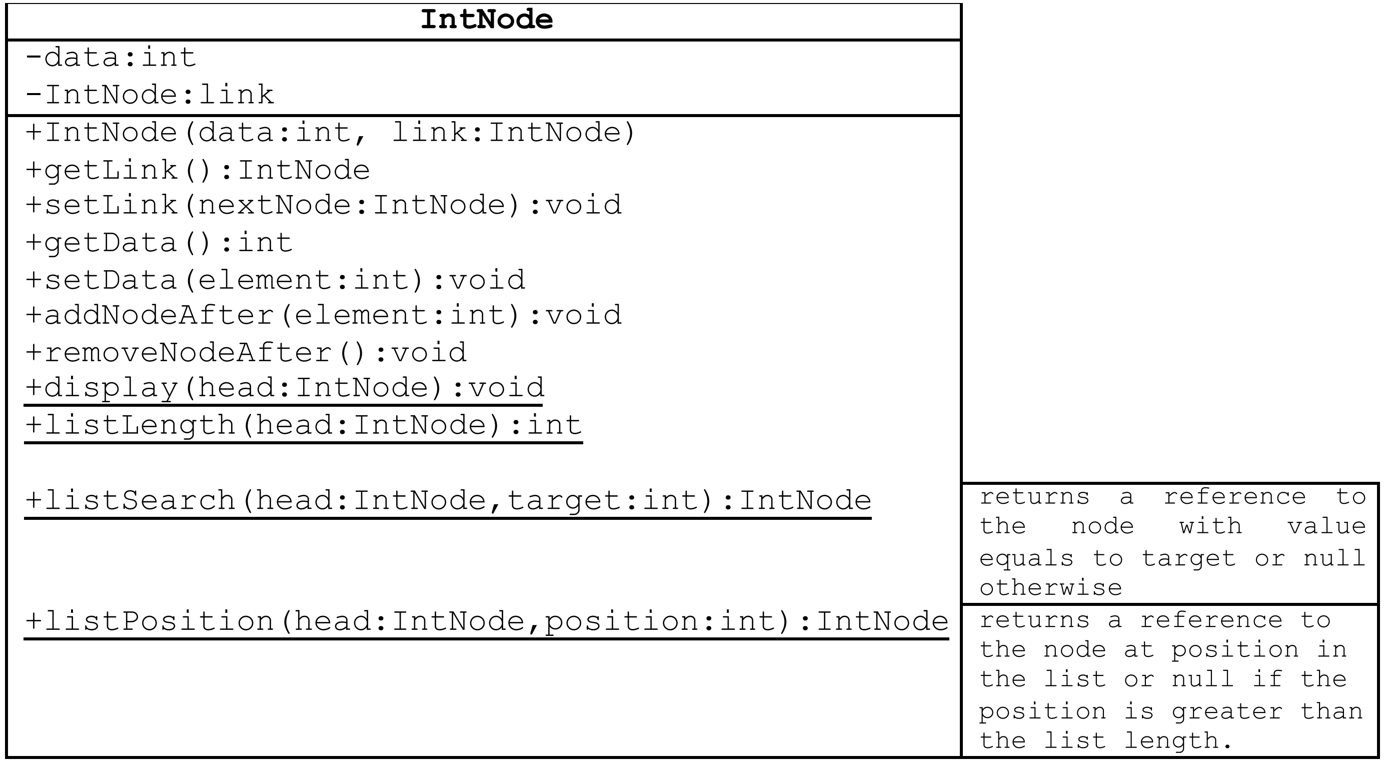
```
IntNode head = new IntNode(10,null);
head.addNodeAfter(20);
IntNode.size(head) // will return 2
//if head is null, then this method returns 0
```

Operations on a linked list

- We will discuss how to implement the following operations as **static** methods in the **IntNode** class
  - display: displaying the contents of a linked list
  - listLength: finds the length (i.e., the number of nodes) of a linked list
  - listSearch: searches for a particular piece of data in a linked list
  - listposition: return a reference to the node that is located at a specific position in the linked list
  - listCopy: create a copy of the list and return as output a reference to the head of that copy.
  - Copying only a part of the linked list given the start and end points
  - Return an array with the same contents as the linked list
  - Other

`IntNode` class: static methods

Modifier and Type	Method and Description
static IntNode	listCopy(IntNode source) Copy a list.
static IntNode[]	listCopyWithTail(IntNode source) Copy a list, returning both a head and tail reference for the copy.
static int	listLength(IntNode head) Compute the number of nodes in a linked list.
static IntNode[]	listPart(IntNode start, IntNode end) Copy part of a list, providing a head and tail reference for the new copy.
static IntNode	listPosition(IntNode head, int position) Find a node at a specified position in a linked list.
static IntNode	listSearch(IntNode head, int target) Search for a particular piece of data in a linked list.



IntNode

-data:int
-IntNode:link

+IntNode(data:int, link:IntNode)
+getLink():IntNode
+setLink(nextNode:IntNode):void
+getData():int
+setData(element:int):void
+addNodeAfter(element:int):void
+removeNodeAfter():void
+display(head:IntNode):void
+listLength(head:IntNode):int

+listSearch(head:IntNode,target:int):IntNode

returns a reference to the node with value equals to target or null otherwise

+listPosition(head:IntNode,position:int):IntNode

returns a reference to the node at position in the list or null if the position is greater than the list length.

Review: How to **traverse** a linked list?

- There is a pattern that can be used whenever you need to step through the nodes of a linked list one at a time

- The steps are as follows:
  - Start a cursor to refer to the head of the list
    `IntNode cursor = head;`

  - to move the cursor to the next node, we use
    `cursor = cursor.getLink()`

  - The loop should terminate when `cursor = null` because this means there are no more nodes in the list

```
IntNode cursor = head;
while (cursor != null){
        //do something with
        //the current node
        cusror = cursor.getLink()
}
```

`display` is a method to print the contents of the linked list.

```
public static void display(IntNode list){
      IntNode cursor = list;
      while (cursor != null){
            System.out.println(cursor.data);
            cursor = cursor.getLink();
      }
}
```

listLength is a method that returns the number of nodes in the linked list.

```
public static int listLength(IntNode head){
      IntNode cursor = head;
      int length = 0;

      while (cursor != null){
            length++;
            cursor = cursor.getLink();
      }
      return length;
}
```

`listSearch` is a method that searches the linked list for a specific value. If found, the method returns a reference to the node that contains the value. This method is used when you need to insert an element after or before a specific value.

```
public static IntNode listSearch(IntNode head, int target){
     IntNode cursor = head;
     while (cursor != null){
          if (cursor.getData() == target)
               return cursor;
          cursor = cursor.getLink();
     }
     return null;
}
```

Example on using listSearch

- Write code to insert the following values in the linked list in the given order.
  - 10,15,20,30

```
IntNode myList = new IntNode(10,null);
myList.addNodeAfter(15);
IntNode n = IntNode.listSearch(myList, 15);
n.addNodeAfter(20);
n= IntNode.listSearch(myList, 20);
n.addNodeAfter(30);
IntNode.display(myList);
```

`listPosition` is a method that returns a reference to the node that is located at a specific position in the linked list or `null` if the number of nodes in the list is less than position. The head node is at position 1

```
public static IntNode listPosition(IntNode head, int position){
     IntNode cursor = head;
     int index = 1;
     while (cursor != null && index < position){
          index++;
          cursor = cursor.getLink();
     }
     return cursor;
}
```

Example: Draw the linked list that results from the following code.

```
IntNode myList = new IntNode(10,null);
myList.addNodeAfter(20);
IntNode n = IntNode.listSearch(myList, 20);
n.addNodeAfter(30);
n = IntNode.listSearch(myList, 20);
n.addNodeAfter(40);
myList.addNodeAfter(40);
n= IntNode.listPosition(myList, 2);
IntNode m = IntNode.listPosition(n,2);
n.addNodeAfter(50);
n= IntNode.listPosition(myList, 3);
myList.addNodeAfter(60);
System.out.println(IntNode.listPosition(myList, 2).getData());
```