ICS 240: Introduction to Data Structures

Module 4 – Part 2

Linked Lists

Linked Lists (continued)

Chapter 4

Reading

- Chapter 4:
    - Sections 4.3 and 4.4

The `Bag` ADT with Linked Lists (Section 4.4)

- `IntNode` methods are used to handle only one node of the linked list.

- **static** methods are used to handle an entire linked list given the head node

- However, for a collection class, we need to be able to perform operations like:
  - `+add(element:int):void`
  - `+remove(target:int):void`

- Any data structure that was implemented using an array, can also be implemented using a linked list.
  - In Assignment 4, you will implement a collection class that stores `Thing`s in a linked list.

IntLinkedBag implementation

- The bag interface should be the same whether the bag is implemented using an **array** or using a **linked list**

- One big difference is that when a bag is implemented using a linked list then we do not have to worry about capacity
  - So capacity handling operations are not needed.

- The linked bag collection is implemented using two instance variables as follows:

```
public class IntLinkedBag{
        private IntNode head;
        private int  manyItems;
}
```

contrast to

```
public class IntArrayBag{
        private int[] data;
        private int  manyItems;
}
```

IntLinkedBag instance methods

Modifier and Type	Method and Description
void	add(int element) Add a new element to this bag.
void	addAll(IntLinkedBag addend) Add the contents of another bag to this bag.
void	addMany(int... elements) Add new elements to this bag.
java.lang.Object	clone() Generate a copy of this bag.
int	countOccurrences(int target) Accessor method to count the number of occurrences of a particular element in this bag.
int	grab() Accessor method to retrieve a random element from this bag.
boolean	remove(int target) Remove one copy of a specified element from this bag.
int	size() Determine the number of elements in this bag.

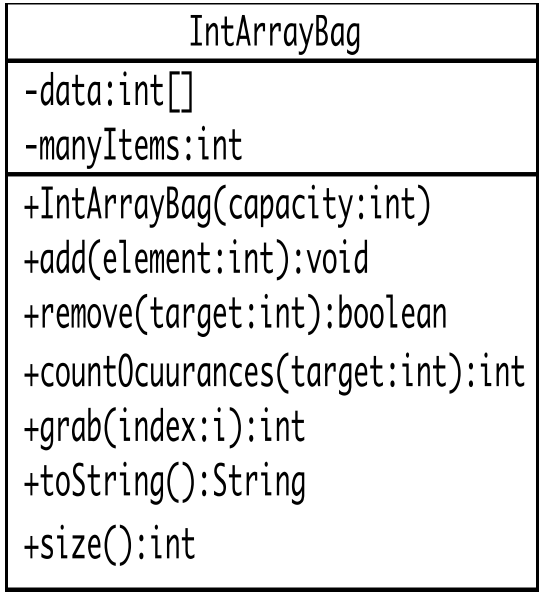
Review: IntArrayBag instance methods

Modifier and Type	Method and Description
void	add(int element) Add a new element to this bag.
void	addAll(IntArrayBag addend) Add the contents of another bag to this bag.
void	addMany(int... elements) Add new elements to this bag.
IntArrayBag	clone()Generate a copy of this bag.
int	countOccurrences(int target)Accessor method to count the number of occurrences of a particular element in this bag.
void	**ensureCapacity**(int minimumCapacity) Change the current capacity of this bag.
int	**getCapacity**() Accessor method to get the current capacity of this bag.
boolean	remove(int target) Remove one copy of a specified element from this bag.
int	size()Determine the number of elements in this bag.
void	**trimToSize**() Reduce the current capacity of this bag to its actual size (i.e., the number of elements it contains).

Difference between instance methods for `IntArrayBag` and `IntLinkedBag`

- The following methods are not included in the `IntLinkedBag` because there is no limit on the capacity of the linked list:
  - `ensureCapacity`
  - `getCapacity`
  - `trimToSize`

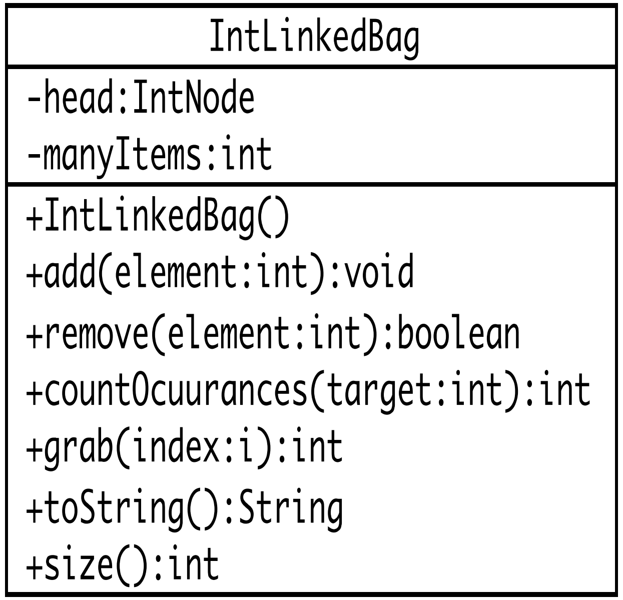
Contrast different Bag implementations



IntArrayBag

-data:int[]
-manyItems:int

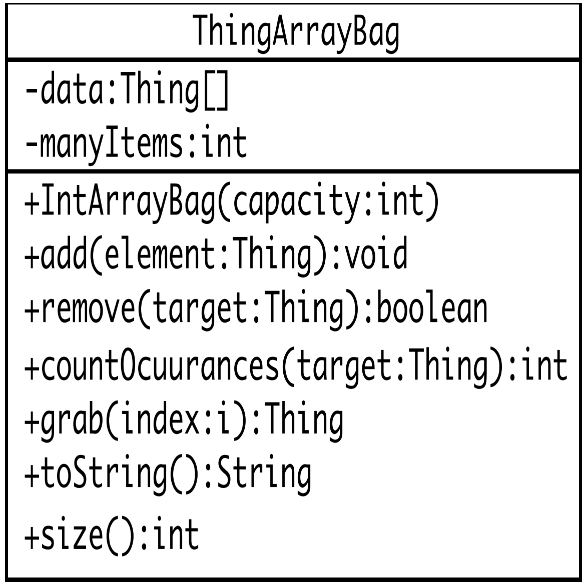
+IntArrayBag(capacity:int)
+add(element:int):void
+remove(target:int):boolean
+countOcuurances(target:int):int
+grab(index:i):int
+toString():String
+size():int



IntLinkedBag

-head:IntNode
-manyItems:int

+IntLinkedBag()
+add(element:int):void
+remove(element:int):boolean
+countOcuurances(target:int):int
+grab(index:i):int
+toString():String
+size():int



ThingArrayBag

-data:Thing[]
-manyItems:int

+IntArrayBag(capacity:int)
+add(element:Thing):void
+remove(target:Thing):boolean
+countOcuurances(target:Thing):int
+grab(index:i):Thing
+toString():String
+size():int

IntLinkedBag Implementation

- **Constructor:** `public IntLinkedBag()`
  - No input is needed because we do not specify capacity
  - Creates an empty list with `head = null` and `manyItems = 0`

- `void add(int element)`**:**
  - adds the input value at the head of the list.

```
public void add(int element){
    this.head = new IntNode(element,head);
    this.manyItems++
}
```

Using `Bag` collections from the Driver

```
IntArrayBag arrayBag  = new IntArrayBag(10);
arrayBag.add(10);
arrayBag.add(20);
arrayBag.add(30);
System.out.println(arrayBag); // [10,20,30]

IntLinkedBag linkedBag = new IntLinkedBag();
linkedBag.add(10);
linkedBag.add(20);
linkedBag.add(30);
System.out.println(linkedBag); //[30,20,10]
```

IntLinkedBag Implementation: countOccurances

```
public int countOccurances(int element){
    int count = 0;
    IntNode cursor = head;
    while (cursor != null){
        if (cursor.getData() == element)
            count++;
        cursor = cursor.getLink();
    }
    return count;
}
```

IntLinkedBag Implementation: remove

- Algorithm to remove a target value from a linked list bag:
  - Get a reference to the node that contains the target value (use IntNode.listSearch)
  - Replace the contents of the node with the value at the head
    - contrast to removing an element from the array bag?
  - Change head to head.getLink()

```java
public boolean remove(int target){
       IntNode targetNode = IntNode.listSearch(this.head, target);
       if (targetNode == null)
              return false;
       else{
              int headData = this.head.getData();
              targetNode.setData(headData);
              manyItems--;
              this.head = this.head.getLink();
              return true;
       }
}
```

What is the output of the following code

```
IntArrayBag arrayBag  = new IntArrayBag(10);
arrayBag.add(10);
arrayBag.add(20);
arrayBag.add(30);
arrayBag.add(40);
arrayBag.add(50);



System.out.println(arrayBag);
System.out.println(arrayBag.countOccurances(30));

arrayBag.remove(20);
System.out.println(arrayBag);
```

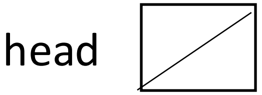
```
IntLinkedBag linkedBag = new IntLinkedBag();
linkedBag.add(10);
linkedBag.add(20);
linkedBag.add(30);
linkedBag.add(40);
linkedBag.add(50);



System.out.println(linkedBag);
System.out.println(linkedBag.countOccurances(30));

linkedBag.remove(20);
System.out.println(linkedBag);
```

`NullPointerException` is common when using linked lists

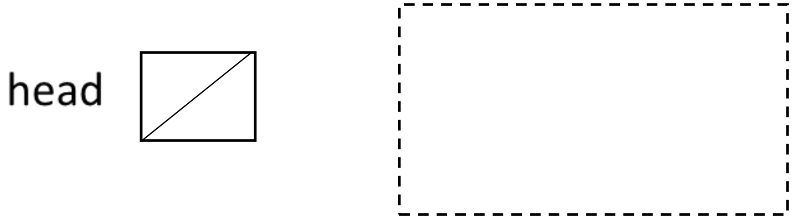
- An exception is an error
- This error often happens when you are programming with linked lists
- A **NullPointerException** happens when dereferencing a null pointer (i.e., the referencing operator (.) used with a null object).
    - Dereferencing a pointer means **getting the value that is stored in the memory location pointed by the pointer**.

Example of NullPointerException

```
IntNode head = null;
```



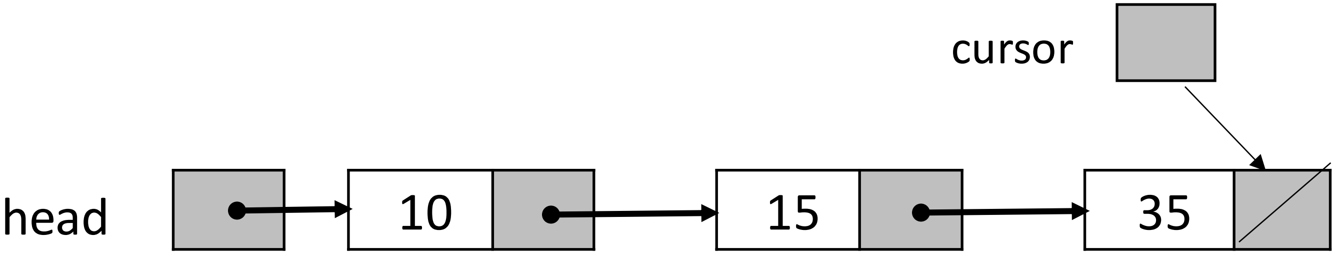
```
head.getData();
```



```
// as if you are using
null.getData();
```

Going Past the End of the List

```
IntNode cursor = head;
// dereferences null at the end of the list
while ((cursor.getData() != 99) && (cursor != null)){
   System.out.println(cursor.getData());
   cursor = cursor.getLink();
}
```



How to Find the Cause?

- Eclipse will tell you which line is causing the problem

```
Exception in thread "main" java.lang.NullPointerException
at IntLinkedBag.yourMethod(IntLinkedBag.java:74)
at IntLinkedBag.main(IntLinkedBag.java:106)
```

- Here, the error occurred on line 74 of the method `yourMethod()`
- Trace your code and find the place where you are dereferencing null
- If you have a compound test, test for `cursor != null` before trying to access the cursor’s instance variables/methods

```
while ((cursor != null) && (cursor.getData() != 99))
```

Test Cases

- What cases are most likely to cause problems?
  - The list is empty
  - The item you are looking for is not in the list
  - The item you are looking for is at the end of the list
  - When you are dealing with the head of the list

LinkedList of Objects

`BookLinkedBag`

BookNode

```
public class BookNode    {
   private Book data;
   private BookNode link;

   public BookNode(Book data, BookNode link)  {
      this.data = data;
      this.link = link;
   }
```

Getters and Setters

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

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

public BookNode getLink()   {
   return link;
}

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

BookLinkedBag

- Convert all IntNode → BookNode
- Convert int data to Book data as appropriate
- Convert comparison operators (<, >, <=, >=, ==, !=) to calls to equals() or compareTo()

```
public class BookLinkedList {
   private BookNode head;
   private int manyItems;

public BookLinkedList()   {
   head = null;
   this.manyItems = 0;
}
```

add

```
public void add(Book element)    {
   this.head = new BookNode(element, head);
   this.manyItems++;
}
```

search

```
public Book search(Book target)     {

   BookNode cursor = head;
   while (cursor != null){
      if (cursor.getData().equals(target))
         return cursor.getData();
      cursor = cursor.getLink();
   }
   return null;
}
```

Implementing an Itertor for `IntLinkedBag`

Iterator is an object that is used to iterate — that is, step through - a data structure of other objects (or primitives). You can let multiple users iterate over your data structure and they can't get in each others way because they are using separate iterators (objects) to go through your data structure

Inner iterator class for IntLinkedBag

```
public class IntLinkedBag implements Iterable<Integer>{

     public IntNode head;

     public class IntLinkedIterator implements Iterator<Integer>{
            //has access to the private head
            private IntNode current = head;
            //No constructor
            @Override
            public boolean hasNext() {}

            @Override
            public boolean next() {}
      }


      @Override
      public Iterator<Integer> iterator() {
            //no input to the iterator constructor
            return new IntLinkedIterator();
      }



}
```

Implementation of class IntLinkedBagIterator

```
public class IntLinkedIterator implements Iterator<Integer>{
        private IntNode current = head;
        @Override
        public boolean hasNext() {
                if (current == null)
                        return false;
                return true;
        }

        @Override
        public Integer next() {
                int data = this.current.getData();
                current = current.getLink();
                return data;
        }
}
```

Using `IntLinkedBagIterator`

```
IntLinkedBag bag1 = new IntLinkedBag();

Iterator<Integer> itr = bag1.iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

Usinf the Enhanced `for-loop` with `IntLinkedBag`

```
IntLinkedBag bag2 = new IntLinkedBag();
```

for each int in bag 2

```
for (int i: bag2){
    System.out.println(i);
}
```

Returning a Linked List from a Method

Returning a Linked List as an Output from a Method

- `listCopy` is a static method that makes a copy of a linked list, returning a reference to the newly created copy

```
public static IntNode listCopy (IntNode source)
```

- to call the method:

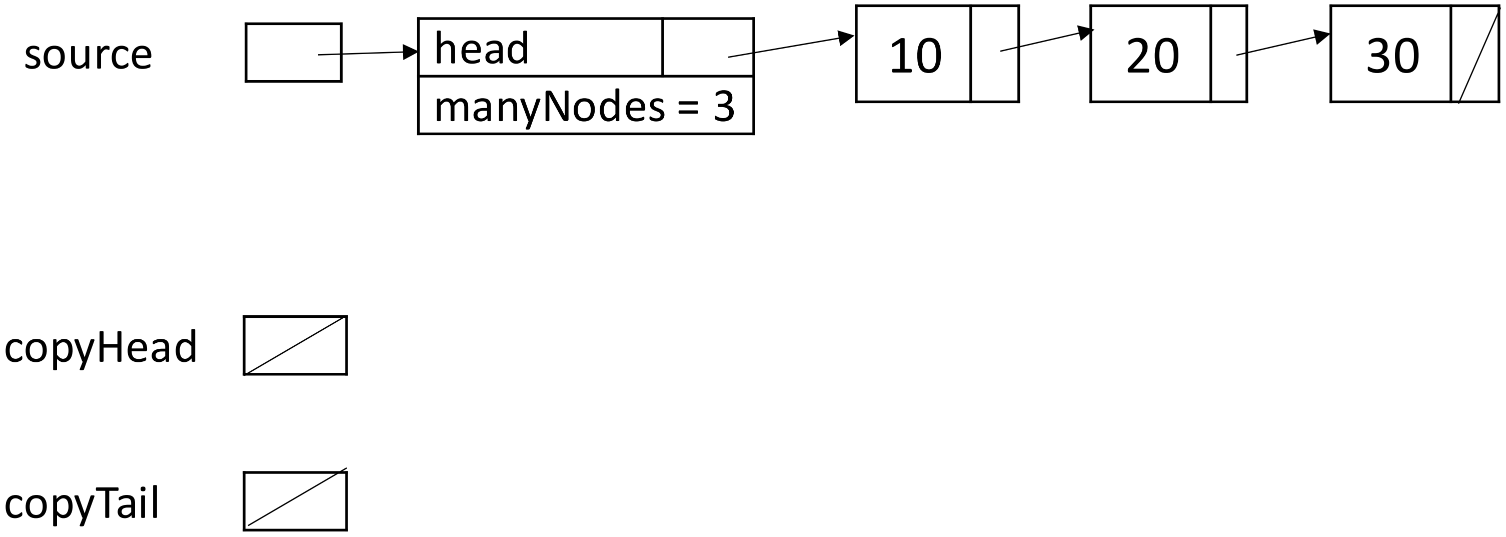
```
IntNode copyList = IntNode.listCopy(list1);
```

Returning a linked list as an output from a method

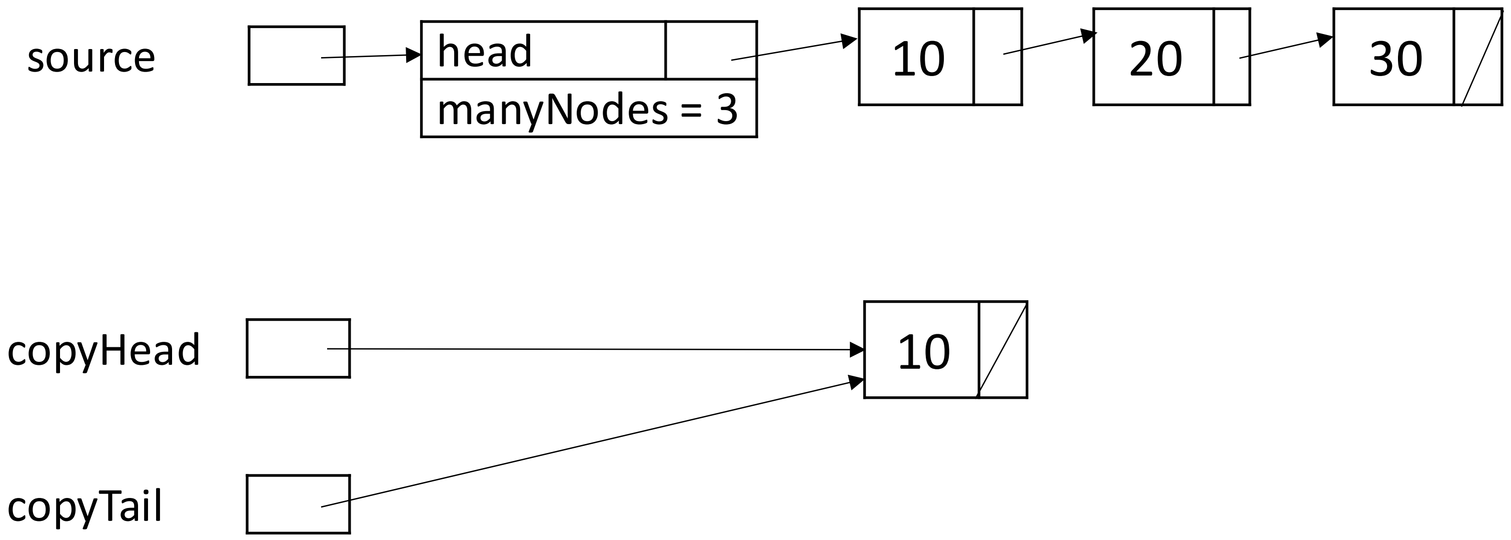
listCopy is a method that makes a copy of a linked list, returning the head reference for the newly created copy.

```
public static IntNode listCopy(IntNode source){
      IntNode copyHead;
      IntNode copyTail;
      if (source == null)
            return null;
      //make the first node in the new list
      copyHead  = new IntNode(source.getData(),null);
      copyTail  = copyHead;
      IntNode cursor = source;
      while (cursor.getLink() !=null){
            cursor = cursor.getLink();
            copyTail.addNodeAfter(cursor.getData());
            copyTail = copyTail.getLink();
      }
      return copyHead;
}
```

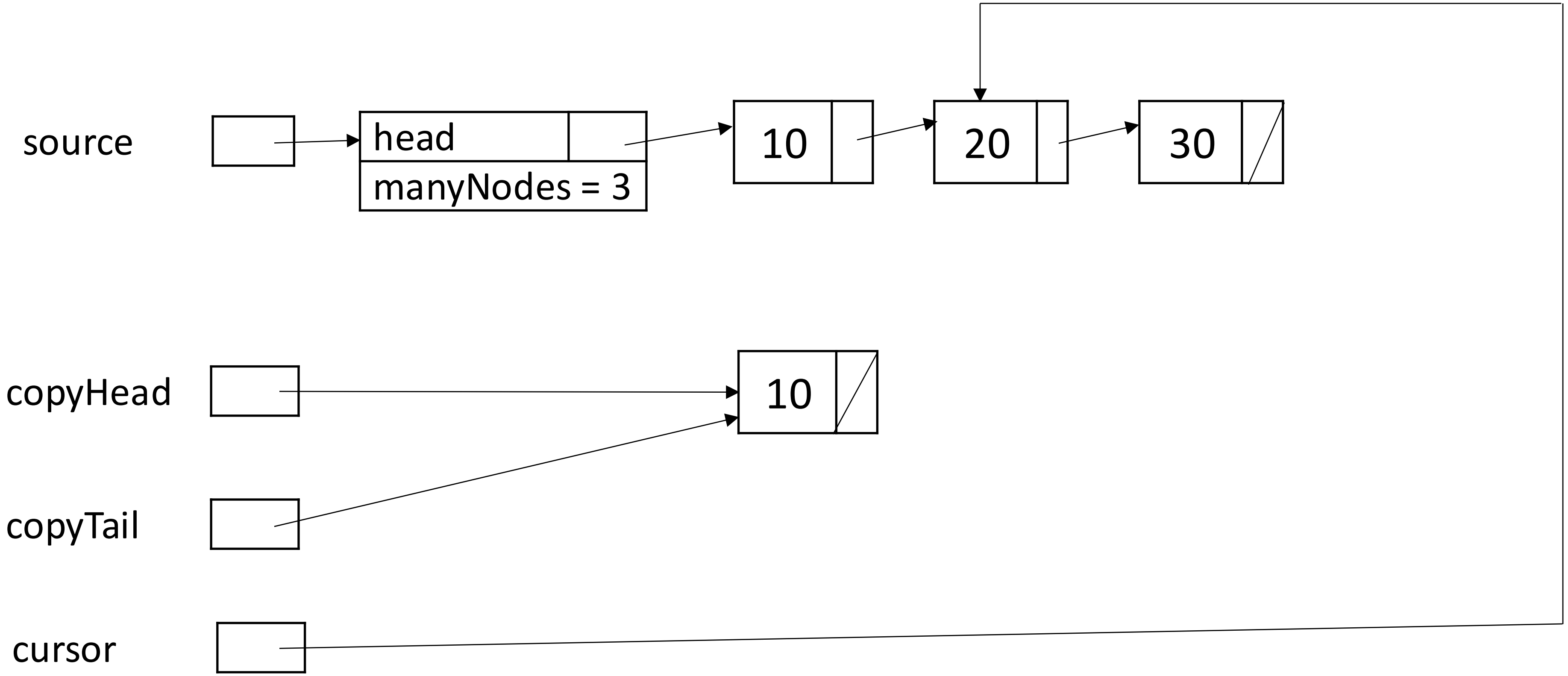
Initial Values



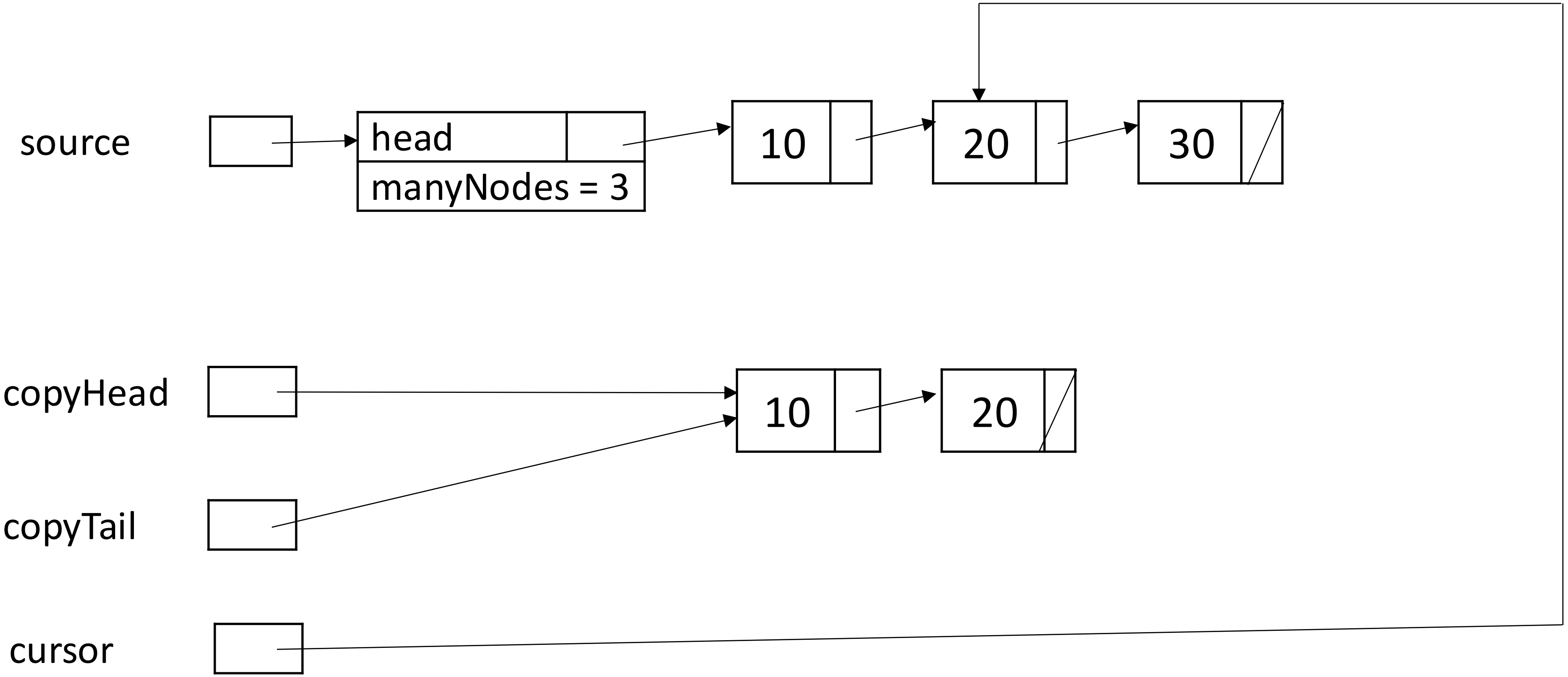
`head` is a special case



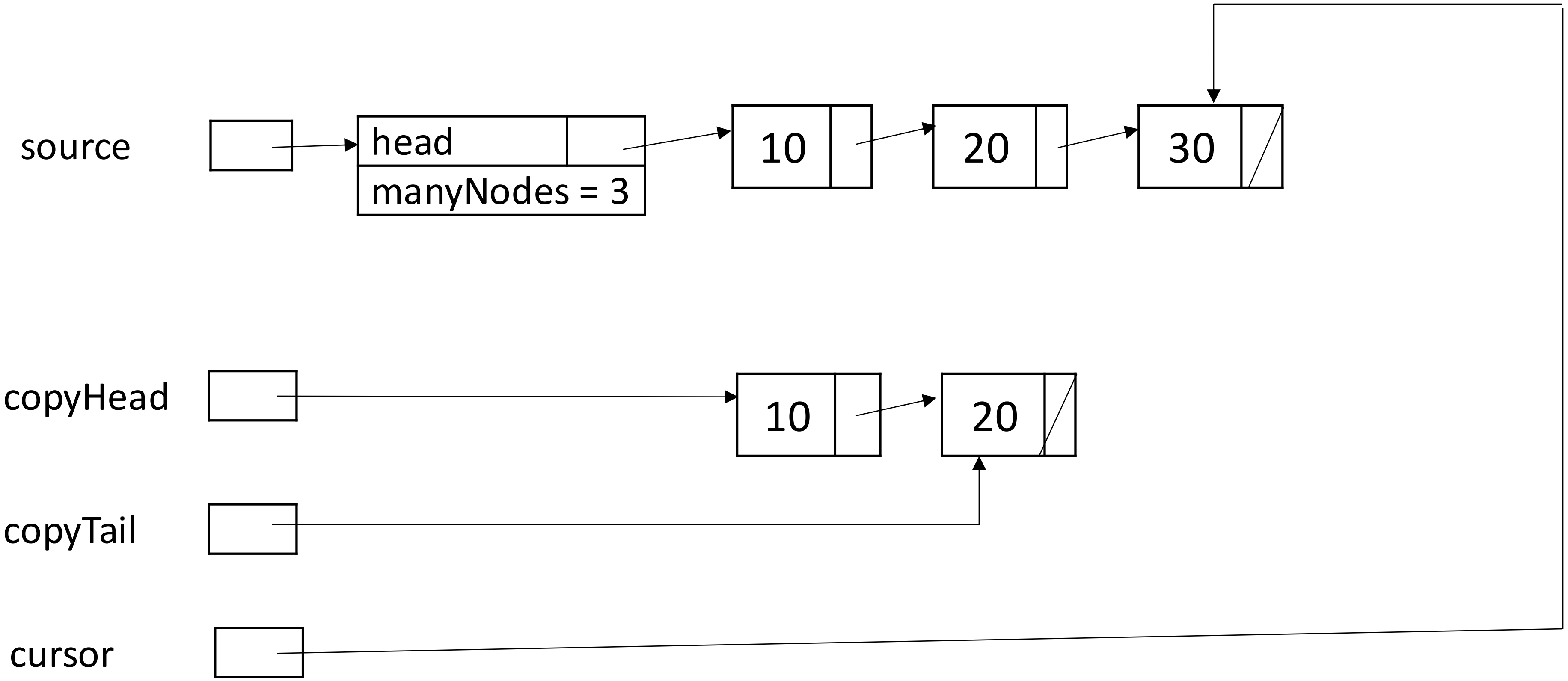
Initialize a `cursor` for `source`



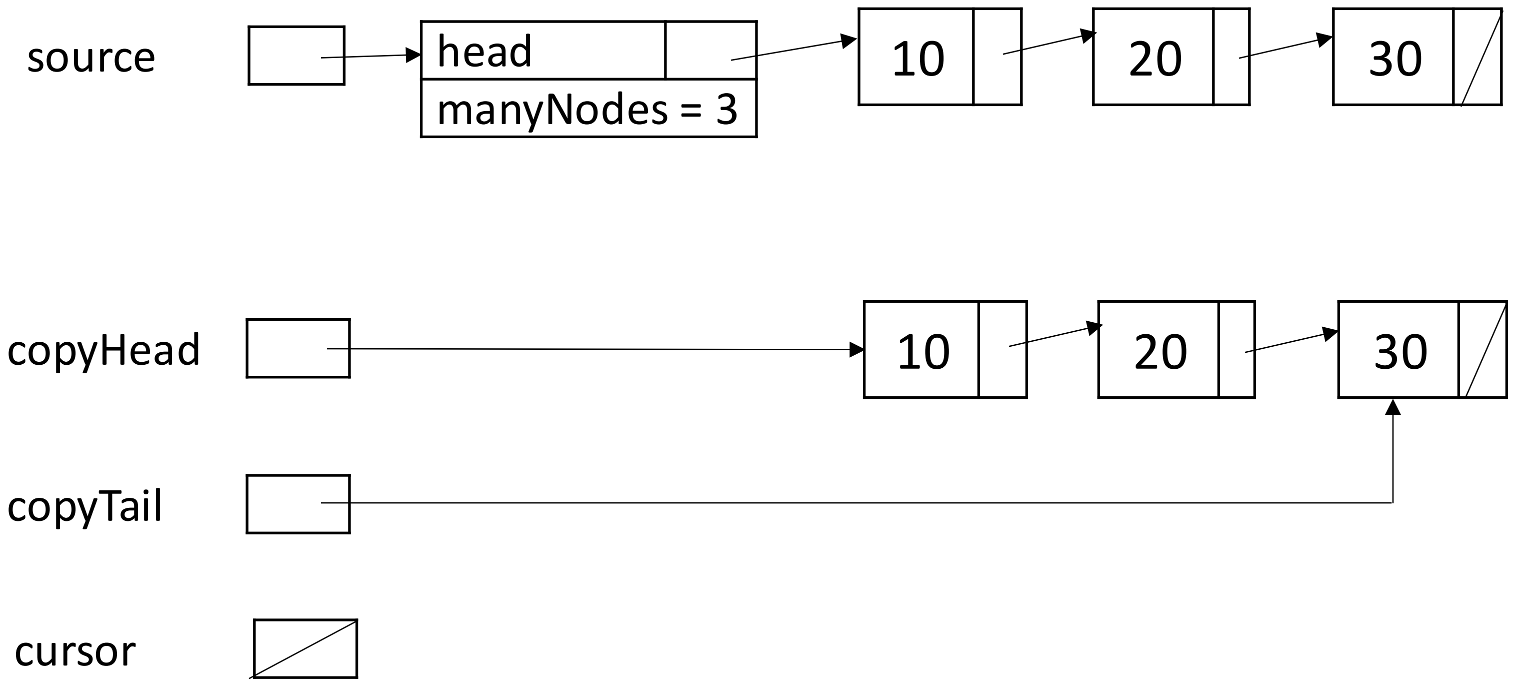
create a copy from the node at cursor and add it after `copyTail`



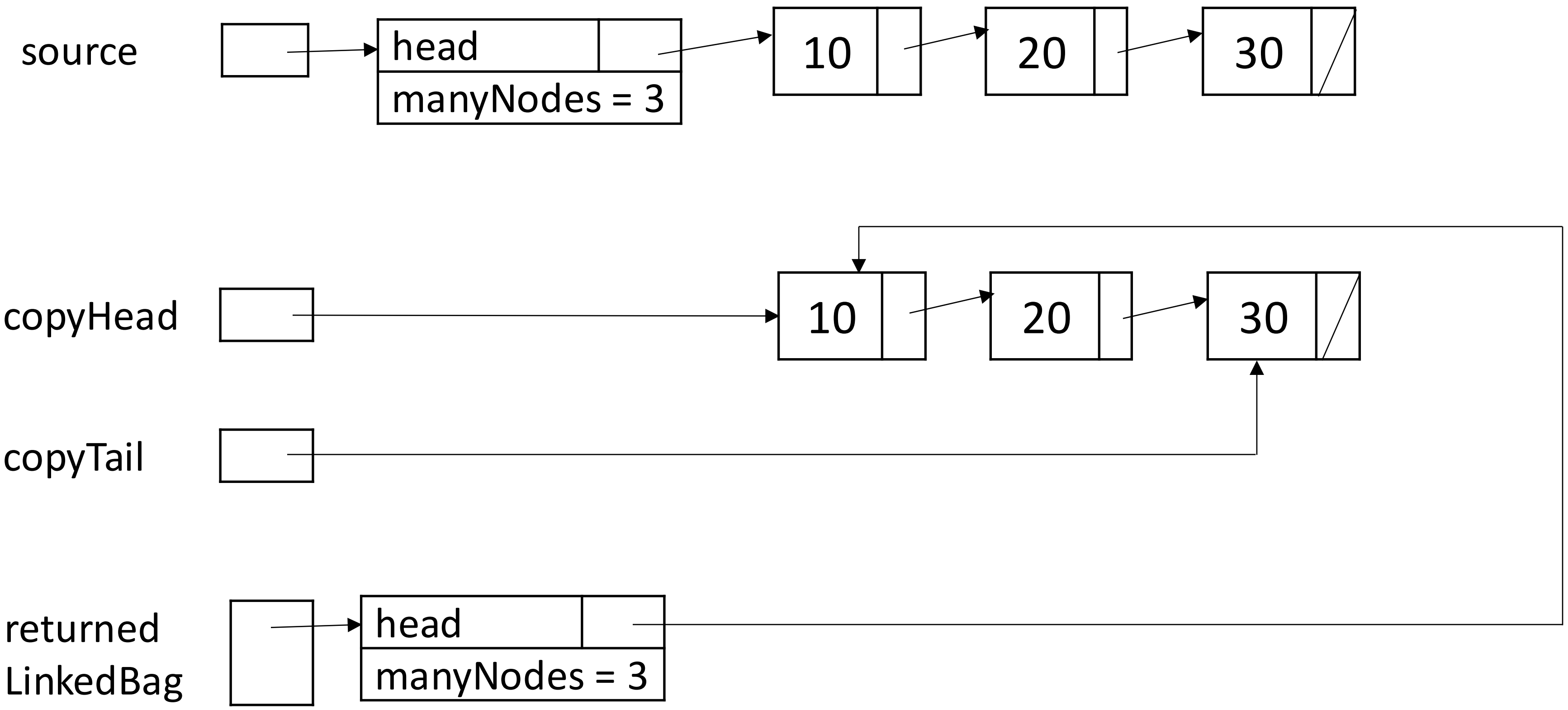
advance `copyTail` and `cursor`



repeat until `cursor == null`



Create an `IntLinkedBag` Object



Pseudocode for `listCopy()`

- Create a reference to the head of the copy
- Create a reference to the tail of the copy
- If the original list is not empty
    - Make a copy of the head node
    - Make both `copyHead` and copyTail point to this single node
    - Set `cursor` to the second node in the original list
    - For each node in the original list (starting with the second node)
        - Make a copy of the node
        - Set `copyTail.getLink()` to this new node
        - Advance `copyTail`
        - Advance `cursor`