Main's book Chapter 4
Linked Structures
IntNode class
Linked Bag class
Linear Data Structures (1)

An array of integers
Contiguous memory locations

A linked list of integers defined by linked nodes
Nodes can be anywhere in memory
Linear Data Structures (2)

An array of objects

A singly linked list of objects
Linear Data Structures (3)

A doubly linked list of objects

list \[ \rightarrow \]

\[ \text{obj0} \quad \text{obj1} \quad \text{obj2} \quad \text{obj3} \]

tail

9/21/2006
Nodes and links (1)

• A node is an object that contains a data part and a link part.

• The link part is a reference to another node which in turn contains a data part and a link part and so on.

• The end of the list is indicated by a null reference in the link part of the node.

• We can follow the links to access the data.
Nodes and links (2)

- A node is a self-referential structure
- We will first consider lists of integers and other primitive types
- Later we will generalize to lists of object type.
- First node of a list is often called the head
- Last node of a list is often called the tail
- It is easy to define nodes in Java
COSC 2006: Data Structures I

Design and Documentation
Implementation
IntNode class
The IntNode class (1)

Main uses this class to define nodes, instance methods that operate on nodes and static methods that operate on entire lists of nodes [ONLY MAIN USES THIS APPROACH]. Later we will discuss the more common approach.

public class IntNode
{
    private int data; // data part of node
    private IntNode link; // link to next node

    // instance methods that operate on nodes

    // static methods that operate on lists
}
The IntNode class (2)

Class design for the constructor and instance methods

```java
public class IntNode
{
    public IntNode(int data, IntNode link) {...}

    public int getData() {...}
    public IntNode getLink() {...}

    public void setData(int data) {...}
    public void setLink(IntNode link) {...}

    public void addNodeAfter(int element) {...}
    public void removeNodeAfter() {...}
    public String toString() {...}

    // static methods on next slide

    // we added this method
```
The IntNode class (3)

Class design for the static methods operating on lists of nodes

```java
public static IntNode listCopy(IntNode source) {...}
public static IntNode[] listCopyWithTail(IntNode source) {...}
public static int listLength(IntNode head) {...}
public static IntNode[] listPart(IntNode start, IntNode end) {...}
public static IntNode listPosition(IntNode head, int position) {...}
public static IntNode listSearch(IntNode head, int target) {...}
```
The list <1, 32, -5, 2> can be constructed one node at a time in reverse order using the following statements which always insert at the head of the list.

```java
IntNode head = new IntNode(2, null);
head = new IntNode(-5, head);
head = new IntNode(32, head);
head = new IntNode(1, head);
```
Constructing a specific list (2)

Here is another way to construct the list <1, 32, -5, 2> in left to right order by inserting at the tail. Here we assume that the private data fields ARE NOT accessible.

```
IntNode head = new IntNode(1, null);
head.setLink(new IntNode(32, null));
head.getLink().setLink(new IntNode(-5, null));
head.getLink().getLink().setLink(new IntNode(2, null));
```
Like previous slide but assuming that the private data fields are directly accessible.

```java
IntNode head = new IntNode(1, null);
head.link = new IntNode(32, null);
head.link.link = new IntNode(-5, null);
head.link.link.link = new IntNode(2, null);
```
The list <1, 32, -5, 2> can be constructed in order using the single statement.

```
IntNode head =
    new IntNode(1,
    new IntNode(32,
    new IntNode(-5,
    new IntNode(2,null))));
```

This technique is useful for creating simple lists to be used in testing the IntNode class.
The toString method

- We have added `toString` since it is useful for displaying a list when testing.
- Implementation given later
Adding node at head of list (1-4)

We want to create a new node and insert it here.

head

1

32

-5

2
Adding node at head of list (2-4)

newNode

Create new node linked to head node

```java
IntNode newNode = new IntNode(99, head);
```
Adding node at head of list (3-4)

Create new node linked to head node

```java
IntNode newNode =
    new IntNode(99, head);
```

Make head reference the new node

```
head = newNode;
```
Adding node at head of list (4-4)

Create new node linked to head node

```
IntNode newNode =
    new IntNode(99, head);
```

Make head reference the new node

```
head = newNode;
```

This can be done in one statement:

```
head = new IntNode(99, head);
```
Adding node at head of list (2)

To add a new node at the head of a list use

```java
head = new IntNode(newData, head);
```

If head is null this also works to give a one-element list

```java
head = new IntNode(newData, null);
```

Construct a one-element list using

```java
IntNode head =
    new IntNode(newData, null);
```
Adding node after a node (1-6)

We want to create a new node and insert it here.

head

1 → 32 → -5 → 2
Adding node after a node (2-6)
Adding node after a node (3-6)

Create new node and set its link

IntNode newNode =
    new IntNode(element, selection.link);

newNode
Adding node after a node (4-6)

Create new node and set its link

\[
\text{IntNode } \text{newNode} = \\
\quad \text{new IntNode(element, selection.link)};
\]

Make selection.link reference the new node

\[
\text{selection.link} = \text{newNode};
\]
Adding node after a node (5-6)

Create new node and set its link

```java
IntNode newNode =
    new IntNode(element, selection.link);
```

Make selection.link reference the new node

```java
selection.link = newNode;
```

Can be done in one statement:

```java
selection.link =
    new IntNode(element, selection.link);
```
Adding node after a node (6-6)

Create new node and set its link

```java
IntNode newNode =
    new IntNode(element, selection.link);
```

Make `selection.link` reference the new node

```java
selection.link = newNode;
```

Can be done in one statement:

```java
selection.link =
    new IntNode(element, selection.link);
```
Adding node after tail

Assume that tail is a reference to the tail of the list
Create new node and set its link
IntNode newNode = new IntNode(element, null);

Make tail.link reference the new node
tail.link = newNode;

Simplify and update tail:
tail.link = new IntNode(element, null);
tail = tail.link;

same as general result with selection.link replaced by null
addNodeAfter method

- If selection is a reference to the node we want to add after then
  - `selection.link = new IntNode(element, selection.link);`
- Letting selection be "this" we get

```java
public void addNodeAfter(int element) {
    link = new IntNode(element, link);
}
```
We can now construct a list by first constructing a one-element list and then using addNodeAfter.
Following statements construct the list <1, 32, -5, 2>:

```
IntNode head = new IntNode(1, null); // <1>
head.addNodeAfter(2); // <1,2>
head.addNodeAfter(-5); // <1,-5,2>
head.addNodeAfter(32); // <1,32,-52>
```

Note that this is a strange way to construct a list since we are always adding after the head: we construct one-element list, then add remaining nodes in reverse order.
Same list <1, 32, -5, 2> can be constructed using the following statements.

```java
IntNode head = new IntNode(1, null);
head.addNodeAfter(32);
head.getLink().addNodeAfter(-5);
head.getLink().getLink().addNodeAfter(2);
```

Note that since we are outside the `IntNode` class we cannot use expressions like `head.link` since `link` is a private data field. Instead we must use `head.getLink()`
Testing addNodeAfter (1)

Testing addNodeAfter to add a node with data 15 after the head
For the list <10,20,30,40> the result is <10,15,20,30,40>

IntNode head =
    new IntNode(10, new IntNode(20,
    new IntNode(30, new IntNode(40, null))));

head.addNodeAfter(15);

System.out.println("After add = " + head);
Testing addNodeAfter to add a node with data 50 after the last node of a list. For the list <10,20,30,40> the result is <10,20,30,40,50>

```java
IntNode head =
    new IntNode(10, new IntNode(20,
        new IntNode(30, new IntNode(40, null))));

IntNode tail =
    head.getLink().getLink(),getLink();
tail.addNodeAfter(50);

System.out.println("After add = " + head);
```
Removing node from head (1-4)

We want to delete this node
Removing node from head (2-4)

head = head.getLink();

If inside the IntNode class we can use
head = head.link;
Removing node from head (3-4)

```java
head = head.getLink();
```

![Diagram showing a linked list with nodes 1, 32, -5, and 2, with the head pointing to the node with value 1. The orphan pointer is shown pointing to the next node in the list.]
Removing node from head (4-4)

```java
head = head.getLink();
```

Works even for a one-element list since `head.getLink()` has the value `null` in this case.
Removing node after a node (1-4)

We want to delete this node

head → 1 → 32 → -5 → 2
Removing node after a node (2-4)

We want to delete this node

head

selection

1

32

-5

2
Removing node after a node (3-4)

```
selection.link = selection.link.link;
```
Removing node after a node (4-4)

```
selection.link = selection.link.link;
```
**removeNodeAfter method**

If selection is a reference to the node we want to remove after then

- `selection.link = selection.link.link;`

Letting selection be "this" we get

```java
public void removeNodeAfter(int element) {
    link = link.link;
}
```
Another way: link = link.link;

Consider 3 nodes labelled a, b, c.

We can delete (remove) node b using

a.link = c;           // skip over b
a.link = b.link;      // since c is b.link
a.link = a.link.link; // since b is a.link
link = link.link;     // letting a be "this"
Testing removeNodeAfter (1)

Testing removeNodeAfter to remove the node after the head
For the list <10,20,30,40> the result is <10,30,40>

```
IntNode head =
    new IntNode(10, new IntNode(20, 
                  new IntNode(30, new IntNode(40, null))));

head.removeNodeAfter();

System.out.println("After remove = " + head);
```

Important Note: To remove the head node it is always necessary to use head = head.getLink();
Testing removeNodeAfter (2)

Testing removeNodeAfter to remove the last node of a list
For the list <10, 20, 30, 40> the result is <10, 30, 40>

```java
IntNode head =
    new IntNode(10, new IntNode(20,
        new IntNode(30, new IntNode(40, null))));

IntNode beforeTail = head.getLink().getLink();
beforeTail.removeNodeAfter();

System.out.println("After remove = " + head);
```

Note: If removeNodeAfter is applied to the last node of a list an exception is thrown.
Length of a list (1)

- The static listLength method needs to count the number of nodes in a list.

- To do this we need to traverse the list:
  - begin with a reference to the head and advance this reference until it reaches the null reference in the last node
  - see toString for another traversal example

- Each time we advance the reference we add 1 to a counter.
Length of a list (2)

- Pseudo-code for list traversal

```
cursor ← first node of list
WHILE cursor is not null DO
    count ← count + 1
    advance cursor to next node
END WHILE
```

- A for loop can also be used
Length of a list (3)

**Cursor is advanced using**  
\[ \text{cursor} = \text{cursor}.\text{link}; \]
Length of a list (4)

Using a while loop

```java
IntNode cursor = head;
int count = 0;
while (cursor != null)
{
    count++;
    cursor = cursor.link;
}
```

Using a for loop

```java
IntNode cursor;
int count = 0;
for (cursor = head; cursor != null;
    cursor = cursor.link)
    count++;
```
**static listLength method**

`listLength` returns number of nodes in a list of type `intNode`. 0 is returned if the list is empty. A for loop is used here but a while loop could also be used.

```java
public static int listLength(IntNode head) {
    int count = 0;
    for (IntNode cursor = head; cursor != null; cursor = cursor.link) {
        count++;
    }
    return count;
}
```
For loop traversal model for a linked list

for (IntNode cursor = head; cursor != null; cursor = cursor.link)
{
    // process the data in the node
    // referenced by cursor
}
General traversal model (2)

while loop traversal model for a linked list

```java
IntNode cursor = head;

while (cursor != null)
{
    // Process the data in the node referenced by cursor
    cursor = cursor.link; // advance to next node
}
```
Example: sum of integers

A traversal can be used to sum the integers in the nodes of an IntNode list

```java
IntNode cursor = head;
int sum = 0;

while (cursor != null)
{
   sum += cursor.getData();
   cursor = cursor.link; // advance to next node
}
```
Example: toString method

We have added `toString` since it is useful for displaying a list when testing.

```java
public String toString()
{
    StringBuffer s = new StringBuffer();
    s.append("IntNode[");
    IntNode current = this;
    while (current != null)
    {
        s.append(current.data);
        if (current.link != null) s.append(",");
        current = current.link;
    }
    s.append("]" );
    return s.toString();
}
```
Searching a list

- The static `listSearch` method searches a list for a given integer and returns a reference to the `IntNode` containing the data.
- If the data is not found then the method returns a null reference.
- The list traversal model can be used here.
static listSearch method

Search a list for a given data element called target and return a reference to the element if it is found else return null.

```java
public static IntNode listSearch(IntNode head, int target)
{
    for (IntNode cursor = head; cursor != null; cursor = cursor.link)
    {
        if (target == cursor.data)
        {
            return cursor;
        }
    }
    return null;
}
```
Testing listSearch

```java
IntNode head =
    new IntNode(10, new IntNode(20,
    new IntNode(30, new IntNode(40, null))));
System.out.println(IntNode.listSearch(head, 10));
System.out.println(IntNode.listSearch(head, 20));
System.out.println(IntNode.listSearch(head, 30));
System.out.println(IntNode.listSearch(head, 40));
System.out.println(IntNode.listSearch(head, 50));
```

Results displayed are
IntNode[10, 20, 30,40]
IntNode[20, 30, 40]
IntNode[30,40]
IntNode[40]
[ ]

these results show the list whose head is the node containing the data found
list search by position

Instead of searching for a given data element in a list and returning a reference to the node containing the data we can search for the node in a given position and return the reference to the node at that position.

We assume here that positions in a list are labelled beginning at 1 instead of 0
The `listPosition` method returns the reference to a node in the list given its position (1, 2, 3, ...). If there is no such position then `null` is returned and if the specified position is not positive then an exception is thrown.

```java
public static IntNode listPosition(
    IntNode head, int position)
{
    if (position <= 0)
        throw new IllegalArgumentException("...");
    IntNode cursor = head;
    for (int i = 1;
         (i < position) && (cursor != null); i++)
        cursor = cursor.link;
    return cursor;
}
```
Testing listPosition

```
IntNode head = new IntNode(10, new IntNode(20, 
    new IntNode(30, new IntNode(40, null))));
System.out.println(IntNode.listPosition(head,1));
System.out.println(IntNode.listPosition(head,2));
System.out.println(IntNode.listPosition(head,3));
System.out.println(IntNode.listPosition(head,4));
System.out.println(IntNode.listPosition(head,5));
```

Results displayed are
```
IntNode[10, 20, 30, 40]
IntNode[20, 30, 40]
IntNode[30, 40]
IntNode[40]
[]
```

these results show the list whose head is the node containing the data found

Non-positive position throw exception, positions > 5 return null
Copying a list (1)

Make a copy of the head of the source list. Set up two references copyHead and copyTail to it.

```java
IntNode copyHead = new IntNode(source.data, null);
IntNode copyTail = copyHead;
```
Now do the statements
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
Now do the same statements again
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;

```
source
1
32
-5
2
copyHead
1
32
-5
copyTail
```
Copying a list (4)

We are done now since `source.link` is null

Now do the same statements again

```java
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
```
Given a list make a copy

```java
public static IntNode listCopy(IntNode source) {
    if (source == null) return null;
    IntNode copyHead =
        new IntNode(source.data, null);
    IntNode copyTail = copyHead;

    while (source.link != null) {
        source = source.link; // advance
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link; // advance
    }

    return copyHead;
}
```
This version does not use addNodeAfter

```java
public static IntNode listCopy(IntNode source) {
    if (source == null) return null;
    IntNode copyHead =
        new IntNode(source.data, null);
    IntNode copyTail = copyHead;

    while (source.link != null) {
        source = source.link; // advance
        copyTail.link =
            new IntNode(source.data, null);
        copyTail = copyTail.link; // advance
    }
    return copyHead;
}
```
First test listCopy on the empty list

```java
IntNode head = null;
IntNode copy = IntNode.listCopy(head);
System.out.println("List = " + head);
System.out.println("List copy = " + copy);
```

Result is [ ] for both lists
Testing listCopy method (2)

Test listCopy on a non-empty list

```java
IntNode head = new IntNode(10, new IntNode(20, new IntNode(30, new IntNode(40, null))));
IntNode copy = IntNode.listCopy(head);

System.out.println("list = " + head);
System.out.println("copy = " + copy);

copy.setData(99); // <99,20,30,40>
copy.addNodeAfter(99); // <99,99,20,30,40>

// original is still <10,20,30,40>

System.out.println("list = " + head);
System.out.println("copy = " + copy);
```
Check cases

- source is an empty list
  - copy will be null

- source is a one element list
  - source.link will be null so the while loop will never be executed

- source is a two or more element list
  - while loop executes until source refers to the last node which is copied before exiting the loop
Sometimes it is useful to have a version of **listCopy** that returns the tail node

- example: appending one list at end of another

If we don't save a tail reference during the copy we will have to traverse the list again to find it in an operation like append

Our method needs to return both the head and tail references to the copy
Returning multiple values

- Java can only return one value from a method.
- We need to return two \texttt{IntNode} references.
- This can be done by returning an array with two elements, element 0 can be the head reference and element 1 can be the tail reference.
- Could also define a pair object and return it.
public static IntNode listCopyWithTail(IntNode source) {
    IntNode[] answer = new IntNode[2];
    if (source == null) return answer;
    IntNode copyHead =
        new IntNode(source.data, null);
    IntNode copyTail = copyHead;
    while (source.link != null) {
        source = source.link; // advance
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link; // advance
    }
    answer[0] = copyHead; answer[1] = copyTail;
    return answer;
}
Copying part of a list (1)

- This is analogous to using substring in the String class to make a string that is a substring of a given String.

- Here we want a method with prototype:
  ```java
  static IntNode[] listPart(IntNode start, IntNode end);
  ```

- Return value [0] is a reference to head of new list and [1] is a reference to the tail.
Copying part of a list (2)

Copy the nodes from start to end to obtain new list.
Given the two nodes start and end of a list make a copy of the sublist from start to end (inclusive) and return a reference to the head of the sublist and the tail of the sublist.

```java
public static IntNode[] listPart(IntNode start, IntNode end)
{
    IntNode copyHead, copyTail, cursor;
    IntNode[] answer = new IntNode[2];

    if (start == null)
        throw new IllegalArgumentException("...");
    if (end == null)
        throw new IllegalArgumentException("...");
```
static listPart method (2)

Now make first node of the new list and copy remaining nodes. Note the check for the end node in the while loop.

copyHead = new IntNode(start.data, null);
copyTail = copyHead;
cursor = start;
while (cursor != end)
{
    cursor = cursor.link;
    if (cursor == null) // end not found
        throw new IllegalArgumentException(".");
    copyTail.addNodeAfter(cursor.data);
    copyTail = copyTail.link;
}
answer[0] = copyHead; answer[1] = copyTail;
return answer;
testing listPart method

Test listPart on list <10,20,30,40> with part given by <20,30>

```java
IntNode list1 = new IntNode(10, new IntNode(20, 
    new IntNode(30, new IntNode(40, null))));

// Make start, end refer to 2nd and 3rd nodes
IntNode start = list1.getLink(); // data 20
IntNode end = start.getLink();   // data 30

IntNode[] part = IntNode.listPart(start, end);

System.out.println("head = " + part[0]);
System.out.println("tail = " + part[1]);
```

head = IntNode[20, 30]
tail = IntNode[30]
Exercise: concatenation

- Write a method with prototype
  ```java
  static IntNode concatenate(
      IntNode list1, IntNode list2);
  ```
  
- The method should use `listCopy` and `listCopyWithTail` to return a new list which is the concatenation of the two given lists `list1` and `list2`.

- `list1` and `list2` are unchanged.
Exercise: another toString

Consider the following toString method

```java
public String toString()
{
    return "IntNode[" + data + ", " + link + "]";
}
```

What output does this method produce?
IntNode class summary (1)

- Defines nodes for integer data
- Unconventional approach because the instance methods `addNodeAfter`, `removeNodeAfter` are part of the node class not a separate LinkedList class.
- This means they cannot be applied to an empty list so we must always construct the head of the list first.
IntNode class summary (2)

- Static methods in the IntNode class are "helper methods"
- Their purpose is to help develop linked implementations of other ADT's
- We will show how to do this for a linked implementation of the IntBag class called IntLinkedBag
The IntLinkedBag class uses IntNode for implementation
IntArrayBag and IntLinkedListBag

- We have implemented the Bag ADT of integers using an array: IntArrayBag
- We can also implement the Bag ADT of integers using our IntNode class to represent the bag as a linked list
IntLinkedBag objects

An IntLinkedBag object showing the number of nodes in the bag and a reference to the head of the list that contains these nodes

private IntNode head;
private int manyNodes;

IntLinkedBag object showing the list of elements

head -> 1 -> 32 -> -5 -> 2
manyNodes = 4
LinkedBag needs only a default constructor. There is no need for an initialCapacity argument or ensureCapacity or trimToSize methods since size is determined by the number of nodes.

```java
public class IntLinkedBag implements Cloneable {
    private IntNode head;
    private int manyNodes;

    public IntLinkedBag() {
        head = null;
        manyNodes = 0;
    }

    // methods go here
}
```
IntLinkedBag invariant

- The elements of the bag are stored in a linked list
- The head reference of the list is stored in the instance variable head
- The total number of elements in the list is stored in the instance variable manyNodes
the add and size methods

Add an element to an IntLinkedBag. Here we add at the head of the list.

```java
public void add(int element) {
    head = new IntNode(element, head);
    manyNodes++;
}

public int size() {
    return manyNodes;
}
```
the addAll method (1)

- Method prototype is
  - public void addAll(IntLinkedBag addend);

- We can implement this by first making a copy of `addend` using `listCopyWithTail`.

- Then we can link the copy with head of "this bag"

- Finally we can link head of "this bag" to the head of the addend copy
the addAll method (2)

```
IntNode[] copyInfo = IntNode.listCopyWithTail(addend.head);
```

This is the copy of addend bag made by listCopyWithTail
the addAll method (3)

copyInfo[1].setLink(head);
The `addAll` method (4)

```java
head = copyInfo[0];
```
the addAll method (5)

Final result is a list of 7 elements
Add the contents of another bag to this bag. We can do this using listCopyWithTail.

```java
public void addAll(IntLinkedBag addend) {
    if (addend == null)
        throw new IllegalArgumentException("...");
    if (addend.manyNodes > 0)
    {
        IntNode[] copyInfo
            IntNode.listCopyWithTail(addend.head);
        copyInfo[1].setLink(head);
        head = copyInfo[0]; // set "this" head
        manyNodes += addend.manyNodes;
    }
}
```
the addMany method

Add new elements to this bag.
We can do this add and the new Java 5 for loop

```java
public void addMany(int ... elements) {
    for (int i: elements)
        add(i);
}
```

The old way of doing this for loop is

```java
for (int k = 0; k < elements.length; k++)
    add(element[k]);
```
the union method

Add the contents of two bags and return the resulting bag. The input bags are unchanged and a new bag is created.

```java
public static IntLinkedBag union(
    IntLinkedBag b1, IntLinkedBag b2)
{
    if (b1 == null)
        throw new IllegalArgumentException("...");
    if (b2 == null)
        throw new IllegalArgumentException("...");
    IntLinkedBag answer = new IntLinkedBag();
    answer.addAll(b1);
    answer.addAll(b2);
    return answer;
}
```
Cloning an IntLinkedBag

```
answer = (IntLinkedBag) super.clone();
answer.head = IntNode.listCopy(head);
```
IntLinkedList clone method

Here we do a deep clone (primitive types)
(1) clone the instance data fields
(2) clone the list

```java
public IntLinkedBag clone()
{
    IntLinkedBag answer;
    try
    {
        answer = (IntLinkedBag) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        throw new RunTimeException("...");
    }
    answer.head = IntNode.listCopy(head);
    return answer;
}
```
The remove method (1)

If we want to remove node with data 42 we can first use listSearch to find this node (target is a reference to it);

```java
IntNode target = IntNode.listSearch(head, target);
```

We run into a problem: To remove the node referenced by target we would need a reference to the previous node so we could link the node containing 8 to the node containing 16. But listSearch doesn't give us the previous reference.
The remove method (2)

Since there is no ordering to the elements in a bag we can solve this problem with the following trick:

First copy the data (99) from the head node to the target node using

```
targetNode.setData(head.getData());
```
Now remove the head node to get rid of one of the nodes containing 99. Since the bag is not ordered it doesn't matter that this changes the order of nodes in the list.

```java
head = head.getLink();
```
The remove method (4)

Now remove the head node to get rid of one of the nodes containing 99. Since the bag is not ordered it doesn't matter that this changes the order of nodes in the list.

```java
head = head.getLink();
```
The remove method (5)

Remove a node from the bag list by copying the head data to the node to be removed and then remove the head. This works because there is no order on a bag

```java
public boolean remove(int target) {
    IntNode targetNode = IntNode.listSearch(head, target);
    if (targetNode == null)
        return false; // target not found
    else {
        targetNode.setData(head.getData());
        head = head.getLink(); // remove head
        manyNodes--;
        return true;
    }
}
```
countOccurrences method (1)

There are two ways to do it

- Make a list traversal and count the number of times the given data element occurs
- Use the listSearch method in the IntNode class in a loop to continually search for the next occurrence of the data element.

Main chooses to use listSearch in a loop
Use the listSearch method from the IntNode class in a loop to count the number of occurrences of a given element. This is Main's approach.

```java
public int countOccurrences(int target) {
    int answer = 0;
    IntNode cursor = IntNode.listSearch(head, target);
    while (cursor != null) {
        answer++;
        cursor = cursor.getLink();
        cursor = IntNode.listSearch(cursor, target);
    }
    return answer;
}
```
Here is an alternate method that doesn't use listSearch

```java
public int countOccurrences(int target) {
    int answer = 0;
    IntNode cursor = head;
    while (cursor != null) {
        if (target == cursor.getData())
            answer++;
        cursor = cursor.getLink();
    }
    return answer;
}
```
A similar method that uses a standard for loop

```java
public int countOccurrences(int target) {
    int answer = 0;
    for (IntNode cursor = head; current != null; 
        cursor = cursor.getLink())
    {
        if (target == current.getData())
            answer++;
    }
    return answer;
}
```
This method was not in IntArrayList
It's purpose is to randomly select a data element from the bag and return it
A random index can be generated using
\[
i = \lfloor \text{Math.random()} \times \text{manyNodes} \rfloor + 1; \quad // i=1,2,3,\ldots
\]
A better way would have been
\[
\text{Random random} = \text{new random();}
\text{i} = \text{random.nextInt(manyNodes)} + 1;
\]
There are two ways to find the list node at position \( i \) (\( i=1,2,3,... \))

- Use the listPosition method from the IntNode class
- Just use an iterator that counts as elements are traversed and stops at ith node.

Main uses the first approach. The listPosition method assumes that indices start at 1.
This method generates a random index in the range 1 to manyNodes. finds that node in the list using the listPosition method in the IntNode class, and returns the data value.

public int grab()
{
    if (manyNodes == 0)
    {
        throw new IllegalStateException("...");
    }
    int i = (int)(Math.random() * manyNodes) + 1;
    IntNode cursor =
        IntNode.listPosition(head, i);
    return cursor.getData();
}
Main does not include a toString method.

```java
public int toString()
{
    StringBuffer s = new StringBuffer();
    s.append("IntLinkedBag[");
    IntNode current = head;
    while (current != null)
    {
        s.append(current.getData());
        if (current.getLink() != null)
        {
            s.append(","alyzed current.getLink());
            current = current.getLink();
        }
    }
    s.append("]");
    return s.toString();
}
```
## Compare Bag implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array bag</th>
<th>Linked bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>$O(\text{capacity})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add</td>
<td>$O(1) / O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>b1.addAll(b2)</td>
<td>$O(n2) / O(n1+n2)$</td>
<td>$O(n2)$</td>
</tr>
<tr>
<td>clone</td>
<td>$O(\text{capacity})$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>countOccurrences</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>ensureCapacity</td>
<td>$O(\text{capacity})$</td>
<td>---</td>
</tr>
<tr>
<td>trimToSize</td>
<td>$O(n)$</td>
<td>---</td>
</tr>
<tr>
<td>union(b1,b2)</td>
<td>$O(\text{capacity1+capacity2})$</td>
<td>$O(n1+n2)$</td>
</tr>
<tr>
<td>size</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Since the bag is an unordered structure there is not much difference between the implementations. The main advantage is that we don't need to worry about the capacity for the linked implementation.
COSC 2006: Data Structures I

The IntLinkedSet class
Programming exercise
Modify the IntLinkedBag class to get an IntLinkedSet class

We did this for IntArrayBag

Make sure add and addAll don't allow duplicates to be added.

Also replace countOccurrences by a method with prototype

public boolean contains(int target);