Lab 6 Linked Bag and List Implementations

Goal

Having constructed array-based classes to represent Bags and Lists in the last lab, we are now going to repeat what we did last week, only using a chain of linked nodes to represent our Bag and Lists objects. For these exercises you may wish to refer to the lecture 3 pdf, which illustrates how chains of linked nodes can be manipulated.

Resources

- Chapter 3, and 14

Part A LinkedBag class

The Bag class was introduced in lab 2, and we wrote an ArrayBag class in lab5a. From an abstract point of view, a LinkedBag is just a Bag (container) we can store objects in. The following Javadoc defines many the exact same methods we wrote for the ArrayBag class. Today we will build a class according to these specifications, now using linked nodes to represent the objects in the bag.

Class LinkedBag <T>

A class of bags whose entries of type T are stored in a chain of linked nodes.

Constructor Summary
LinkedBag<T>()   Creates an empty bag whose initial capacity is 25.
LinkedBag<T>(int capacity)   Creates an empty bag having a given initial capacity.

Method Summary	
boolean	add(T newEntry) Adds a new entry to this bag.
void	clear() Removes all entries from this bag.
T	contains(T anEntry) Tests whether this bag contains a given entry.
int	getCurrentSize() Gets the current number of objects in this bag.
int	getFrequencyOf(T anEntry) Counts the number of times a given entry appears in this bag.
boolean	isEmpty() Sees whether this bag is empty.
boolean	isFull() Sees whether this bag is full.
Object	remove() Removes one unspecified entry from this bag, if possible.
boolean	remove(T anEntry) Removes one occurrence of a given entry from this bag.
Object[]	toArray() Retrieves all entries that are in this bag.
String	toString() Converts all the data in the bag into one big String

Pre-Lab Visualization of the Linked Chain-based Bag

The linked bag will use an internal chain of **Node** objects to maintain the items in the bag. As we saw in lecture, the items in the bag will be added to the chain at the beginning/front of the chain. We will still maintain a **mySize** instance variable to keep track of how many items the bag contains.

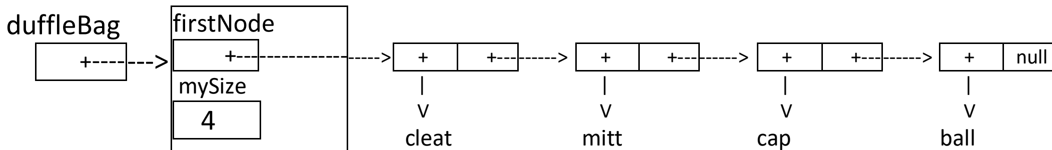
Here is some code that creates one of our simple linked bags:

```
LinkedBag duffleBag = new LinkedBag();
duffleBag.add("ball");  duffleBag.add("cap");
duffleBag.add("mitt");  duffleBag.add("cleat");
```

Notice we are not using type specifiers in angle brackets again. Like the ArrayBag we made last week this LinkedBag class uses Object as the data type, which is somewhat riskier but simpler to implement.

A (crude) memory map of the above LinkedBag object might look something like this

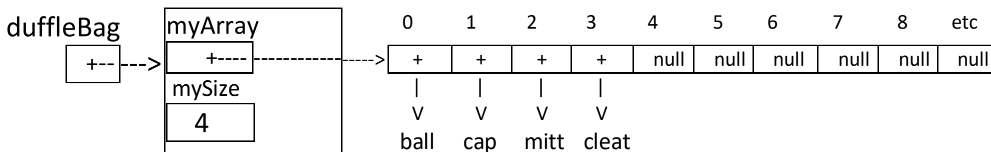
LinkedBag Diagram of the duffleBag object



In other words, **duffleBag** is a reference to the **LinkedBag** object, which has an internal **Node** reference and a **mySize** field. The **mySize** field indicates the total number of nodes in the bag. Because it's easier to add to the beginning of the chain we have chosen to always add new items at the front.

Compared to the **ArrayBag** diagram of **duffleBag** from last week (see below), the sequence of items in the linked bag appear reversed; however, logically speaking, both bags represent the same state of the Bag.

Array Bag Diagram (different implementation showing the exact same <u>logical</u> state as the previous diagram)



Using the above as a guide, draw pictures of the two LinkedBags that are formed by the following statements (note that the add method adds to the beginning of the chain and the remove method removes the last item added):

```
LinkedBag bag1 = new LinkedBag();
LinkedBag bag2 = new LinkedBag();
bag1.add("a"); bag1.add("b"); bag1.add ("c"); bag1.add ("d"); bag1.remove();
bag2.add(bag1.remove()); bag2.add("e");
```

bag1

bag2

Directed Lab Work

We will now start developing and testing the various methods for the LinkedBag class.The in class demonstration will show how the following methods are implemented:

constructors(2), isFull, isEmpty, getCurrentSize

Your task will now be to complete the following methods:

add, toString, getFrequencyOf, contains, remove, clear

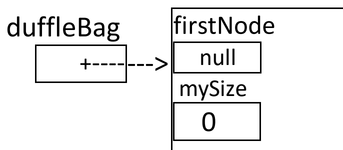
And, if time permits, 2 optional methods:

remove(anEntry), toArray

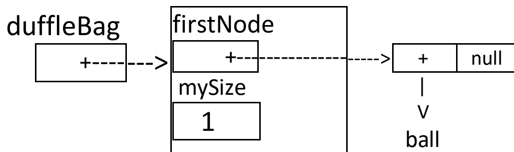
Problem 1a: the add method

When the LinkedBag is initially created, **firstNode** is null and **mySize** is 0.

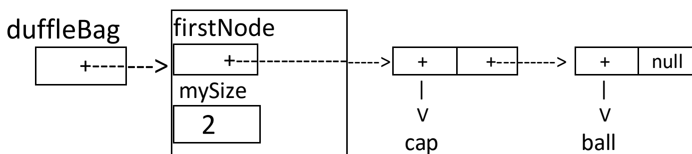
LinkedBag Diagram



After we execute the statement `duffleBag.add("ball");` the **duffleBag** object should undergo the following changes:

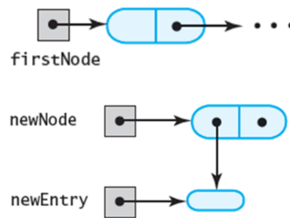


And after we execute the statment `duffleBag.add("cap");` the **duffleBag** object should undergo the following changes:

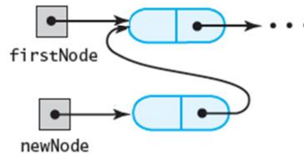


With these observations in mind, we can formulate a basic algorithm for the add method that will work whether the bag is empty or already has items stored in it. Our pictures below will assume there is already a chain of nodes. The add method should:

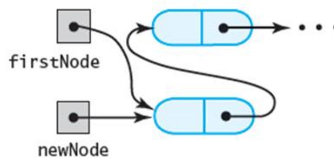
a) create **newNode** as a new **Node** with data set to **newEntry**. There is a constructor that will help you do this in the **Node** class. After this step the inside of your bag object will look something like this:



b) assign to the **next** member of **newNode** the value (address) stored in **firstNode**. (**newNode.next = firstNode**)
This means newNode will refer to the first node (if any) in the chain currently representing the bag.



c) assign to **firstNode** the value (address) stored in **newNode**. This means **firstNode** will now refer to the **newNode**, which now refers to the rest of the chain.

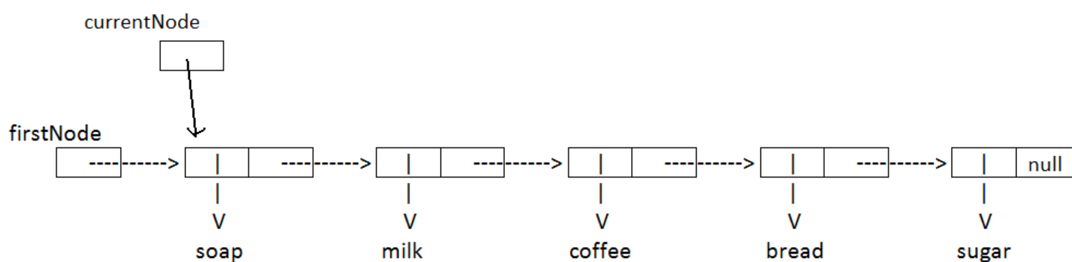


d) add one to **mySize** so we keep track of the correct number of nodes in the chain.

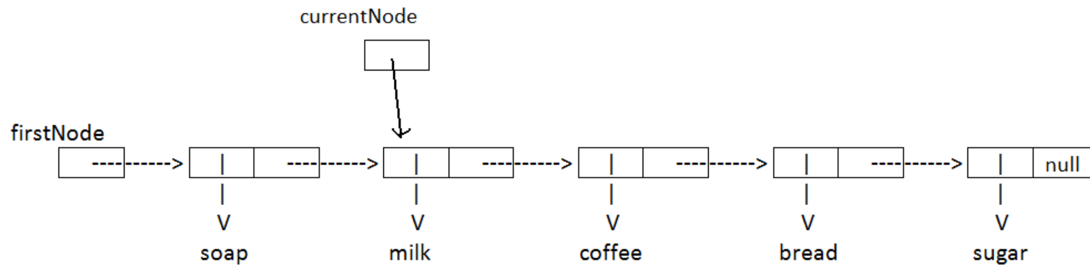
Recall that the add method **is supposed to return true if it was successful and false otherwise.** But, could a LinkedBag ever be full? It would take a long time! **Therefore, we are safe in always returning true from this add method.** Although that may seem strange, we are just satisfying the requirements put on us by the documentation that defined how a Bag is supposed to behave. Go ahead and write your modified definition for add now. Then test your add method in the Problem1 section of the lab6a test program.

Problem 1b: the toString method

The **toString** method will require some statements that will step a node reference variable **currentNode** from the first node to the last, so it can add the **data** in each node to a String variable **result**. This is called a traversal. The following diagrams illustrate how **currentNode** starts at the **firstNode** of the chain and advances to touch each **Node**.



The **currentNode** variable will start with the same value (address) as **firstNode**, allowing us to refer to "soap" using the expression **currentNode.data**;   Whenever we want to advance **currentNode** to the next **Node** we will use the expression **currentNode = currentNode.next;** This copies the **next** field of the node that **currentNode** refers to into **currentNode**, making **currentNode** now refer to (in this case) the node with "milk" in it.



We can continue to repeat this operation, referring to each node in the chain, until **currentNode** gets the value **null**. At that point, we have reached the end of the chain and must stop the loop.

To begin the **toString** method, we'll need to create a **String result** that initially assigned an empty string, or **"";** Then, at each node in the chain we'll add **currentNode.data** to our **result** String, with a space in quotes to prepare for the next item.

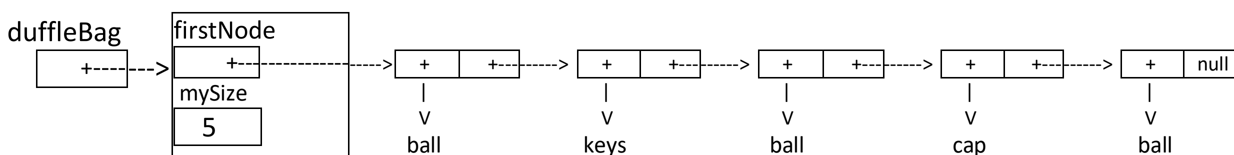
Here is the psuedocode for the basic **toString** method:

a) define **String result** as an empty string (that means assign **""** to it)
b) define **Node currentNode** and give it the initial value **firstNode** (as shown in the top diagram above)
c) as long as **currentNode** is not **null** (this is a while loop)
  i) add **currentNode.data** to the end of the **result** String
  ii) move **currentNode** to the next node in the chain (see explanation above)
d) after the loop completes, **return** the **result** variable

As an **extra challenge**, modify your toString so it inserts commas after each item except the last, and encloses the whole string in square brackets, for example: **[soap, milk, coffee, bread, sugar]**

Problem 2: the getFrequencyOf method

When the LinkedBag has a few items in it, we may wish to know how many times an item occurs in the bag. For example, given the following state of our bag,



The statement `duffleBag.getFrequencyOf("ball");` will return the value 3.

Now that you know how to traverse a linked chain of nodes, the getFrequency method will be fairly straightforward.

In order to determine the number of times **anEntry** occurs in the bag, we'll need to use a loop with an external counter. Here is an approach to solving this problem:

1) declare and set **numTimes = 0**
2) declare a **Node** reference variable **currentNode** and set equal to **firstNode**
3) make **currentNode** traverse over all the Nodes in the chain until it becomes null (while loop)
    a. If the object referred to by **currenNode** "equals" the **anEntry** object
        i. add one to **numTimes**
4) after the loop, return the value in **numTimes**

Write this method now and test it in the lab6a test program.

Problem 3: the contains method

This method returns true if **anEntry** is stored in the bag. It can be written in one of two ways.

Option 1 is to run through (traverse) the items in the chain using a loop. If there is ever an object in the bag that "equals" **anEntry**, return **true**. **After the loop**, just return **false** (meaning no match was found)

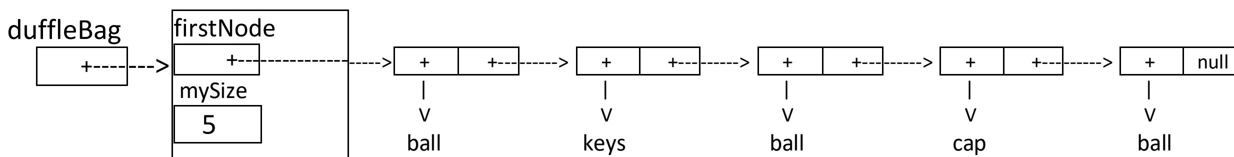
Although option 1 is fine, there is a simpler way. The advantage of writing numerous methods for a class is that some methods can be written in terms of others. Here is the alternative:

Option 2 is to just use **getFrequencyOf** You could simply return the logical expression that **getFrequencyOf** is greater than 0.

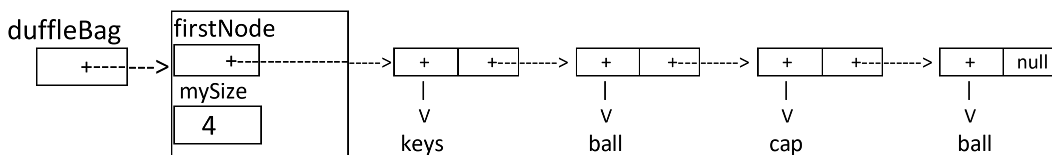
Write this method now and test it in the lab6a test program.

Problem 4: the remove method

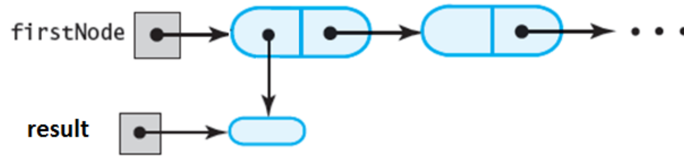
Suppose our bag at some point looks like the following:



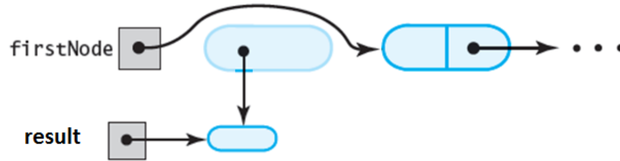
In the **ArrayBag** class, we found it easiest to remove the last item in the array. For the **LinkedBag** class it turns out to be much easier to remove the **first** node of the linked chain of nodes. So, the statement `duffleBag.remove();` would make the bag now look like:



Notice the changes that have taken place on the **dufflebag** with the single **remove** statement. Remember, your method has to **return** the entry being removed. To accomplish a remove, we need to create a variable **result** to refer to the data in the first node, as shown in this diagram:



Then advance **firstNode** so it refers to the next node in the chain, like so:



Finally, we simply return the value of result, the data in the node being deleted. Well, it's not quite that simple, because if the bag is empty, if we do not have any nodes, then we must return a result of null.

Here's the psuedocode for the **remove** method:

a) define **Object result** and initialize it to **null** (in case the bag has no nodes)

b) if **mySize** is greater than 0 (we have something real to return besides null)

- i) assign to **result** a reference to the data in the node referred to by **firstNode**
- ii) assign to **firstNode** the value (address) in the **next** field of the first node in the chain
  (this makes **firstNode** point to the second node)
- iii) decrement the value in **mySize** by one

c) after the if statement, **return** the value in **result** (either null, or a reference to the data field of the first node.

Write this method now and test it in the lab6a test program.

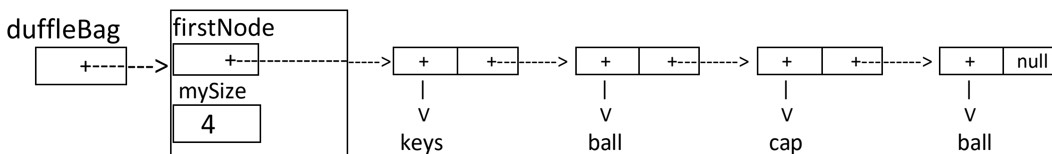
Problem 5: the clear method

The clear method removes all of the items in the array. It could be written in at least two different ways. Come up with an approach and code it up and test it in problem 5 in the test program.

Problem 6: method remove(anEntry)

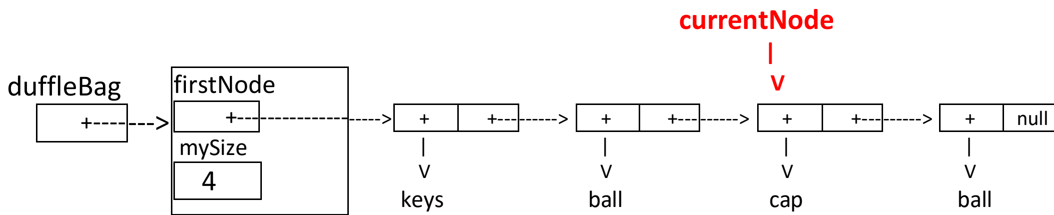
There is another method that removes items from the bag. This one requires the user to specify which item to remove. We can solve this one similarly to how we solved it last week. First, we'll locate the node containing the item we wish to remove, then we'll copy the firstNode's data over that node's data. Finally, we'll remove the first node.

For example, given the following duffleBag object:

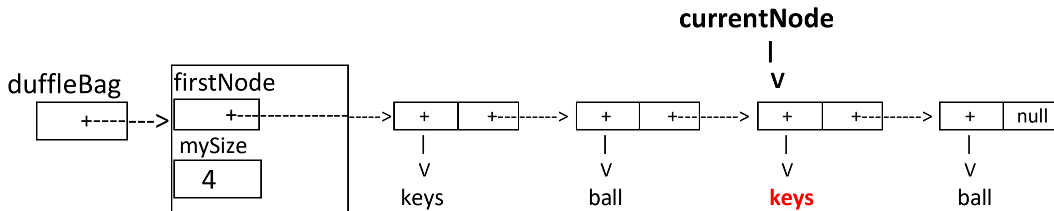


If we execute the statement **duffleBag.remove("cap")**;

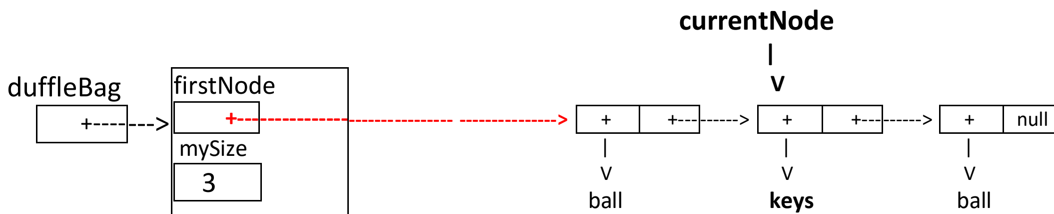
We will first set up a **currentNode** reference to locate the Node in the chain where **anEntry** is stored (if at all):



Then, we'll copy **firstNode.data** into the node where currentNode is referring to.



Finally, we'll remove the first Node, similar to how remove() works (you could actually invoke **remove** to do this!)



Note: we also have to return true or false indicating whether we actually removed an object or not. This will require a little extra logic to account for. Code up your method and test it now in problem 7 in lab6a.

Problem 7: Converting the LinkedBag to a generic class

We have been using **Object** as the data type for all of the objects added to the bag, and for the base type of **myArray**. Although this approach works as a means to create a working **LinkedBag** class, it does have some defects. The problem occurs when we accidently insert something unintended into our Bag, such as an array of String rather than a single String object. These mistakes would go unnoticed by the compiler and not appear until run time, where they are usually much harder to find. In order to allow the compiler to flag incorrect insertions into our **LinkedBag**, we will now make a simple modification that allows us to specify the data type our LinkedBag can hold.

The steps to do this are: 1) add **<T>** after the class declaration, and 2) Change all occurences of **Object** to **T** in the remaining code.

Since no arrays are being created, there is no need to use any special warning suppression in the constructor like we did for the ArrayBag class.

Problem 8: the toArray method

The toArray method is defined incorrectly in our text—it does not need to be cast into an array of T since Java does not remember what T is for arrays—although the basic idea is correct. The steps are as follows: 1) make a new array of Object called result that has space for mySize items; 2) write a for loop that copies all the data items from the linked nodes over into the result array (you may need to cast the data references as Object); and 3) return the result array. Write your toArray method and then test it in the lab6a program. ***Note that the toArray method should return Object[] not T[].***

Problem 9: the union method (do Problems 7 & 8 first)

The **union** method is commented out in the **LinkedBag** class file because it is written with a parameter of type **BagInterface<T>**. You won't be able to uncomment this method until you complete problems 7 and 8.

Since the union method takes a second BagInterface<T> as a parameter and returns a new BagInterface<T> object containing the contents of both bags, this is a good time to review the purpose of interfaces.

Interfaces help us manage and define classes that perform similar behaviors. The BagInterface is a like contract that specifies the methods a class must contain to be considered a Bag.

To make our **LinkedBag** fully functional, we now add the phrase **<u>implements BagInterface<T></u>** to the end of the first line of the class file: **public class LinkedBag<T> implements BagInterface<T>** this tells the compiler that our LinkedBag satisfies the BagInterface contract (and the compiler will make sure it does).

In order for the union method to work effectively it should be able to combine **ArrayBag** objects with **LinkedBag** objects (they are both Bags after all). So we use a **BagInterface** parameter because that can refer to either an **ArrayBag** or **LinkedBag** object.

Also, if we don't care if an object variable refers to either an ArrayBag or a LinkedBag, we could declare it as a BagInterface variable.

To complete this problem,

a) add **implements BagInterface<T>** to the LinkedBag class definition so that LinkedBag objects can be referred to with BagInterface variables
b) open the BagInterface file and uncomment the union method prototype
c) define the union method in LinkedBag. This method will
    1) create a new bag called result
    2) (non-destructively) copy all the items from anotherBag into result
       (hint: first dump all the items in anotherBag into a temp array using toArray)
    3) (non-destructively) copy all the items from this bag (the host bag) into result
       (hint: make a Node currentNode variable to traverse the Nodes in this bag)
    4) return result

Write this method now and test it in Problem 9 of lab6a

Part B LinkedList class

The List class was introduced in lab 4 and an ArrayList was developed in lab 5. A List is a container that stores objects in a particular order, and allows deleting and adding items at any point in the list. The following Javadoc defines the methods for the LinkedList, which are the same as the ArrayList. Today we will build a class according to these specifications. Although we will use the generic type specifier T, we will ignore the use of intefaces to keep things simple.

Class LinkedList <T>

A class of lists whose entries of type T are stored in a chain of linked nodes of <T>.

Constructor Summary
LinkedList<T>() Creates an empty list whose initial capacity is 25.

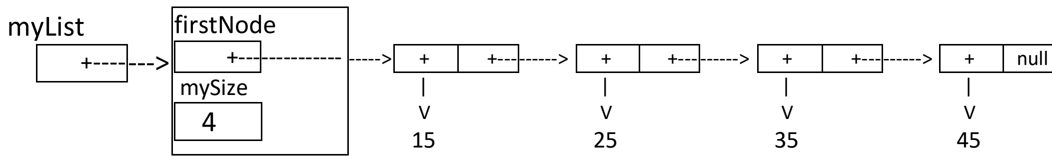
Method Summary	
boolean	**add**(int newPosition, T newEntry) Adds a new entry at a specified position within this list.
void	**add**(T newEntry) Adds a new entry to the end of this list.
void	**clear**() Removes all entries from this list.
boolean	**contains**(T anEntry) Sees whether this list contains a given entry.
T	**getEntry**(int givenPosition) Retrieves the entry at a given position in this list.
int	**getLength**() Gets the length of this list.
boolean	**isEmpty**() Sees whether this list is empty.
T	**remove**(int givenPosition) Removes the entry at a given position from this list.
boolean	**replace**(int givenPosition, T newEntry) Replaces the entry at a given position in this list.
Object[]	**toArray**() Retrieves all entries that are in this list in the order in which they occur in the list.

Pre-lab visualization

A LinedList object will be made of a set of linked nodes, looking very much like the LinkedBag object from part A. The list created by the following statements,

```
LinkedList<String> myList = new LinkedList<String>();
   myList.add("15");           myList.add("25");
   myList.add("35");           myList.add("45");
```

will look like the following in memory. Note that add now puts items at the end of the chain, not the beginning as in the LinkedBag class.



Directed Lab Work

We will now start developing and testing the various methods for the LinkedList class. Since the LinkedList shares many of the same features as the LinkedBag class, some of the method definitions are the same for both. In these cases (the **constructors(2), isEmpty, getLength, contains, clear, toArray and toString**) definitions have already been provided.

Your task will now be to complete and test the following methods:

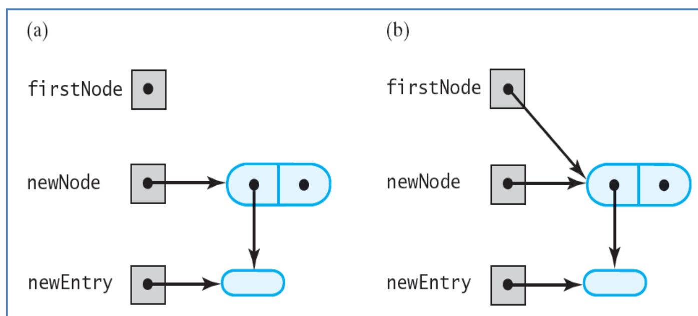
getEntry(position), add(newEntry), add(position, newEntry), remove(position), replace(position, newEntry)

And, if time permits, 2 optional methods:

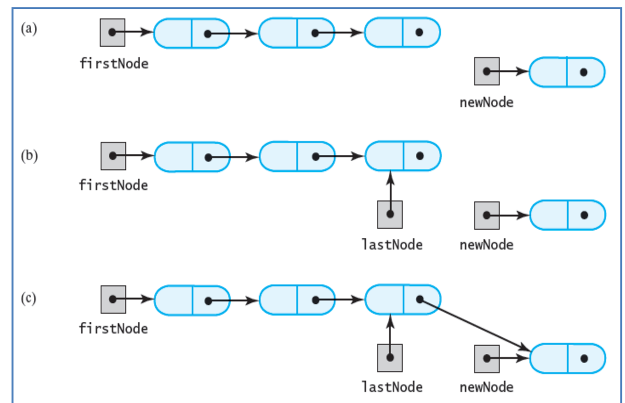
getPosition(anEntry), moveToEnd(position)

Problem 1a: the add(newEntry) method (add to end of list)

This method adds an item to the end of the chain of nodes in the list. There are two possible cases for this method: either the list is empty or it is not. Here are a couple diagrams illustrating these two situations:



ADDING to an EMPTY LIST (set firstNode = newNode)



ADDING at END of LIST (traverse lastNode to end)

The general algorithm for adding to the end of this list is

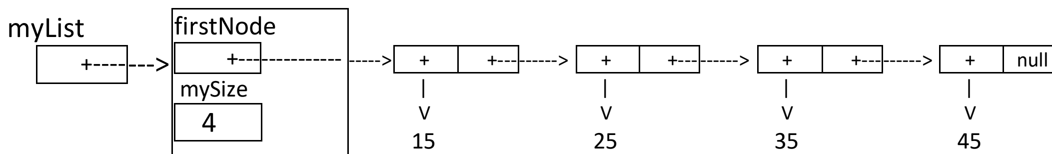
1) construct a new Node referred to by **newNode** that contains the data to be we wish to add (newEntry)
2) if mySize is 0 (empty list)
   a. assign to firstNode the address in newNode
   b. add one to mySize
3) otherwise
   a. make a variable lastNode and initialize it to firstNode

b. advance lastNode to the end of the list: as long as lastNode.next is not null, advance lastNode to the next node.
c. assign to lastNode.next the address of the new Node
d. add one to mySize

Problem 1b: the getEntry(position) method

The **getEntry** method returns the item that sits at **position** in the list. Remember that our list defines position 1 as the start of the list. This is not as confusing as with arrays, but we should still keep alert to the location we are moving to.

As an example, for the following situation with **myList**, if we print the expression **myList.getEntry(3)** we should see a "35" on the screen, NOT "45"!!



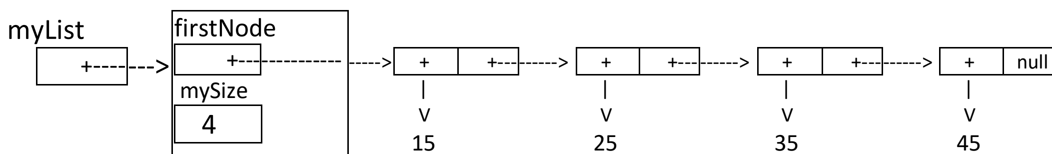
Another task that **getEntry** must perform is to ensure that only legal index values are used to access the data in the list -- a user might accidently execute **myList.getEntry(-4)** or **myList.getEntry(5)**, both of which are outside the range of allowable values for **index**. Should this happen **getEntry** should return a value of null.

Write this method now (you'll need to traverse a Node pointer **curentNode** to the correct node) and verify that your program prints the correct output. It's a simple method that just returns the desired data from the bag, or null if the desired cell is not valid.

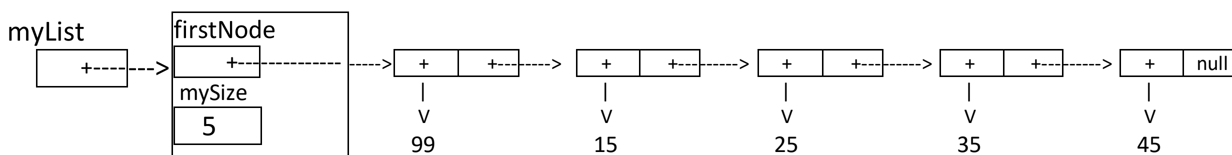
Problem 2: the add(position, newEntry) method

This method takes a **position** and places **newEntry** at the desired location (remember, list positions start at 1. This method also returns **true** if the add was successful (the position was a valid location) and **false** if the add could not be performed (the position requested was outside the allowable range of cells in the array—it would've created a gap in the array or gone past the array bounds).

Example 1: if we have the following list defined:

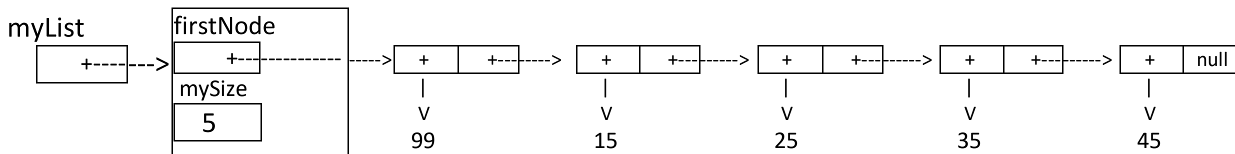


and we execute **myList.add(1,"99");** the result on **myList** will be to change it as follows:

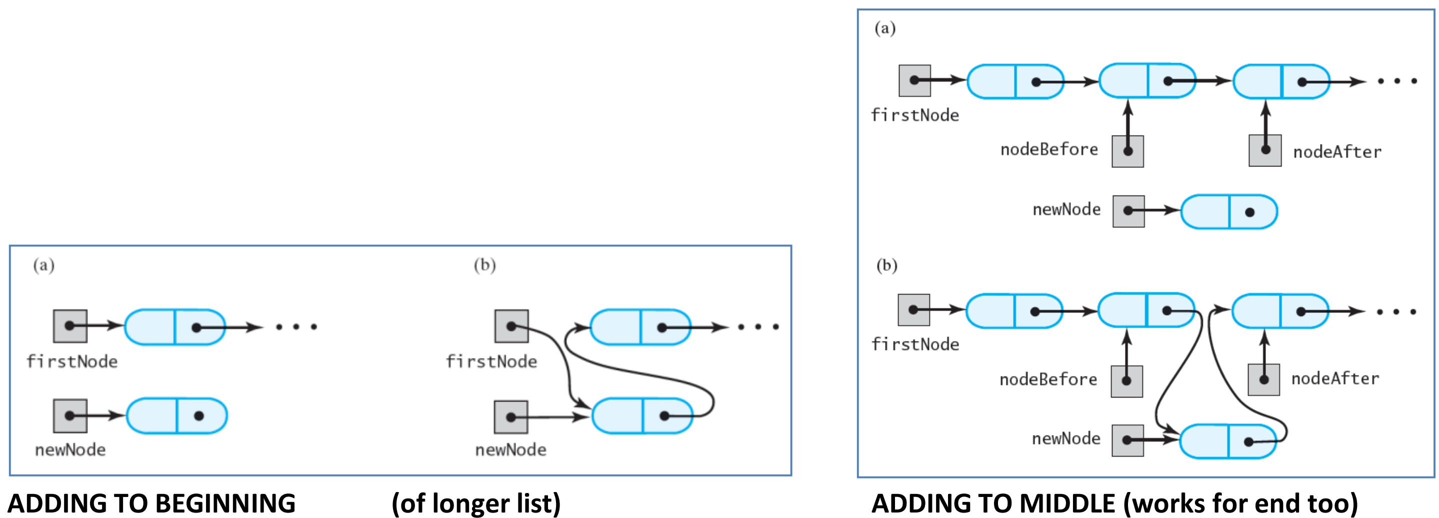


and the **add** method would return **true**, meaning the add was successful. Notice that a new node with 99 has been inserted at the head of the list.

Example 2: However, if we now execute the statement: **myList.add(7"77");** **myList** will remain **unchanged** and a result of **false** will be returned since this command attempts to write at item 7 in the list, leaving a gap at position 6.



There are two cases for the add at position method: adding to the beginning of the list, and adding after the beginning. The following diagrams illustrate these two cases:



ADDING TO BEGINNING **(of longer list)** **ADDING TO MIDDLE (works for end too)**

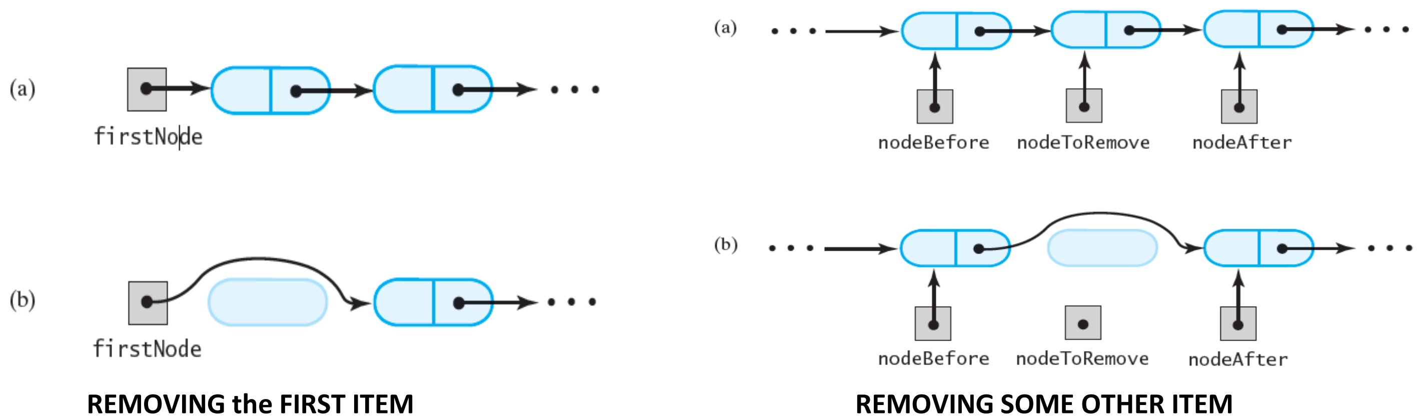
Here is an algorithm for adding at a position:

1) construct a new Node referred to by **newNode** that contains the data to be we wish to add (newEntry)
2) if the newPosition is legal (for a list of 5 entries, this would be positions 1 through 6 (6 appends at end)
    a. if the newPosition is at the beginning of the list
        i. make newNode's next pointer refer to the list Node that firstNode points to
        ii. assign to firstNode the address in newNode
        iii. add one to mySize
        iv. return true (success)
    b. otherwise (newPosition is somewhere else, deeper in the list)
        i. define Node `nodeBefore` and initialize it to point to the first Node in the chain
        ii. use a for loop to advance `nodeBefore` to the node before the insertion point (if newPosition is 3, the number of advances is 1)
        iii. define `nodeAfter` and initialize it to the node that comes after `nodeBefore`
        iv. set the next field of `newNode` so it refers to the `nodeAfter` node.
        v. set the next field of `nodeBefore` so it refers to the `newNode` node
        vi. add one to mySize
        vii. return true (success)
3) otherwise, return false (unsuccessful or illegal insertion point)

Problem 3) remove, replace

Remove saves a reference to the Node that is about to be removed before disconnecting the removed node from the chain, so it can return the data item it contains when the method finishes.

There are two cases for the remove method: removing the item at the beginning of the list, or removing an item somewhere after the beginning. The following diagrams illustrate these two situations:



REMOVING the FIRST ITEM **REMOVING SOME OTHER ITEM**

Here are the steps to remove an item from a linked list:

1. If the givenPosition is legal (example: 1 to 5 would be legal for a list of 5 items, 0 or 6 would be illegal)
    a. if the givenPosition is 1 (first item being removed)
        i. define Node `nodeToRemove` and initialize it to point to the first Node in the chain
        ii. advance `firstNode` to the next node in the chain
        iii. return the entry stored in `nodeToRemove`
    b. otherwise (some other position in chain)
        i. define nodeBefore similar to how you did for add in problem 2, advancing it to the node just prior to the one you want to remove
        ii. define `nodeToRemove` and assign it the node after `nodeBefore`
        iii. define `nodeAfter` and assign it the node after `nodeToRemove`
        iv. rearrange pointers as illustrated in the diagram
        v. return the entry stored in `nodeToRemove`

Replace: Using similar techniques as demonstrated in remove and add, write the replace method (or write the method in terms of remove and add). If the **givenPosition** is legal, this method copies over the item in the Node at **givenPosition** with a reference to **newEntry**, and returns **true** (success). Otherwise it returns **false** (illegal position).

Problem 4) getPosition, moveToEnd

Note: there are no stubs for these methods so you'll have to add them completely, with javadocs, to the LinkedBag file.

Read the example test code in the Lab6b file to see how these methods are used. The method getPosition should return the position (remember, we start counting at 1) of an item in the list. If it is not found it should return the flag -1.

the method moveToEnd should move the item at givenPosition to the end of the list.