CS526
Homework Assignment 3

This assignment has two parts.

Part 1 (10 points)

The *ArrayList* class in pages 260 – 261 of the textbook implements an array list using an array as an underlying storage. Your task is to implement additional methods in this class and also to make the list a dynamic array whose capacity automatically grows as more elements are added beyond the current capacity of the list. Before you begin this assignment, you may want to study the *ArrayList* class carefully.

Implement the following additional methods within the *ArrayList* class. Note that the *ArrayList* in this assignment is not Java's *ArrayList*. All references to *ArrayList* in this program are references to the *ArrayList* class within which you are implementing the additional methods.

- addAll(ArrayList *l*):
  - Input: An ArrayList *l*.
  - Output: None
  - Postcondition: All elements in the list *l* have been added to the end of this list, in the order that they are in *l*.
- ensureCapacity(int minCapacity):
  - Input: The new capacity of this list.
  - Output: None
  - Postcondition: The capacity of this list has been increased to ensure that it can hold at least the number of elements specified by the minimum capacity argument. If the *minCapacity* is not greater than the current capacity of this list, the current capacity is unchanged.
- remove(E e):
  - Input: The element to be removed.
  - Output: Returns true if the element *e* exists in this list, and returns false otherwise.
  - Postcondition: The first occurrence of the element *e* has been removed from this list.
- removeRange(int fromIndex, int toIndex):
  - Input:
    - fromIndex: index of the first element to be removed
    - toIndex: index after the last element to be removed
  - Output: None
  - Precondition: *fromIndex* and *toIndex* must be valid indexes.
  - Postcondition: All elements whose index is between *fromIndex*, inclusive, and *toIndex*, exclusive, have been removed from this list.
- trimToSize( ):
  - Input: None
  - Output: None
  - Postcondition: The capacity of this list has been changed to the list's current size.

- *main* method: An incomplete main method is included in the provided *ArrayList.java* code and you need to add more code segments to test the methods you implemented.

The *ensureCapacity* method allows a list to grow when more elements are added to this list than can be accommodated by the current list capacity, making the list a dynamic array list.

You also need to modify the *add* method so that it may increase the capacity of this list to the twice the current capacity using the *ensureCapacity* method, when adding a new element would increase the number of elements beyond the current capacity of this list. For example, suppose that the capacity of this list is 100 and there are currently 100 elements in this list. If an add method is invoked, then the capacity of this list must be increased to 200 first before the new element is added.

You also need to modify the *addAll* method in a similar way. Suppose that the capacity of this list is 100 and currently there are 80 elements in this list. If the *addAll* method is invoked with another list which has 40 elements, then the capacity of this list must be increased to 240 (= 2 * (80 + 40)).

An incomplete code of the *ArrayList.java* is posted on blackboard. Complete the code by implementing the above methods.

Documentation

No separate documentation is needed. However, you must include the specification of each method in your program, right above each method, and you must include sufficient inline comments within your program.

Grading

Program correctness is worth 80% and documentation is worth 20%.

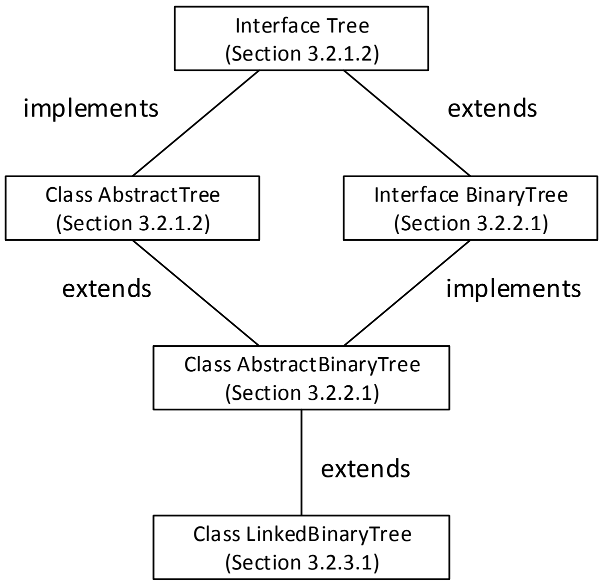
The methods you implemented will be tested and points will be deducted if your methods do not behave as expected.

Points will be deducted if you do not include specifications of methods or sufficient inline comments.

Part 2 (10 points)

This part is about a binary tree that uses a linked data structure. Binary trees are discussed in Section 3.2.2 and an implementation is discussed in Section 3.2.3.1 (and also in Section 8.2 and Section 8.3.1, respectively, in the textbook).

The *LinkedBinaryTree.java* (which comes with the textbook and is also posted with this assignment) is a concrete implementation of a binary tree, which uses a linked structure. It extends the *AbstractBinaryTree* class. The relevant class hierarchy is shown below:


Interface Tree (Section 3.2.1.2)
implements
extends
Class AbstractTree (Section 3.2.1.2)
Interface BinaryTree (Section 3.2.2.1)
extends
implements
Class AbstractBinaryTree (Section 3.2.2.1)
extends
Class LinkedBinaryTree (Section 3.2.3.1)

The *LinkedBinaryTree* inherits methods from its superclasses and it also implements its own methods.

Your task is as follows:

1. Define the *IntLinkedBinaryTree* class that stores integers (of Integer type) in the tree nodes as a subclass of the *LinkedBinaryTree* class.

   The *LinkedBinaryTree* is a generic class that can store elements of arbitrary object type in the tree nodes. For this assignment, you are required to define a subclass of the *LinkedBinaryTree* class that can store Integer objects, and name it *IntLinkedBinaryTree*.

2. Implement two additional methods within the *IntLinkedBinaryTree* class, which will make the tree a *binary search tree*.

   A binary search tree is a binary tree with additional properties, as described in Section 3.2.4.3 (also in Section 8.4.3 of the textbook). Binary search trees will be discussed in more detail in Module 4.

   For this assignment, you are required to implement two “add” methods that add new nodes to a binary tree in such a way that the resulting tree always satisfies the binary search properties.

   The first method you need to implement is the *add* method. The signature of the method is:

   ```
   public Position<Integer> add(Position<Integer> p, Integer e)
   ```

   The pseudocode of the method is given below:

```
Algorithm add(p, e)
  Input parameters:
    p: The position of the root of the subtree to which a new node is added
    e: The integer element of the new node to be added
  Output: Returns the position of the new node that was added.
    If there already is a node with e in the tree, returns null.

  if p == null // this is an empty tree
    create a new node with e and make it the root of the tree
    return the root

  x = p
  y = x
  while (x is not null) {
    if (the element of x) is the same as e, return null
    else if (the element of x) > e{
      y = x
      x = left child of x
    }
    else {
      y = x
      x = right child of x
    }
  } // end of while

  temp = new node with element e
  if (the element of y) > e
    temp becomes the left child of y
  else
    temp becomes the right child of y
  increment size // size is the number of elements currently in the tree
  return temp
```

You may want to read and study the pseudocode carefully to fully understand what it does before writing a code.

Note that this method returns null if the tree already contains the element to be added. So, all elements in this binary tree are distinct.

The second method you need to implement is the *addMultiple* method, whose signature is given below:

```
   public void addMultiple(Integer... a )
```

This method receives as its argument a list of integers, separated by commas, and add all these integers into the tree. You can invoke this method using the following syntax:

```
   t.addMultiple(35, 200, 130, 15, 300, 400, 50, 10);
```

Here, *t* is an instance of the *IntLinkedBinaryTree* class.

3. Write a code segment to experimentally decide an average height of a binary search tree.

   First, you need to generate 1,000 random integers in the range [0, 999999] and add them, one at a time, to an initially empty binary tree, using the *add* method you implemented. Make sure that the resulting tree has 1,000 distinct integers. Then, determine the height of the tree. Repeat this 100 times and calculate the average height of these 100 binary search trees.

An incomplete code of *IntLinkedBinaryTree.java* is posted on Blackboard. The incomplete code also includes an incomplete main method that you can use to test your methods and perform a required experiment. If you want, you may implement additional methods. But, you must implement, at the minimum, the above methods.

Documentation

No separate documentation is needed. However, you must include the specification of each method in your program, right above each method, and you must include sufficient inline comments within your program.

Grading

Program correctness is worth 80% and documentation is worth 20%.

The methods you implemented will be tested and points will be deducted if your methods do not behave as expected.

Points will be deducted if you do not include specifications of methods or sufficient inline comments.

Deliverables

For Part 1, you must complete the *ArrayList.java* file. For Part 2, you must complete the *IntLinkedBinaryTree.java* file. Combine these files into a single archive file, such as a *zip* file or a *rar* file, and name it *LastName_FirstName_hw3.EXT*, where *EXT* is an appropriate file extension (such as *zip* or *rar*). Upload this file to Blackboard.