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. 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*):
  - Adds all elements in *l* to the end of this list, in the order that they are in *l*.
  - Input: An ArrayList *l*.
  - Output: None
  - Postcondition: All elements in the list *l* have been added to this list.
- remove(E e):
  - Removes the first occurrence of the element *e* from this list.
  - Input: The element to be removed.
  - Output: Returns true if the element *e* exists in this list, and returns false otherwise.
- removeRange(int fromIndex, int toIndex):
  - Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
  - 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.
- trimToSize( ):
  - Trims the capacity of this list to be the list's current size.
  - Input: None
  - Output: None
  - Postcondition: The capacity of this list has been trimmed 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.

Note that the *addAll* method must increase the size of the array if needed (as was done in the existing *add* method). Suppose that the current capacity of the list is 20 and the currently size of the list is 15 (this means there are currently 15 elements in the list of capacity 20). If the list to be added has 10 elements, then there is not enough space in the current list and you need to increase the capacity to 50. The new capacity 50 is calculated as 2 * (size of this list + size of the list to be added) = 2 * (15 + 10) = 50. Note that the “size” is not the capacity of the underlying data structure. It refers to the number of elements that are currently in the list.

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, and you must include sufficient inline comments within your program.

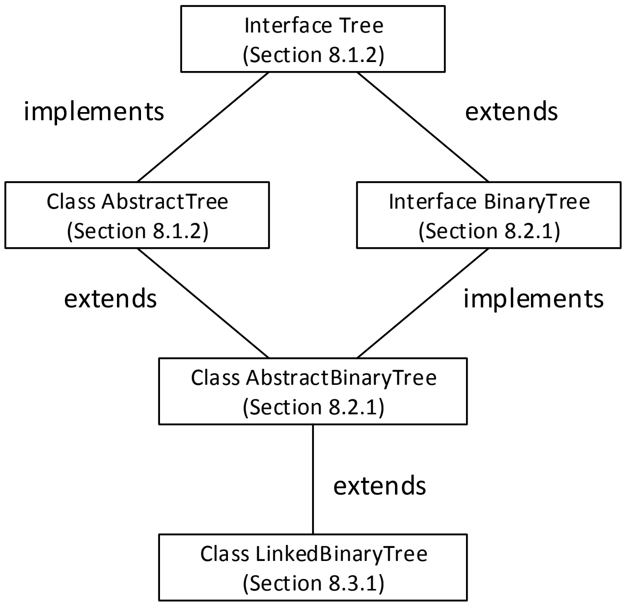
Grading

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 8.2 and Section 8.3.1 in the textbook.

The *LinkedBinaryTree.java*, which comes with the textbook, is a concrete implementation of a binary tree that uses a linked structure. It extends the *AbstractBinaryTree* class. The relevant class hierarchy is shown below:



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

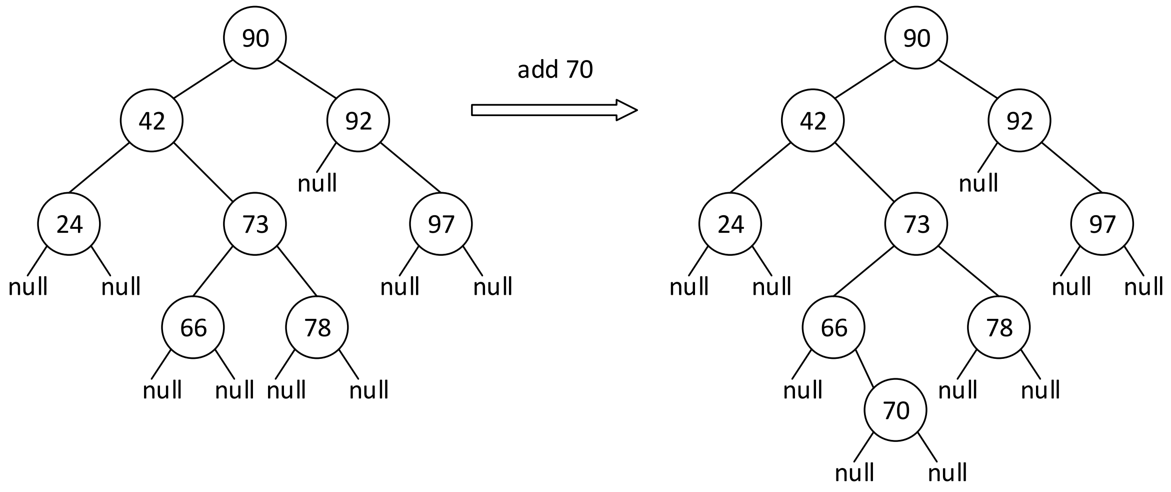
Your task is as follows:

1. 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* that can store Integer objects, and name it *IntLinkedBinaryTree*.

2. Implement an add method within the *IntLinkedBinaryTree* class that adds a new node to a binary tree in such a way that the resulting tree always satisfies the following *binary search tree property*.

   For each internal position *p*:
   - Elements stored in the left subtree of *p* are less than *p*'s element.
   - Elements stored in the right subtree of *p* are greater than *p*'s element.

   A binary tree with the binary search tree property is called a *binary search tree*. The following figure shows two binary search trees. If you add a position with 70 to the tree on the left, the result is the tree on the right.



   The pseudocode of the add 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
  y becomes the parent of temp
  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 and the codes of *LinkedBinaryTree* and its superclasses/super interfaces carefully before writing a code.

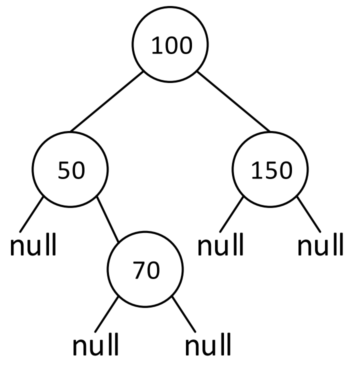
In your program, write a *main* method to test your implementation. An example is shown below:

```java
public static void main(String[] args) {

      IntLinkedBinaryTree t =   new IntLinkedBinaryTree();
      t.add(t.root, 100);
      t.add(t.root, 50);
      t.add(t.root, 150);
      t.add(t.root, 70);

      Iterator<Position<Integer>> it = t.inorder().iterator();
      while (it.hasNext()){
            System.out.print(it.next().getElement() + " ");
      }
      System.out.println();
```

The tree built by the above example is:



And, the expected output is: 50 70 100 150

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. If you want, you may implement additional methods.

<u>Documentation</u>

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

<u>Grading</u>

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.

<u>Deliverables</u>

For Part 1, you must complete the *ArrayList.java* file. For Part 2, you must complete the *IntLinkedBinaryTree.java* file. Combine these two files (and other files if any) 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 it to Blackboard.