CS202: Project 2
A Size-Balanced Binary Search Tree With Range Operations
Due: Monday April 8 at 9am

In this project you will complete the implementation of a Java class for storing sets of **Comparable** elements in a binary search tree. The key things that separates this class from the "vanilla" binary search trees we've studied are:

> Maintaining a size-balanced property ensuring logarithmic worst-case lookup and amortized logarithmic insertion and deletion.
>
> Supporting order-statistics operations.
>
> Supporting "range query" operations.

You are given a java file (also listed at the end of this document) which contains stubs for the methods you must complete. The class contains no data members yet -- that is part of your job. It is recommended that you create a nested static class for representation of tree nodes. However, you could have a node class in a separate class.

Requirements for each method are given in header comments above each method.

Size-Balancing

Your tree will maintain logarithmic height by enforcing a "size-balanced" property.

> **Definition:** size-balance property for a node. Consider a node *v i*n a binary tree with n_l nodes in its left subtree and n_r nodes in its right subtree; we say that ***v* is size-balanced** if:
>
> $$max(n_l,\ n_r) \le 2 \times min(n_l,\ n_r) + 1$$
>
> (so roughly, an imbalance of up to ⅓ - ⅔ is allowed)

> **Definition:** size-balance property for a tree. We say that a binary tree *t* **is size-balanced** if all nodes *v* in t are size-balanced.
>
> $$max(n_l,\ n_r) \le 2 \times min(n_l,\ n_r) + 1$$

Your implementation must always ensure that the tree is size-balanced. Only the insert and remove operations can result in a violation. When an operation results in a violation, you must rebalance **the violating subtree closest to the root.** You do not in general want to rebalance at the root each time there is a violation (only when there is a violation at the root).

You should reuse some of your code for building a balanced bst from a sorted array to restore the balanced property.

Submssion

You will submit your completed file `SBTreeSet.java` and any additional `.java` files you created for helper classes (if any).

You will also submit a proof of the following claim:

> Any size-balanced binary tree with *n* nodes has height $O(log\ n)$

Your proof should be in a file called `proof.pdf.`

Submit all of your files in a single archive.

```
import java.util.Collection;

public class SBTreeSet<T extends Comparable<T>> {

        /**
         * Default constructor initializes an empty SBTreeSet
         */
        public SBTreeSet(){

        }

        /**
        * Constructs an SBTreeSet that is as balanced as possible from the
        * given array a[] under the assumption that a[] is sorted and contains
        * no duplicates; if not, null is returned.
        *
        * Runtime:  O(n)
        *
        * @param a array of set elements in sorted order with no duplicates
        * @return an SBTreeSet containing the given elements on success; null
        * on failure (given array not sorted or has duplicates).
        */
        public static <E extends Comparable<E>> SBTreeSet<E> fromSortedArray(E a[]){
                return null;
        }
        /**
      * Ensures that element x is a member of the set.  Returns true
      * if the set changed (i.e., x was not previously a member) and
      * false if set is  unchanged (x already a member).
      *
      * Runtime:  O(log n) amortized
      *
      * @param x element being inserted
      * @return true if set changed, false otherwise
      */
        public boolean insert(T x){
                return false;
        }

        /**
      * Determines if x is an element of the set
```

```
 *
 * Runtime:  O(log n)
 *
 * @param x element being tested for membership
 * @return true if x is an element of the set, false otherwise.
 */
   public boolean contains(T x){
         return false;
   }

   /**
 * Removes x from set if already a member; does not modify set
 * if x not a member.
 *
 * Runtime:  O(log n) amortized
 *
 * @param x element being removed
 * @return true if set modified (i.e., x was actually a member),
 * false otherwise.
 */
   public boolean remove(T x){
         return false;
   }

/**
 * Returns the number of elements in this set.
 *
 * Runtime:  O(1)
 *
 * @return the number of elements in this set
 */
   public int size(){
         return 0;
   }

/**
 * Returns the height of the tree
 *
 * Runtime:  O(1)
 *
 * @return the height of the tree
 */
   public int height(){
         return 0;
   }


/**
 * Returns the ith element in the ordered set where i ranges from 0..n-1;
 * in other words, if the elements were in a sorted array, the element in
 * index i would be returned. Returns null if i is out of range
 *
 * Runtime:  O(log n)
 *
 * @param i
 * @return element at position i in sorted order (the min being at position 0);
 * null if i is out of range.
 */
   public T atPosition(int i){
         return null;
   }
/**
 * Returns the number of elements x in the set where
 * min <= x <= max.
 *
 * Runtime:  O(log n)
 *
 * @param min lower-bound of range specified
 * @param max upper-bound of range specified
 * @return the number of elements in this set
 */
   public int rangeSize(T min, T max){
         return 0;
   }


/**
 * Returns a collection (e.g., ArrayList<T>) of elements x in the set where
 * min <= x <= max.
 *
 * Runtime:  O(log n + m) where m is the number of elements
 * in the range for the particular query.
 *
 * @param min lower-bound of range specified
 * @param max upper-bound of range specified
 * @return a Collection (e.g., an ArrayList) containing all
```

```
     * elements in the range
     */
      public Collection<T> extractRange(T min, T max){
             return null;
      }

      /**
     * Prints the following
     *
     *   The current size of the set.
     *   The current height of the tree.
     *   The total number of successful insertions since creation.
     *   The total number of successful deletions since creation.
     *   The total number of rebalancing operations performed since creation
     *   The total "work" done over all rebalancing operations; an individual
     *      rebalancing operation does work equal to the size of the subtree
     *      being rebalanced since the operation is linear in the size of the
     *      subtree being rebalanced.
     */
      public void stats(){

      }
}
```