CS 367 Homework 3

Alex Pizzuto

1 Question One

We assume that general trees are implemented using a `Treenode` class that includes the following fields and methods:

```
// fields
private T data;
private List<Treenode<T>> children;

// methods
public T getData() { return data; }
public List<Treenode<T>> getChildren() { return children; }
```

With this, we will write a method, `isBinary`, with the following header:

```
public boolean isBinary( Treenode<T> n )
```

that will determine if the general tree rooted by `n` is also a binary tree. Our method is as follows:

```
public boolean isBinary( Treenode<T> n ) {
   //First, we define a base case
   boolean value = true;
   if (n.getChildren.size() == 0 ) {
      return value;
   }
   if (n.getChildren.size() > 2 ) {
      return false;
   }
   Iterator<T> itrt = n.getChildren.iterator();
   while (itrt.hasNext() ) {
      value = value && isBinary(itrt.next());
   }
   return value;
}
```

2 Question Two

Next, we write the `findNegatives` method for a binary tree implemented using a `BinaryTreenode` class with the following header:

```
public static List<Integer> findNegatives( BinaryTreenode<Integer> n)
```

Our method will return a list containing all the negative values in a binary tree containing `Integer` data. First, we establish the rules for recursion of this function:

- The list of negative values in an empty tree is the empty list.
- The list of negative values in a tree with one node is a list containing the node's data if the node's data is negative
- The list of negative values in a tree with more than one node is the concatenation of lists of the negative values in each subtree rooted by the node's children.

With this, we can write the method (assuming an `ArrayList` implementation of a `List`).

```
public static List<Integer> findNegatives( BinaryTreenode<Integer> n) {
   List<Integer> negs = new ArrayList<Integer>();
   if (n.getData() instanceof Integer && n.getData() < 0 ) {
      negs.add(n.getData());
   }
   if(n.getLeft() != null) {
      negs.addAll(findNegatives(n.getLeft()));
   }
   if(n.getRight() != null ) {
      negs.addAll(findNegatives(n.getRight()));
   }
   return negs;
}
```

3 Question Three

We assume that binary search trees are implemented using a BSTnode class that includes the following fields and methods:

```
// fields
private K key;
private BSTnode<K> left, right;

// methods
public K getKey() { return key; }
public BSTnode<K> getLeft() { return left; }
public BSTnode<K> getRight() { return right; }
public void setLeft(BSTnode<K> newL) { left = newL; }
public void setRight(BSTnode<K> newR) { right = newR; }
```

where K is a class that implements the `Comparable` interface. With this, we write the `secondSmallest` method:

```
public K secondSmallest(BSTnode<K> n) {
   BSTnode<K> tmp;
   while(n.getLeft() != null ) {
      if(n.getLeft().getLeft != null ){
         n = n.getLeft(); //If there are more grandchildren, move down the tree
      }else if(n.getLeft().getRight() == null ) {
         return n;
```

```
            //If both the left and right child of the left child are null,
            //then return the partn
        } else if (n.getLeft().getRight() != null) {
            n = n.getLeft().getRight();
            while(n.getLeft() != null ) {
                n = n.getLeft();
            }
            return n;
            //If the left child of the left child is null, but the left child
            //has a right child, return the smallest value of the tree
            //rooted by the right child of the left child
        }
    }
}
```