CS 367 Homework 3

Mingren Shen (mshen32@wisc.edu)

Question 1

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; }
```

For efficiency, use an iterator to access the children in the list returned by the method `getChildren` (as we've done in lecture). You may assume that `getChildren` never returns `null`: if a node is a leaf, then `getChildren` will return a non-`null` list containing zero elements.

Write an `isBinary` method whose header is given below.

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

The method should determine if the general tree rooted by `n` is also a binary tree. A binary tree is recursively defined as:

- An empty tree is binary.
- A leaf node is a binary tree.
- A node with 1 or 2 children is binary if each child is itself a binary tree.

Answer

```
    // import java.util.Iterator; if needed

    public boolean isBinary( Treenode<T> n)
    {
             /* Base case */
            // An empty tree is a binary tree
            if ( n == null)
            {
                return true;
```

```
            }

            // A leaf node is a binary tree
            if ( n.getChildren().isEmpty() )
            {
                return true;
            }

            /*General case */
            // A node with 1 or 2 Children is binary only when each child is
binary
            if ( n.getChildren().size() == 1 || n.getChildren().size() == 2)
            {
                // recursively use isBinary for each children tree
                Iterator<Treenode<T>> iter = n.getChildren().iterator();
                while ( iter.hasNext() )
                {
                    // if this child is not binary
                    if ( !isBinary(iter.next()))
                    {
                        return false;
                    }
                }
                // all children are binary
                return true;
            }
            // size is 3 or larger
        return false;
    }
```

Question 2

Assume that binary trees are implemented using a `BinaryTreenode` class that includes the following fields and methods:

```
// fields
private T data;
private BinaryTreenode<T> left, right;

// methods
public T getData()                { return data; }
public BinaryTreenode<T> getLeft()        { return left; }
public BinaryTreenode<T> getRight()       { return right; }
public void setLeft(BinaryTreenode<T> newL)  { left  = newL; }
public void setRight(BinaryTreenode<T> newR) { right = newR; }
```

Write the `findNegatives` method whose header is given below.

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

The method should return a list containing all the negative values in a binary tree containing Integer data. For example, if the tree pointed to by n looks like this:

```
       -6
       / \
      /   \
    -4     5
    /     / \
   9     8  -1
  / \     \   \
 7  -3     2   0
```

then `findNegatives(n)` should return a list containing -6, -4, -1, and -3 (not necessarily in this order). If the same value appears more than once in the tree, it should also appear more than once in the result list.

Part A: First, complete the English descriptions of the base and recursive cases, like what was given above for Question 1.

- 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 (**the list containing the value of that node if the node's value is negatives**)
- The list of negative values in a tree with more than one node is (**the list containing the value itself if it is smaller than 0 and the values in the list of the node's leftChild and the list of the node's rightChild**)

Part B: Now write the findNegatives method. You may assume that the List used to hold negative values is implemented as an `ArrayList`.

```
    public static List<Integer> findNegatives( BinaryTreenode<Integer> n)
    {
            ArrayList<Integer> result = new ArrayList<Integer>();

            // empty tree and leaf node return empty list
            if ( ( n == null ) )
            {
                return new ArrayList<Integer>();
            }

            //  leaf child node
            //  list containing the value of that node if the node's value is
negatives
            if ( n.getLeft() == null && n.getRight() == null)
            {
                ArrayList<Integer> tmp = new ArrayList<Integer>();

                if ( n.getData() < 0)
                {
                    tmp.add( n.getData() );
                }
                return ( tmp );
            }

            // general cases
            //  list containing  the value itself if it is smaller than 0
            if( n.getData() < 0)
            {
                result.add( n.getData() ) ;
            }
            // containing the list of the node's leftChild
            result.addAll( findNegatives( n.getLeft()) );
            // containing the list of the node's rightChild
            result.addAll( findNegatives( n.getRight()) );
            return result;
    }
```

Q3

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. For this question you will **write the secondSmallest method** whose header is given below.

`public K secondSmallest(BSTnode<K> n)`

The method should return the second smallest item in the tree or null if the tree is empty or only has one item. (Note: your method is not required to be recursive.)

Answer

Because BST is an ordered structure so we can do inorder travesal of the tree which return the nodes of the tree from smallest to the largest. And we only need to insert the elements into a `ArrayList` and check the size of this list and return when it gets the second elements.

```
public K secondSmallest(BSTnode<K> n)
    {
        // check whether the tree is empty
        if ( n == null )
        {
            return null;
        }
        // create the list to store the elements in order
        List<K> inOrderList = new ArrayList<K>();
        // call helper method
        secondSmallestHelper(n, inOrderList);
        // check result to determine retunrning results
        if ( inOrderList.size() == 0 || inOrderList.size() == 1 )
        {
            return null;
        }
        return inOrderList.get( 1 );
    }
```

```
private void secondSmallestHelper(BSTnode<K> m, List<K> returnList)
{
    // perform in order traversal for BST
    if ( m == null )
    {
        return;
    }
    // only when we do not have 2 elements in the list
    // we process to the next elements
    if( returnList.size() < 2 ) {
        secondSmallestHelper( m.getLeft(), returnList);
        if ( returnList.size() < 2 )
        {
            returnList.add( m.getKey() );
            secondSmallestHelper( m.getRight(), returnList);
        }
    }
}
```