CMPSCI 187 (Spring 2018) Lab 09: Binary Search Tree (BST)

The goal of this lab is to get familiar with BST, by implementing two methods (search in a BST, and check the validity of a BST). To work on the assignment:

- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account
- Follow the instructions to complete the assignment
- When you are done, go to ***File -> Download As -> PDF Document***
- Log in to Gradescope and submit your PDF

The lab is due by 5:00 pm today. Please make sure that you submit to Lab 9, not Project!

Section 1: Multiple Choice [3 x 2 points = 6 points]

One correct answer per question. **Select your choice by making that option red and bold.**

1. **If a full binary tree contains a total of N nodes, how many interior (i.e. non-leaf) nodes and leaf nodes are there?**

(a) Interior N/2 , leaf N/2
(b) Interior (N+1)/2, leaf (N-1)/2
(c) Interior (N-1)/2, leaf (N+1)/2
(d) Interior (N/2)-1, leaf (N/2)
(e) Interior (N/2)+1, leaf (N/2)+1

2. **We are given a set of n distinct elements (i.e. no two are equal) and an unlabeled binary tree with n nodes (i.e. the structure of the tree is given). In how many ways can we populate the tree with the given set of n elements so that it's a valid BST?**

(a) 0
(b) 1
(c) n
(d) n!
(e) None of the above

3. **In a BST, which of the following is true about the inorder successor of a given node?**

(a) The inorder successor is always a leaf node.
(b) The inorder successor cannot have a right child.
(c) The inorder successor may be the parent of the given node.
(d) The inorder successor cannot have a left child.
(e) The inorder successor cannot be the right child of the given node.

Section 2: Search in BST [5 points]

In this section you will implement the BST **search** method (in lectures we covered a recursive version, here you **must write an iterative version**, meaning using loops and do NOT use recursion). Also, the test program shows you the difference in performance comparing linear search (in an unsorted array) vs. using BST search. The cost of linear search (in an unsorted array) is `O(N)`. The cost for search in BST is `O(h)`, where `h` is the height. For a balanced BST this is `O(log N)`, so the difference in performance will be quite significant for a large `N`.

The following classes are in the `BST_student.zip` code zip file posted on piazza:

- a generic `BSTNode` class (note that all data members are public so you can directly access them)
- a generic `BST` class (the `search` and `isValid` methods are what you need to work on).
- a `RunTests` class that provides some basic tests to help you implement your methods.

First, implement the `search` method in `BST.java`. This must be an iterative version using a loop. You are **NOT allowed** to use recursion. Use `RunTests.java` to check whether your implementation is correct. You can also see a performance comparison between linear search vs. `BST` search, which hopefully will make you appreciate the big performance gain provided by BST search.

```
// return null if elem does not exist; or the node containing it if it exists
public BSTNode<T> search(T elem) {
    BSTNode<T> search = root;
    while (search != null) {
        if (search.data.equals(elem))
              return search;
        if (search.data.compareTo(elem) < 0) {
              if (search.right == null)
                    break;
              else
                    search = search.right;
        } else {
              if (search.left == null)
                    break;
              else
                    search = search.left;
        }
    }
    return null;
}
```

Section 2: Validity of BST [10 points]

In this section, you will write a method to check the validity of a BST. First, review the definition of BST (what are the BST rules?). Next, think about how to write a method to check whether a binary tree is a valid BST or not. This may not be as obvious as you think. For example, you may think that it's sufficient to check at every node whether its value is in between its two children. This is actually wrong, because a node may have grand-children that violate the BST rules.

There are several possible solutions to implement the `isValid()` method, all using recursion. The first is to perform an inorder traversal (which is recursive), and then use the traversal result to check validity. Think about how. This is the easiest solution, but it requires `O(N)` memory, because it needs that memory to store the traversal result. You can use Java's `ArrayList` class for this question.

In a second solution, you can avoid additional memory but directly using the definition of BST. Specifically, you can pass a min value and a max value down to the recursive calls to help you track the allowed range of values to the children nodes. For example, if at the current node the allowed range is given by [min, max], when I make recursive calls to the left child, what would be its value range? How about the right child? What would be the min and max values to pass on to the initial call (i.e. the root node)?

Alternatively, you can have the recursive calls return the min and max values of each subtree, and use those values to track the allowed range of values of the parent node.

For this question you are allowed to create a new method such as a recursive helper method. Any method created by you must be included in this document, otherwise you will lose points. After implementing the method, you can run RunTests class, which includes a few test cases. Remember, since this is a generic class, you CANNOT use < or >, instead, you must use the `compareTo` method.

```
public boolean isValid() {
    ArrayList<T> sorted = new ArrayList<T>(size());
    bstToSortedArray(sorted, root);
    for (int i = 0; i < sorted.size() - 1; i++)
        if (sorted.get(i).compareTo(sorted.get(i + 1)) > 0)
              return false;
    return true;
}

private void bstToSortedArray(ArrayList<T> list, BSTNode<T> subtree) {
    if (subtree != null) {
        bstToSortedArray(list, subtree.left);
        list.add(subtree.data);
        bstToSortedArray(list, subtree.right);
    }
}
```