# CSC172 LAB

# BINARY SEARCH TREES

## 1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in his own work, but every student must list the name of the lab partner (if any) on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

## 2 A Binary Search Tree

The goal of this lab it to gain familiarity with simple binary search trees.

1. Begin this lab by implementing a simple class that represents a “node” in a binary search tree, as follows. Compile the following class.

```
public class MyTreeNode {
     public Comparable data ;
     public MyTreeNode leftchild;
     public MyTreeNode rightchild;
     public MyTreeNode parent;
```

```
}
```

2. Have the second member of your pair type in the code for the simple binary search tree interface. You should in us a `private MyTreeNode root;` variable to act as the root of the tree in your class. [Note: as discussed in your text book, the use of Comparable in this context is not type safe. The proper use of generics would be advisable. However, since we only want to illustrate the manipulation of references in this lab exercise, we will allow the use of Comparable.]

```
public interface MyBST {
     public void insert(Comparable x);
     public void delete(Comparable x);
     public boolean lookup(Comparable x);
     public void printPreOrder();
     public void printInOrder();
     public void printPostOrder();
}
```

3. Implement your own binary search tree class that implements `MyBST.` Write a constructor and the `insert()` method. Your insert method should check for an empty root node and construct a root node if necessary. If the root node already exists, then you should call insert on the root node. To do this, you will need to implement a recursive version of insert in the MyTreeNode class.
4. Implement the `printPreOrder(), printInOrder() and printPostOrder()` methods. The best strategy for this is to implement simple versions of these methods in your binary search tree class. Then implement a recursive version of these methods in the MyTreeNode class. The methods in the your binary search tree class should only call the MyTreeNode version on the root node, if it exists. Also implement a test program class with a main method that inserts a few objects into your binary search tree class and then prints the data in the appropriate order.
5. Implement the `lookup()` method. The lookup method should return true if the object is contained in the tree and return false otherwise. Modify your insert() method so as to prevent duplicate items (only insert an item if lookup() returns false). Modify your test program to demonstrate that the lookup method works.
6. Implement the delete() method. The delete method should do nothing if the item is not found in the tree. If the item is found then it should modify the tree to remove the item. Recall that there are three cases. (Be sure to modify your main test program to illustrate that your delete method works for all three cases.)
    1. If the item to be deleted is a leaf on the tree, delete it simply by setting the parent's reference to null.
    2. If the item to be deleted has a single child, then delete it simply by setting the parent's reference to the single child.
    3. If the item to be deleted has two children, then you need to

1. find the leftmost member of the right child.
2. Replace the node's data item with the data item found
3. Delete the node that contained the found data item (which itself can have at most one child).

# 3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped”) file that contains the following.

1. A README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.).
2. Several JAVA source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file. It is expected that you will have a file for the Node class, a file for the MyBST interface, a file for your own binary search tree class and a file for the test program class.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

# 4 Grading

172/grading.html

Each section (1-6) accounts for 15% of the lab grade (total 90%)

(README file counts for 10%)