Learning Outcomes

Upon completion of this lab, you should be able to do the following in Eclipse:
• Understand inorder traversals of Binary Search Trees and
• Observe how Binary Search Trees look when they are built in different ways.

General Lab Instructions to Help Labs Run Smoothly

• Read through the lab instructions before coming to the lab.
• Do the pre-lab preparation.

Preparation. Review the lecture notes on the Binary Search Trees. A binary search tree is a binary tree with the following property: for each internal node \( p \) the data stored in any node in the left subtree of \( p \) is smaller than the data stored in \( p \), and the data stored in any node in the right subtree of \( p \) is larger than the data stored in \( p \).

IMPORTANT: Make sure you attend your lab session and sign an attendance sheet, show the result(s) of each exercise to your TA, and submit your .java files through OWL by 11:55pm on the same day as your lab session to get your marks for the lab.

Exercise 1: Traversing a Binary Search Tree in Descending Order

Recall that an inorder traversal of a Binary Tree does the (recursive) traversal like this: traverse left subtree, visit root, traverse right subtree. For a Binary Search Tree, this results in the tree items being visited in ascending order. To visit the items in descending order, we switch the left and right traversals in the recursive algorithm.

Let us call the method that performs the right, root, left variation of inorder traversal \texttt{iteratorInOrderDescending()}. You need to create class \texttt{IterableBinarySearchTreeList}, where you will implement the following two methods:
• \texttt{public Iterator iteratorInOrderDescending()}, which returns an iterator that iterates over all elements in the tree in descending order. \texttt{Hint:} see the \texttt{iteratorInOrder()} method in \texttt{LinkedBinaryTree}. You can copy its code to this method, and change it to use your \texttt{inorderDescending()} method.
• \texttt{private inorderDescending(BinaryTreeNode node, ArrayUnorderedList tempList)}, which performs a recursive inorder traversal in descending order. \texttt{Hint:} this is nearly identical to the \texttt{inorder()} method in \texttt{LinkedBinaryTree}. Here, instead of going left to right, we want to go right to left.
Exercise 2: Creating a Balanced Binary Search Tree List

Download the files BuildBSTList.java and iteratorLevelOrder.txt and put the code of the iteratorLevelOrder() method into your LinkedBinaryTree.java from Lab 10 (this is the only method you did not complete in Lab 10). Download this version of LinkedBinarySearchTree.java. Note that it requires several classes to compile; we provide most of them here and here.

BuildBSTList.java is an (incomplete) program that creates an ArrayOrderedList storing the integers from 1 to 10, and then builds a BinarySearchTreeList by just taking the items from the ordered list (in increasing order) and adding them one by one to the binary search tree. What do you expect that the binary search tree will look like? This will be confirmed when you complete the program and run it; it will print the content of the nodes of the tree in level-order. You should look at the output produced by the program and draw on a piece of paper the corresponding tree.

The program then builds another BinarySearchTreeList by calling method buildBalanced() that tries to build a tree where either both subtrees of an internal node have the same height or they differ in height by at most 1. (This tree is called a balanced tree.) The heart of this method is method buildBalancedRec() that you need to write. This is a recursive algorithm that receives 4 parameters: an array tempArr of integers, two integer values start and end, and a BinarySearchTreeList object bstList. The algorithm works as follows:

- The base case is when start > end. The algorithm does not need to do anything in this case.
- The recursive case happens when start <= end. In this situation the values in tempArr[start],...tempArr[end] need to be stored in bstList. To do this, first compute the position of the middle element in the subarray tempArr[start],...tempArr[end] (middle = (start+end)/2). Add tempArr[mid] to bstList and then make two recursive calls: One to store the elements tempArr[start],...tempArr[mid-1] into bstList and the other to store the elements tempArr[mid+1],...tempArr[end] into bstList.

Run your completed program and note the difference in the level-order representation of the first tree created (by just adding the items in order) and the second tree created (by adding the items so that the tree is as balanced as possible). Draw a picture of the two trees from the level-order representation, and show it to the TA.

The program will finally call your method iteratorInOrderDescending() on the second tree. Verify that the values stored in the tree are printed in decreasing order of value. Note: In OWL instead of drawing the above 2 trees, write in level order the nodes of each tree:

Tree 1
level 0 nodes: ...
level 1 nodes: ...
...
Tree 2
level 0 nodes: ...
...