AVL Versus Simple Binary Search Trees

Out: 2/14
Due: 3/6 by 11:59 PM

"To iterate is human, to recurse divine."[L. Peter Deutsch]

Learning Objective

- Compare Attributes of AVL and Simple Binary Search Trees

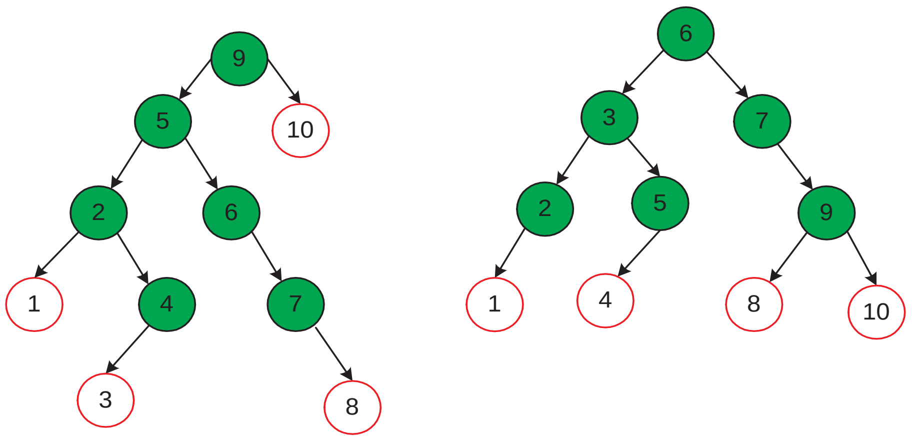
In class we showed that searching a binary search tree is $O(h)$, where h is the height of the tree. On average, a binary search tree with n keys generated from a random series of insertions has expected height $O(\lg n)$. The self-balancing property of an AVL tree generally ensures that its height is shorter than a simple binary search tree for the same series of insertions. The difference in heights between an AVL tree and a binary search tree after inserting sorted data in parallel in the trees is relatively large. Inserting random data in parallel in the trees leads to relatively small difference in height, if any. The choice regarding which tree will give the overall better performance depends on, among other factors, how critical this difference in height is to the performance of the application and also the time that is required to restore balance to an AVL tree after insertion and deletion operations.

In this project you will augment the implementations of a parametric extensible AVL tree and a simple binary search tree abstract data types. You will then write a program that instantiates an AVL tree and a simple binary search tree to store strings. Your program will execute a series of *insert*, *remove* and *traversal* instructions on them. The program will be called *HarvestAnalyzer*. It will take the name of a *command file* as a command line argument and execute the instructions in the file while performing a trace of the instructions as they are executed. The command file consists of instructions in one of these formats:

1. *insert word*
2. *remove word*
3. *traverse*

The *insert* command in the file is followed by a word. When your program reads this instruction, it inserts the uppercase version of the word into the AVL and binary search trees. Whenever a word is inserted, a message is displayed showing the word that is inserted along with the size and height of each tree and whether the tree is *full* and *complete*. Additionally, the number of leaf nodes and their levels, in left-to-right order, are also displayed.

Definition 1. The **Level** of a leaf node is the length of the path from the root to the leaf node. Observe that the root node in a tree with only one node is a leaf node and its level is 0.



A binary search tree with 4 leaf nodes 1, 3, 8 and 10 at levels 3, 4, 4 and 1, respectively.

A binary search tree with 4 leaf nodes 1, 4, 8 and 10 each at level 3.

Figure 1: The Levels of the Leaf Nodes

For the command **insert ignominious**, a typical trace would be generated and formatted as shown in Listing 1.

Listing 1: A Sample Trace After Insertion

```
inserted: IGNOMINIOUS in the AVL
inserted: IGNOMINIOUS in the BST
Type: size      height    full?     complete?
AVL:  1         0         true      true
BST:  1         0         true      true

AVL:
Number of Leaf Nodes: 1
Levels of Leaf Nodes: 0
BST:
Number of Leaf Nodes: 1
Levels of Leaf Nodes: 0
```

When the **remove** instruction is executed, the program determines the depth of the word in the tree and displays the word, its depth in the tree and then removes the word from the tree. It also displays the size and height of each tree and whether the tree is *full* and *complete*. Additionally, the number of leaf nodes and their levels, in left-to-right order, are also displayed. A typical trace would appear in the format shown in Listing 2.

Listing 2: A Sample Trace After Deletion

```
removed: IGNOMINIOUS from a depth of 0 in the AVL
removed: IGNOMINIOUS from a depth of 0 in the BST
Type: size      height    full?     complete?
AVL:  0         -1        true      true
BST:  0         -1        true      true

AVL:
Number of Leaf Nodes: 0
Levels of Leaf Nodes: N/A
BST:
Number of Leaf Nodes: 0
Levels of Leaf Nodes: N/A
```

See another example of a trace in Listing 3, this time when there are two of more leaf nodes.

Listing 3: A Sample Trace After Deletion

```
inserted: ASTUTE in the AVL
inserted: ASTUTE in the BST
Type: size      height    full?     complete?
AVL:  8         3         false     false
BST:  8         4         false     false

AVL:
Number of Leaf Nodes: 4
Levels of Leaf Nodes: 2   3   2   2
BST:
Number of Leaf Nodes: 3
Levels of Leaf Nodes: 1   4   3
```

When the *traverse* command is executed it performs an in-order traversal of each tree and displays the contents of the trees in tabular format, one word per line, followed by the word count. A typical trace would appear as shown in Listing 4.

Listing 4: A Sample Trace After traverse

```
AVL (in-order):
ASCERTAIN
ASTUTE
BASTION
-----------------------------------
word count: 3

BST (in-order):
ASCERTAIN
ASTUTE
BASTION
-----------------------------------
word count: 3
```

To run the program, the name of the command file is entered as a command line argument. I have included three sample command text files, *smalltrees.ops*, *sortedtrees.ops* and *bigtrees.ops*, and a text file, *smalltrees.out*, containing the output that the program should generate when *smalltrees.ops* is used as the command file. You can use the files with the *small* prefix to verify the correctness of your program. Your program should also work correctly on *bigtrees.ops*, *sortedtrees.ops* and any valid command file. These files are included in the starter code zip archive for the project. Exhaustively test your program including with additional command files. See project files for details on assigned tasks.

Submitting Your Work

1. All source code files you submit must have a header with the following:

```
/**
 * Describe the purpose of this file
 * Course: CS3102.01<br>
 * Programming Project #: 2<br>
 * Instructor: Dr. Duncan<br>
 * @see the list of all the files that this file references
 * @author Programmer(s)
 * @since 9999-99-99
 */
```

2. Verify that your code has no syntax error and that it is ISO C++11 or JDK 8 compliant. Be sure to provide documentation, where applicable, for the methods that you have been asked to write. When you augment the starter code, add your name after the @author tag and put the last date modified after the @since tag..

3. Enclose your source files-

   (a) for Java programmers, **AVLTreeAPI.java**, **AVLTree.java**, **BSTreeAPI.java**, **BSTree.java** and **HarvestAnalyzer.java**

   (b) for C++ programmers, **AVLTree.h**, **AVLTree.cpp**, **BSTree.h**, **BSTree.cpp** and **HarvestAnalyzer.cpp**

   - in a zip file. Name the zip file *YOURPAWSID_proj02.zip*, and submit your project for grading using the digital drop box on the course Moodle.