San José State University
Department of Computer Engineering

CMPE 180-92
Data Structures and Algorithms in C++

Spring 2017
Instructor: Ron Mak

Assignment #13

100 points
(with extra credit)

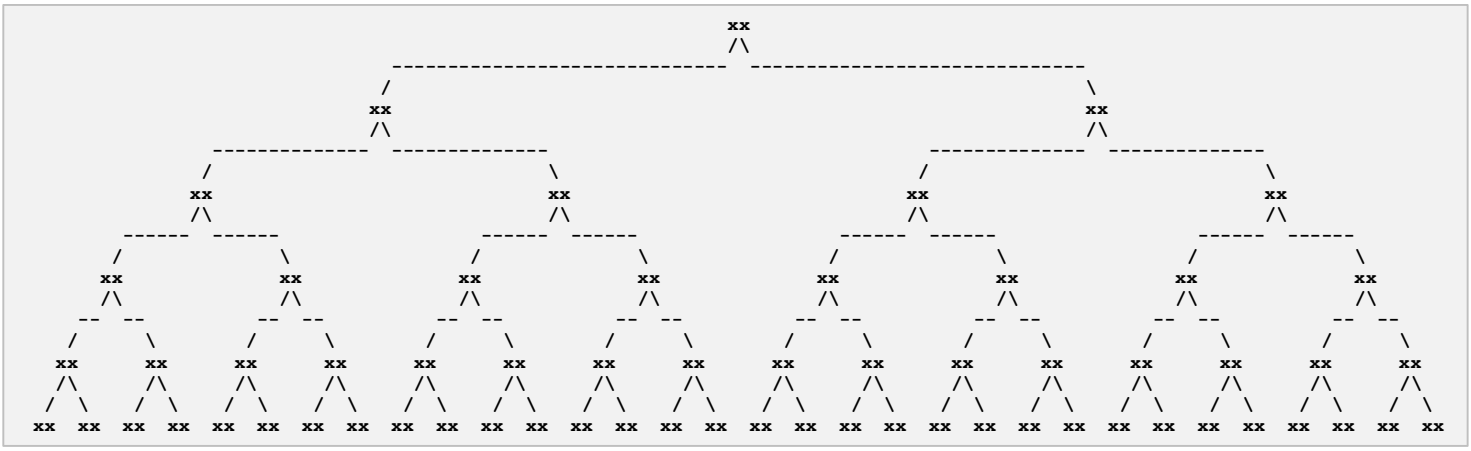
Assigned: Friday, April 28
Due: Thursday, May 4 at 5:30 PM
URL: http://codecheck.it/codecheck/files/17042822569gg4l3tcehvkl7r75mqcafqvi
Canvas: Assignment 13. BST and AVL trees
Points: 200

BST and AVL trees

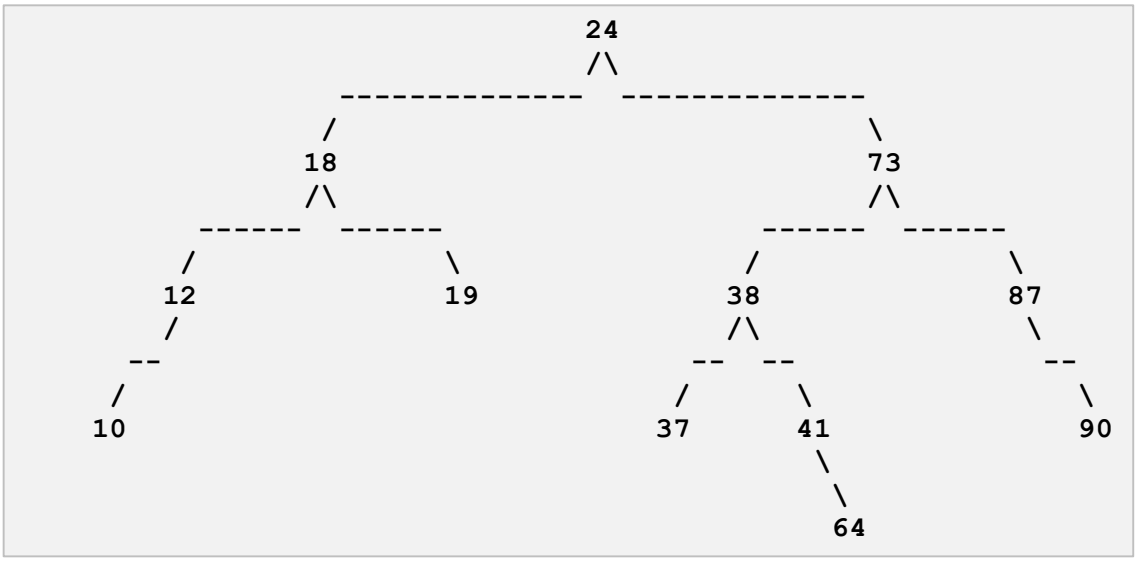
This assignment will give you practice with binary search trees (BST) and balanced Adelson-Velskii and Landis (AVL) trees.

A tree printer

You are provided a **`TreePrinter`** class with a **`print`** method that will print any arbitrary binary tree. A template for how it prints a tree:



`TreePrinter` can print trees with height up to 5, i.e., 32 node values on the bottom row. An example of an actual printed tree:



Part 1

The first part of the assignment makes sure that you can successfully insert nodes into and delete nodes from BST and AVL trees. **Do this part in CodeCheck.**

```
Inserted node 62:

  62

Inserted node 71:

    62
     \
      \
      71

Inserted node 29:

    62
    /\
   /  \
  29  71

Inserted node 88:

        62
        /\
      --  --
     /      \
    29      71
              \
               \
               88
```

First, create a BST node by node. You will be provided the sequence of integer values to insert into the tree one at a time. Print the tree after each insertion. The tree will be unbalanced.

Then repeatedly delete the root of the tree. Print the tree after each deletion to verify that you did the deletion correctly. Stop when the tree becomes empty.

(The BST code is provided for you.)

Second, create an AVL tree, node by node, by inserting the same sequence of integer values. Print the tree after each insertion to verify that you are keeping it balanced. Each time you do a rebalancing, print a message indicating which rotation operation and which node. For example:

```
Inserted node 10:
    --- Single right rotation at node 21
```

Then, as you did with the BST, repeatedly delete the root of your AVL tree. Print the tree after each deletion to verify that you are keeping it balanced.

A handy AVL tree balance checker:

```
int AvlTree::checkBalance(BinaryNode *ptr)
{
    if (ptr == nullptr) return -1;

    int leftHeight  = checkBalance(ptr->left);
    int rightHeight = checkBalance(ptr->right);

    this->probe_count += 6;

    if ((abs(height(ptr->left) - height(ptr->right)) > 1)
        || (height(ptr->left)  != leftHeight)
        || (height(ptr->right) != rightHeight))
    {
        return -2;       // unbalanced
    }

    return height(ptr);  // balanced
}
```

Expected output for Part 1

See http://www.cs.sjsu.edu/~mak/CMPE180-92/assignments/13/output.txt

CodeCheck will check your output. Unfortunately, since the output is long, CodeCheck will only show the first part of the output.

Part 2

The second part of the assignment compares the performance of a BST vs. an AVL tree. Part 2 should be a separate program from Part 1. Adjust all counts accordingly for slower machines. **Do this part outside of CodeCheck.** You will not be provided code for this part, but you should re-use code from Part 1 with any necessary modifications.

First, generate n random integer values. n is some large number, explained below. Insert the random integers one at a time into the BST and AVL trees. For each tree, collect the following statistics for all the insertions:

- **Probe counts.** A probe is whenever you visit a tree node, even if you don't do anything with the node other than use its left or right link to go to a child node.
- **Comparison counts.** A comparison is a probe where you also check the node's value.
- **Elapsed time** in milliseconds.

Do not print the tree after each insertion. Be sure to count probes and comparisons during AVL tree rotations.

Then generate another n random integer values. For each of the BST and AVL trees, count the total probes and comparisons and compute the total elapsed time to search for all the values one at a time. It doesn't matter whether or not a search succeeds.

Choose values of n large enough to give you consistent timings that you can compare. Try values of n = 10,000 to 100,000 in increments of 10,000. Slower machines can use a different range of values for n.

Print tables of these insertion and search statistics for the BST and AVL trees as comma-separated values. Use Excel to create the following graphs, each one containing two plots, one for BST and one for AVL:

- insertion probe counts
- insertion compare counts
- insertion elapsed time
- search probe counts
- search compare counts
- search elapsed time

Code

You can use any code from the lectures or from the textbook or from the Web. Be sure to give proper citations (names of books, URLs, etc.) if you use code that you didn't write yourself. Put the citations in your program comments.

What to submit

Submit into Canvas: Assignment #13:

- The signed zip file from CodeCheck. (Only do Part 1 in Canvas.)
- A text copy of the output from Part 1. (CodeCheck truncates the output in its report.)
- A text copy of insertion and search statistics and their graphs from Part 2.

Rubrics

Criteria	Maximum points
Part 1	**100**
• AVL tree rotations printed correctly.	• 30
• The AVL tree remains balanced after each node insertion.	• 35
• The AVL tree remains balanced after each node deletion.	• 35
Part 2	**100**
• BST insertion statistics	• 10
• AVL insertion statistics	• 10
• BST search statistics	• 10
• AVL search statistics	• 10
• Insertion probe counts graph	• 10
• Insertion compare counts graph	• 10
• Insertion elapsed time graph	• 10
• Search probe counts graph	• 10
• Search compare counts graph	• 10
• Search elapsed time graph	• 10