CMPT 225 SECTION D100

ASSIGNMENT #4: AVL TREES

DUE DATE WITH CANVAS: SUNDAY, JULY 27, 2025 AT 11:59 PM

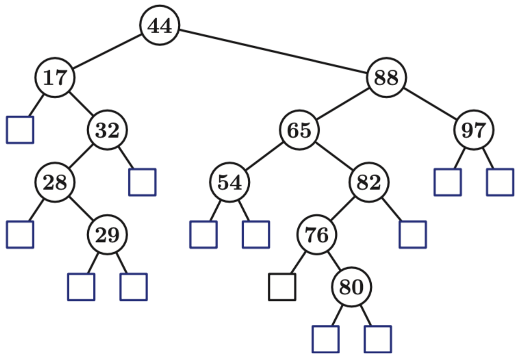
PROBLEM AT HAND

Given is a mostly complete Binary Search Tree (BST) and AVL tree. Our two goals in this assignment are:

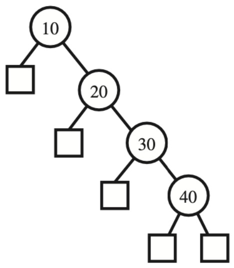
- Completing the BST and AVL classes
- Testing the performance of both BST and AVL

An AVL is a type of BST, so it is important that we first understand a BST. The rule of a BST is that:

- For any node v:
    - All keys in the left subtree must be less than the key of v
    - All keys in the right subtree must be greater than the key of v



For simplicity, we will not include duplicate keys in our tree.

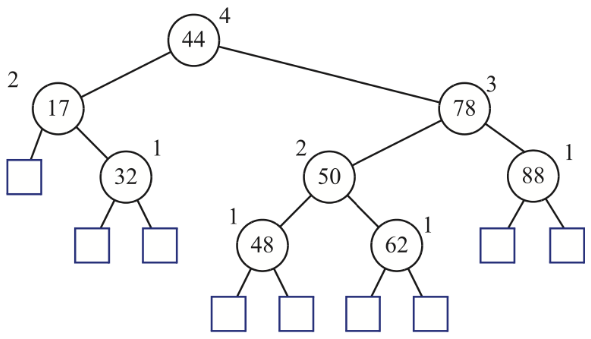


Our implementation only stores values in the internal nodes of the tree, and all external nodes are dummy nodes.

The major downside of these trees is the potential for imbalance. If keys are inserted in-order, then the tree can become similar to a linear linked list. This means with n keys in the tree, the height of the tree can be $h = n - 1$

AVL trees enforce the **Height-Balance Property**: "For every internal node v of T, the heights of the children of v differ by at most 1."

This gives us trees that can have their height bounded by $h < 2\log n + 2$. Since the search time of a BST and an AVL tree are both dependent on the height of the tree, AVL trees can give us a search time of $O(\log n)$. Much better than a traditional BST.

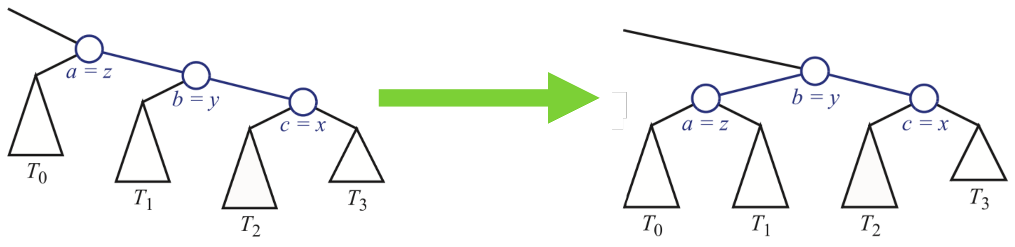


AVLs require more work when inserting keys to keep the height-balance property. Our job will be to implement aspects of the searching and inserting of BST and AVL trees.

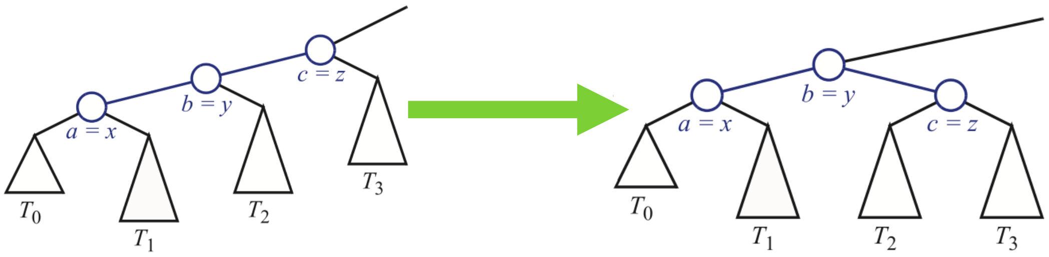
ASSIGNMENT INSTRUCTIONS

Read the code provided for the BST class and the AVL class to understand how functions are invoking other functions before beginning to implement the undefined functions.

- The BST class is defined in: BST.h and BST.cpp. The implementation of each function must be written in the BST.cpp. Do not change any of the current public members of BST. The functions you must complete are:
  - `Node* treeSearch(int k, Node* v) const;` - This private helper function returns the node containing the given key k. This must be written recursively. When first invoked by the public member function search `Node* v` is the root of the tree. With each recursive call, follow the path that would lead to the position of the key k, if it exists. If key k is not found, return the dummy node that would have contained the key.
  - `int getHeight(Node* node) const;` - This private helper function returns the height of the tree. This must be written recursively. When first invoked by the public member function getHeight `Node* v` is the root of the tree
- The AVL class is defined in: AVL.h and AVL.cpp. The implementation of each function must be written in the AVL.cpp. Do not change any of the current public members of AVL. The functions you must complete are:
  - `NodeAVL* treeSearch(int k, NodeAVL* v) const;` - This private helper function returns the node containing the given key k. This must be written recursively. When first invoked by the public member function search `Node* v` is the root of the tree. With each recursive call, follow the path that would lead to the position of the key k, if it exists. If key k is not found, return the dummy node that would have contained the key. This is identical to the BST version.
  - `void leftRotation(NodeAVL* x);` - Perform a single left rotation given the node x. For further details on the purpose of a left rotation, please see the textbook.



    - `void rightRotation(NodeAVL* x);` - Perform a single right rotation given the node x. For further details on the purpose of a right rotation, please see the textbook.



    - `void restructure(NodeAVL* z, NodeAVL* y, NodeAVL* x);` - Given an unbalanced node z, restructure must determine which type of rotation is required and perform such rotation. It must make use of the `leftRotation` and `rightRotation` functions. The four types of rotations are:
        - Single left rotation
        - Single right rotation
        - Double left-right rotation
        - Double right-left rotation.
- In the main.cpp, write test cases to determine if AVL performs better than the BST. For each both the BST and AVL tests, test with n = 1000, n = 10000, and n = 100000:
    - Test a worst-case tree with sorted numbers inserted
        - The height of the tree
        - The average time of a successful search
        - The average time of an unsuccessful search
    - Test an average case tree with numbers inserted in random order
        - The height of the tree
        - The average time of a successful search
        - The average time of an unsuccessful search
    - Note: how to test unsuccessful searches? Only insert even numbers in the tree, then search for odd numbers. The searches will always be unsuccessful, and this can test many different branches of the trees.
- To make and run the file, enter the commands:
    - `make`
- To clean up unneeded object files and executables, enter the command:
    - `make clean` (for windows)
    - `make remove` (for mac/linux)

TO SUBMIT WITH CANVAS AS A SINGLE ZIP FILE:

Remove any executables or object files from your project folder. Your folder should only contain:

- BST.h
- BST.cpp
- AVL.h
- AVL.cpp
- main.cpp
- makefile

Zip this folder and name it as: `firstname_lastname_studentID.zip` and submit this file on Canvas by the due date. Any last submission will result in a 10% penalty per day. No late assignments will be accepted after 48 hours.