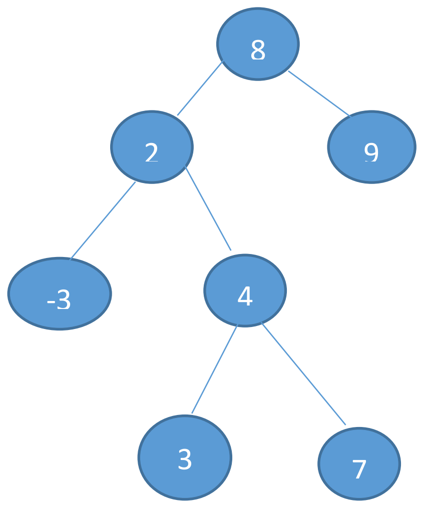
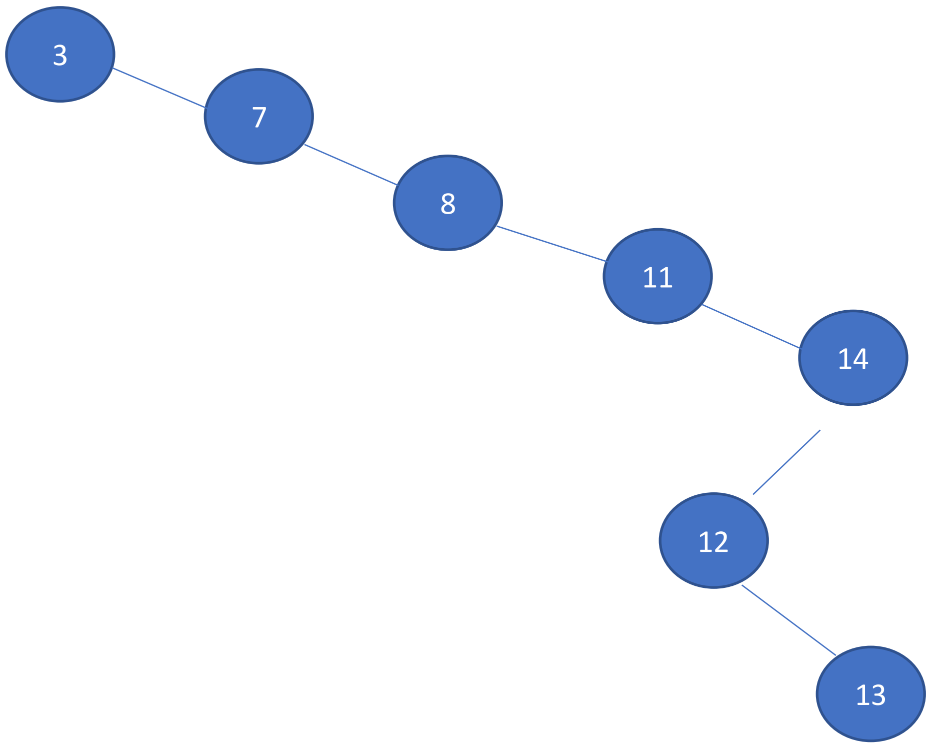
Lab09: Balanced Binary Search Trees: AVL

A **BINARY SEARCH** tree is a binary tree in which the data (keys) are stored in order such that all nodes to the right of node N have keys bigger than N and all nodes to the left of node N have keys smaller than N. BSTs give us a good data structure to implement the dictionary ADT (insert, find, delete, create, print). Here is a simple BST where the keys are ints:



Here is another example of a BST. In the worst-case, we have a tree that looks like a list:



Choose any node in the tree. Its left subtree descendants are less than the node's value. Its right subtree descendants are greater than the node's value.

Complexity

Insert new data node: how long does this take given a tree with N nodes?

What is the "worst-case" tree? (long, skinny)

May need to compare new data with every other node in tree

O(N)

Find data in tree: how long does this take given a tree with N nodes?

What is the "worst-case" tree?

O(N)

Delete node: how long does this taken given a tree with N nodes?

Find takes O(N) time; if not found, this is the complexity

Leaf case: O(N) to get to leaf, deleting leaf takes O(1); total: O(N)

Non-full interior node: O(N) to get partway down; updating pointer takes O(1); total: O(N)

Full interior node: O(N) to get partway down, O(N) to get to next successor, O(1) to copy data value, O(1) to delete leaf or do update pointer for non-full interior node

O(N)

How is this better than a linked list?

How could we re-order the insertions to make a complete tree (i.e., a tree that has most of the levels filled up)? Note that by creating complete trees, the height of the tree is closer to lgN instead of N for a tree with N nodes. That means find, insert, and delete run in time closer to O(lgN) instead of O(N).

How do we get Binary Search Trees that are complete?

Option 1: If we know all the data in advance, we can randomize the data before insertion. There is a high probability that this will give us a height lgN tree. (Try it…)

That does not work if we have an algorithm that uses trees where data is being inserted, searched for, and deleted over time. We need another approach.

In this lab, we explore and implement a self-balancing binary search tree. Whereas there are many techniques to balance a binary search tree, this lab explores a technique first proposed by Adelson-Velsky and Landis. The technique is named after them, and the ensuing balanced binary search tree is called an **AVL Tree**. The technique modifies the insertion method such that whenever a node is created and inserted there is no node in the entire tree whose right and left subtrees heights differ than height greater than 1.

In class, we identified 4 problem cases to solve when implementing an AVL tree.

- Left Left Case
- Left Right Case
- Right Right Case
- Right Left Case

These four cases, only need **two rotation techniques: Rotate Right and Rotate Left.**

In this lab, you have been provided with starter code as follows (Please pull the code from the class github repo).

- AVLMain.c – a driver program
- AVLtree.c – implementation file for all the functions used
- AVLtree.h – the ADT interface for Balanced Binary Search Trees

Your task is to implement the following four functions:

```
int getBalancefactor(TNode* node); //lab exercise   1
void postOrderTraversal(TNode *root)//lab exercise  2
TNode* rightRotation(TNode* nodey); //lab exercise  3
TNode* leftRotation(TNode* nodex); ; //lab exercise 4
```

Submission:

1) Make sure that your code runs in the khoury server
2) Push your entire code files (including the .h and the main driver) to your github repo.
3) During the ensuing lab or prior present the work to TA/Prof by performing a code walkthrough –
    a. Notice that there is a 20% penalty if code walkthrough is not carried out.
    b. Code walkthroughs helps you to verbally communicate your implementation strategy and helps professor and TA to give you instant feedback that can be implemented in your already finished work.

Please see in the next page, how BST balancing was carried out for the four problem cases.

Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - - ->    /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - - ->  /  \      - - - - - - - - -> / \    / \
T1   x                          y    T3                   T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

Right Right Case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - - ->  / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - - ->   /  \   - - - - - - - - -> / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                                  /  \
T2   T3                               T3   T4
```