# Data Structures and Algorithms — CSci 230
## Chapter 4 — Trees

## Overview

- We will cover sections 4.1 through 4.4 in class.
- We assume you have been introduced to binary search trees. These will not be covered in detail until we get to balanced trees — AVL trees.

## Basics

- Difference between a **rooted tree** and a **tree**. This chapter studies only rooted trees.
- Terminology:
  - node, root, internal node, leaf node
  - edge, path
  - parent, child, sibling
  - depth of a node, height of a node, height of a tree
  - m-ary trees
  - binary trees
- Implementation of the nodes in a rooted tree:
  - trees with no limit on the number of children at any given node,
  - binary trees.
- Pre-order vs. In-order vs. Post-order traversals of trees.
- Analyzing the time required for a tree algorithm.

## Exercises on the Basics

Here are two exercises exploring properties of rooted trees.

1. A *full* $m$-ary tree is a tree where each internal node has exactly $m$ children. One interesting property of full $m$-ary trees is that you can write a formula relating the number of internal nodes $i$, the number of leaf nodes $l$, and the total number of nodes $n$ (obviously, $n = i + l$). If a $m$-ary tree has $i$ internal nodes, how many leaf nodes does it have? (Hint: start from a tree having just a root node (it is a leaf node), then successively change leaf nodes into internal nodes and determine how $i$ and $l$ change.) Prove your answer by mathematical induction on $i$, the number of internal nodes in the full $m$-ary tree.

2. Sketch an algorithm to calculate the height of a node in an $m$-ary tree. What is the time complexity of your algorithm? What would the time complexity be for an algorithm computing the height of all nodes in the tree?

## Binary Search Trees

- Binary search trees represent our first step toward a data structure allowing logarithmic time insertion, deletion and search.
- **Definition**: a binary search tree is a binary tree such that at each internal node, $v$, the value stored at $v$ is greater than (or equal) any value stored in $v$'s left subtree and is less than (or equal) any value stored in $v$'s right subtree.
- Review of key operations: **Search**, **Insert** and **Remove**.
- See the text for code. The only code from Chapter 4 that we will discuss in detail is the code for AVL trees.

## Exercises on Binary Search Trees

- Draw a binary search tree after inserting the following integer values, in order: 15, 3, 19, 12, 8, 7, 2. Show the state of the tree after deleting 8 and 15.
- Suppose a binary tree is created that contains $n$ nodes.
  - What is the maximum possible height of the tree? What type of input causes this height to occur?
  - What is the minimum possible height of the tree? Hint: remember that we proved a complete binary tree of height $h$ contains exactly $2^{h+1} - 1$ nodes.

## AVL trees

- AVL trees are a type of binary search tree that **guarantees** each search, insert, and remove operation on a tree storing $n$ values (containing $n$ nodes), will require a worst-case of $O(\log n)$ time.
  - Red-black trees are another type of binary search tree with the similar time guarantees. These are used in the implementation of STL `set`, `map` and `multimap`.
  - Splay-trees have a slightly different but related time guarantee.
- At each node in an AVL tree the difference in height between the left subtree and the right subtree is guaranteed to be -1, 0 or 1.
  - Note that leaves are considered height 0 and null subtrees (null pointers from a tree node) height -1.
- Maintaining this guarantee requires special "rotation" operations: single rotations and double rotations. These operations distinguish AVL trees from ordinary binary search trees.
- Insertion operations will require at most one rotation. Remove operations may require at most $O(\log n)$ rotations. This worst-case behavior is rare.
- The maximum height of an AVL tree containing $n$ nodes is approximately $1.44 \log n$.
- A modified version of the code from the Weiss text will be is attached here and will be discussed in detail in class. This includes code to remove nodes from the trees, which Weiss does not provide.

  The source code for the demonstrations done in class is available in

  ```
  /dept/cs/cs230/public/examples/avl_tree
  ```

## AVL Tree Exercises

1. Starting from an empty tree, show the structure of the tree after each of the following is inserted: 30, 50, 70, 80, 90, 15, 85, 97, 82. Draw the tree after each insert (before rebalancing) and after rebalancing. Write the heights of the subtrees if it helps.

2. Show the state of the tree after removing 15.

3. In examining the AVL tree code, you will notice that pointers to tree nodes are frequently passed by reference. For example, look at the function `Insert`. Why is this necessary?

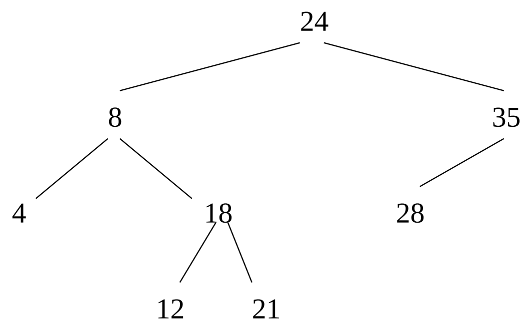
4. Create an AVL tree that will require two rotations after a certain node is deleted. The smallest one I could create contained 12 nodes. Hint: you will need to make the tree as unbalanced as possible while still maintaining the AVL property.

## Review Problems

Here are review exercises for trees. Not covered by these exercises are problems involving maps and multimaps from the standard library. Since these were the subject of Lab 5, they are "fair game" for the quiz.

1. Suppose each non-leaf node of a $m$-ary has exactly $m$ children — in other words, each node of the tree has either exactly 0 or exactly $m$ children. Suppose we build such a tree of height $h \geq 0$ having the **minimum** possible number of nodes. What is the number of nodes in this "minimal" tree? Prove your answer using mathematical induction.

2. Consider the following binary search tree.

   Assuming it is an AVL tree, show the state of the tree after inserting 10 and then again after deleting 28.

3. Given an empty AVL tree of integers, show the structure of the tree after each of the values 5, 2, 4, 6, 7, 1, 8 is inserted and then show the change to the resulting tree when 4 is deleted. Indicate where rotations are done to rebalance the tree.

4. Given a binary search tree containing $n$ floating point values and having a height that is $O(\log n)$ (i.e. it is balanced), write an algorithm to count the number of nodes storing values greater than `x0` and less than `x1`. What is the running time of your algorithm? (More credit will be given for an efficient algorithm.)

   You may assume the following declaration for tree nodes:

   ```
   struct TreeNode {
      float x;
      TreeNode * left;
      TreeNode * right;
   }
   ```

   Start from the following prototype and assume the function is initially passed a pointer to the root:

```
int Count( TreeNode * T, float x0, float x1 )
```

5. It is possible to turn a sorted array of $N$ floating point values into an AVL tree that is as balanced as possible in $O(N)$ time. Write a function to do this. (Hint: think about which value should be stored at the root.) Assume the following structure declaration for AVL nodes.

```
struct Avl_Node {
  float Element;
  Avl_Node *Left;
  Avl_Node *Right;
  int Height;
};
```

Your function should calculate the heights of the nodes and it should return a pointer to the root of the tree. You do not need to prove that the result is an AVL tree and you do not need to prove the function requires $O(N)$ time. Here is a prototype

```
Avl_Node * ArrayToAVLTree( float values[], int N  )
```