# CSE 674 Advanced Data Structures

## Binary Search Trees

Andrew C. Lee

EECS Dept, Syracuse U.

# Contents

We will discuss

1. The implementation of the main operations of a Binary Search Trees
2. Introduction to AVL Trees

# Search Trees: Introduction

1. A tree structures that store objects that are identified by a key
2. two keys can be compared in constant time
3. comparisons are used to guide the search
4. Often used to implement a dictionary data structure
5. have different models
    - go left when the query key is smaller than node key; otherwise take the right branch
    - go left when the query key is smaller than node key; go right if the query key is larger than the node key and take the object store in the node if they are equal

# Binary Search Trees

In an object oriented langauage (e.g. `C++`), Binary Search Trees are typically implemented via two classes:

1. A class that represent a node
2. A class that represent the binary search tree

**Example** We will walk through a typical textbook example here. Note that the implementation of the major operations are often divided into several functions

# Binary Search Trees

Recall that the following questions last time:
the three major operations are:

1. Insertion
2. Deletion
3. Search

Can you write down the pseudocode of the following operations ?

# The contains (Search) operation

## Discussions

```
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
```

**Figure 4.18** contains operation for binary search trees

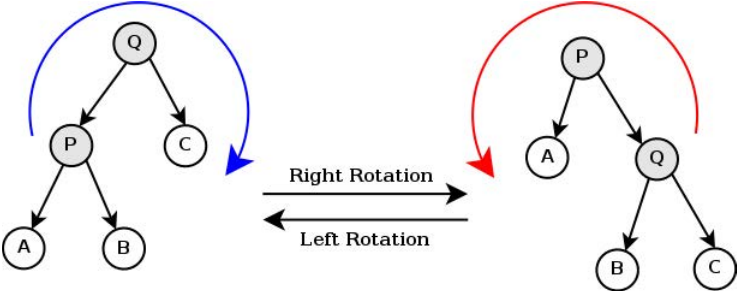
# Binary Search Trees

What are the worst case running time of the following operations if the binary search trees contain $n$ keys ?

1. Insert
2. delete
3. search

How can we improve the performance of these operations?

# Rotations




Note: The ordering of the nodes (in-order) is preserved but the height of the tree may be changed

# DSW Algorithm

**Definition** A binary search tree $T$ is said to be balanced when the difference between the heights of its subtrees (i.e. left and right subtrees) is at most 1.
DSW algorithm will turn a binary search tree into a balanced binary search tree in $O(n)$ time.
(Example handout provides an example and the pseudocode for DSW.
**Question** When will we apply DSW algorithm ?

# AVL Trees: a self-balancing Binary Search Tree

AVL Trees

1. named after Adelson, Velskii and Landis
2. It show that, we can applying balancing technical incrementally so that the binary search trees is guaranteed to have $O(lg\ n)$ height, irrespective of the order of insertions and deletions
3. It uses the following *balance condition*:
   For every node of the tree, the height of its left subtree and right subtree differ by at most 1.
4. The height of the empty tree is -1, by convention

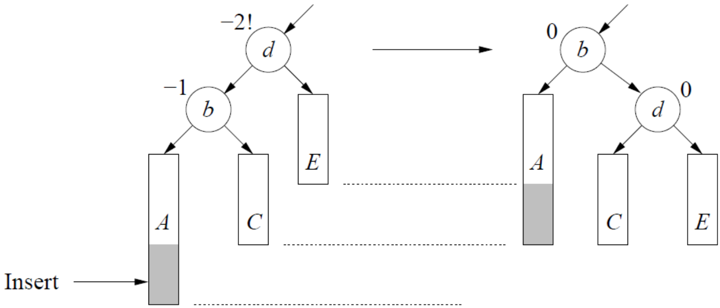
# AVL: Rebalancing Operations

Main idea for AVL tree

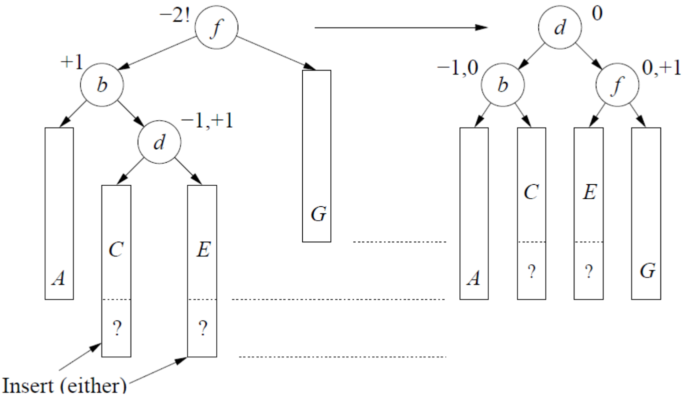
1. keep track of balancing information
2. Each time after an operation (insert or delete) is performed, if it cause imbalance, apply an appropriate rebalancing operations (a suitable type of rotations)
3. Type of rotations: single rotation (left, right) ; double rotation (left-right, right-left);

Let''s begin with an animation:
`https://www.youtube.com/watch?v=aQS9DqLWxw4`

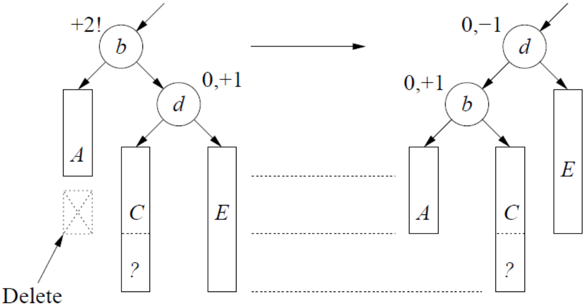
# Rebalancing operations I: Single Rotation for insertion



# Rebalancing operations II: Left-Right Double Rotation for insertion



# Rebalancing operations III: Single Rotation (deletion)



# Rebalancing operations IV: Double Rotation (deletion)