AVL Trees

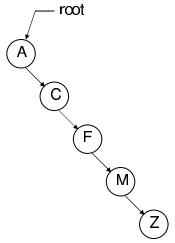
CS 367 – Introduction to Data Structures

Binary Tree Issue

- One major problem with the binary trees we have discussed up to now
  - they can become extremely unbalanced
    - this will lead to long search times
  - in the worst case scenario, inserts are done in order
    - this leads to a linked list structure
  - possible O(n) performance
    - this is not likely but it's performance will tend to be worse than O(log n)

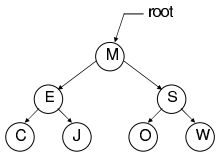
Binary Tree Issue

```
BinaryTree tree = new BinaryTree();
tree.insert(A);
tree.insert(C);
tree.insert(F);
tree.insert(M);
tree.insert(Z);
```



Balanced Tree

- In a perfectly balanced tree
  - all leaves are at one level
  - each non-leaf node has two children

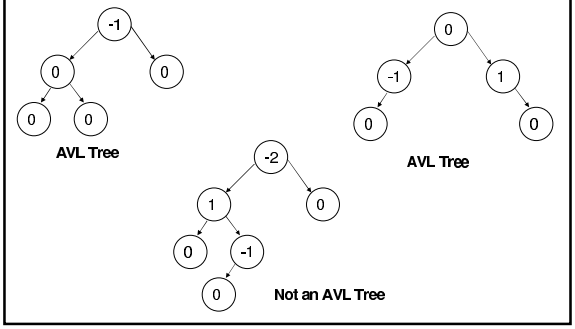


- Worst case search time is log n
  - n is the number of levels – 1

AVL Tree

- AVL tree definition
  - a binary tree where the maximum difference in the height of any nodes right and left sub-trees is 1 (called the balance factor)
    - balance factor = height(right) – height(left)
- AVL trees are usually not perfectly balanced
  - however, the biggest difference in any two branch lengths will be no more than one level

AVL Tree



AVL Tree Node

- Very similar to regular binary tree node
  - must add a balance factor field
- For this discussion, we will consider the key field to also be the data
  - this will make things look slightly simpler
    - they will be confusing enough as it is ☹

AVL Tree Node

```
class TreeNode {
    public Comparable key;
    public TreeNode left;
    public TreeNode right;
    public int balFactor;

    public TreeNode(Comparable key) {
        this.key = key;
        left = right = null;
        balFactor = 0;
    }
}
```

Searching AVL Tree

- Searching an AVL tree is exactly the same as searching a regular binary tree
  - all descendants to the right of a node are greater than the node
  - all descendants to the left of a node are less than the node

Searching an AVL Tree

- Psuedocode

```
Object search(Comparable key, TreeNode subRoot) {
  I) if subRoot is null, empty tree or key not found
       A) return null
  II) compare subRoot's key (Kr) to search key (Ks)
       A) if Kr < Ks
            -> recursively call search with right subTree
       B) if Kr > Ks
            -> recursively call search with left subTree
       C) if Kr == Ks
            -> found it! return data in subRoot
}
```

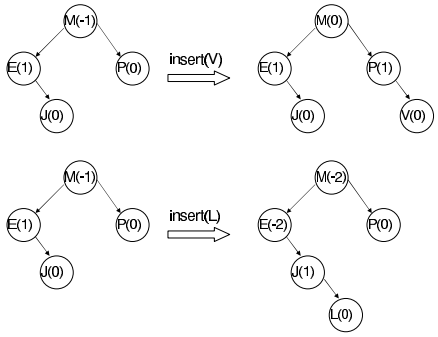
Inserting in AVL Tree

- Insertion is similar to regular binary tree
  - keep going left (or right) in the tree until a null child is reached
  - insert a new node in this position
    - an inserted node is *always* a leaf to start with
- Major difference from binary tree
  - must check if any of the sub-trees in the tree have become too unbalanced
    - search from inserted node to root looking for any node with a balance factor of ±2

Inserting in AVL Tree

- A few points about tree inserts
  - the insert will be done recursively
  - the insert call will return true if the height of the sub-tree has changed
    - since we are doing an insert, the height of the sub-tree can only increase
  - if *insert()* returns true, balance factor of current node needs to be adjusted
    - balance factor = height(right) – height(left)
      - left sub-tree increases, balance factor decreases by 1
      - right sub-tree increases, balance factor increases by 1
  - if balance factor equals ±2 for any node, the sub-tree must be rebalanced

Inserting in AVL Tree



This tree needs to be fixed!

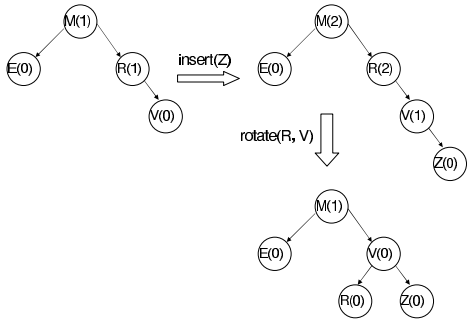
Re-Balancing a Tree

- To check if a tree needs to be rebalanced
  - start at the parent of the inserted node and journey up the tree to the root
    - if a node's balance factor becomes ±2 need to do a rotation in the sub-tree rooted at the node
    - once sub-tree has been re-balanced, guaranteed that the rest of the tree is balanced as well
      - can just return false from the *insert()* method
  - 4 possible cases for re-balancing
    - only 2 of them need to be considered
      - other 2 are identical but in the opposite direction

Re-Balancing a Tree

- Case 1
  - a node, N, has a balance factor of 2
    - this means it's right sub-tree is too long
  - inserted node was placed in the right sub-tree of N's right child, N_{right}
    - N's right child have a balance factor of 1
  - to balance this tree, need to replace N with it's right child and make N the left child of N_{right}

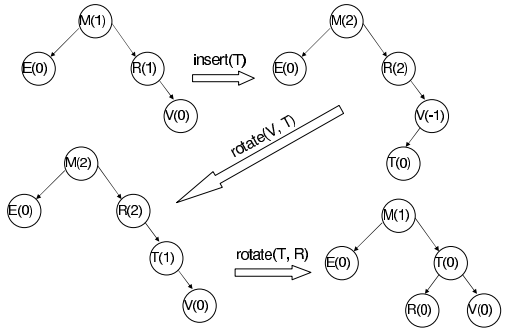
Case 1



Re-Balancing a Tree

- Case 2
  - a node, N, has a balance factor of 2
    - this means it's right sub-tree is too long
  - inserted node was placed in the left sub-tree of N's right child, N_{right}
    - N's right child have a balance factor of -1
  - to balance this tree takes two steps
    - replace N_{right} with its left child, $N_{grandchild}$
    - replace N with it's grandchild, $N_{grandchild}$

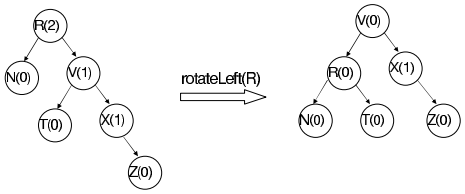
Case 2



Rotating a Node

- It can be seen from the previous examples that *moving* nodes really means rotating them around
  - rotating left
    - a node, N, has its right child, N_{right}, replace it
    - N becomes the left child of N_{right}
    - N_{right}'s left sub-tree becomes the right sub-tree of N
  - rotating right
    - a node, N, has its left child, N_{left}, replace it
    - N becomes the right child of N_{left}
    - N_{left}'s right sub-tree becomes the left sub-tree of N

Rotate Left



Re-Balancing a Tree

- Notice that Case 1 can be handled by a single rotation left
  - or in the case of a -2 node, a single rotation right
- Case 2 can be handled by a single rotation right and then left
  - or in the case of a -2 node, a rotation left and then right

Rotate Left

- Psuedo-Code

```
void rotateLeft(TreeNode subRoot, TreeNode prev) {
  I) set tmp equal to subRoot.right
        A) this isn't necessary but it makes things look nicer
  II) set prev equal to tmp
        A) be careful – must consider rotating around root
  III) set subRoot's right child to tmp's left child
  IV) set tmp's left child equal to subRoot
  V) adjust balance factor
        subRoot.balFactor -= (1 + max(tmp.balFactor, 0));
        tmp.balFactor -= (1 – min(subRoot.balFactor, 0));
}
```

Re-Balancing the Tree

- Psuedo-Code

```
void balance(TreeNode subRoot, TreeNode prev) {
  I) if the right sub-tree is out of balance (subRoot.factor = 2)
        A) if subRoot's right child's balance factor is -1
              -> do a rotate right and then left
        B) otherwise (if child's balance factor is 1 or 0)
              -> do a rotate left only
  I) if the left sub-tree is out of balance (subRoot.factor = -2)
        A) if subRoot's left child's balance factor is 1
              -> do a rotate left and then right
        B) otherwise (if child's balance factor is -1 or 0)
              -> do a rotate right only
}
```

Inserting a Node

- Psuedo-Code

```
boolean insert(Comparable key, TreeNode subRoot, TreeNode prev) {
   I) compare subRoot.key to key
        A) if subRoot.key is less than key
              1) if subRoot doesn't have a right child
                    -> subRoot.right = new node();
                    -> increment subRoot's balance factor by 1
              2) if subRoot does have a right subTree
                    -> recursively call insert with right child
                    -> if true is returned, increment balance by 1
                    -> otherwise return false
        B) if the balance factor of subRoot is now 0, return false
        C) if ballance factor of subRoot is 1, return true
        D) otherwise, the balance factor of subRoot is 2
              1) rebalance the tree rooted at subRoot
}
```