AVL Tree (1962)

Georgy M. Adelson-Velsky
1922-2014

Evgenii Mikhailovich Landis
1921-1997

AVL Trees

- A balanced binary search tree (BST).
- The insert and delete operations are more complicated.
  - Need to maintain the balanced property.
- Remains fairly simple to understand and implement.
- Worst case for searching is now O(log N).

AVL Tree: Insertion

- Recall: Insertion in BST

```
void InsertItem(Tree &tree, int Data){
   if (tree == 0)
      tree = MakeNode(Data);
   else if (Data < tree->data)
      InsertItem(tree->left, Data);
   else if (Data > tree->data)
      InsertItem(tree->right, Data);
   else
      cout << "Error, duplicate item" << endl;
}
```

AVL Tree: Insertion

1. Insert the item into the tree using the same algorithm for BSTs. Call this new node x.
   - While traversing the tree looking for the appropriate insertion point for x, push the visited nodes onto a stack. (Actually, you are pushing pointers to the nodes.)
   - It is not necessary to push x onto the stack.

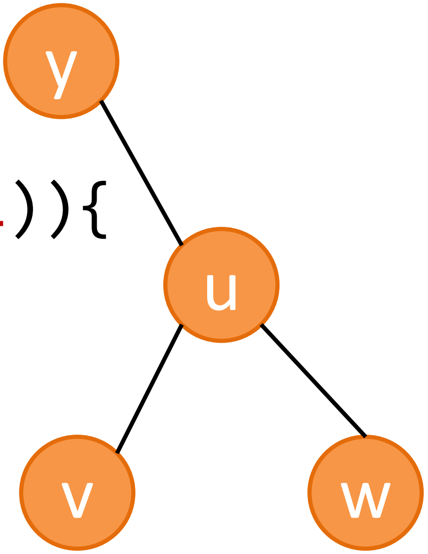
2. Check if there are more nodes on the stack.
   a) If the stack is empty, the algorithm is complete and the tree is balanced.
   b) If any nodes remain on the stack, go to step 3.

3. Remove the top node pointer from the stack and call it y

4. Check the height of the left and right subtrees of y.
   a) If they are equal or differ by no more than 1 (hence, balanced), go to step 2.
   b) If they differ by more than 1, perform a rotation on one or two nodes as described below. After the rotation(s), the algorithm is complete and the tree is balanced.

AVL Tree: Balancing

- Compute the height of the left and right subtree of y:

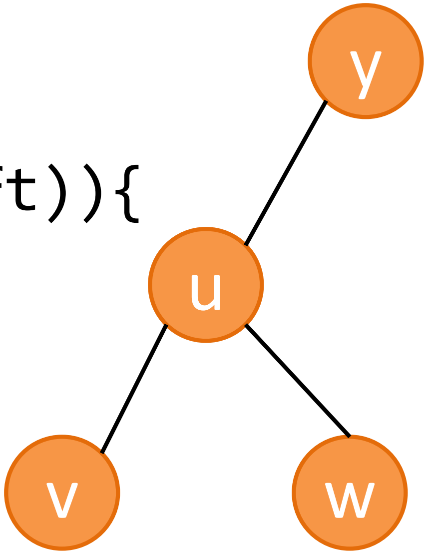



```
if(height(y->right)>(height(y->left)+1)){
   if(height(v)>height(w))
      //Promote v twice
   else
      //Promote u
}
```

AVL Tree: Balancing

- Compute the height of the left and right subtree of y:

```
if((height(y->right)+1)<height(y->left)){
   if(height(v)>height(w))
      //Promote u
   else
      //Promote w twice
}
```



AVL Tree: Insertion

- Insert: 1, 2, 3, 4, 5, 6, 7, 8
- Insert: 5, 2, 9, 8, 12, 7

AVL Tree: Partial Implementation

```
// Client calls this instead of InsertItem
void InsertAVLItem(Tree &tree, int Data){
  stack<Tree *> nodes;
  InsertAVLItem2(tree, Data, nodes);
}
```

```
void BalanceAVLTree(stack<Tree *> nodes){
  while (!nodes.empty()){
    Tree *node = nodes.top();
    nodes.pop();

    // implement algorithm using functions that
    // are already defined (Height, RotateLeft, RotateRight)
}
```

AVL Tree: Partial Implementation

```
// Auxiliary function with the stack of visited nodes
void InsertAVLItem2(Tree &tree, int Data, stack<Tree*> nodes){
  if (tree == 0){
    tree = MakeNode(Data);
    BalanceAVLTree(nodes); // Balance it now
  }
  else if (Data < tree->data){
    nodes.push(&tree); // save visited node
    InsertAVLItem2(tree->left, Data, nodes);
  }
  else if (Data > tree->data){
    nodes.push(&tree); // save visited node
    InsertAVLItem2(tree->right, Data, nodes);
  }
  else
    cout << "Error, duplicate item" << endl;
}
```

Height of AVL Tree

- In AVL trees, the subtrees of each node differ by at most 1 in their height.

- Claim: The height of AVL trees is O(log N), where N is the number of nodes in the tree.

AVL Tree: Deletion

- Similar to insertion algorithm.
  - Delete as you would delete a node in a BST.
  - Push the visited nodes onto a stack.
  - While the stack is not empty
    - Pop item from stack -> y
    - Balance the tree if necessary
    - Continue until stack is empty
- Note: we need to continue until the stack is empty.

AVL Tree: Deletion

- Insert: 17, 6, 20, 3, 14, 19, 26, 5, 7, 16, 23, 11
- Delete 19
- Delete 17