AVL Trees

A simple balanced tree

AVL trees

- Invented by two Russian Mathematicians: Adel'son – Vel'skii and Landis.
- Essentially 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 to understand and implement.
- Worst case for searching is now **O(lgN)**

AVL Tree: Insertion

1. Insert the item into the tree using the same algorithm for BSTs. Call this new node x.
    1. While traversing the tree looking for the appropriate insertion point for x, pushed 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
    1. If the stack is empty, the algorithm is complete and the tree is balanced.
    2. If any nodes remain on the stack, **go to step 3.**
3. Remove the top node pointer from the stack (pop) and call is y.
4. Check the height of the **<u>left</u>** and **<u>right</u>** subtrees of y
    1. If they are equal or differ by no more than **1**(hence, balanced) **go to step 2.**
    2. If they differ by more than 1, perform a rotation on one or two nodes as described in the next slide. After the rotation(s), the algorithm is complete and the tree is balanced.

AVL Tree: Balancing

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

$$h(y \rightarrow left), h(y \rightarrow right)$$

$if\,(h(y \rightarrow right) > h(y \rightarrow left))$

- $u = y \rightarrow right$
- $v = u \rightarrow left$
- $w = u \rightarrow right$
- $if\big(h(v) > h(w)\big)$
    - Promote v twice(Rotate **right** around u, Rotate **left** around y).
- Otherwise:
    - Promote u(Rotate **left** around y).

AVL Tree: Balancing (2)

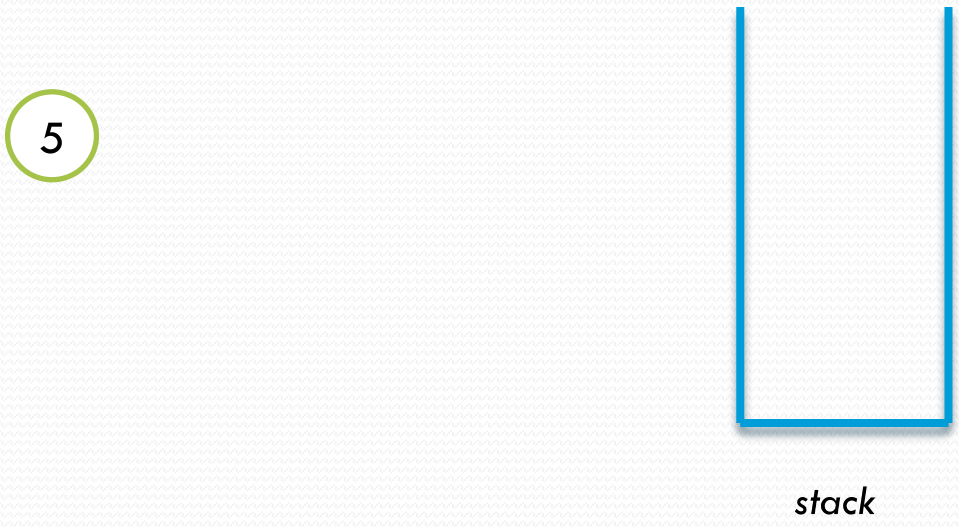
- The algorithm for balancing is symmetrical:

$if(h(y \rightarrow left) > h(y \rightarrow right))$

- $u = y \rightarrow left$
- $v = u \rightarrow left$
- $w = u \rightarrow right$
- $if\left(h(v) > h(w)\right)$
  - Promote u(Rotate **right** around y).
- Otherwise:
  - Promote w twice(Rotate **left** around u, Rotate **right** around y).

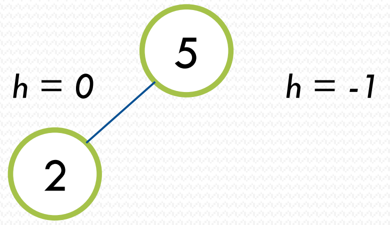
Example: Insert “5, 2, 9, 8,12, 7”

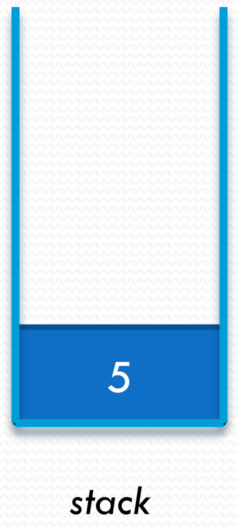
- Inserting 5 at root.



Example: Insert “5, 2, 9, 8,12, 7”

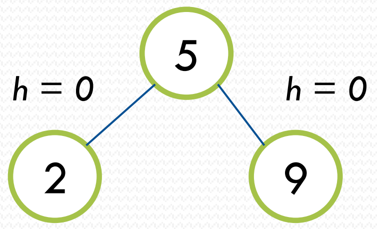
- Inserting 2:

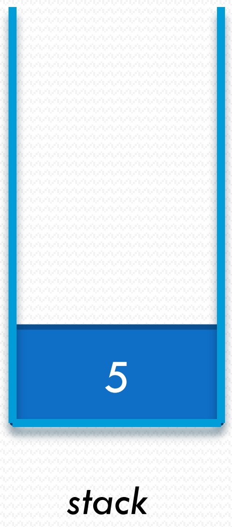




Example: Insert "5, 2, 9, 8,12, 7"

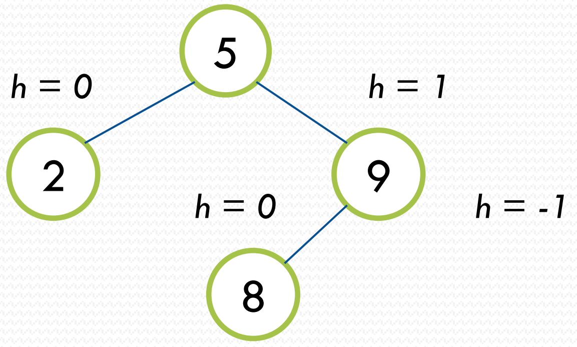
- Inserting 9:

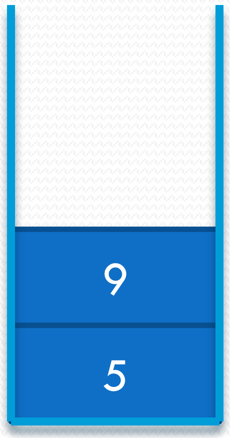




Example: Insert "5, 2, 9, 8,12, 7"

- Inserting 8:

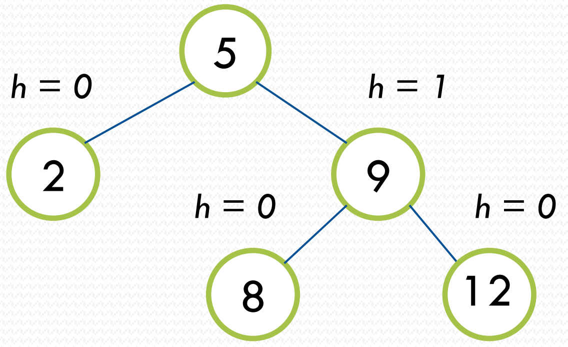


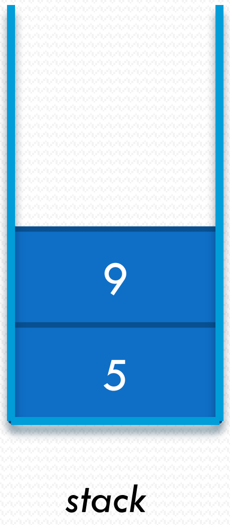


stack

Example: Insert “5, 2, 9, 8,12, 7”

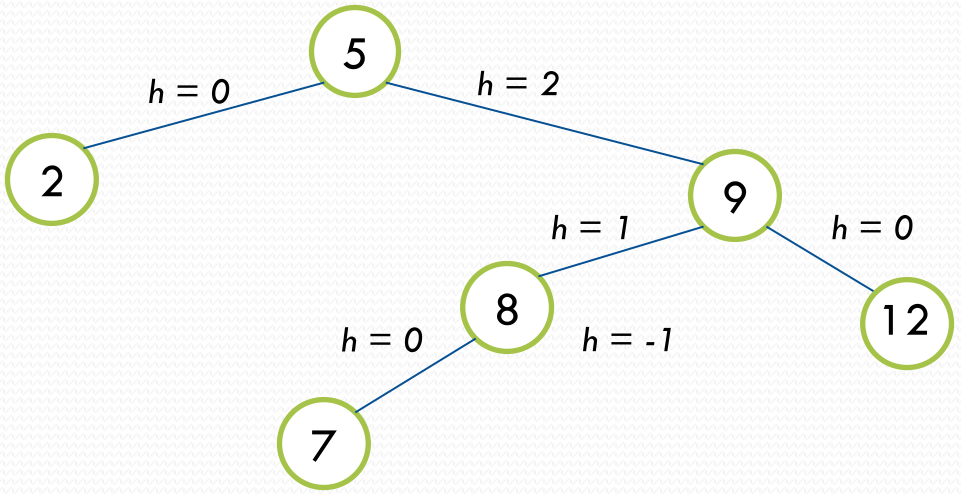
- Inserting 12:

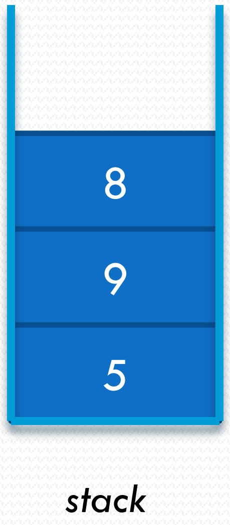




Example: Insert “5, 2, 9, 8,12, 7”

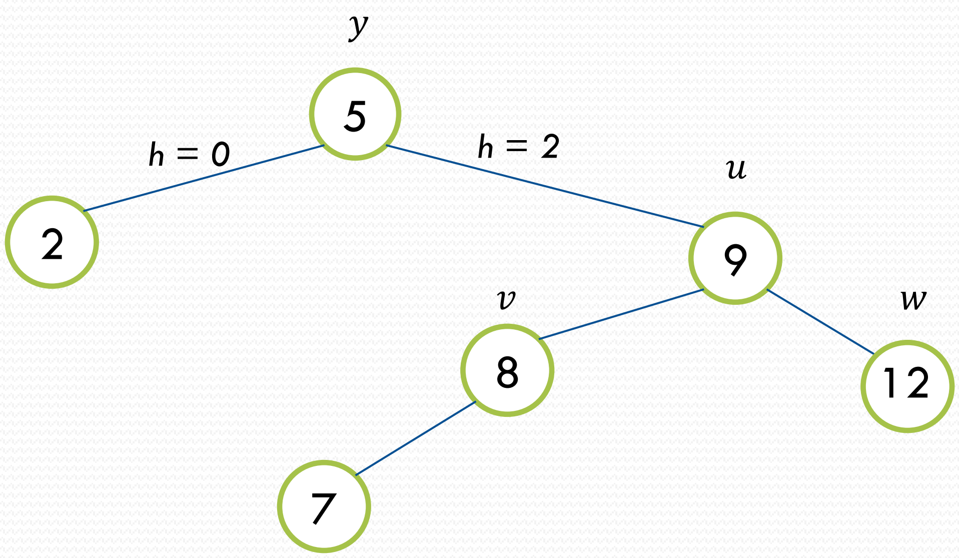
- Inserting 7:

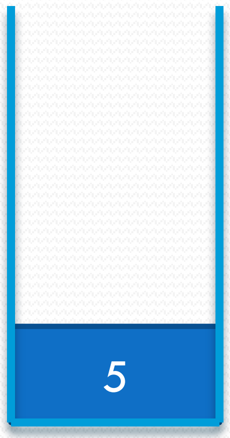




Example: Insert “5, 2, 9, 8,12, 7”

- Rebalancing Tree: $h(y \rightarrow right) > h(y \rightarrow left)$
  - $y = 5, u = y \rightarrow right, v = u \rightarrow left, w = u \rightarrow right$

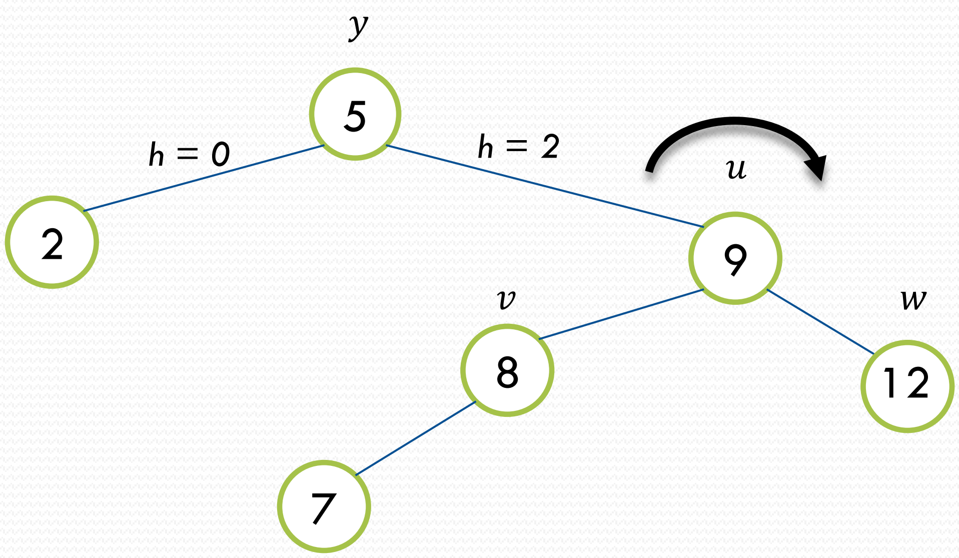


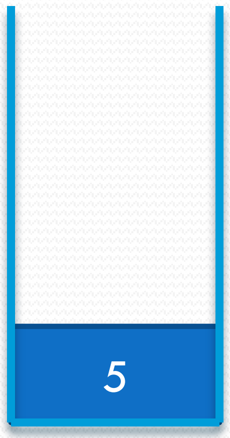


stack

Example: Insert "5, 2, 9, 8,12, 7"

- Rebalancing Tree: $h(y \rightarrow right) > h(y \rightarrow left)$
  - $h(v) > h(w) \rightarrow Promote\ v\ twice$



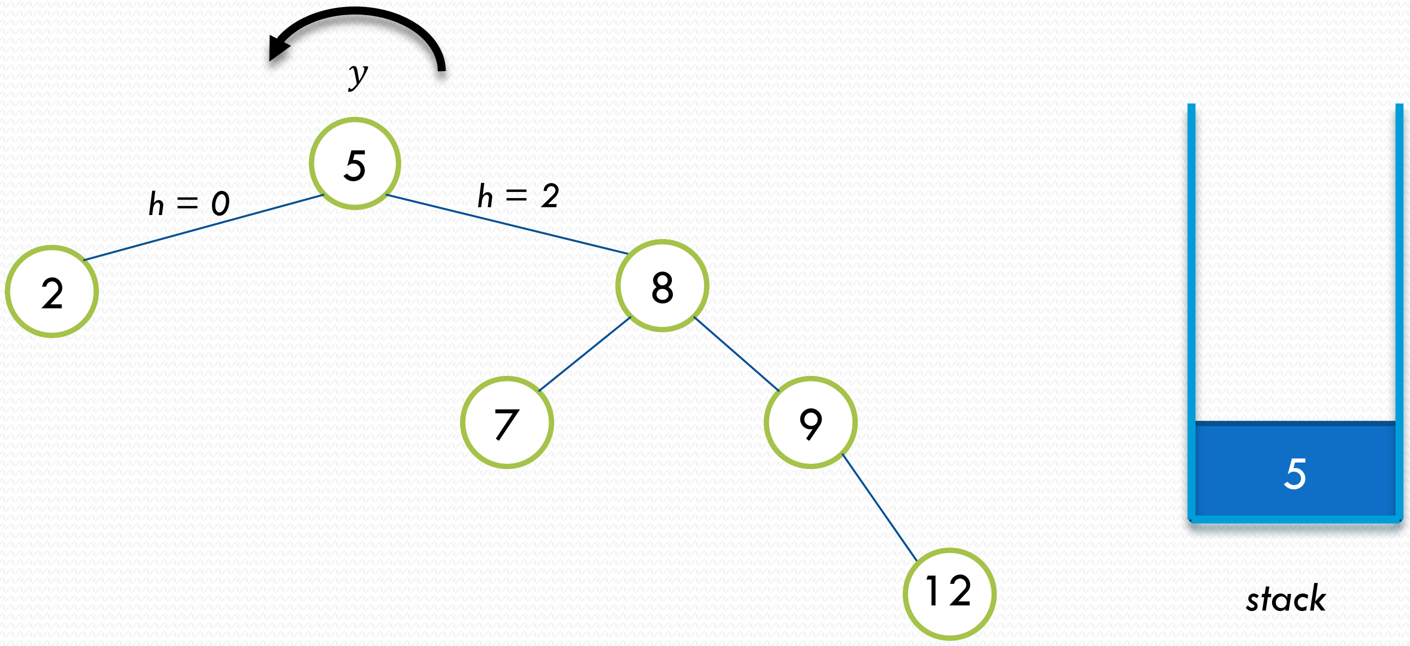


stack

Rotate RIGHT around u

Example: Insert “5, 2, 9, 8,12, 7”

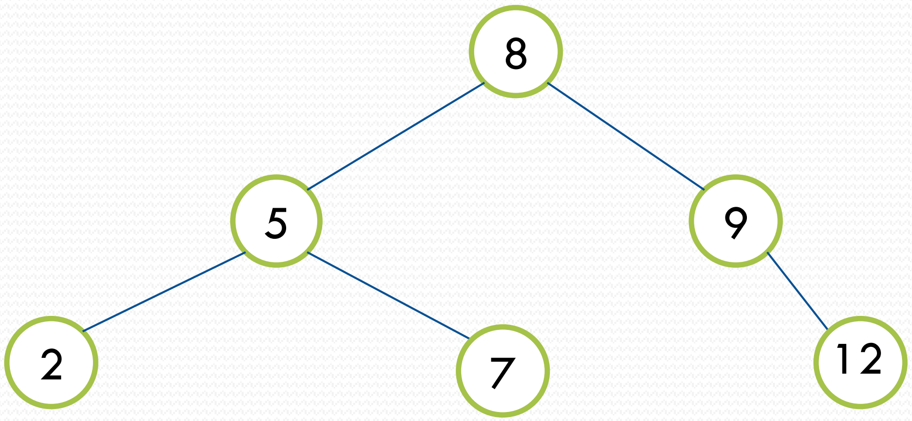
- Rebalancing Tree: $h(y \rightarrow right) > h(y \rightarrow left)$
  - $h(v) > h(w) \rightarrow Promote\ v\ twice$

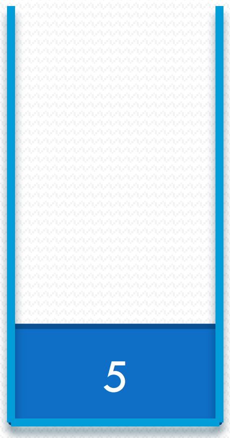


Rotate LEFT around y

Example: Insert “5, 2, 9, 8,12, 7”

- Rebalancing Tree: $h(y \rightarrow right) > h(y \rightarrow left)$
  - $h(v) > h(w) \rightarrow Promote\ v\ twice$



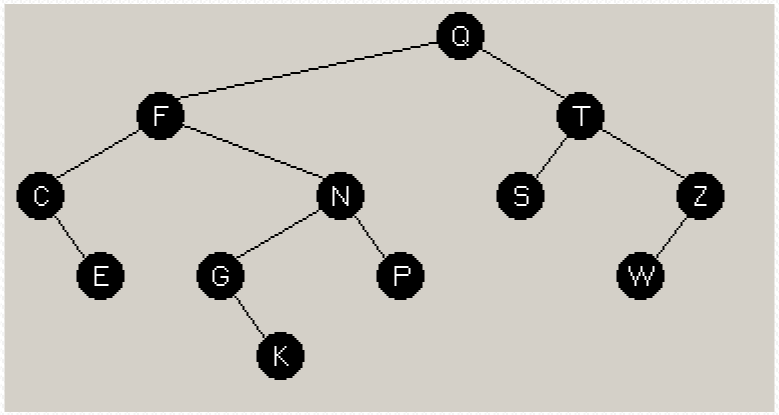


stack

AVL Tree: Deleting

- Deleting a node in an AVL tree is very similar to inserting.
  - Delete as you would delete a node in a BST.
    - Push down the stack every visited node
  - While the stack is not empty
    - Pop item from stack $\rightarrow y$
    - Balance the tree if needed
    - Continue until stack is empty
- Exactly the same as inserting algorithm, except we continue until the stack is empty.

Deleting Example:

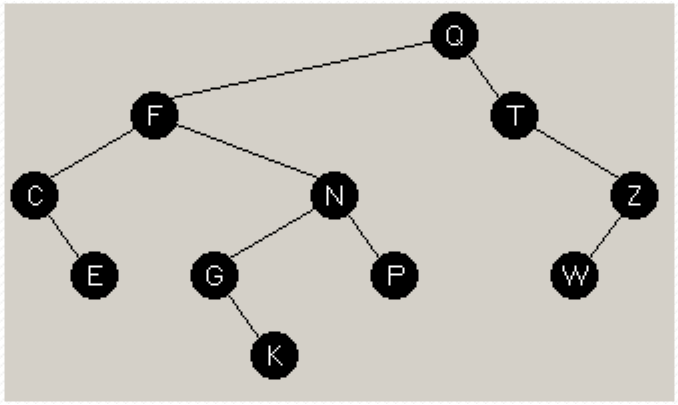


The first picture shows the initial tree created by inserting these letters in this order:

Q F T C N S Z E G P W K

This is a valid AVL tree with 12 nodes and height 4.

Deleting Example:

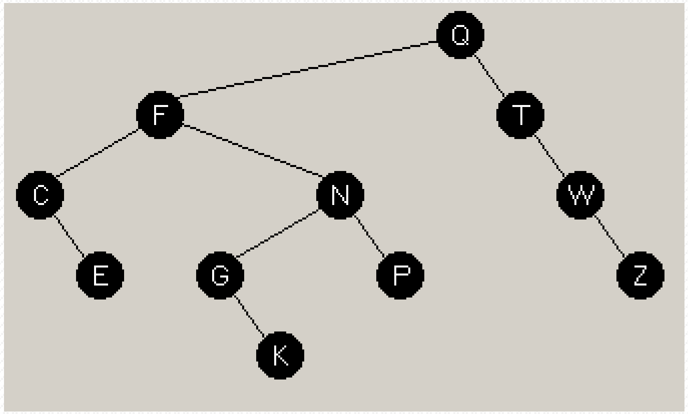


If you delete 'S', you get the picture on the left.

Now, walking up the tree (to make sure all nodes are still balanced) we get to the first node, 'T', and see that it is out of balance.

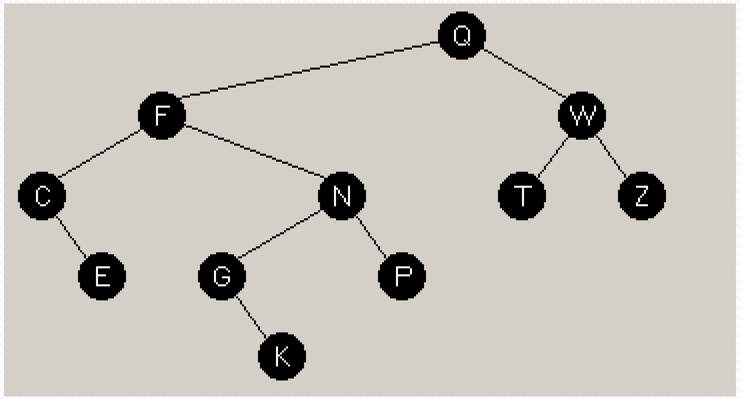
To balance it, we have to do 2 rotations. (It's a zig-zag situation.)

Deleting Example:



So, we rotate right about Z (promote node W), giving us the picture on the left.

Deleting Example:



Then, we rotate left about T (promote node W again), giving us the picture on the left.

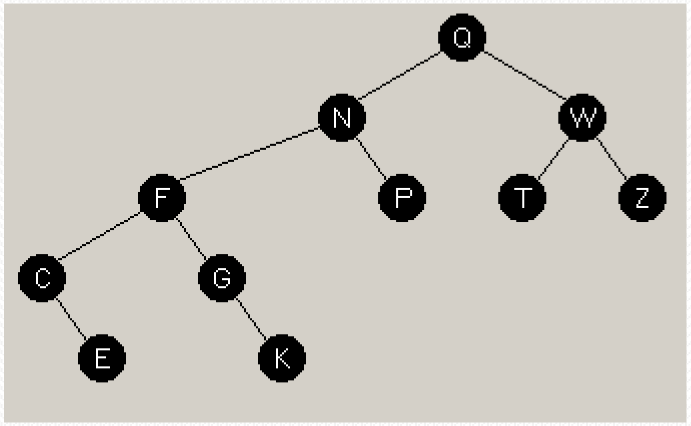
The tree is still unbalanced at the root.

Height of the left subtree rooted at node Q is 3 and the height of the right subtree rooted at node Q is 1.

This node is not balanced. We discovered this as we continued up the tree.

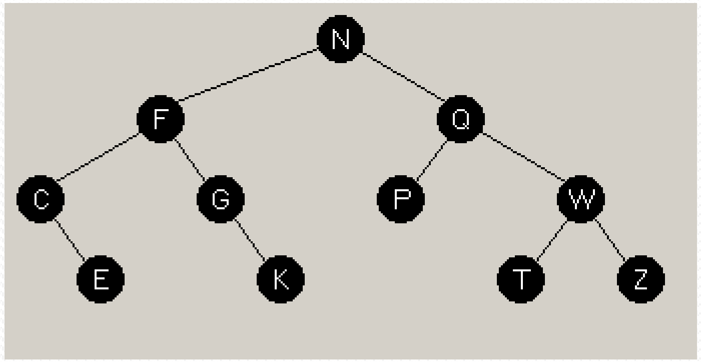
The left subtree is in a zig-zag orientation, so we have to perform another double rotation.

Deleting Example:



The first rotation is a left rotation about F (promoting node N), giving us the picture on the left.

Deleting Example:



The second rotation is a right rotation about Q (promoting node N again), giving us the final picture on the left.

This tree is now a valid, balanced AVL tree

Partial Implementation

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

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;
}
```

Partial Implementation

```
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)

}
```