CS 261 – Data Structures

AVL Trees
Binary Search Tree

- Complexity of BST operations:
  - proportional to the length of the path from a node to the root

- Unbalanced tree: operations may be $O(n)$
  - E.g.: adding elements in a sorted order
Balanced Binary Search Tree

• Balanced tree: the length of the longest path is roughly $\log n$

• BALANCE IS IMPORTANT!
Complete Binary Tree is Balanced

• Has the smallest height for any binary tree with the same number of nodes

• The longest path guaranteed to be \( \leq \log n \)

• => Keep the tree complete
Requiring Complete Trees

• However, it is very costly to maintain a complete binary tree
Requiring Complete Trees

• However, it is very costly to maintain a complete binary tree
Height-Balanced Trees

• For each node, the height difference between the left and right subtrees is $\leq 1$. 

`3(3)`

`2(1)`

`5(1)`

`8(2)`

`9(0)`

`1(0)`

`4(0)`

`6(0)`

indicates maximum height
Height-Balanced Trees

• Are locally balanced, but globally (slightly) unbalanced
Height-Balanced Trees

• Mathematically, the longest path has been shown to be, at worst, 44% longer than $\log n$.

• Algorithms that run in time proportional to the path length are still $O(\log n)$.
  
  – Why?
AVL Trees

• Named after the inventors’ initials:
  – Adelson-Velskii and Landis

• Maintain the height balanced property of Binary Search Trees
AVL Trees

• Add an integer height field to each node:
  – Null child has a height of –1
  – A node is *unbalanced* when the absolute height difference between the left and right subtrees is *greater than one*
struct AVLNode {
    TYPE val;
    struct AVLNode *left;
    struct AVLNode *rght;
    int hght; /* Height of node*/
};
Get Height

```c
int _height(struct AVLNode *cur) {
    if (cur == 0)
        return -1
    else return cur->hght;
}
```
void _setHeight(struct AVLNode *cur) {
    int lh = _height(cur->left);
    int rh = _height(cur->rght);
    if(lh < rh)
        cur->hght = 1 + rh;
    else
        cur->hght = 1 + lh;
}
Maintaining the Height Balanced Property

• When unbalanced, perform a “rotation” to balance the tree

![Diagram of tree rotation]
Left Rotation

1. Input: current
2. New top = current's right child
1. Input: current

2. New top = current's right child

3. New top's new left child = current

Left Rotation
Left Rotation

1. Input: \textit{current}

2. New top = \textit{current}'s right child

3. New top’s new left child = \textit{current}

4. Current’s new right child = new top's left child
1. Input: current
2. New top = current's right child
3. New top’s new left child = current
4. Current’s new right child = new top's left child
5. Set height of current
6. Set height of new top node
Left Rotation

```
2(3)       Rotate left
           4(2)
           /   \
1(0)   3(0)   5(1)
        /   \
6(0)   4(2)
       /   \
2(1)   5(1)
      /   \
1(0)   3(0)
     /   \
6(0)   1(0)
```

20
Right Rotation
Right Rotation

1. Input: current
2. New top = current's left child
3. New top’s right child = current
4. Current’s new left child = new top's right child
5. Set height of current
6. Set height of new top node
Double Rotation Left

• A single rotation may not fix the problem:
  – When the right child is heavy, i.e.,
    • its parent is unbalanced
    • has only a right subtree

Unbalanced “top” node

“Heavy” right child

Doesn’t work!!!
Double Rotation Left

• *Rotate the child* before the regular rotation:
  1. Rotate the heavy right child to the **right**
  2. Rotate the “top” node to the **left**
Double Rotation

• A single rotation may not fix the problem:
  – When the **left** child is **heavy**, i.e.,
    • its parent in unbalanced from the left
    • has only a left subtree

Unbalanced “top” node

“Heavy” left child

Doesn’t work!!!
Double Rotation Right

• This case requires *rotating the child* before the regular rotation:

1. Rotate the heavy left child to the **left**
2. Rotate the “top” node to the **right**
Balancing an Unbalanced Node

If left child is taller than right child{ /* Rotation right */
    If left child is heavy{ /* Double rotation right*/
        Rotate left the heavy left child
    }
    Rotate right the node
} else{ /* Rotation left */
    If right child is heavy { /* Double rotation left */
        Rotate right the heavy right child
    }
    Rotate left the node
}
Return node
Example: Add 7 to the tree

Height-Balanced Tree

Add data: 7

Unbalanced Tree

“Heavy” left child

Added to right side of heavy left child
Example – Suppose We Used Single Rotation

Unbalanced Tree

```
3(4)  
+---+  
|   |   
| 2(1) 8(3) |
|   +---+   |
| 1(0) 5(2) 9(0) |
|      +---+   |
|      4(0) 6(1) |
|            +---+   |
|            7(0)  
```

Single right rotation

```
3(4)  
+---+  
|   |   
| 2(1) 5(3) |
|   +---+   |
| 1(0) 4(0) 8(2) |
|      +---+   |
|      6(1) 9(0) |
|            +---+   |
|            7(0)  
```

Tree Still Unbalanced

Unbalanced “top” node (still)
Example – Double Rotation Right

Unbalanced Tree

Tree Still Unbalanced, but …

Rotate left the heavy left child

“Heavy” left child
Example – Double Rotation Right

Unbalanced Tree (after 1st rotation)

Rotate right top node

Tree Now Balanced

Unbalanced “top” node
Your Turn

• Any questions

• Worksheet:
  – Start by inserting values 1-7 into an empty AVL tree
  – Then write code for left and right rotations