CS 261 – Data Structures

AVL Trees
Binary Search Tree

• Complexity of BST operations:
  – proportional to the length of the root-node

• For unbalanced tree, operations become $O(n)$:
  – E.g.: when elements are added to the tree in sorted order
Balanced Binary Search Tree

• In a well balanced tree, the length of the longest path is roughly $\log n$
  
  – E.g.: longest path of $\log_2 1,048,576 = 20$.

• BALANCE IS IMPORTANT!
Complete Binary Tree is Balanced

• Has the shortest overall path length for any binary tree

• The longest path guaranteed to be $\leq \log n$

• $\Rightarrow$ Keep the tree complete
Complete Binary Tree is Balanced

• Has the shortest overall path length for any binary tree

• The longest path guaranteed to be $\leq \log n$
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

• Instead, use *height*-balanced binary trees:

  – for each node, the height difference between the left and right subtrees is ≤ 1

```
       3(3)
      /   \
     2(1) 8(2)
    / |    |   \
   1(0) 5(1) 9(0)
  / |       |   \
 4(0) 6(0)     
```

indicates maximum depth
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

• 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*
AVL Implementation

```c
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->right);
    if(lh < rh)
        cur->hght = 1 + rh;
    else
        cur->hght = 1 + lh;
}
Maintaining the Height Balanced Property

• When unbalanced, performs a “rotation” to balance the tree
Rotation Pseudocode: Rotate Current "Top" Node Left

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

```plaintext
Rotate left

Current

2(3) -> 4(2)
1(0)   5(1)
3(0)   6(0)

New “top” node

4(2)
2(1)   5(1)
1(0)   3(0)
6(0)
```
Rotation Pseudocode: Rotate Current "Top" Node Left

1. Input: current = current node
2. New top node is current's right child
3. New top's left child = current

```
Current

2(3)
  /   \
1(0)   4(2)
  /     /
3(0)   5(1)

New "top" node
```

```
2(1)
  /   \
1(0)   3(0)
  /     /
6(0)  5(1)
  /     /
6(0)  3(0)
```
Rotation Pseudocode: Rotate Current "Top" Node Left

1. Input: \texttt{current} = current node
2. New top node is current's right child
3. New top's left child = current
4. Current's new right child = new top node's left child
Rotation Pseudocode: Rotate Current "Top" Node Left

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

• Sometimes a single rotation may not fix the problem:

  – When an insertion is made on the left side of a node that is itself a heavy right child

Unbalanced “top” node

“Heavy” right child

Doesn’t work!!!
AVL Trees: **Double Rotation**

- Fortunately, this case is easily handled by rotating the *child* before the regular rotation:

1. First rotate the heavy right (or left) child to the right (or left)

2. Rotate the “top” node to the left (or right)
Balancing pseudocode (to rebalance an unbalanced node):

If left child is tallest:

If left child is heavy on the right side: // Double rotation needed.
    Rotate the left child to the left
    Rotate unbalanced ("top") node to the right

Else: // Right child is the tallest.

If right child is heavy on the left side: // Double rotation needed.
    Rotate the right child to the right
    Rotate unbalanced ("top") node to the left

Return new "top" node
AVL Trees: Double Rotation Example

Balanced Tree

3(3)
/   \
2(1)   8(2)
/   /   /
1(0) 5(1) 9(0)
/   /   /
4(0) 6(0)  \\

Unbalanced Tree

3(4)
/   \
2(1)   8(3)
/   /   /
1(0) 5(2) 9(0)
/   /   /
4(0) 6(1) 7(0)

Added to right side of heavy left child

Unbalanced "top" node

"Heavy" left child

Add data: 7
AVL Trees: **Double Rotation Example**

Unbalanced Tree

```
  3(4)
  /   \
 2(1) 8(3)
/     /   \
1(0)  5(2) 9(0)
   /   /   \
 4(0) 6(1) 7(0)
```

Single rotation

Tree Still Unbalanced

```
  3(4)
  /   \
 2(1) 5(3)
/     /   \
1(0)  4(0) 8(2)
   /   /   \
 6(1) 9(0) 7(0)
```

Unbalanced “top” node (still)
AVL Trees: Double Rotation Example

Unbalanced Tree

Tree Still Unbalanced, but …

Rotate heavy child

“Heavy” left child
AVL Trees: Double Rotation Example

Unbalanced Tree
(after 1st rotation)

Unbalanced "top" node

Rotate top node

Tree Now Balanced

Rotate top node
AVL Tree: Comparison with Skip Lists

• Other types of height balanced trees:
  – JAVA library uses red/black trees (class `TreeSet`) → Similar idea, slightly faster still
AVL Trees: Sorting

• An AVL tree can easily sort a collection of values:
  1. Copy the values of the data into the tree: \(O(n \log_2 n)\)
  2. Copy them out using an in-order traversal: \(O(n)\)

• Execution time \(\rightarrow O(n \log n)\):
  – Matches that of quick sort in benchmarks
  – Unlike quick sort, AVL trees don’t have problems if data is already sorted or almost sorted (which degrades quick sort to \(O(n^2)\))

• However, requires extra storage to maintain both the original data buffer (e.g., a DynArr) and the tree structure
Your Turn

• Any questions

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