AVL Trees

CSE 373
Data Structures
Unit 7

Reading: Section 4.4

Binary Search Tree - Best Time

• All BST operations are \( O(d) \), where \( d \) is tree depth
• minimum \( d \) is \( d = \lfloor \log_2 N \rfloor \) for a binary tree with \( N \) nodes
  › What is the best case tree?
  › What is the worst case tree?
• So, best case running time of BST operations is \( O(\log N) \)

Binary Search Tree - Worst Time

• Worst case running time is \( O(N) \)
  › What happens when you insert elements in ascending order?
    • Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  › Problem: Lack of “balance”:
    • compare depths of left and right subtree
  › Unbalanced degenerate tree

Balanced and unbalanced BST
Approaches to balancing trees

• Don't balance
  › May end up with some nodes very deep

• Strict balance
  › The tree must always be balanced perfectly

• Pretty good balance
  › Only allow a little out of balance

• Adjust on access
  › Self-adjusting

Balancing Binary Search Trees

• Many algorithms exist for keeping binary search trees balanced
  › Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  › Splay trees and other self-adjusting trees
  › B-trees and other multiway search trees

Perfect Balance

• Want a complete tree after every operation
  › tree is full except possibly in the lower right

• This is expensive
  › For example, insert 2 in the tree on the left and then rebuild as a complete tree

AVL - Good but not Perfect Balance

• AVL trees are height-balanced binary search trees

• Balance factor of a node
  › height(left subtree) - height(right subtree)

• An AVL tree has balance factor calculated at every node
  › For every node, heights of left and right subtree can differ by no more than 1: For every node \( t \), \( h(t.left) - h(t.right) \in \{-1, 0, 1\} \)
  › Store current heights in each node

---

[Diagram showing the insertion of 2 into a binary search tree, followed by rebuilding to a complete tree]
Height of an AVL Tree

- $N(h) = \text{minimum number of nodes in an AVL tree of height } h$.
- **Basis**
  - $N(0) = 1$, $N(1) = 2$
- **Induction**
  - $N(h) = N(h-1) + N(h-2) + 1$
- **Solution** (recall Fibonacci analysis)
  - $N(h) \geq \phi^h$ ($\phi \approx 1.62$)

Node Heights

Tree A (AVL)  
```
          2
         /\  
        /  \  
       1   6
```

Tree B (AVL)  
```
          2
         /\  
        /  \  
       1   6
```

Node Heights after Insert 7

Tree B (AVL)  
```
          2
         /\  
        /  \  
       1   6
```

Tree C (not AVL)  
```
          3
         /\  
        /  \  
       1   6
```

- Height of node = $h$
- Balance factor = $h_{\text{left}} - h_{\text{right}}$
- Empty height = -1
- $n \geq N(h)$ (because $N(h)$ was the minimum)
- $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
- $h \leq 1.44 \log_2 n$ (i.e., Find takes $O(\log n)$)
Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference $h_{\text{left}} - h_{\text{right}}$) is 2 or –2, adjust tree by rotation around the node

Insertions in AVL Trees

Let the node that needs rebalancing be $\alpha$.

There are 4 cases:

Outside Cases (require single rotation):
1. Insertion into left subtree of left child of $\alpha$.
2. Insertion into right subtree of right child of $\alpha$.

Inside Cases (require double rotation):
3. Insertion into right subtree of left child of $\alpha$.
4. Insertion into left subtree of right child of $\alpha$.

The rebalancing is performed through four separate rotation algorithms.

Single Rotation in an AVL Tree

AVL Insertion: Outside Case
AVL Insertion: Outside Case

Inserting into X destroys the AVL property at node j
\[(h+2) - h\]

Becomes
\[h + 2\]

Do a “right rotation”

Single right rotation

Outside Case Completed

“Right rotation” done!
(“Left rotation” is mirror symmetric)

AVL property has been restored!
Consider a valid AVL subtree.

AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j.

AVL Insertion: Inside Case

Does “right rotation” restore balance?

AVL Insertion: Inside Case

“Right rotation” does not restore balance… now k is out of balance.

AVL Insertion: Inside Case

Consider the structure of subtree Y…
AVL Insertion: Inside Case

\[ Y = \text{node } i \text{ and subtrees } V \text{ and } W \]

\[ \begin{align*}
  h + 2 & : k \\
  h & : i \\
  h \text{ or } h - 1 & : X
\end{align*} \]

\[ \begin{align*}
  h & : Z \\
  V & : W
\end{align*} \]

Double rotation: first rotation

\[ \text{left rotation complete} \]

Double rotation: second rotation

\[ \text{Now do a right rotation} \]
Double rotation: second rotation

Double rotation complete
Balance has been restored

Single Rotation

```
RotateFromRight(n : reference node pointer) {
  p : node pointer;
  p := n.right;
  n.right := p.left;
  p.left := n;
  n := p
}
```

We also need to modify the heights or balance factors of `n` and `p`

Double Rotation

- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer) {
  ?????
}
```

Implementation

Another possible implementation: do not keep the height; just the difference in height, i.e. the balance factor (1,0,-1).

In both implementations, this has to be modified on the path of insertion even if you don't perform rotations.

Once you have performed a rotation (single or double) you won't need to go back up the tree.
Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference $h_{left} - h_{right}$) is 2 or $-2$, adjust tree by rotation around the node

Insert in BST

```java
Insert(T : reference tree pointer, x : element) : integer {
    if T = null then
        T := new tree; T.data := x; return 1; //the links to children are null
    else { case
        T.data = x : return 0; //Duplicate do nothing
        T.data > x : return Insert(T.left, x);
        T.data < x : return Insert(T.right, x);
    } Endcase
    T.data > x : return Insert(T.left, x);
    T.data < x : return Insert(T.right, x);
    T.data = x : return 0;
    if ((height(T.left) - height(T.right)) = 2){
        if (T.left.data > x ) then //outside case
            T = RotatefromLeft (T);
        else //inside case
            T = DoubleRotatefromLeft (T);
    }
    T.data = x : return Insert(T.left, x);
    code similar to the left case
    T.height := max(height(T.left),height(T.right)) +1;
    return 1;
}
```

Example of Insertions in an AVL Tree
Example of Insertions in an AVL Tree

Now Insert 45

Single rotation (outside case)

Now Insert 34

Double rotation (inside case)

AVL Tree Deletion

• Similar but more complex than insertion
  › Rotations and double rotations needed to rebalance
  › Imbalance may propagate upward so that many rotations may be needed.
Pros and Cons of AVL Trees

Arguments for AVL trees:
1. Search is $O(\log N)$ since AVL trees are always balanced.
2. Insertion and deletions are also $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:
1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have $O(N)$ for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Double Rotation Solution

```java
DoubleRotateFromRight(n : reference node pointer) {
    RotateFromLeft(n.right);
    RotateFromRight(n);
}
```

![Double Rotation Diagram](image-url)