Readings

• Reading Sec. 4.4

Binary Search Tree - Best Time

• All BST operations are $O(d)$, where $d$ is tree depth
• minimum $d = \lceil \log_2 N \rceil$ 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?
  › 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)
  - Weight-balanced trees
  - Red-black trees
  - Splay trees and other self-adjusting trees
  - B-trees and other (e.g. 2-4 trees) 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 Trees (1962)

- Named after 2 Russian mathematicians
  - Georgii Adelson-Velsky (1922 - ?)
  - Evgenii Mikhailovich Landis (1921-1997)

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
  - Store current heights in each node

Height of an AVL Tree

- $N(h)$ = 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)
  - $N(h) \geq \phi^h$ ($\phi \approx 1.62$)

Height of an AVL Tree

- $N(h) \geq \phi^h$ ($\phi \approx 1.62$)
- Suppose we have $n$ nodes in an AVL tree of height $h$.
  - $n \geq N(h)$
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes $O(\log n)$)
AVL Trees

Node Heights

Tree A (AVL)  
<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Tree B (AVL)  
<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

height of node = h  
balance factor = h_{left} - h_{right}  
empty height = -1

AVL Trees

Node Heights after Insert 7

Tree A (AVL)  
<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Tree B (not AVL)  
<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

height of node = h  
balance factor = h_{left} - h_{right}  
empty height = -1

AVL Trees

Worst-case AVL Trees

- 4 nodes: $h = 3$ vs $3$
- 7 nodes: $h = 4$ vs $3$
- 12 nodes: $h = 5$ vs $4$
- 20 nodes: $h = 6$ vs $5$
- 33 nodes: $h = 7$ vs $6$
- 54 nodes: $h = 8$ vs $6$
- 88 nodes: $h = 9$ vs $7$
- 143 nodes: $h = 10$ vs $8$
- 232 nodes: $h = 11$ vs $8$
- 376 nodes: $h = 12$ vs $9$
- 609 nodes: $h = 13$ vs $10$
- 986 nodes: $h = 14$ vs $10$
- 1596 nodes: $h = 15$ vs $11$

AVL Trees

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 $h_{left} - h_{right}$) is 2 or −2, adjust tree by rotation around the node

AVL Trees

Single Rotation in an AVL Tree

AVL Trees

Double rotation

Insertion of 34
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.

AVL Insertion: Outside Case

Consider a valid AVL subtree

```
                   j
                  / \       \\
                k   Z
               /   \        \__________
           X     Y
```

Inserting into \( X \) destroys the AVL property at node \( j \).

AVL Insertion: Outside Case

```
                   j
                  / \       \\
                k   Z
               /   \        \__________
           X     Y
```

Do a "right rotation"

```
                   j
                  / \       \\
                k   Z
               /   \        \__________
           X     Y
```

"Right rotation" done! ("Left rotation" is mirror symmetric)

```
                   j
                  / \       \\
                k   Z
               /   \        \__________
           X     Y
```

AVL property has been restored

Outside Case Completed

```
                   j
                  / \       \\
                k   Z
               /   \        \__________
           X     Y
```

AVL Trees 19

AVL Trees 20

AVL Trees 21

AVL Trees 22

AVL Trees 23

AVL Trees 24
AVL Insertion: Inside Case

Consider a valid AVL subtree:

```
     j
    / \  
   k   h
  /   /  
X   Y   h
```

Inserting into Y destroys the AVL property at node j:

```
     j
    /  
   k   h+1
  /   /  
X   Y   Z
```

Does "right rotation" restore balance?

```
     j
    /  
   k   h
  /   /  
X   Y   h+1
```

"Right rotation" does not restore balance… now k is out of balance:

```
     k
   /   
  j   h
 /   /  
X   h+1   h
```

Consider the structure of subtree Y…

```
     j
    /  
   k   h
  /   /  
X   Y   h
```

We will do a left-right "double rotation"…

```
     j
    /  
   k   h
  /   /  
X   i   h
```

Y = node i and subtrees V and W

```
     j
    /  
   k   h
  /   /  
X   i   h+1
```

h or h-1

```
     j
    /  
   k   h
  /   /  
X   i   h
```

We will do a left-right "double rotation"…

```
     j
    /  
   k   h
  /   /  
X   i   Z
```

V and W

```
     j
    /  
   k   h
  /   /  
X   i   Z
```

h or h-1
Double rotation: first rotation

Double rotation: second rotation

Double rotation: second rotation

Double rotation

Implementation

Single Rotation

Double Rotation

You can either keep the height or just the difference in height, i.e., the balance factor; 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.

You also need to modify the heights or balance factors of n and p.
Insert in AVL trees

Insert(T : tree pointer, x : element) : {
  if T null then
    T := new tree; T.data := x; height := 0;
  case
    T.data = x : return ; //Duplicate do nothing
    T.data > x : return Insert(T.left, x);
    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.right, x);
  code similar to the left case
  Endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}

Example of Insertions in an AVL Tree

Example of Insertions in an AVL Tree

Single rotation (outside case)

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.
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).

Pros and Cons of AVL Trees

Non-recursive insertion or the hacker's delight

- Key observations;
  - At most one rotation
  - Balance factor: 2 bits are sufficient (-1 left, 0 equal, +1 right)
  - There is one node on the path of insertion, say $S$, that is "critical". It is the node where a rotation can occur and nodes above it won't have their balance factors modified
Non-recursive insertion

• Step 1 (Insert and find S):
  - Find the place of insertion and identify the last node S on the path whose BF ≠ 0 (if all BF on the path = 0, S is the root).
  - Insert

• Step 2 (Adjust BF’s)
  - Restart from the child of S on the path of insertion. (note: all the nodes from that node on the path of insertion have BF = 0.) If the path traversed was left (right) set BF to −1 (+1) and repeat until you reach a null link (all the place of insertion)

Non-recursive insertion (ct’d)

• Step 3 (Balance if necessary):
  - If BF(S) = 0 (S was the root) set BF(S) to the direction of insertion (the tree has become higher)
  - If BF(S) = -1 (+1) and we traverse right (left) set BF(S) = 0 (the tree has become more balanced)
  - If BF(S) = -1 (+1) and we traverse left (right), the tree becomes unbalanced. Perform a single rotation or a double rotation depending on whether the path is left-left (right-right) or left-right (right-left)