More Balanced Search Trees

CS 321 Spring 2016

Steve Cutchin
Rotation in Binary Trees

Changes Structure but leaves total order of Nodes Intact.
Inorder, pre-order, post-order outputs are unchanged.

Picture comes to us courtesy of Wikipedia.
Rotation Examples

R Rotation:

L Rotations:

LR Rotation:

RL Rotation:
Right Rotation

- rightRotate(root) // assume have left child.
  - lchild = root->left;
  - Root->left = lchild->right;
  - lchild->right = root;
Left Rotation

- `leftRotate(root) // assume have right child.`
  - `rchild = root->right;`
  - `root->right = rchild->left;`
  - `rchild->left = root;`
LR = Left Right Rotation

leftRight(root)
• leftRotation(root->left);
• rightRotation(root);

LR Rotation:

Before:

```
     3
   /   \
  1     Z
 /     /\    
W   X  Y  
```

After:

```
     2
   /   \
  1     3
 /     /\    
W   XY  Z  
```

Left Rotation:

```
     5
   /   \
  3     A
 /     /\    
B   C  D  
```

Right Rotation:

```
     4
   /   \
  5     A
 /     /\    
B   C  D  
```
RL = Right Left Rotation

- rightLeft(root)
  - rightRotation(root->right);
  - leftRotation(root);
What’s in a Name?

• AVL = Adelson-Velskii and Landis, named after its two soviet inventors.

• Reminder: A Balanced Binary Search Tree is a binary search tree with a height of $\Theta(\log n)$ where $n$ is the # nodes in the tree.
AVL = Balanced Search Tree

• Self-balancing search tree.
• First invented. Invented in 1962
• Lookup, insertion, deletion all take $O(\log n)$. 
• balanceFactor is maintained between $[-1, 1]$. 
• balanceFactor = height(left) – height(right)
AVL Tree

– An AVL tree is a BST.

– For each node, the difference between the height of its left and right subtrees is either $+1$, $0$, or $-1$.

Ex:

An AVL tree

not an AVL tree
AVL Insertion: Example

Insert 15:

Insert 9:
Height of AVL Trees

Given an AVL tree with height $h$.

The maximum # of nodes: the tree is full.

\[ n \leq 2^0 + 2^1 + \ldots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \]

\[ \implies n \leq 2^{h+1} - 1 \]

\[ \implies h \geq \log(n + 1) - 1 \]

\[ = \Omega(\log n) \]
Height of AVL Trees

The minimum # of nodes:

$h = 0$

$h = 1$

$h = 2$

$h = 3$

$h = 4$
Height of AVL Trees

Let $n_h$ be the minimum # of nodes of an AVL tree with height $h$

$$n_h = \begin{cases} 
1 & \text{if } h = 0 \\
2 & \text{if } h = 1 \\
n_{h-1} + n_{h-2} + 1 & \text{if } h \geq 2 
\end{cases}$$

Recall the Fibonacci numbers as follows.

$$F_h = \begin{cases} 
0 & \text{if } h = 0 \\
1 & \text{if } h = 1 \\
F_{h-1} + F_{h-2} & \text{if } h \geq 2 
\end{cases}$$

Thus, we have $n_h > F_h$, $\forall h \geq 0$.

Since $F_h = \frac{\phi^h - \overline{\phi}^h}{\sqrt{5}}$, where $\phi = 1.61803$ and $\overline{\phi} = -0.61803$.

Binet’s formula

$$n_h > F_h = \frac{\phi^h - \overline{\phi}^h}{\sqrt{5}} \approx \frac{\phi^h}{\sqrt{5}} \quad \text{if } h \text{ is large}$$

$$\implies n \geq n_h > \frac{\phi^h}{\sqrt{5}}$$

$$\implies \log_\phi(\sqrt{5} \cdot n) > h$$

$$\implies h < \log_\phi(\sqrt{5}) + \log_\phi n$$

$$= O(\log n)$$

Therefore, the height of any AVL tree is $\Theta(\log n)$. 
Applications of BST

- Any time want to keep an incrementally sorted linked list:
- No matter the size insertion = $O(\log n)$.
- Sorted vertices are extremely good candidates.
- Polygons, Voxels, octrees all are very good candidates for BST sorting.
Running Times for AVL Trees

- a single restructure is $O(1)$
  - using a linked-structure binary tree
- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
In-Order Traversal of BST

Inorder(root):
    inorder(root->left);
    Print root;
    inorder(root->right);

In-order Traversal: 2,4,5,7,8,10,12,20
Balanced Delete in AVL tree

• Very similar to our Balance Insert Solution
• deleteBST(root, key); // simple delete
• //magic code goes in here to get closest node.
• parent = closestnode;
• While (parent)
  – balanceTree(parent); // start at bottom
  – Parent = parent->parent; // traverse up path to root.
AVL Deletion: Example

Delete 7:

```
    10
   /  \
  5    20
 /  \\  /  \\
4    7 12 2
   /  \
 2    8
```

BST-Insert

```
    10
   /  \
  5    20
 /  \\  /  \\
4    8 12 2
   /  \
 2    8
```

No Rotation necessary

Delete 10:

```
    10
   /  \
  5    20
 /  \\  /  \\
4    7 12 2
   /  \
 2    8
```

BST-Insert

```
    12
   /  \
  5    20
 /  \\  /  \\
4    7 12 2
   /  \
 2    8
```

R on 12

```
    5
   /  \
  4    12
 /  \\  /  \\
2    7 20 8
   /  \
 2    7
```

Delete 4:

```
    10
   /  \
  5    20
 /  \\  /  \\
4    7 12 6
   /  \
 2    8
```

BST-Insert

```
    10
   /  \
  5    20
 /  \\  /  \\
4    7 12 6
   /  \
 2    8
```

RL on 5

```
    6
   /  \
  5    12
 /  \\  /  \\
7    7 20 8
   /  \
 2    7
```
Magic Code Comment

• The closest node in our delete is:
  – The parent of the node we deleted.
  – Or the parent node of the node returned by findMax that we swapped with the node to delete.
  – Or root if no parent.
**Case 2:** removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

**Removing 7**

What other element can be used as replacement?
Simple Removal - BST

• Remove Node with given key from Tree.

• 1 Helper Methods:
  • Node findMax(root) // return the largest child of the given node.
    – { result = root;
    – while (result->right) result = result->right;
    – return result;
    – }

• Or – the right most child of a node.
Simple Removal - BST

• Returns root of this subtree, even if changed.
• Parent is passed as a convenience.
• Node deleteBST(root, key, parent)
  – If (root==null) return root;  // empty tree.
  – If (key < root->key) {deleteBST(root->left, key, root); return root; }
  – else if (key > root->key ) {deleteBST(root->right, key, root); return root; }
  – else if (key != root->key) return root;  // key not in tree.
  – else  // key == root->key so remove this node
    • Go to next slide

Problems with code: Does Not Clean Up Pointers to Dropped Memory.
Simple Removal - BST

else // key == root->key so remove
  • If { (root->left ) && ( root->right)) // node has two children, swap.
  • {
    • rep = findMax(root->left);
    • deleteBST(root->left,rep->key,root);
    • root->key=rep->key;
    • root->val=rep->val;
    • return root
  • }
Simple Removal - BST

else // key == root->key so remove ---- continued

• else if (root->left) // has only one child, point parent at child.
• {
  • if (parent) parent->left = root->left;
  • return root->left;
  • }
• else if (root->right)
• {
  • if (parent) parent->right = root->right;
  • return root->right;
  • }

Problems with code: Does Not Clean Up Pointers to Dropped Memory.
Simple Removal - BST

else // key == root->key so remove
  • else // node has no children
  • {
    • if (!parent) {root = null; return root;} // tree is now empty.
    • else if (parent->left == root) {parent->left = null; return null; }
    • else if (parent->right == root) parent->right = null; return null; }
  • }
  • Return root; // this should not be executed.

Problems with code: Does Not Clean Up Pointers to Dropped Memory.
Calculating Balance Factor

- Cost of the Naïve Way?
- Cost of the ‘Incremental’ way?
Calculating Balance Factor

• 3 total cases: \( p \) = parent, \( n \) = new node.
• Case 1: \( p \)'s \( \text{bf} \) was 0, now it is -1 or +1.
  – Propogate up tree: height of subtree changed!
• Case 2: \( n \) is added to \( p \)'s shorter side: \( \text{bf}=0 \)
  – Change does not propagate up tree.
• Case 3: \( n \) is added to \( p \)'s taller side: \( \text{bf}=+-2! \)
  – Rebalance will fix!. Do not propogate.
Calculate BalanceFactor

- `calcBalanceFactorTree(nn)`
- `tn = nn;`
- `p = tn->parent;`
- `while(p != null) && (p->bf == 0) {`
- `if (p->left == tn) p->bf +=1; else p->bf -=1;`
- `tn = p; p = tn->parent;`
- `}`
Calculate BalanceFactor

• Continued // p == -1 or 1
• If (p == null) return; // done.
• nbf = 0;
• if (nn == find(p->left,nn->key)) nbf =1;
• else nbf = -1;
• p->bf += nbf;
• // if p->bf == nbf unbalanced! Rotation will fix.
• // if p->bf != nbf, then more balanced!
Components of a Node

• Node has the following components:
  – **left** = pointer to left subtree.
  – **right** = pointer to right subtree.
  – **key** = the numeric key for the node. Tree is Sorted on the keys in the tree.
  – **val** = a pointer to the data associated with the key.
  – Adding: **parent** = pointer to parent.
  – Adding: **bf** = balance factor.
Simple Insertion in BST with BF

- Node insertBST(root, key, value)
  - If (root == null) { newnode = Node(key,value); return newnode; }  
  - if (key < root->key) { root->left = insertBST(root->left,key,value);}
  - else if (key > root->key) { root->right = insertBST(root->right,key,value); }
  - else {root->val = value; } // keys match, replace old value.
  - Return newnode;
Balanced Insertion

- \( \text{nn} = \text{insertBST} (\text{root, key}) \) \ // \ simple \ insertion
- \( \text{calcBalanceFactorTree} (\text{nn}) \);
- \( \text{parent} = \text{nn->parent} \);
- While (\text{parent}) \ // \ rebalance \ tree
  - \( \text{balanceTree} (\text{parent}) \); \ // \ start \ at \ bottom
  - \( \text{Parent} = \text{parent->parent} \); \ // \ traverse \ up \ path \ to \ root. \)
Balanced Insertion

• balanceTree(root):
  – If (root->bf == 2) {
    • If ((root->left->bf) == -1) leftRotate(root->left);
    – rightRotate(root);
  – }
  – else if (root->bf == -2) {
    • If (root->right->bf == 1) rightRotate(root->right);
    – leftRotate(root);
  – }
  – root->bf = 0;root->left->bf = root->right->bf = 0;