Overview

• Balance Binary Search Tree
Objective

- Understand the principles of binary search trees that assure that tree remains balanced at all times

Binary Search Tree Problem

- Unfortunately, search tree of height $n$ can be constructed by inserting keys in sorted order
- In this case, performance of put method is $O(n)$
- Similar for get, in, del
**Balanced Binary Search Tree**

- Special kind of binary search tree
- Automatically assures that tree remains balanced at all times
- Tree is called AVL tree, names after inventors: G. M. Adelson-Velskii and E. M. Landis

**Balanced Binary Search Tree**

- AVL tree implement Map ADT just like regular binary search tree
- Difference lies in its performance
- Need to keep track of balance:
  - Height of left and right subtree of each node

\[
\text{balanceFactor} = \text{height(leftSubTree)} - \text{height(rightSubTree)}
\]
Balanced Binary Search Tree

- Left-heavy: Balance factor > 0
- Right-heavy: Balance factor < 0
- Perfectly in balance: Balance factor = 0
- Definitions: Tree is balanced if balance factor is -1, 0, or 1
- Outside that range tree needs to be brought back in balance

Balanced Binary Search Tree

- Unbalance, right-heavy tree
- Balance factor at each node
AVL Tree Performance

• Most unbalanced left-heavy tree

<table>
<thead>
<tr>
<th>Height</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1 + 1 = 2</td>
</tr>
<tr>
<td>2</td>
<td>1 + 1 + 2 = 4</td>
</tr>
<tr>
<td>3</td>
<td>1 + 2 + 4 = 7</td>
</tr>
</tbody>
</table>

• General: $N_h = 1 + N_{h-1} + N_{h-2}$
AVL Tree Performance

• Similarity to Fibonacci sequence:
  • $F_0 = 0$
  • $F_1 = 1$
  • $F_i = F_{i-1} + F_{i-2}$ for all $i \geq 0$
  • With ”golden Rule”: $F_i = \phi^i / \sqrt{5}$

• With approximation: $N_h = F_{h+2} - 1$, $h \geq 1$
  • $N_h = \frac{\phi^{h+2}}{\sqrt{5}} - 11$
  • $\log N_h = (H + 2)\log \phi - \frac{1}{2} \log 5$
  • $h = \frac{\log N_h + 1 - 2\log \phi + \frac{1}{2} \log 5}{\log \phi}$
  • $h = 1.44 \log N_h \Rightarrow O(\log N)$
AVL Tree Implementation

• Base implementation on binary search tree:
  • New keys will be as leaf nodes
    • Balance factor for leaf node = 0
  • Must update balance factor for parent
  • If new node == right child balance factor of parent reduced by 1
  • If new node == left child balance factor of parent increased by 1

• relation can be applied recursively to the grandparent of new node

• updating balance factors:
  • The recursive call has reached the root of the tree.
  • The balance factor of the parent has been adjusted to zero. Once balance factor is zero, balance of its ancestor nodes does not change.
AVL Tree Implementation

- Implement the AVL tree as a subclass of BinarySearchTree.
- override the _put method
- new updateBalance helper method.

```python
def _put(self, key, val, currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key, val, currentNode.leftChild)
        else:
            currentNode.leftChild = TreeNode(key, val, parent=currentNode)
            self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
            self._put(key, val, currentNode.rightChild)
        else:
            currentNode.rightChild = TreeNode(key, val, parent=currentNode)
            self.updateBalance(currentNode.rightChild)
```
AVL Tree Implementation

- updateBalance method is where most of the work is done.
- updateBalance first checks if current node is out of balance enough to require rebalancing
- If current node does not require rebalancing => balance factor of parent is adjusted
- If balance factor of the parent is non-zero then the algorithm continues

```python
def updateBalance(self, node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.isLeftChild():
            node.parent.balanceFactor += 1
        elif node.isRightChild():
            node.parent.balanceFactor -= 1
        if node.parent.balanceFactor != 0:
            self.updateBalance(node.parent)
```
AVL Tree Rebalancing

- How to perform rebalancing
- => rotations on the tree

AVL Tree Left Rotation

- Promote right child (B) to be root of subtree
- Move old root (A) to be left child of new root
- If new root (B) already had left child then make it right child of new left child (A)

- While procedure is fairly easy in concepts, implementation is tricky
AVL Tree Right Rotation

• Promote the child (C) to be root of subtree
• Move old root (E) to be right child of new root
• If new root (C) already had a right child (D) then make it left child of new right child (E)
def rotateLeft(self, rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
            rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
    min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 +
    max(rotRoot.balanceFactor, 0)

AVL Tree Rotate Implementation

AVL Tree Balance Factors

- How to update balance factors without completely recalculating heights of new subtrees?
AVL Tree Balance Factors

- B and D are pivotal nodes; A, C, E are their subtrees
- Let $h_x$ be height at subtree rooted at node $x$:
  - $newBal(B) = h_A - h_C$
  - $oldBal(B) = h_A - h_D$

\[ D \text{ can also be given by } 1 + \max(h_C, h_E) \]
- $h_C$ and $h_E$ have not changed
- $\Rightarrow oldBal(B) = h_A - (1 + \max(h_C, h_E))$
AVL Tree Balance Factors

- \( \text{newBal}(B) - \text{oldBal}(B) = h_A - h_C - (1 + \max(h_C, h_E)) \)
- \( \text{newBal}(B) - \text{oldBal}(B) = h_A - h_C - h_A + (1 + \max(h_C, h_E)) \)
- \( \text{newBal}(B) - \text{oldBal}(B) = (1 + \max(h_C, h_E)) - h_C \)

AVL Tree Balance Factors

- With \( \max(a,b) - c = \max(a - c, b - c) \)
- \( \text{newBal}(B) - \text{oldBal}(B) = h_A - h_C - h_A + (1 + \max(h_C, h_E)) \)
- \( \text{newBal}(B) = \text{oldBal}(B) + 1 + \max(h_C - h_C, h_E - h_C) \)
AVL Tree Balance Factors

• Since $h_E - h_C = -oldBal(D)$ and $max(-a, -b) = -min(a, b)$
• $newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))$
• $newBal(B) = oldBal(B) + 1 - min(0, oldBal(D))$

AVL Tree - Not Done Yet

Left rotation =>
AVL Tree - Not Done Yet

• To correct problem:
  • If a subtree needs left rotation, first check balance factor of right child. If right child is left heavy then do a right rotation on right child, followed by original left rotation.
  • If a subtree needs right rotation, first check balance factor of left child. If left child is right heavy then do a left rotation on left child, followed by the original right rotation.
AVL Tree – Rebalance

```python
def rebalance(self, node):
    if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
        else:
            self.rotateLeft(node)
    elif node.balanceFactor > 0:
        if node.leftChild.balanceFactor < 0:
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
        else:
            self.rotateRight(node)
```

AVL Tree – Analysis

• Keeping the tree in balance all times => get runs in $O(\log n)$ time.

• Insertion (put):
  • New node inserted as leaf => $\log n$
  • Balance => $O(1)$
## Search – Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sorted List</th>
<th>Hash Table</th>
<th>Binary Search Tree</th>
<th>AVL Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>get</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>in</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>del</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

## Next Steps

- Next lecture on Tuesday: Exam Review
- No lecture on 10/9
- Exam on 10/10