6.006 Recitation

Build 2008.7
Outline

• Basic concepts review
• AVL algorithms
• Python implementation for AVLs
BST Invariants

- Binary rooted tree
- All left descendants have keys < node’s key
- All right descendants have keys > node’s key
Node Height

- Leaves: height = 0
- Inner nodes: height = \( \text{max(children height)} + 1 \)
- Null tree: height = -1
- Rationale:
  - A subtree operation takes \( O(h) \) time
Node Height

- Leaves: height = 0
- Inner nodes: height = \text{max(children height)} + 1
- Null tree: height = -1
- Rationale:
  - a subtree operation takes $O(h)$ time
Node Height

- Leaves: height = 0
- Inner nodes: height = \text{max(children height)} + 1
- Null tree: height = -1
- Rationale:
  - A subtree operation takes $O(h)$ time
Node Height

- Leaves: height = 0
- Inner nodes: height = max(children height) + 1
- Null tree: height = -1
- Rationale:
  - a subtree operation takes $O(h)$ time
Node Height

- Leaves: height = 0
- Inner nodes: height = \text{max(children height)} + 1
- Null tree: height = -1

Rationale:
- A subtree operation takes $O(h)$ time
Node Height

- Leaves: height = 0
- Inner nodes: height = max(children height) + 1
- Null tree: height = -1
- Rationale:
  - a subtree operation takes $O(h)$ time
Balanced Trees

- Small tree height means fast operations
- Pack many nodes in trees with low heights
- Perfectly balanced tree: \( 2^{h+1} - 1 \) nodes
- We only care about asymptotic notation
  - Nodes = \( f(\text{height}) \) must be exponential
AVL Trees

- Regular BST with extra invariants:
  - absolute value(left child height - right child height) <= 1
  - Each subtree is AVL
Least dense AVL
Least dense AVL
Least dense AVL
Least dense AVL
Least dense AVL
Least dense AVL
Least dense AVL
Least dense structure

- Nodes(-1) = 0
- Nodes(0) = 1
- Nodes(h) = 1 + Nodes(h-1) + Nodes(h-2)
- Looks like Fibonacci, must be exponential
Pwnage with AVLs 101

- Goals
  - Reuse the code we wrote before
  - Start with an AVL, end up with an AVL
- Managerial Input (the ‘doh’ words)
  - Insert and delete like it’s a BST
  - Patch to make it an AVL again
Key Observation
Key Observation

Adding or removing a node only upsets the heights on a single path to the root.
Pwnage with AVLs 201

- Will obviously have to move nodes around
- But must keep track of
  - Height
  - Augmented data
  - Invariants for AVL, BST
- Need a tool that preserves most structure
Uberpoke (rotations)
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Huh? Do that again?
Rebalancing

- Rotations are quite teh uberpoke
- Need a master plan for using them
  - Managerial Input: call it ‘rebalancing’
  - Divide and conquer: start from the bottom, fix up the tree level by level
Rebalancing: easy

\[
\begin{align*}
&\text{Rebalancing: easy} \\
&\text{or: (} \\
\end{align*}
\]
Rebalancing: easy

A

k-1

or

B

C

k

Rotate left

k+1

y

x

:((
Rebalancing: easy

Rotate left

#: (x
 y
  k+1
 k-1
 A
 B
 C
 k-1
 k
 or k

y
 k+1
 k
 or k+1
 x
 C
 k
 A
 B
 C
 k-1
 k
 or k
 or k
Rebalancing: easy?
Rebalancing: easy?

Rotate left

>:(
Rebalancing: easy?

Rotate left:

Before:
- Node x
- Left child y
- Right child k+1
- k-1

After:
- Node y
- Left child k+1
- Right child k

A → B → C
WTF?

Rotate left

Before:

\[ k-1 \]

\[ k \]

\[ k+1 \]

\[ A \]

\[ B \]

\[ C \]

After:

\[ k+1 \]

\[ k-1 \]

\[ y \]

\[ x \]

\[ A \]

\[ B \]

\[ C \]
WTF?

Rotate left

\[
\begin{align*}
&k-1 & & k & & k-1 \\
&k-1 & & y & & k+1 \\
&A & & B & & C \\
\end{align*}
\]
WTF?

```
A
k-1

k

B

k+1

y

x

:(

B cannot be taller than C
```

Rotate left

```
A
k-1

k

B

k+1

y

x

:(

k-1

C

k

B

k+1
```

WTF?
Rebalancing: hack it up

\[ : (x) \]

\[ y \]

\[ k+1 \]

\[ A \]

\[ k-1 \]

\[ z \]

\[ k \]

\[ C \]

\[ k-1 \]

\[ k-1 \]

\[ D \]

\[ k-1 \]

\[ E \]

\[ k-2 \]

\[ k-2 \]
Rebalancing: hack it up

Rotate right

A

B

C

D

E

:(
Rebalancing: hack it up

Rotate right

...
and in the end it’s right
and in the end it’s right

Rotate left

A

D

E

x

z

y

k

k+1

k-1

k-1

k-1 or k-2

k-1 or k-2

k-1

C
and in the end it's right

Rotate left

::(x

A

k-1

D

k-1

or

k-2

Y

k

E

k-1

or

k-2

z

k+1

x

A

k-1

D

k-1

or

k-2

k

y

k

z

k+1

E

k-1

or

k-2

C

k-1
Rebalancing one level

- AVL violation at current node
- Right is than left?
  - Right.left taller than right.right?
    - Rotate right to the right
  - Either way, rotate current to the left
- Left is heavier than right: symmetry
Rebalancing wrap-up

- Know how to fix one level, use that to fix everything along the path to the root
- Must recompute height on-the-go
  - If recomputing for all nodes along the path on each rotation, $O(\log^2(h))$
- Why is rebalancing $O(\log(h))$?
Python Code

‘cause you can’t live on bubbles and lines
AVL Design

- BST
  - incorporate the deletion hack
- AVL
  - inherited from BST, uses AVLnode
- AVLnode
  - does all the heavy lifting
Return values matter!

- `insert`: returns the newly inserted node
- `delete`: returns the deleted node (its parent link still indicates where it was hanging)
```python
class BST(object):
    def __init__(self, NodeType=BSTnode):
        self.root = None
        self.NodeType = NodeType
        self.psroot = self.NodeType(None, None)

    def reroot(self):
        self.root = self.psroot.left

    def insert(self, t):
        if self.root is None:
            self.psroot.left = self.NodeType(self.psroot, t)
            self.reroot()
            return self.root
        else:
            return self.root.insert(t, self.NodeType)
```

class AVL(BST):
    def __init__(self):
        BST.__init__(self, AVLnode)

    def insert(self, t):
        node = BST.insert(self, t)
        node.rebalance()
        self.reroot()

    def delete(self):
        node = BST.delete(self)
        node.parent.rebalance()
        self.reroot()
AVLNode: helpers

```python
1 def height(node):
2     if node is None:
3         return -1
4     else:
5         return node.height

class AVLNode(BSTNode):
10     def update_stats(self):
11         self.height = max(height(self.left), height(self.right)) + 1
12         BSTNode.update_stats(self)
```
class AVLNode(BSTNode):
    def left_rotate(self):
        x = self; y = x.right
        y.parent = x.parent
        if y.parent.left is x:
            y.parent.left = y
        elif y.parent.right is x:
            y.parent.right = y
        x.right = y.left
        if x.right is not None:
            x.right.parent = x
        y.left = x
        x.parent = y
        x.update_stats()
        y.update_stats()
        return y
class AVLnode(BSTnode):
    def rebalance(self):
        if self.key is None:
            return

        self.update_height()
        if height(self.left) >= 2 + height(self.right):
            if height(self.left.left) < height(self.left.right):
                self.left.left_rotate()
                self.right_rotate()
            elif height(self.right) >= 2 + height(self.left):
                if height(self.right.right) < height(self.right.left):
                    self.right.right_rotate()
                    self.left_rotate()
        self.parent.rebalance()