AVL Trees
AVL Tree Definition

- Adelson-Velsky and Landis
- binary search tree
- balanced
  - each internal node v
    - the heights of the children of v can differ by at most 1

An example of an AVL tree where the heights are shown next to the nodes.
Height of an AVL Tree

Fact: The height of an AVL tree storing $n$ keys is $O(\log n)$.

Proof (by induction): $n(h)$: the minimum number of internal nodes of an AVL tree of height $h$.

- $n(1) = 1$ and $n(2) = 2$
- For $n > 2$, an AVL tree of height $h$ contains the root node, one AVL subtree of height $n-1$ and another of height $n-2$.
- That is, $n(h) = 1 + n(h-1) + n(h-2)$
- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So
  $n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), \ldots$ (by induction),
  $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{\frac{h}{2} - 1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is $O(\log n)$
Insertion

- Insertion is as in a binary search tree.
- Always done by expanding an external node.
- Insert 54:

Before insertion:

After insertion:
Insertion

- Insertion is like a binary search tree.
- Always done by expanding an external node.
- Insert 54:

Before insertion:

- 44
- 17
- 32
- 48
- 50
- 62
- 78
- 88

After insertion:

- 44
- 17
- 32
- 48
- 50
- 62
- 78
- 54

Imbalance Node z

Node w

Insert Node w
Insertion

- $z = \text{first unbalanced node encountered while travelling up the tree from } w$.
  - $y = \text{child of } z \text{ with the larger height}$,
  - $x = \text{child of } y \text{ with the larger height}$

- \text{trinode restructuring} to restore balance at $z$
Overview of 4 Cases of Trinode Restructuring

Case 1

Case 2

Case 3

Case 4

z ->
y ->

x ->

© 2014 Goodrich, Tamassia, Goldwasser
Red-Black Trees
Rotation operation

Consider subTree points to y and we also have x and y

1. $y.left = x.right$
2. $x.right = y$
3. subTree = x

With a linked structure
- Constant number of updates
- $O(1)$ time
Trinode Restructuring: Case 1

- Keys: \( a < b < c \)
- Nodes: grandparent \( z \) is not balanced, \( y \) is parent, \( x \) is node

**Single Rotation:**

- Not balanced at \( a \), the smallest key
- \( x \) has the largest key \( c \)
- Result: middle key \( b \) at the top
Example for Case 1

Case 1

T0
T1
T2
T3

z
y
x

© 2014 Goodrich, Tamassia, Goldwasser Red-Black Trees
Trinode Restructuring: Case 2

• Single Rotation:
• Not balanced at c, the largest key
• x has the smallest key a

Result: middle key b at the top

- Keys: a < b < c
- Nodes: grandparent z is not balanced, y is parent, x is node
Example for Case 2

Case 2

T0  T1  T2  T3

T0  T1

T2

T3
Trinode Restructuring: Case 3

double rotation:

- Keys: \(a < b < c\)
- Nodes: grandparent \(z\) is not balanced, \(y\) is parent, \(x\) is node

- Not balanced at \(a\), the smallest key
- \(x\) has the middle key \(b\)
- \(x\) is rotated above \(y\)
- \(x\) is then rotated above \(x\)

- Result: middle key \(b\) at the top
Example for Case 3
Trinode Restructuring: Case 4

- double rotation
- Not balanced at c, the largest key
- x has the middle key b
- x is rotated above y
- x is then rotated above z

- Result: middle key b at the top

Keys: a < b < c
Nodes: grandparent z is not balanced, y is parent, x is node
Example for Case 4
Insert 54 (Case 3 or 4?)

unbalanced...

...balanced

Draw the double rotation
### Trinode Restructuring summary

<table>
<thead>
<tr>
<th>Case</th>
<th>imbalance/ grandparent $z$</th>
<th>Node $x$</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smallest key $a$</td>
<td>Largest key $c$</td>
<td>single</td>
</tr>
<tr>
<td>2</td>
<td>Largest key $c$</td>
<td>Smallest key $a$</td>
<td>single</td>
</tr>
<tr>
<td>3</td>
<td>Smallest key $a$</td>
<td>Middle key $b$</td>
<td>double</td>
</tr>
<tr>
<td>4</td>
<td>Largest key $c$</td>
<td>Middle key $b$</td>
<td>double</td>
</tr>
</tbody>
</table>
### Trinode Restructuring Summary

<table>
<thead>
<tr>
<th>Case</th>
<th>imbalance/ grandparent ( z )</th>
<th>Node ( x )</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smallest key ( a )</td>
<td>Largest key ( c )</td>
<td>single</td>
</tr>
<tr>
<td>2</td>
<td>Largest key ( c )</td>
<td>Smallest key ( a )</td>
<td>single</td>
</tr>
<tr>
<td>3</td>
<td>Smallest key ( a )</td>
<td>Middle key ( b )</td>
<td>double</td>
</tr>
<tr>
<td>4</td>
<td>Largest key ( c )</td>
<td>Middle key ( b )</td>
<td>double</td>
</tr>
</tbody>
</table>

The resulting balanced subtree has:
- middle key \( b \) at the top
- smallest key \( a \) as left child
  - \( T_0 \) and \( T_1 \) are left and right subtrees of \( a \)
- largest key \( c \) as right child
  - \( T_2 \) and \( T_3 \) are left and right subtrees of \( c \)
Removal

- Removal begins as in a binary search tree
  - the node removed will become an empty external node.
  - Its parent, \( w \), may cause an imbalance.
- Remove 32, imbalance at 44

Before deletion of 32

After deletion
Rebalancing after a Removal

- \( z \) = first unbalanced node encountered while travelling up the tree from \( w \).
  - \( y \) = child of \( z \) with the larger height,
  - \( x \) = child of \( y \) with the larger height

- **Trinode restructuring** to restore balance at \( z \)—Case 1 in example
Rebalancing after a Removal

- this restructuring may upset the balance of another node higher in the tree

  - continue checking for balance until the root of T is reached
Rebalancing after a Removal

[Slide added –jyp]

In the case below, restructuring the subtree rooted at 44 created a new subtree (incidentally now rooted at 62) which is has height decreased by 1

This might cause an unbalanced situation at an ancestor of this subtree
Balanced tree

© 2014 Goodrich, Tamassia, Goldwasser

AVL Trees
Delete 80
Not balanced at 70
Single rotation
Anything wrong?
Not balanced at 50!
AVL Tree Performance

n entries

- \(O(n)\) space
- A single restructuring takes \(O(1)\) time
  - using a linked-structure binary tree

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst-case Time Complexity</th>
<th>Worst-case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get/search</td>
<td>(O(\log n))</td>
<td>Up to height (\log n)</td>
</tr>
<tr>
<td>Put/insert</td>
<td>(O(\log n))</td>
<td>(O(\log n)): searching &amp; restructuring</td>
</tr>
<tr>
<td>Remove/delete</td>
<td>(O(\log n))</td>
<td>(O(\log n)): searching &amp; restructuring up to height (\log n)</td>
</tr>
</tbody>
</table>
AVL Trees

- balanced Binary Search Tree (BST)
- Insert/delete operations include rebalancing if needed
- Worst-case time complexity: $O(\log n)$