Data Structures in Java

Lecture 10: AVL Trees.

10/12/2015

Daniel Bauer
Balanced BSTs

• Balance condition: Guarantee that the BST is always close to a complete binary tree (every node has exactly two or zero children).

• Then the height of the tree will be $O(\log N)$ and all BST operations will run in $O(\log N)$. 
AVL Tree Condition

- An AVL Tree is a Binary Search Tree in which the following **balance condition** holds after each operation:
  - For every node, the height of the left and right subtree differs by at most 1.

![AVL Tree Diagrams](https://via.placeholder.com/150)

- Not an AVL tree
AVL Trees

• Height of an AVL tree is at most
  \[ \sim 1.44 \log(N+2) - 1.328 = O(\log N) \]

• How to maintain the balance condition?
  • Rebalance the tree after each change (insertion or deletion).
  • Rebalancing must be cheap.
“Outside” Imbalance

node $k_2$ violates the balance condition

- left subtree of left child too high
- right subtree of right child too high

- Solution: Single rotation
“Inside” Imbalance

node k₂ violates the balance condition

right subtree of left child too high

left subtree of right child too high

• Solution: Double rotation
Single Rotation

Maintain BST property:
• $x$ is still left subtree of $k_1$.
• $z$ is still right subtree of $k_2$.
• For all values $v$ in $y$: $k_1 < v < k_2$
  so $y$ becomes new left subtree of $k_2$. 
Maintain BST property:
- x is still left subtree of \( k_1 \).
- z is still right subtree of \( k_2 \).
- For all values \( v \) in \( y \): \( k_1 < v < k_2 \) so \( y \) becomes new left subtree of \( k_2 \).
Single Rotation

Maintain BST property:
• $x$ is still left subtree of $k_1$.
• $z$ is still right subtree of $k_2$.
• For all values $v$ in $y$: $k_1 < v < k_2$ so $y$ becomes new left subtree of $k_2$.

Modify 3 references:
• $k_2$.left = $k_1$.right
• $k_1$.right = $k_2$
• parent($k_2$).left = $k_1$ or parent($k_2$).right = $k_1$ or
Maintaining Balance in an AVL Tree

- Assume the tree is balanced.
- After each insertion, find the lowest node $k$ that violates the balance condition (if any).
- Perform rotation to re-balance the tree.
- Rotation maintains original height of subtree under $k$ before the insertion. No further rotations are needed.
Single Rotation Example

insert(3)
Single Rotation Example

insert(3)
insert(2)
Null Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
insert(6)  rotate_right(2)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
insert(6)  rotate_right(2)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
insert(6)  rotate_right(2)
insert(7)  rotate_right(5)
Single Rotation Example

insert(3)
insert(2)
insert(1)  rotate_left(3)
insert(4)
insert(5)  rotate_right(3)
insert(6)  rotate_right(2)
insert(7)  rotate_right(5)
Single Rotation does not work for “Inside” Imbalance
Single Rotation does not work for “Inside” Imbalance

Result is not an AVL tree.
Now \( k_1 \) is violates the balance condition.
Problem: Tree \( y \) cannot move and it is too high.
Double Rotation (1)

- y is non-empty (imbalance due to insertion into y or deletion from z)
- so we can view y as a root and two sub-trees.
Double Rotation (1)

- $y$ is non-empty (imbalance due to insertion into $y$ or deletion from $z$)
- so we can view $y$ as a root and two sub-trees.

- either $y_l$ or $y_r$ is two levels deeper than $z$ (or both are empty).
Double Rotation (2)

Maintain BST property:
- $x$ is still left subtree of $k_1$.
- $z$ is still right subtree of $k_3$.
- For all values $v$ in $y_l$: $k_1 < v < k_2$ so $y_l$ becomes new right subtree of $k_1$.
- For all values $w$ in $y_r$: $k_2 < w < k_3$ so $y_r$ becomes new left subtree of $k_3$. 
Double Rotation (2)

Maintain BST property:
• $x$ is still left subtree of $k_1$.
• $z$ is still right subtree of $k_3$.
• For all values $v$ in $y_l$: $k_1 < v < k_2$
  so $y_l$ becomes new right subtree of $k_1$.
• For all values $w$ in $y_r$: $k_2 < w < k_3$
  so $y_r$ becomes new left subtree of $k_3$. 
Double Rotation (2)

These are actually two single rotations:
First at $k_1$, then at $k_3$. 
Double Rotation (2)

These are actually two single rotations: First at $k_1$, then at $k_3$. 

![Diagram of tree structure with nodes at $k_1$, $k_2$, $k_3$, $x$, $y_l$, $y_r$, and $z$.]
Double Rotation (2)

These are actually two single rotations: First at $k_1$, then at $k_3$. 
Double Rotation (3)

Modify 5 references:
• parent($k_3$).left = $k_2$ or parent($k_3$).right = $k_2$
• $k_2$.left = $k_1$
• $k_2$.right = $k_3$
• $k_1$.right = root($y_l$)
• $k_3$.left = root($y_r$)
Double Rotation Example
Double Rotation Example

insert(16)
Double Rotation Example

insert(16)
insert(7) rotate(7)
Double Rotation Example

insert(16)
insert(7) rotate(7)
Double Rotation Example

insert(16)
insert(7) rotate(7)
insert(14) rotate(6)
Double Rotation Example

insert(16)
insert(7) rotate(7)
insert(14) rotate(6)
Double Rotation Example

insert(16)
insert(7) rotate(7)
insert(14) rotate(6)