Lecture 9: AVL TREES
definition, properties, insertion
BST Performance

- BST with $n$ nodes and of height $h$
  - methods find, insert and remove take $O(h)$ time
- $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case

worst case

best case
Balanced Tree Motivation

- Need to insure height $h$ of BST is $O(\log n)$
  - then worst case complexity for find, insert, remove is $O(\log n)$
- height $h$ is $O(\log n)$ for a balanced tree
- Informally, a tree is balanced, if for any node
  - the size of its left subtree not too different from size of the right subtree
  - or, equivalently, left and right subtree heights are not too different
- More formally, a tree is balanced if its height is $O(\log n)$
Balanced Trees

- Complete tree is an example of a balanced tree
  - used for a priority queue
  - heap supports `insert` and `removeMin`, but it does not support `find` and `remove`
  - thus heap does not work for an ordered dictionary

- There are other examples of balanced trees that support `find`, `insert`, `remove`
  - AVL trees
Definition: Height of a Node

- **height a tree node** $v$ is the height of the subtree rooted at node $v$
  - recall that tree height is the maximum over tree node depths

![Tree Diagram]

- 6
- 3
- 2
- 1
- 8
- 1
- 2
- 1
- 1
- 0
- 0
- 0
- 0
- 0
- 0
Definition: AVL Tree

- Inventors: Adel'son-Vel'skii and Landis (1962)
  - "An algorithm for organization of information", Doklady Akademii Nauk USSR
- AVL Tree is BST that satisfies the **height-balance property**
  - for every node $v$, the heights of its left and right children differ by at most 1
- Height-balance property ensures that height is logarithmic in the number of nodes
Theorem: Height $h$ of AVL tree storing $n$ keys is $O(\log n)$

Proof: Let $n(h)$ be number of internal nodes in smallest AVL tree of height $h$

- $n(1) = 1$ and $n(2) = 2$

- $n(h) = 1 + n(h-1) + n(h-2) > 1 + n(h-2) + n(h-2) > 2n(h-2)$
- solving: $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(h-6)$, ... , $n(h) > 2^i n(h-2i)$
- base case: $h-2i=1$, solving for $i$, we get $i=(h-1)/2$
- therefore $n(h) > 2^{(h-1)/2}$ $n(1)=2^{(h-1)/2}$
- take logarithms of both sides: $h < 2\log n(h) + 1$
Operations in AVL Tree

- The height of an AVL tree is $O(\log n)$
- Thus find is $O(\log n)$
  - performed just like in BST since AVL tree is BST
- Have to show how to insert and remove in AVL trees, while maintaining
  1. the height-balance property
  2. the binary search tree order
Insertion in an AVL Tree

- Insertion starts as in BST
  - expanding at external node
- Example: insert 54

```
before inserting

```

```
after inserting

```

insertion node w
Insertion in an AVL Tree

- After inserting at external, height-balance property is likely lost
- Need to restructure the tree

- Will use “pictorial” notation
Rebalancing After Insertion

- Node is **unbalanced** if difference in heights of its left and right children is more than 1
- After insertion, only heights of ancestors of insertion node \( w \) could change
  - if change, increase by exactly 1
- Need to check only ancestors of \( w \)
- Follow the path from \( w \) to the root, updating the heights and correcting any unbalanced nodes
  - in about 50% of the cases, insertion causes no unbalance
Rebalancing After Insertion

- Suppose $z$ is the first unbalanced node on the path from $w$ to the root
- Height difference between the left and the right child of $z$ is more than 1
  - tree was balanced before the insertion
  - insertion can change height only by 1
- Thus this height difference is **exactly** 2
- One subtree has height $p$, other height $p+2$
  - $w$ was inserted into the higher subtree
  - the higher subtree could be on the left as well
Rebalancing After Insertion

- Let $S$ be the higher subtree
  - could be either on the left or on the right of $z$
- $z$ was balanced before insertion, so height of $S$ was $p+1$ before insertion

- $S$ had at least one internal node before insertion, since its height is $p+1$
- Therefore $S$ has a non-leaf root
- Let $y$ be the root of $S$
Rebalancing After Insertion

- Let $y$ be the root of $S$
  - $y$ balanced after insertion
- Expand structure of $S$ before insertion
- **Case 1:** both subtrees of $S$ have height $p$
  - $y$'s balanced after insertion
- **Case 2:** one subtree of $S$ has height $p$, another height $p - 1$
  - $w$ will be inserted in higher subtree
  - impossible, since $y$ would become unbalanced after insertion
- \( w \) could have been inserted into the right subtree of \( y \)
- $S$ could be either the left or right subtree of $z$
- $w$ could have been inserted either in left or right subtrees of $y$
- Let $R$ be the higher subtree of $y$
Rebalancing After Insertion

- Let $R$ be the higher subtree of $y$
  - $R$ contains $w$, which is an internal node
    - therefore $R$ has at least one internal node
- Let $x$ be the root of $R$
  - height of $R$ went from $p$ before insertion to $p+1$ after insertion
  - $x$ was balanced before insertion and stays balanced after insertion
    - both subtrees of $x$ had height $p-1$ before insertion
    - After insertion, one subtree of $x$ has height $p$, the other height $p-1$
- y could be either left or right child of z
- x could be either left or right child of y
Before Insertion vs after Insertion
BST order is preserved

- \( z \leq x \leq y \)
- all keys in grey subtree larger than \( z \)
- all keys in yellow subtree smaller than \( y \)
This case is when $y$ to the right of $z$, $x$ to the left of $y$

Three other cases

- $y$ to the right of $z$, $x$ to the right of $y$
- $y$ to the left of $z$, $x$ to the left of $y$
- $y$ to the left of $z$, $x$ to the right of $y$
Rebalance, other cases

- y to the right of z, x is to the right of y
- y to the left of z, x is to the left of y
Rebalance, other cases

- y to the left of z, x is to the right of y
Trinode Restructuring

- **trinode restructuring** handles all cases

Call

- node $x$ as the middle (has middle key)
- node $y$ as the smallest (has smallest key)
- nodes $z$ as the largest (has largest key)

Make

- middle node new subtree parent
- smallest node its left child
- largest node its right child
- subtrees of middle node are orphaned
  - left subtree, if present, goes to the new left child
  - right subtree, if present, goes to the new right child
PseudoCode for Trinode Restructuring

Algorithm TriNodeRestructure(x,y,z)

Input: Unbalanced node z, y is the taller child of z, x is the taller child of y

Output: position of the node that goes in the place of z in the tree structure

if z.right() = y and y.left() = x then a = z; b = x; c = y
if z.right() = y and y.right() = x then a = z; b = y; c = x
if z.left() = y and y.left() = x then a = x; b = y; c = z
if z.left() = y and y.right() = x then a = y; b = x; c = z
if (z = root) then
    root = b; //root changes after triNodeRestructure
    b.parent = null
else
    // reconnect parent of z to the node replacing z
    if z.parent.left() = z then MakeLeftChild(z.parent, b);
    else MakeRightChild(z.parent, b);

if b.left() ≠ x and b.left() ≠ y // left orphan present
    MakeRightChild(a, b.LeftChild);
if b.right() ≠ x and b.right() ≠ y // right orphan present
    MakeLeftChild(c, b.RightChild);
MakeLeftChild(b,a);
MakeRightChild(b,c);
return b
Pseudo-Code for Trinode Restructuring

Algorithm TriNodeRestructure(x, y, z)
Input: Unbalanced node z, y is the taller child of z, x is the taller child of y
Output: position of the node that goes in the place of z in the tree structure

if z.right() = y and y.left() = x then a = z; b = x; c = y
if z.right() = y and y.right() = x then a = z; b = y; c = x
if z.left() = y and y.left() = x then a = x; b = y; c = z
if z.left() = y and y.right() = x then a = y; b = x; c = z
PseudoCode for Trinode Restructuring

Algorithm TriNodeRestructure(x,y,z)
Input: Unbalanced node z, y is the taller child of z, x is the taller child of y
Output: position of the node that goes in the place of z in the tree structure

if ( z = root ) then
    root = b; //root changes after triNodeRestructure
    b.parent = null
else // reconnect parent of z to the node replacing z
    if z.parent.left() = z then MakeLeftChild(z.parent, b)
    else MakeRightChild(z.parent, b)
Algorithm TriNodeRestructure(x, y, z)
Input: Unbalanced node z, y is the taller child of z, x is the taller child of y
Output: position of the node that goes in the place of z in the tree structure

if (z = root) then
    root = b; // root changes after triNodeRestructure
    b.parent = null
else // reconnect parent of z to the node replacing z
    if z.parent.left() = z then MakeLeftChild(z.parent, b)
    else MakeRightChild(z.parent, b)
Algorithm TriNodeRestructure(x,y,z)

Input: Unbalanced node z, y is the taller child of z, x is the taller child of y

Output: position of the node that goes in the place of z in the tree structure

if b.left() ≠ x and b.left() ≠ y // left orphan present
    MakeRightChild(a, b.LeftChild);
if b.right() ≠ x and b.right() ≠ y // right orphan present
    MakeLeftChild(c, b.RightChild);
Algorithm TriNodeRestructure(x, y, z)
Input: Unbalanced node z, y is the taller child of z, x is the taller child of y
Output: position of the node that goes in the place of z in the tree structure

if b.left() ≠ x and b.left() ≠ y // left orphan present
    MakeRightChild(a, b.LeftChild);
if b.right() ≠ x and b.right() ≠ y // right orphan present
    MakeLeftChild(c, b.RightChild);
Algorithm TriNodeRestructure(x, y, z)

Input: Unbalanced node z, y is the taller child of z, x is the taller child of y

Output: position of the node that goes in the place of z in the tree structure

MakeLeftChild(b, a)
MakeRightChild(b, c)

return b
PseudoCode for Trinode Restructuring

- **MakeLeftChild**(a, b) makes node b left child of node a
  - a.leftchild = b
  - b.parent = a

- **MakeRightChild**(a, b) makes node b right child of node a
  - a.rightchild = b
  - b.parent = a

- Trinode restructuring is **O(1)**
  - constant number of comparisons and assignments
After one trinode restructure, height-balance property is restored globally
- can stop checking ancestor path after trinode restructure
Algorithm AVLtreeInsert($k, o$)

Input: key $k$ and value $o$; Output: node where the entry was inserted

$w = \text{TreeInsert}(k, o, T.\text{root})$  // $w$ holds position of new entry ($k, o$)

// now need to check and if needed, restore height-balance property
$z = w$

while ($z \neq \text{null}$)  // traverse up the tree, checking for imbalance
    setHeight($z$) // reset the height of $z$ since it may have changed
    if $|\text{getHeight}(z.\text{left})-\text{getHeight}(z.\text{right})| > 1$ then
        $z = \text{TriNodeRestructure}($tallerChild(tallerChild($z$)), tallerChild($z$), $z$)$
        \text{setHeight}(z.\text{left}); \text{setHeight}(z.\text{right}); \text{setHeight}(z)$;
        break  // exit while loop, tree is balanced after 1 trinodeRestructure
    
$z = \text{parent}(z)$

return $w$

- setHeight($z$) = $1 + \max(z.\text{left}.\text{height}, z.\text{right}.\text{height})$
- tallerChild($v$) returns the child of $v$ with larger height
Insertion Example Continued

after inserting

after inserting
Inserting in AVL Tree, Summary

- After inserting into at node $w$, go up the tree, following ancestor path from $w$, checking if any node on this path has become unbalanced.

- If an unbalanced node is found, perform tri-node restructuring.
  - Tree becomes balanced and can return from insert right after trinode restructuring.

- Trace at most one path in the tree, performing constant number of operations at each node, thus insert is $O(\log n)$. 