Lecture 9: AVL Trees

- Definition
- Properties
- Insertion
BST Performance

- Recall that for a binary search tree with \( n \) nodes and of height \( h \)
  - methods find, insert and remove take \( O(h) \) time
- The height \( h \) is \( O(n) \) in the worst case and \( O(\log n) \) in the best case
- Thus in the worst case, find, insert and remove take \( O(n) \) time
Balanced Tree Motivation

- If we find a way to make sure the height $h$ of a binary search tree is always $O(\log n)$, then search tree will be much more efficient, the worst case complexity for find, insert, remove will be $O(\log n)$, significantly better than $O(n)$ for a BST.

- To make sure height $h$ of a tree is always $O(\log n)$, the tree must be “balanced”, that is for any node its left subtree should not be much higher than its right subtree.
Balanced Trees

- We have already seen an example of a balanced tree, that is the complete binary tree.

- Complete binary tree is not the only example of a balanced tree, that is tree with logarithmic height.

- How to implement a balanced tree which allows operations find, remove, insert in $O(\log n)$?

- One way is with AVL trees.
Recall that the **height of a tree** is the maximum over all node depths, or, equivalently, the longest path in the tree.

The **height of a tree node** $v$ is defined as the height of the subtree rooted at node $v$. 
AVL Tree Definition

- An AVL Tree is a binary search tree which satisfies the **height-balance property**:
  
  - for every internal node \( v \) of \( T \), the heights of the children of \( v \) can differ by at most 1.

- We will say that AVL trees are balanced.

- Inventors: Adel'son-Vel'skii and Landis 1962.

- The **height-balance property** guarantees that the height of an AVL tree is logarithmic in the number of items in the tree.

![Example of an AVL Tree](image)
**Height of an AVL Tree**

- **Theorem:** Height $h$ of AVL tree storing $n$ keys is $O(\log n)$.

- **Proof:** Let us bound $f(h)$: the minimum number of internal nodes of an AVL tree of height $h$.
  
  We easily see that $f(1) = 1$ and $f(2) = 2$.
  
  For $h > 2$, an AVL tree of height $h$ contains the root node, one AVL subtree of height $h-1$ and the other of height at least $h-2$.

  That is, $f(h) = 1 + f(h-1) + f(h-2)$.

  Knowing $f(h-1) > f(h-2)$, we get $f(h) > 2f(h-2)$.

  - The number of nodes doubles after 2 steps down the AVL tree.
  - So $f(h) > 2f(h-2), f(h) > 4f(h-4), f(h) > 8f(h-6), \ldots \ldots$, $f(h) > 2^{i}f(h-2i)$.

  Solving the base case ($h-2i=1$) we get $i=(h-1)/2$.

  Therefore $f(h) > 2^{(h-1)/2} f(1) = 2^{(h-1)/2}$.

  Taking logarithms of both sides: $h < 2 \log f(h) + 1$.

  Thus the height of an AVL tree is $O(\log n)$. 
Operations in an AVL Tree

- The height of an AVL tree is $O(\log n)$
- Thus the search operation takes $O(\log n)$
  - Performed just like in BST since any AVL tree is a BST
- All that’s left to do is 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 a binary search tree
- Always done by expanding an external node.
- Example:

Before inserting 54:

```
        44
       /   \
      17    78
     /     /   \
   32     50    88
  /     /     /    \
48     62     54    88
```

After insertion:

```
        44
       /   \
      17    78
     /     /   \
   32     50    88
  /     /     /    \
48     62     54    88
```
Insertion in an AVL Tree

- After inserting a new item at a leaf, the height-balance property of the AVL tree is very likely lost.

- Recall that **height-balance** property requires that for any internal node the height of its children can differ by at most 1.

- To make it an AVL tree again, need to restore the balance by restructuring the tree.

- “Pictorial” notation:
Analysis After Insertion

- Let us call a node unbalanced if the difference in heights of its left and right subtrees is more than 1.
- After insertion, the heights could change (increase) only for ancestors of the insertion node $w$.
  - height of a node is the length of the longest path from that node to a leaf.
- Thus the only possibly unbalanced nodes are the ancestors of insertion node $w$.
- we should search up the tree from the insertion node $w$, looking for any unbalanced nodes and correcting this unbalance somehow.
  - also update the height of each node on the path from $w$ to the root, as the heights of nodes on this path may have changed.
Analysis After Insertion

- So after inserting at node $w$, we follow the path from $w$ to the root (the path of ancestors of $w$), checking the balance of nodes
  - Usually, in about 50% of cases there is no unbalanced node after insertion
- Suppose the first unbalanced node (as you go from $w$ up the tree) is at position $z$
  - Height difference between the left and the right subtree of $z$ is more than 1
    - tree was balanced before the insertion
    - each insertion can change height only by 1
    - Thus this height difference is exactly 2
  - One subtree has height $p$, the other height $p+2$
    - $w$ was inserted into the higher subtree
Analysis After Insertion

- Height difference between left and right subtrees of $z$ is exactly 2
- One subtree has height $p$, the other height $p+2$
  - $w$ was inserted into the higher subtree
- There are 2 cases: right subtree is higher or left subtree is higher

**case 1:**

- Name the highest subtree $S$
- Since tree was balanced before insertion, height of $S$ was $p+1$ before insertion
- Name the root of $S$ with $y$
- Since $y$ is balanced after insertion, and $z$ is not balanced after insertion, both subtrees of $y$ have height exactly $p$ before insertion
Analysis After Insertion

**Case 1:**
- Height $p+2$
- Height $p$

**Case 2:**
- Height $p+2$
- Height $p$

*S before insertion, height $p+1*~
*S after insertion, height $p+2*~

**Case a**
- Height $p$
- Height $p$

**Case b**
- Height $p+1$
- Height $p+1$
- Height $p$
After Insertion: 4 cases

- **case 1:**
  - Height $p$
  - Height $p$

- **case 2:**
  - Height $p$

- **case 3:**
  - Height $p$
  - Height $p + 1$

- **case 4:**
  - Height $p$
  - Height $p + 1$
Analysis After Insertion

case 1:

- Let $R$ be the name of the right 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 $R$
- There are 2 cases
  - Case 1: $x = w$ in which case $p = 0$ and both subtrees of $w$ are leafs
  - Case 2:
    - height of $R$ went from $p$ before insertion to $p+1$ after insertion
    - $x$ was balanced before insertion and is 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 $p-1$
## Analysis After Insertion: 4 cases

**Case 1:**
- Tree structure with nodes labeled `z`, `y`, and `x`.
- Heights: `height p`, `height p`, `height p`, and `height p`, respectively.

**Case 2:**
- Tree structure with nodes labeled `z`, `y`, and `x`.
- Heights: `height p`, `height p`, and `height p`, respectively.

**Case 3:**
- Tree structure with nodes labeled `z`, `y`, and `x`.
- Heights: `height p`, `height p`, and `height p`, respectively.

**Case 4:**
- Tree structure with nodes labeled `z`, `y`, and `x`.
- Heights: `height p`, `height p`, and `height p`, respectively.
One More Picture

after insertion

before insertion
Rebalance, Finally! Case 1

- Tree is unbalanced at node $z$
  - left subtree of $z$ has height $p$
  - right subtree of $z$ has height $p+2$

- Tree is balanced at node $y$
  - left subtree of $y$ has height $p+1$
  - right subtree of $y$ has height $p+1$

- Tree is balanced at nodes $z$, $x$
- The binary search tree order is preserved
Rebalance, Case 2, 3, 4

Case 4:

Case 3:

Case 2:

height p, height p-1
Trinode Restructuring

- All 4 cases can be coded with the same algorithm, called **trinode restructuring**

- In all 4 cases, out of 3 nodes \( x, y, z \), we make
  - the node with the **middle** key the new parent
  - the smallest key node as its left child
  - the largest key node as its right child
  - for the “new parent” the old 1 or 2 subtrees not rooted at \( x, y, z \) need to be put in the appropriate locations
    - The left subtree (if present) goes with the new left child
    - The right subtree (if present) goes with the new right child
PseudoCode for Trinode Restructuring

Algorithm $\text{TriNodeRestructure}(x,y,z)$

Input: node $x$, its parent $y$, its grandparent $z$. Node $z$ is not balanced
Output: position of the node that goes in the place of $z$ in the tree structure

if $\text{key}(z) \leq \text{key}(x)$ and $\text{key}(x) \leq \text{key}(y)$ then $a = z; b = x; c = y$
if $\text{key}(z) \geq \text{key}(x)$ and $\text{key}(x) \geq \text{key}(y)$ then $a = y; b = x; c = z$
if $\text{key}(z) \leq \text{key}(y)$ and $\text{key}(y) \leq \text{key}(x)$ then $a = z; b = y; c = x$
if $\text{key}(z) \geq \text{key}(y)$ and $\text{key}(y) \geq \text{key}(x)$ then $a = x; b = y; c = z$

if $(z = \text{root})$ then
    $\text{root} = b;$ //In this case, root changes after triNodeRestructure
    $\text{b.parent} = \text{null}$
else // reconnect parent of $z$ to the node replacing $z$
    if $z.\text{Parent.leftChild} = z$ then MakeLeftChild($z.\text{Parent}$, $b$);
    else MakeRightChild($z.\text{Parent}$, $b$);

if $b.\text{LeftChild} \neq x$ and $b.\text{LeftChild} \neq y$ and $b.\text{LeftChild} \neq z$ then
    MakeRightChild($a$, $b.\text{LeftChild}$);
if $b.\text{RightChild} \neq x$ and $b.\text{RightChild} \neq y$ and $b.\text{RightChild} \neq z$ then
    MakeLeftChild($c$, $b.\text{RightChild}$);

$\text{MakeLeftChild}(b, a)$;
$\text{MakeRightChild}(b, c)$;

return $b$
PseudoCode for Trinode Restructuring

- On the previous slide, method
  - \textit{MakeLeftChild}(a,b): makes node \textit{b} the left child of node \textit{a}. This involves 2 steps
    - \textbf{a.leftchild} = \textit{b}
    - \textbf{b.parent} = \textit{a}

- \textit{MakeRightChild}(a,b): makes node \textit{b} the right child of node \textit{a}. This involves 2 steps:
  - \textbf{a.rightchild} = \textit{b}
  - \textbf{b.parent} = \textit{a}

- Trinode restructuring takes \textit{O(1)}
  - no loops, no recursive calls, just a constant number of comparisons and changes in parent-child relationships
**Insertion and Trinode Restructuring**

- **Only 1 trinode restructuring is necessary per insertion**
  - After restructuring, the height of the subtree formerly rooted at $z$ (now rooted at $x$ or $y$) is the same as before insertion, namely $p+2$, and the tree was balanced before insertion.
  - Thus, after insertion, trinode restructuring restores height-balance order *globally*.

Before insertion:

- $z$ (height $p+2$)
- $y$ (height $p+1$)

After insertion, but before trinode restructuring:

- $z$ (height $p+3$)
- $w$ (height $p+2$)

After insertion and trinode restructuring:

- $x$ or $y$ (height $p+2$)
PseudoCode for Insertion into AVL Tree

Algorithm $AVLtreeInsert(k,o)$
Input: key k and value o;  Output: node where the entry was inserted

\[ w = TreeInsert(k, o, T.root) \]  \( w \) 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
  \[ \text{setHeight}(z); \]  // reset the height of \( z \) since it may have changed
  if \( |\text{getHeight}(z.left) - \text{getHeight}(z.right)| > 1 \) then
    \[ z = \text{TriNodeRestructure}(\text{tallerChild(\text{tallerChild}(z))}, \text{tallerChild}(z), z) \]
    \[ \text{setHeight}(z); \text{setHeight}(z.left); \text{setHeight}(z.right); \text{setHeight}(z); \]
    break;  // exit while loop, tree is balanced after 1 trinodeRestructure
  \[ z = \text{parent}(z) \]

return \( w \)

- $\text{setHeight}(z) = 1 + \max(z.left.height, z.right.height)$
- $\text{tallerChild}(\text{node})$ gives the child with larger height. Thus $y = \text{tallerChild}(z)$, and $x = \text{tallerChild(tallerChild}(z))$
- complexity of $AVLtreeInsert(k,o) : O(\log n)$
Insertion Example, continued

unbalanced...

...balanced
Insertion in AVL Tree: Summary

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

- If an unbalanced node is found, perform **tri-node restructuring**
  - Since after 1 such restructuring, the tree has become balanced and thus is an AVL tree, can return right after tri-node restructuring.

- Travel up the tree is $O(\log n)$, and tri-node restructuring is $O(1)$.