Lecture 4
Prof. Piotr Indyk
Lecture Overview

- Review: Binary Search Trees
- Importance of being balanced
- Balanced BSTs
  - AVL trees
    - definition
    - rotations, insert
Binary Search Trees (BSTs)

- Each node $x$ has:
  - key[$x$]
  - Pointers: left[$x$], right[$x$], p[$x$]

- Property: for any node $x$:
  - For all nodes $y$ in the left subtree of $x$:
    \[
    \text{key}[y] \leq \text{key}[x]
    \]
  - For all nodes $y$ in the right subtree of $x$:
    \[
    \text{key}[y] \geq \text{key}[x]
    \]
The importance of being balanced

h = Θ(log n)

h = Θ(n)
Balanced BST Strategy

- **Augment** every node with some data
- Define a local **invariant** on data
- Show (prove) that invariant guarantees $\Theta(\log n)$ height
- Design algorithms to maintain data and the invariant
AVL Trees: Definition

[Adelson-Velskii and Landis’62]

• **Data**: for every node, maintain its height ("augmentation")
  
  - Leaves have height 0
  - NIL has "height" -1

• **Invariant**: for every node x, the heights of its left child and right child differ by at most 1
AVL trees have height $\Theta(\log n)$

**Invariant:** for every node $x$, the heights of its left child and right child differ by at most 1

- Let $n_h$ be the minimum number of nodes of an AVL tree of height $h$
- We have $n_h \geq 1 + n_{h-1} + n_{h-2}$
  \[ \Rightarrow n_h > 2n_{h-2} \]
  \[ \Rightarrow n_h > 2^{h/2} \]
  \[ \Rightarrow h < 2 \log n_h \]
- The constant “2” can be improved

How can we maintain the invariant?
Rotations maintain the inorder ordering of keys:

\[ a \in \alpha, \ b \in \beta, \ c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c. \]
Insertions

• Insert new node $u$ as in the simple BST
  – Can create imbalance
• Work your way up the tree, restoring the balance
• Similar issue/solution when deleting a node
Balancing

• Let $x$ be the lowest “violating” node
  – We will fix the subtree of $x$ and move up
• Assume the right child of $x$ is deeper than the left child of $x$ ($x$ is “right-heavy”)
• Scenarios:
  – Case 1: Right child $y$ of $x$ is right-heavy
  – Case 2: Right child $y$ of $x$ is balanced
  – Case 3: Right child $y$ of $x$ is left-heavy
Case 1: $y$ is right-heavy

\text{LEFT-ROTATE}(x)
Case 2: \( y \) is balanced

Same as Case 1
Case 3: \( y \) is left-heavy

\[
\begin{array}{c}
x
\end{array}
\xrightarrow{\text{LEFT-ROTATE}(x)}
\begin{array}{c}
y
\end{array}
\]

Need to do more …
Rotations

Rotations maintain the inorder ordering of keys:
• \( a \in \alpha, \ b \in \beta, \ c \in \gamma \ \Rightarrow \ a \leq A \leq b \leq B \leq c. \)
Case 3: $y$ is left-heavy

Right-Rotate ($y$)  
Left-Rotate ($x$)

And we are done!
Conclusions

- Can maintain balanced BSTs in $O(\log n)$ time per insertion
- Search etc take $O(\log n)$ time
Examples of insert/balancing

Insert(23)

x = 29: left-left case

Done

Insert(55)

x = 65: left-right case

Done
Balanced Search Trees …

- AVL trees (Adelson-Velsii and Landis 1962)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Scapegoat trees (Galperin and Rivest 1993)
- Treaps (Seidel and Aragon 1996)
- …
BST for runway reservation system

- $R = (37, 41, 46, 49, 56)$ current landing times

- remove $t$ from the set when a plane lands
  - $R = (41, 46, 49, 56)$

- add new $t$ to the set if no other landings are scheduled within $< 3$ minutes from $t$
  - 44 => reject (46 in $R$)
  - 53 => ok

- delete, insert, conflict checking take $O(h)$, where $h$ is the height of the tree
And some people like to do nothing