6.006 Lecture 4: BST’s & Balanced BSTs

- BST’s & BST operations in $\Theta(h)$ time
- Balance $\Rightarrow$ height $= \Theta(\lg n)$
- AVL Trees

> CLRS 13.1 & 13.2, but Red-Black trees, not AVL trees
Binary Search Trees

BST property:

BST operations start at the root and only visit vertices along a path to a leaf

- find
- findmax: go right until can't
  - findmin: go left until can't
- delete min:

next-larger: a bit trickier
delete: even more :-)
Balanced Trees

operations cost bounded by path from root to a leaf; how long can the path be?
(Height of tree is defined as longest such path).

perfectly balanced
\( h = \Theta(\lg n) \)

same \( n \), poorly balanced
\( h = \Theta(n) \), expensive operations
Balanced BST Strategy

- Augment every node with some property
- Define a local invariant on property
- Show (prove) that invariant guarantees $\Theta(\log n)$ height
- Design algorithms to maintain property & to fix up the invariant.

AVL trees (Adelson-Velskii - Landis 1962)

- Property: height of most distant leaf
- Invariant: heights of left & right children differ by at most $\pm 1$
  (define height of None as $-1$)

- Thm: Height of AVL tree $\leq 2 \log n$

Proof: Let $N_h$ be min # nodes in height-$h$ AVL tree.

\[ N_h = N_{h-2} + N_{h-1} + 1 \quad \text{(picture above)} \]

\[ > 2N_{h-2} \]

\[ N_0 = 1 \Rightarrow N_h > 2^{h/2} \Rightarrow \frac{h}{2} < \log N_h \Rightarrow h < 2\log N_h \]

For a given tree with height $h$ and $N$ nodes,

$N_h = \log h$.
When you insert (delete, etc), the invariant may get violated.

Example: Insert(23)

Example: Insert(55)

A single rotation on 29

A double rotation
General Rotation Rules

Case 1:

\[
\begin{array}{ccc}
\text{X} & \Rightarrow & \\
\text{A} & & \text{Y}
\end{array}
\]

\[
\begin{array}{ccc}
\text{B} & & \text{C}
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k & k+1
\end{array}
\]

\[
\begin{array}{ccc}
k & k-1 & k
\end{array}
\]

Case 2:

\[
\begin{array}{ccc}
\text{X} & \Rightarrow & \\
\text{A} & & \text{Y}
\end{array}
\]

\[
\begin{array}{ccc}
\text{B} & & \text{C}
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k & k
\end{array}
\]

\[
\begin{array}{ccc}
k+1 & k-1 & k
\end{array}
\]

Case 3:

\[
\begin{array}{ccc}
\text{X} & \Rightarrow & \\
\text{A} & & \text{Y}
\end{array}
\]

\[
\begin{array}{ccc}
\text{B} & & \text{C}
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k & k-1
\end{array}
\]

\[
\begin{array}{ccc}
k+1 & k-1 & k
\end{array}
\]

\[
\text{Violates our property; not a useful rotation on this tree}
\]

Case 3: (again)

\[
\begin{array}{ccc}
\text{X} & \Rightarrow & \\
\text{A} & & \text{Y}
\end{array}
\]

\[
\begin{array}{ccc}
\text{B}_1 & & \text{B}_2
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k & k-1
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k-1 & k
\end{array}
\]

\[
\begin{array}{ccc}
k-1 & k-2 & k-1
\end{array}
\]

\[
\text{In general, you can use rotations to transform any shape tree to any other shape using rotations.}
\]

- Here we assumed that right subtree of x is higher other case is symmetric
- We also assumed that x is off-balance by 1
- During insertions, this is always the case & we can fix up the balance going up the tree in \( \Theta(\log n) \) time.
Other search trees

- AVL trees
- 2-3 trees \(\Rightarrow\) B-trees
- weight-balanced trees a.k.a. \(BB[\alpha]\)
- red-black trees
- skip lists
- splay trees
- scapegoat trees
  - Rivest & Galperin
  - amortized insert, delete

Why did people invent so many trees?

- B-trees: very shallow, very little random access
- \(BB[\alpha]\): tune insert/delete vs search
- red-black: only 1 extra bit per node
- splay, scapegoat: no extra data in nodes, lower constants, but only amortized guarantees

(Insertions into lists in Python take constant amortized time; once in a while, the list is copied to a larger array to make room for more insertions; when do you trigger this?)

- van Emde Boas trees: \(O(\log \log u)\) operations for integer key \(1 \ldots u\)