csce750 — Analysis of Algorithms
Fall 2019 — Lecture Notes: Balanced Binary Search Trees

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

CLRS 12, 13

A **binary search tree** is a data structure that supports these operations:

- INSERT(k)
- SEARCH(k)
- DELETE(k)

Basic idea: Store one key at each node.

- All keys in the left subtree of n are less than the key stored at n.
- All keys in the right subtree of n are greater than the key stored at n.

Search and insert are trivial. Delete is slightly trickier, but not too bad.

2 BST Analysis

Each operation can be done in time $O(h)$ on a BST of height h.

Worst case: $\Theta(n)$

Aside: Does randomization help?

- Answer: Sort of. If we know all of the keys at the start, and insert them in a **random** order, in which each of the $n!$ permutations is equally likely, then the expected tree height is $O(\lg n)$. (See CLRS 300.)

3 Balancing

To be a useful improvement over a linked list, a BST must be kept **balanced**, ensuring that its height remains $O(\lg n)$.

There are lots of schemes to keep BSTs balanced:

- AVL trees: Heights of left and right subtrees differ by at most 1.
- Red-black trees: Heights of left and right subtrees differ at most by a factor of 2.

You have likely seen one or both of these at some point.

We'll have a look at a more exotic variation: Treaps.

4 Rotations

Treaps (and AVL trees, and Red-Black trees, and ...) use a pair of operations called **rotations** to change the structure of the BST *without breaking the BST property.*

- Left rotation
- Right rotation

Pseudocode: CLRS 313

5 It's a tree! It's a heap! It's a treap!

In a treap, every node is labeled with both a unique key and a unique numerical **priority**.

For each node v:

- $v.\mathrm{key}$ is greater than all keys in the subtree rooted at $v.\mathrm{left}$
- $v.\mathrm{key}$ is less than all keys in the subtree rooted at $v.\mathrm{right}$
- $v.\mathrm{left.priority} < v.\mathrm{priority}$
- $v.\mathrm{right.priority} < v.\mathrm{priority}$

Since a treap is a BST, the standard search algorithm works.

6 Treap insertion

To insert a new key k with priority p into a treap:

- Use the standard BST insertion algorithm to add a new leaf (k, p).
- Visit each node v on the path back from this new node to the root.
  - If $v.\mathrm{priority} < v.\mathrm{left.priority}$, then $\textsc{RotateR}(v)$.
  - If $v.\mathrm{priority} < v.\mathrm{right.priority}$, then $\textsc{RotateL}(v)$.
  - Otherwise, stop.

7 Treap example

8 Treap analysis

Two observations:

1. Given the keys and their priorities, the shape of the treap is fully determined. (Proof by induction: The heap property ensures that the highest priority node is the root. The BST property uniquely partitions the remaining nodes, which form sub-treaps. Base case: Empty treap.)
2. The uniquely determined shape is identical to the shape that would result from inserting the elements into a standard (unbalanced) BST, in order of decreasing priority.

Therefore: The analysis for inserting known keys in random order applies here, because we get the same tree. The expected height of a treap of n nodes is $O(\lg n)$.

Therefore: Searching, inserting, and deleting in a treap each take worst-case expected time $O(\log n)$.