1 Introduction

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.

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/10-treaps.pdf

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 $O(lg n)$. (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 tree, 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.key is greater than all keys in the subtree rooted at v.left
- v.key is less than all keys in the subtree rooted at v.right
- v.left.priority < v.priority
- v.right.priority < v.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.priority < v.left.priority, then ROTATER(v).
  - If v.priority < v.right.priority, then 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 \( O(\log n) \).

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