Recap

We have seen many variants on the binary search tree

- **(Standard) Binary search trees**: No balance. $O(\log n)$ height/time if operations are random
- **AVL trees**: A classic, height-balanced binary tree. $O(\log n)$ performance guaranteed. Good, but not the fastest in practice
- **2-3 trees**: A tree that allows nodes to have 2 or 3 children. $O(\log n)$ performance guaranteed. Some space wastage
- **Red-black trees**: A binary implementation of 2-3 (actually 2-3-4) trees. $O(\log n)$ performance guaranteed. Considered among the fastest deterministic structures
- **AA trees**: A kinder, simpler red-black tree
- **Treap and Skiplists**: Randomized search structures. $O(\log n)$ performance in expectation (over random choices). Very simple and practical
Recap

Are we done yet?

- There are still many interesting extensions
  - **Order-statistic queries**: Find the $k$th smallest key
  - **Range queries**: Count/sum/report all the keys in the interval $[x_0, x_1]$
  - **Split/Merge**: Given a tree $T$ and key $x$, split $T$ into subtrees $T_1$ and $T_2$, such that keys in $T_1$ are at most $x$, and keys in $T_2$ are greater than $x$. **Merge** reverses this, melding two trees (with one having keys smaller than the other) into a single tree
  - **Expected-case Optimal Trees**: Given access probabilities for the elements, build a tree that minimizes the expected search time. (Static optimality)
Recap

Optimal Binary Search Trees

- **Optimal Search Trees:**
  - Let \( \{x_1, \ldots, x_n\} \) be the keys
  - Let \( p_i \) denote the access probability of \( x_i \). Where, \( 0 \leq p_i \leq 1 \), and \( p_1 + \cdots + p_n = 1 \).
  - High-probability items should be stored near root
  - Can be solved by dynamic programming

- **Static optimality:** We assume that access probabilities never change

- **Dynamic optimality?**
  - Suppose that access probabilities do change.
  - Can we build a tree that automatically adjusts to the current distribution?
  - Yes! Splay trees
Splay Trees

Intuition

- We seek a tree structure that readjusts itself, depending on the access pattern
- Want low-probability nodes near the bottom and high-probability nodes near top
- Intuition:
  - Keys near the bottom of long access chains have high cost
  - Whenever we access a key, let’s pull it up to the root
  - Frequently accessed keys will tend to “rise to the top,” leading to faster access and better expected performance
- But how do we pull a node up to the root?
  - Need to preserve inorder structure - use rotations!
Splay Trees
A good idea, that doesn’t work

- Here is an idea for a restructuring operation, that *doesn’t work*
  - Let $p$ be the node we wish to access
  - Apply rotations along the path from $p$ back to the root, thus pulling $p$ up to the root
- Unfortunately, while this brings $p$ to the root, the rest of the tree structure may remain poorly balanced
Splay Trees

Fixing our idea

- There is an easy fix, however. Perform rotations two at a time!
- If done properly, the search path length reduces by roughly half

- Can we make this idea rigorous?
Let T be a splay tree. The operation T.splay(p) rotates a node p to the root.

Case 1: (Zig-zig) p is the left-left or right-right grandchild of some node
  - Do two rotations. First at p’s grandparent, then at p’s parent
Splay Trees

Basic Splay Operations

- **Case 2:** *(Zig-zag)* \( p \) is the left-right or right-left grandchild of some node
  - Do two rotations. First at \( p \)'s parent, then at \( p \)'s grandparent
Splay Trees

Basic Splay Operations

- **Case 3**: *(Zig)* p is a child of the root
  - Do two a single rotation, pulling p up to the root

- **Case 4**: *(End)* p is the root - We’re done
Splay Trees
A Self-Adjusting Tree Structure

- **T.splay(x):**
  - Apply a standard tree descent to find $x$ in the tree.
  - Let $p$ be the node containing $x$ (if present) or the last node visited before falling out (if not). Note that $p$ either contains $x$ or its inorder predecessor or successor.
  - Apply zig-zig, zig-zag rotations until almost to root.
  - If needed, apply one final zig rotation to finish things off.
Splay Trees
A Self-Adjusting Tree Structure

- $T$.splay(3):
Splay Trees

Dictionary Operations

- **T.find(x):**
  - T.splay(x). Check whether root contains key x

- **T.insert(x, v):**
  - T.splay(x). If root contains x, duplicate!
  - Let y be root. If y < x, link subtrees together as shown below (other case symmetrical)
Splay Trees

Dictionary Operations

- **T.delete(x):**
  - T.splay(x). Check that x is at root (if not, key not found!)
  - Let L and R be left and right subtrees. If either is null, return the other
  - If both are non-null, do R.splay(x)
  - New root y is smallest key in R (so its left child is null)
  - Relink trees as shown below
Splay Trees

Amortized Analysis (Optional)

- **Potential:**
  - A function $\Phi$ that represents how imbalanced the tree $T$ is
  - $\Phi$ is like a bank account that can be spent to balance the tree
  - There must always be sufficient funds in this account

- **Amortized cost:** For any operation, there are two costs to consider:
  - The actual cost of the $i$th operation (number of rotations): $C_i$
  - The change in the tree’s potential: $\Delta \Phi_i = \Phi_i - \Phi_{i-1}$
  - Amortized cost of $i$th operation is defined to be: $A_i = C_i + \Delta \Phi_i$
  - **Objective:** Prove that amortized cost is $O(\log n)$ for every operation

- **Intuition:** Can tolerate a high actual cost, if there is a large decrease in potential
Splay Trees
Amortized Analysis (Optional)

- **Potential:**
  - For each node $p$ in the tree, $size(p) = \text{number of nodes in } p\text{'s subtree}$
  - Define $rank(p) = \lg size(p)$ (intuitively, this is ideal height of $p\text{'s subtree}$)
  - $\Phi(T) = \sum_{p \in T} rank(p)$

- **Rotation Lemma:** Given any node $p$, let $rank(p)$ and $rank'(p)$ be its rank before and after a rotation operation. Then:
  - Amortized cost of zig-zig or zig-zag is $\leq 3(rank'(p) - rank(p))$
  - Amortized cost of zig is $\leq 1 + 3(rank'(p) - rank(p))$
  - (See lecture notes for the proof...it’s not easy!)

- **Splay Lemma:** Amortized cost of $T.splay(p)$ is $\leq 1 + 3(rank(root) - rank(p))$

- **Corollary:** Amortized cost of $T.splay(p)$ is $O(\log n)$
Splay Trees

Splay trees have an amazing set of properties

- Consider any sequence $S$ of $m$ accesses to a splay tree of size $n$
- **Balance Theorem**: The running time of $S$ is $O(m \log n + n \log n)$
- **Static Optimality**: Let $q_x$ be the number of times that $x$ is accessed in $S$. Then the running time of $S$ is $O(m + \sum q_x \log \frac{m}{q_x})$. This is theoretically optimal (the Entropy of the access distribution)
- **Dynamic Finger Theorem**: Number the elements 1 through $n$. Given a sequence of accesses $x_1, ..., x_m$, the running time of $S$ is $O(m + \sum \log(|x_i - x_{i-1}|+1))$
- **Working-Set Theorem**: Each time we access $x$, let $t(x)$ denote the number of accesses since the last time $x$ was accessed, then the running time of $S$ is $O(m + \sum \log(t(x) + 1))$
- **Scanning Theorem**: The time to access all elements in order is $O(n)$
Splay Trees
- Self-adjusting binary search tree
- Basic operation splay(x) - Brings x to root and reorganizes the tree
  - Zig-zig
  - Zig-zag
  - Zig
- Splay has the effect of turning long stringing search paths into bushier ones
- Amortized cost is $O(\log n)$ per dictionary operations (find, insert, delete)
- Splay trees satisfy an impressive set of optimality properties
- Not widely used, however, because constant factors are high