Lecture 24

Advanced Tree Structures

- Binary Search Trees
  - Efficiency
- Advanced Tree Structures
  - Splay Trees
  - Red-Black Trees
  - Skew Heaps
Binary Search Trees
continued ...
Efficiency
Discussion on Efficiency

The operations `find / insert / delete` will take $O(h)$-time where $h$ is the height of the tree.

**Question:** How does $h$ relate to the number of values in the tree $n$?

**Answer:** In the worst case $h = n-1$. Thus, these operations take $O(n)$-time in the worst-case. In the best case $h = \log(n)$.

**Question:** What is the "expected" height of a binary search tree?

**Answer:** $O(\log n)$. Thus, the operations take $O(\log n)$-time in an expected sense. (Recall Quick Sort.)

Why is the expected height the same as the best case height, instead of the worst-case height? Intuitively, it is because there are more ways in which the better cases can be created. We'll investigate this in the next activity.
Activity: `insert` Sequences

We will consider binary search trees with values 1, 2, ..., n where \( n = 2^m - 1 \). This \( n \) simplifies the activity. The total number of possible binary trees with \( n \) nodes is the \( n \)th Catalan number \( C(n) \) (seen earlier).

- How many binary trees have the worst-case height of \( h = n - 1 \)?
- How many binary trees have the best-case height of \( h = \lceil \log(n) \rceil = m - 1 \)?

Discuss with a neighbor for 2 minutes.
Then again for 2 more minutes.

Oh No! The above answers imply that there are more worst-case trees than best-case trees. Now consider how a binary search tree is created by a sequence of \( n \) calls to `insert`. There are \( n! \) possible sequences. For example, `insert(1), insert(2), ...` is one sequence.

- How many sequences create one of the worst-case trees with height \( h = n - 1 \)?
- How many sequences create the best-case tree with height \( h = m - 1 \)?

In general, the majority of sequences create binary search trees that have height closer to \( m \) than \( n \).
Advanced Tree Structures
Improving Upon the Tree Data Structures

We have now seen two specific ways to store values within a binary tree:

- A binary search tree’s values are $\leq$ in the left subtree and $\geq$ in the right subtree.
- A binary heap’s values are $\geq$ in the children, and it has a fixed shape.

These are the simplest structures of their respective types.

- We can improve the practical performance of a BST using Splay Trees (§14.5–14.6).
- We can obtain $O(\log n)$-time guarantees for BSTs using Red-Black Trees (§14.7).
- We can efficiently merge two heaps using Skew Heaps (§13.4.3).

More generally, it is important to view this course as an introduction to data structures.
Splay Trees
Splay Trees (§14.5–14.6)
A splay tree rearranges its nodes after each insert or delete using a splay operation, which involves a small number of tree rotations.

It improves run-times in practice, but it does not provide $O(\log n)$-time guarantees.

This idea of optimizing the links within a structure comes up in other data structures.

For example, path compression is a common feature of disjoint-set (or “union-find”) data structures, which you may see in CSCI 256.

The splay operation involves rotating the binary tree. The goal is to keep the tree as balanced as possible.
Red-Black Trees
Red-Black Trees (§14.7)

A self-balancing binary search tree performs additional work to ensure that it always has guaranteed logarithmic height in the number of nodes.

One example is a red-black tree, named for having two different types of nodes. It maintains several conditions, including this property: every path from a node to a leaf has the same # of black nodes.

- This requires a number of delicate cases involving some constant-time tree rearrangements.
- Difficult to implement correctly.

In this Red-Black Tree every path from the root to a leaf goes through exactly 3 black nodes (including null nodes).

Other self-balancing trees include splay trees, AVL trees, B trees, and more.

Related: A 2-3 tree has both regular nodes and big nodes with 2 values and 3 children.
Skew Trees
**Skew Heaps (§13.4.3)**

In some situations it can be helpful to *merge* two binary heaps. In other words, we want to create a new heap that has the union of values in two heaps.

The implementation that we studied does not provide any logarithmic benefits when merging. It takes $O(n)$-time, where $n$ is the sum of the nodes.

A skew heap allows for merging in $O(\log n)$-time.

- Also see leftist tree for $O(\log n)$-time merging.

Given the efficiency of this operation, it makes sense to reformulate the other operations in terms of it.

- An insert (add) runs merge with the heap and a new singleton heap containing the new value.
- A delete-min (remove) deletes the root, and then runs merge on the two subheaps.