Introduction

Remember binary search trees and how they can be used to store values using the BST property. This week, we will look at improving the worse case performance of BSTs. We want to guarantee logarithmic costs for \texttt{insert}, \texttt{find}, and \texttt{delete}. By enforcing the invariant that the BST always has a height of roughly $\lg n$, we will ensure that all of the operations require only $O(\lg n)$ comparisons.

2-3 Search Trees

We will first consider 2-3 Search Trees, which can contain both regular BST nodes (one key and 2 children) and nodes with two keys and 3 children: 2-nodes and 3-nodes, respectively. A 2-node has two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys. A 3-node, with two keys (and associated values) has three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node’s keys, and a right link to a 2-3 search tree with larger keys. We want to maintain a perfectly balanced 2-3 search tree, which is one whose null links are all the same distance from the root. In class we covered insertion under the following cases: insert into a 2-node (makes a 3-node), insert into a single 3-node (creates a 4-node then adds a level to the tree), insert into a 3-node with a 2-node as a parent (makes the parent a 3-node), insert into a 3-node with a 3-node parent (makes a 4-node, then splits and makes the parent a 4-node, which again splits and increases the size of its parent till we reach the root, in which case we get to one of the other cases). Importantly, we can see that the only case where the height of the tree increases is when we split a 4-node at the root, which ensures we keep the perfect balance invariant.

![Figure 1: Insertion Cases for 2-3 Trees](image)
Red-Black BSTs

Directly implementing perfectly balanced 2-3 trees might be quite painful! Keeping track of different kinds of nodes and making multiple moves around the tree would be very cumbersome. Instead, we will look at ways to represent 2-3 trees as BSTs. We can represent 2-nodes exactly the same we used to, and for 3-nodes we will add left-leaning “red” links so that we split as follows:

![Figure 2: A 3-node in a red-black tree](image)

Insert in a Red-Black Tree

Keeping track of which links are red allows us to view pairs of nodes connected by them 3-nodes, allowing us to keep many of the same concepts we used for balancing 2-3 trees. We can ensure that our red links are only for left children and that no node has two links by using tree rotations and color flips, as shown in class. The extra cases can be accounted for in the insert by fixing three cases: red right child and black left child (`rotateLeft`), both red children (`flipColors`), and two red left links in a row (`rotateRight`).

![Figure 3: The 3 Red Black Operations described below](image)

Problems

Valid BST

Design an algorithm to decide if a given binary tree is a valid binary search tree.

Practice Inserting in a Red-Black Tree

Insert the key 29 into the following red-black tree:

![Image of red-black tree with key 29 inserted](image)
Prove Height of Red-Black Tree is Logarithmic

Prove the height of a red-black tree with \( n \) internal nodes is \( O(\log n) \).

Theory of Rotations

Given two BSTs \( T_1 \) and \( T_2 \) that contain the same keys, prove that it is always possible to transform one of the BSTs into the other using some sequence of left- and/or right-rotates.