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 insert, find, and delete. By enforcing the invariant that the BST always has a height of roughly $\log n$, we will ensure that all of the operations require only $O(\log n)$ 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-url)
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.

Solution. This problem is a bit trickier than it may at first seem. The first thing that may come to mind is to simply visit every node in the tree and check that the current node’s key is greater than the key of its left child and less than the key of its right child. However, this algorithm does not work on the following tree:
We can see that every node is in the correct order in relation to its immediate children, but 4 should not be in the left subtree of 3. So we will have to refine our algorithm a bit.

We want to use a similar algorithm, but we need a way of knowing the valid values that can appear in the current subtree of the BST. So we can define a recursive algorithm that begins at the root of the tree and works down. Each recursive call takes in a node to visit, and a minimum and maximum value that define the range of values that can appear in the subtree rooted at that node. We begin at the root x and set $\text{min} = -\infty$ and $\text{max} = \infty$. Now we check the keys of the children of x to make sure that they are in the correct order in comparison to x and ensure that the keys are in the range strictly between $\text{min}$ and $\text{max}$. If these conditions are not met, then return false. Otherwise, recursively call this algorithm on the left child of x with the same $\text{min}$ value but with $\text{max} = x.key$. We also recursively call the algorithm on the right child of x with the same $\text{max}$ value but with $\text{min} = x.key$. If this procedure visits the entire tree without returning false, then the binary tree is indeed a BST.

Another solution, which is particularly elegant, is to simply perform an inorder traversal on the given tree and check that the elements are visited in sorted order. If they are not in sorted order, then the tree is not a BST. Otherwise, it is a BST. This gives us an easy $O(n)$ time algorithm to check if a binary tree is a binary search tree.

**Practice Inserting in a Red-Black Tree**

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

```
34B
 / \
30B 36B
 / \
28R
```

**Solution.**

```
34B
 / \
29R 36B
 / \
28B 30B
```

**Prove Height of Red-Black Tree is Logarithmic**

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

**Solution.** Remember that an invariant of red-black trees is that for a given node in the tree, the number of black nodes along the path from that node to any of its leaves will be the same. Thus, let us define the black-height of a node $x$, $bh(x)$, as the number of black nodes on a simple path from that node to a leaf. First, we will show that the number of internal nodes at a subtree rooted at node $x$ is at least $2^{bh(x)} - 1$. We will do this by inducting on the height of the tree:

**Base case:** $h = 0$: The node is a leaf and contains no internal nodes. $0 = 2^0 - 1$.

**Induction step:** Each child of a node $x$ has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether the child is red or black. By the induction hypothesis (since the children have a height less than the parent, but this is technically strong induction since we don’t know both have a height one less than the parent), each of the children has at least $2^{bh(x) - 1} - 1$ internal nodes. Thus, node $x$ has at least $2 \times (2^{bh(x) - 1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

Next, we observe that a node’s black-height must be at least half of the node’s total height; that is, $bh(root) \geq$
This is due to the invariant that there can’t exist two red nodes in a row on any path down the tree; thus the worst case would be an alternating sequence of red and black nodes. Using this information, we can upper bound the height of a red-black tree with \( n \) nodes:

\[
2^{h} - 1 \leq n \leq 2^{h} - 1
\]

Solving for \( h \):

\[
2^{\frac{h}{2}} - 1 \leq n \\
2^{\frac{h}{2}} \leq n + 1 \\
h \leq 2 \ast \lg(n + 1)
\]

Thus, the height of a red-black tree is \( O(\lg n) \).

**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.

**Solution.** Rotate the smallest key in \( T_1 \) to the root; then repeat the same process on the resulting right subtree until you have a right-leaning tree of height \( n \) (every left link will be null). Do the same with \( T_2 \). To transform \( T_1 \) into \( T_2 \), simply perform on \( T_1 \) a series of rotations that “undo” the rotations used to make \( T_2 \) right-leaning. Transforming \( T_2 \) into \( T_1 \) is symmetric.

**Remark:** It is unknown whether there exists a polynomial-time algorithm for determining the minimum number of rotations needed to transform one BST into the other (even though the rotation distance is at most \( 2n - 6 \) for BSTs with at least 11 nodes).