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 worst 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 $\lg n$, we will ensure that all of the operations require only $O(\lg n)$ comparisons.

Tree Rotations

Tree rotations are at the core of balanced BST implementations. A tree rotation is a constant time operation that changes the shape of a local area of a BST. There are two types of tree rotations, left rotations, and right rotations. Both involve making one child the new root, the root one of the children, and then swapping the “inner” subtrees of the two nodes that changed.

By using tree rotations to preserve additional invariants for a BST, you can ensure it is balanced (or roughly balanced), and keep the favorable asymptotic running time of a BST in even the worst cases. The key difference between different balanced BST implementations is that they keep track of different invariants and values in order to determine when tree rotations are needed to restructure the tree.

Red-Black BSTs

Properties of Red-Black Trees

A red-black tree is a BST with one extra bit of storage per node: its color, which can be either RED or BLACK. Each node of the tree contains the attributes color and key, and maintains pointers to left, right, and parent. If a child of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We call these NILs leaves. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from it to descendant leaves contain the same number of black nodes.
We call the number of black nodes on any simple path from, but not including, a node \( x \) down to a leaf the **black-height** of the node, denoted \( bh(x) \). By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root. Given these properties of red-black trees, it can be shown (see practice problems below) that a red-black tree with \( n \) internal nodes has height at most \( 2 \lg(n + 1) \).

Thus, we can implement the operations **find**, **minimum**, **maximum**, **successor**, and **predecessor** in \( O(\lg n) \) time on red-black trees, since each operation can run in \( O(h) \) time on a binary search tree of height \( h \).

**Insert in a Red-Black Tree**

When we insert into a red-black tree, we insert as we would in a normal BST and then color the inserted node red. This can violate property 4 of our tree. To resolve this, there are 3 cases we must examine. We describe the cases below using the following notation: \( z \) is the current node we are iterating on. At the point of insertion \( z \) is set to the inserted node. \( z.p \) refers to \( z \)'s parent, and \( z.p.p \) refers to \( z \)'s grandparent. \( y \) refers to the uncle of \( z \), that is \( y \) is the child of \( z.p.p \) that is not \( z.p \). In all cases described below the violation of property 4 is that both \( z \) and \( z.p \) are red. Note that these cases are not mutually exclusive, so we iterate through them in order when restructuring our tree.

**NOTE:** The cases laid out here are not identical to the cases laid out in the slides. In the lecture slides case 2 is different than the case 2 below. The slide version of case 2 is equivalent to combining one call to the below case 2 and then one call to case 3. They are equivalent operations at the end, they are just described differently.

**Case 1:** \( z \)'s uncle is red.

Solution: Change \( z.p \) and \( y \) to be black, and change \( z.p.p \) to be red. Change the pointer of \( z \) to \( z.p.p \).

**Case 2 (zig-zag):** \( z \)'s uncle is black, and \( z \)'s is not the same type of child to \( z.p \) as \( z.p \)'s is to \( z.p.p \).

Solution: We perform a left or right rotation on \( z \) and \( z.p \), then set \( z \) to be either \( z.left \) or \( z.right \) respectively. This keeps \( y \) constant.

**Case 3 (zig-zig):** \( z \)'s uncle is black, and \( z \)'s is the same type of child to \( z.p \) as \( z.p \)'s is to \( z.p.p \).

Solution: We perform a left or right rotation on \( z.p \) and \( z.p.p \), and then recolor \( z.p \) to be black.

Below is given a diagram going through each of these cases and the corresponding solution.
Delete in a Red-Black Tree

Deletion starts similar to that in normal BSTs. We will examine the procedure of delete(x). You first search for x and check the number of children it has in the tree. If x has two children, replace x with successor(x) and call delete(successor(x)). Recursively run this until you reach a case when the node we are deleting has 0 or 1 children. We delete normally as in a BST. That is, replace x with either a null in the 0 children case or its singular child in the 1 children case. Call this replacement v. If the x was red, then we are done. If x was black, then deleting it causes problems. This is because we have deleted a black node, so now the path to all nodes who were children of x contains one less black node than before. To correct this, we add one extra "unit" of blackness to v. This causes all of our black paths to again be the same length.

Here we run into another issue: we are only permitted red/black nodes in our tree, so we have no representation in our data structure of this extra blackness unit. We deal with two cases. If v were red, we can simply change it to black. This is because it had 0 units of blackness as a red node, so now it simply has one total blackness unit which just makes it a black node. Adding a black node in this manner violates no properties. The issue is when v is black. This node has 2 total units of blackness, which has no interpretation.
in our data structure. We will call this a "doubly black" node. The four cases below outline how to resolve this issue. Note that as in insert, this cases are not mutually exclusive, so iterate in the order they are listed.

**NOTE:** The cases here switch cases 3 and 4 from the lecture slides:

**Case 1: x’s sibling w is red**
Case 1 occurs when node w, the sibling of node x, is red. Since w must have black children, we can switch the colors of w and x.p and then perform a left-rotation on x.p without violating any of the red-black properties. The new sibling of x, which is one of w’s children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the colors of w’s children

**Case 2: x’s sibling w is black, and both of w’s children are black**
In case 2, both of w’s children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to x.p, which was originally either red or black. We do so by repeating our cases with x.p as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original x.p was red. Hence, the value c of the color attribute of the new node x is RED, and our cases terminate as we can just change the node to black.

**Case 3: x’s sibling w is black, w’s left child is red, and w’s right child is black**
Case 3 occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child w.left and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

**Case 4: x’s sibling w is black, and w’s right child is red**
Case 4 occurs when node x’s sibling w is black and w’s right child is red. By making some color changes and performing a left rotation on x.p, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root marks the end of all possible cases.
Splay Trees

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. All normal operations on a binary search tree are combined with one basic operation, called splaying. That is, every time a node is accessed, it is pushed to the root by a series of tree rotations known as splaying.

A sequence of $m$ insert, find, and delete operations on a splay tree takes at most $O(mlgn)$ time. Thus, we say that a single operation takes $\frac{O(mlgn)}{m} = O(lgn)$ amortized time. Note, however, that the worst case for any particular BST operation may still be $O(n)$. Therefore, amortized $O(lgn)$ is not as good as worst-case $O(lgn)$ time.

The operations insert, find, and delete are implemented as follows:

- **insert**: First perform a standard BST insertion. Then, splay inserted node
- **find**: Search for the node. If found, splay it. Otherwise, splay the last node accessed on the search path
- **delete**: Splay the element to be removed. If the element to be deleted is not found, splay the last node on the search path. Otherwise, disconnect the right and left subtrees of the root. Splay the maximum element of the left subtree, then connect the right subtree to the root of the left subtree
Splaying

When a node $x$ is accessed, a splay operation is performed on $x$ to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves $x$ closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

During each splay step we must consider the following factors:
- Whether $x$ is the left or right child of its parent node, $p$
- Whether $p$ is the root or not, and if not
- Whether $p$ is the left or right child of its parent, $g$ (the grandparent of $x$)

There are three types of splay steps, each of which has a left- and right-handed case. For the sake of brevity, only one of these two is shown for each type. These three types are:

**Zig Step:** This step is done when $p$ is the root. The tree is rotated on the edge between $x$ and $p$. Zig steps will be done only as the last step in a splay operation and only when $x$ has odd depth at the beginning of the operation.

![Zig Step Diagram](image)

**Zig-Zig Step:** This step is done when $p$ isn’t the root and $x$ and $p$ are either both right children or both left children (hence the name "zig-zig"). The picture below shows the case where $x$ and $p$ are both left children. The tree is rotated on the edge joining $p$ with its parent $g$, then rotated on the edge joining $x$ with $p$.

![Zig-Zig Step Diagram](image)

**Zig-Zag Step:** This step is done when $p$ is not the root and $x$ is a right child and $p$ is a left child or vice versa (hence the name "zig-zag"). The tree is rotated on the edge between $p$ and $x$, and then rotated on the resulting edge between $x$ and $g$.

![Zig-Zag Step Diagram](image)
Problems

Inserting into a Splay Tree
Given the following splay tree, give the final state after 5 is inserted.

Inserting into a Red-Black Tree
Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.
Deleting from a Red-Black Tree

Using the red-black tree constructed in the previous problem, show the red-black trees that result from the successive deletion of the keys in the order: 8, 12, 19, 31, 38, 41.

Prove Height of Red-Black Tree is Logarithmic

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