CMSC 420: Splay Trees

Justin Wyss-Gallifent

April 12, 2023

1 Introduction

Wouldn't it be nice if we could create a binary search tree which somehow tended stay balanced and did so without a lot of maintenance but didn't require storing any balance information to do so? This may seem impossible, but it's not!

2 Applications

Splay trees are used:

- in Windows NT to manage files; In a filesystem the keys are the filenames and the values are the locations (on disk, in memory, etc.) As files are created and deleted we want to be able to insert and delete them into our tree quickly. Moreover it's reasonable to ask that the files we access most often ought to be close to the top of the tree.
- In the `sed` string editor. This is a Unix utility which (broadly speaking) processes files line-by-line and applies rules to strings, such as "replace x by y". Because it might handle very large files the editor builds a tree in which the keys are the strings it needs to work on and the values are the rules it needs to apply. In that sense we want to be able to access the rules as quickly as possible and the rules we use most often ought to be close to the top of the tree.

3 Overview

Splay trees (1985) are binary search trees which are fascinating because:

- They tend to balance themselves. Just to be clear - they are not balanced in the sense that AVL trees are balanced, but rather that they restructure themselves they are more likely to be balanced than just using a BST.
- The contain no balance information.
- The time complexity of search, insert, and delete is average case $\mathcal{O}(\lg n)$ amortized time. Recall that this means that while we may have expensive operations we will have enough cheap operations to "save up" for the expensive ones.

You might wonder how this happens - if no balance information is saved, how does the tree tend to balance itself?

The trick is this: When we search, insert, and delete, we stir up the tree in a manner which tends to balance it out, meaning on average the height is $\Theta(\lg n)$. In addition this stirring is done in a manner which tends to bring frequently accessed nodes close to the root, meaning that frequently accessed nodes tend to be more quickly accessed.

Not only is this interesting, it's practical, because it makes sense that frequently-accessed elements should be close to the root and hence quick (cheap!) to access.

4 Splay

4.1 Introduction

Given a target key (which may or may not exist in the tree), this is what *splay* does: We first search for the node containing the target key just as with any binary search tree. If we find it, great, otherwise we find the last node before we fall out of the tree. We then embark upon a series of operations which brings the found node to the root.

Appreciate that if the target key is not in the tree the splay operation still does something! In such a case as a result of how binary search trees works it brings either the inorder predecessor or successor to the root.

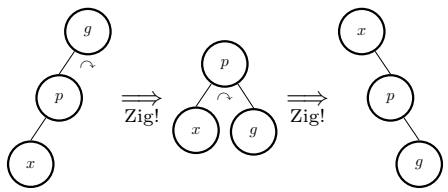
These splay operations are as follows. They are all familiar to you, they just get new names here:

4.2 Zig

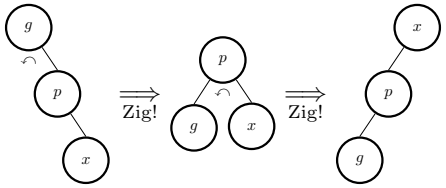
Zig is used when the target node x is the left or right child of the root p. Zig is effectively just a left or right rotation and will bring x to the root. If x is the left child of p, rotate right at p. If x is the right child of p, rotate left at p. If Zig is ever done, it will be the last operation.

4.3 Zig-Zig

Zig-Zig is used when the target node x has parent p and grandparent g and x is the left-left or right-right grandchild. Zig-Zig is effectively just two rotations and will bring x to g's position. If x is the left child of p which is the left child of g, rotate right at g and then right at p. If x is the right child of p which is the right child of g, rotate left at g and then left at p.

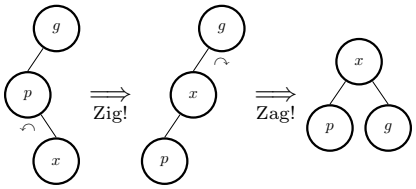


and

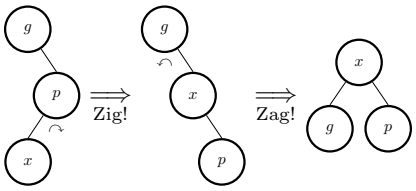


4.4 Zig-Zag

Zig-Zag is used when the target node x has parent p and grandparent g and x is the left-right or right-left grandchild. Zig-Zag is effectively just two rotations and will bring x to g's position. If x is the left child of p which is the right child of g, rotate right at p and then left at g. If x is the right child of p which is the left child of g, rotate left at p and then right at g.



and



4.5 Implementing Splay

Definition 4.5.1. We say that we *splay* a node when that node is not the root and we wish to transport it to the root. We basically ask:

(a) Is x the root's child? If so, Zig.

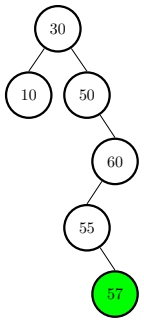
(b) Is x a left-left or right-right child? If so, Zig-Zig.

(c) Is x a left-right or right-left child? If so, Zig-Zag.

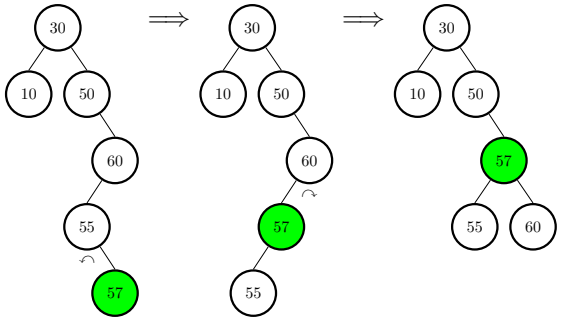
(d) Is x at the root? If not, go back to (a).

Note 4.5.1. These operations are mutually exclusive in the sense that only one will apply at any time.

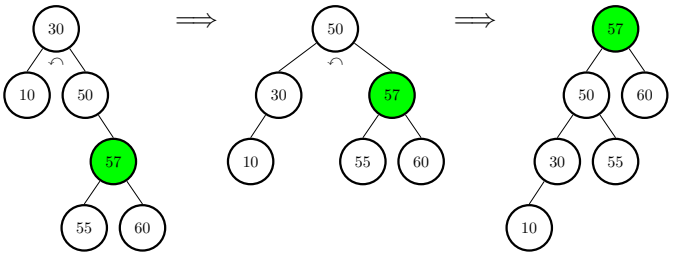
Example 4.1. Let's apply splay to the node containing 57. Notice how unbalanced this tree is at the start and how balanced it is at the end!



Since 57 is the right child of 55 which is the left child of 60 we Zig-Zag. Specifically we rotate left at 55 and then rotate right at 60:



Since 57 is the right child of 50 which is the right child of 30 we Zig-Zig. Specifically we rotate left at 30 and then we rotate left at 50:



Notice that at the start the distances of the keys to the root were: 30 (distance 0), $10, 50$ (distance 1), 60 (distance 2), 55 (distance 3), and 57 (distance 4).

At the end the distances of the keys to the root were: 57 (distance 0), $50, 60$ (distance 1), $30, 55$ (distance 2), and 10 (distance 3).

Most of the close nodes remain close, the one exception being 10.

5 Search

5.1 Algorithm

For search we first call splay on the target key. The result is either that the target key or its inorder predecessor or successor is at the root. We then respond appropriately, for example if the target key is not the root then we might error.

There are two things of note here. First, assuming that the target is in the tree this will bring it to the root. This is a feature (not a bug!) in the sense that it suggests that a key we're interested in ought to be close to the root and quick to access. Note that elements previously close to the root will tend to stay close to the root.

Second, if the target is not in the tree you may notice that we have now reorganized the tree anyway. There are two positive spins we can put on this, primarily that it will reorganize the tree a bit, which is possibly good from a balance perspective, but also that at least we have brought something possibly meaningful to the root which may be useful later.

5.2 Time Complexity

See in the later section.

6 Insert

6.1 Insertion Note

In theory we could insert as with a standard BST and be done but the desire that splay trees tend to stay balanced insists that we shuffle the tree around a bit as much as possible and so this will include during insertion.

6.2 Algorithm 1

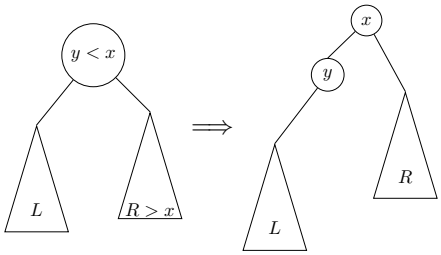
Suppose we wish to insert the key x. We proceed as follows: First we call splay on x. If x exists it will now be at the root and we can respond accordingly, probably by throwing an error.

Otherwise the root y is either the inorder predecessor or inorder successor of x.

Suppose y is the inorder predecessor of x. This means that y's left subtree L has $L < y$ and that y's right subtree R has $R > x > y$.

We then create a new tree using x as the root, y and its left subtree L as x's left subtree and R as x's right subtree.

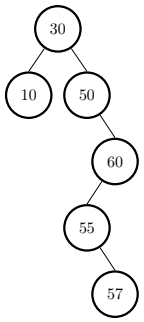
Here is an illustration:



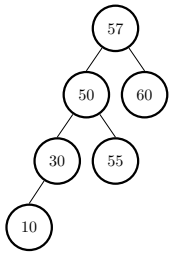
The case where y is the inorder successor is symmetric.

Note that the newly inserted element (which is possibly important) is now at the root of the tree and quick to access.

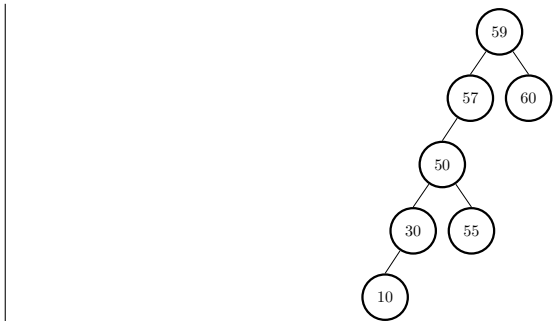
Example 6.1. Suppose we wish to insert 59 into the following splay tree:



You may recognize this as the tree from earlier. We call splay on 59. This hunts for 59 in the tree but gets to 57 and would fall out. Thus it brings 57 to the root. This is the same process as above, so we just show the result:



Now we observe that 57 is the inorder predecessor of 59 so we put 59 as the new root and hang 57's tree off it as the left child:



6.3 Algorithm 2

An alternate approach to inserting a new key is to insert it as with a standard binary search tree and then splay it to the root.

6.4 Time Complexity

See in the later section.

7 Delete

7.1 Deletion Note

In theory we could delete as with a standard BST and be done but the desire that splay trees tend to stay balanced insists that we shuffle the tree around a bit as much as possible and so this will include during deletion.

7.2 Algorithm 1

Suppose we wish to delete the key x. We proceed as follows: First we call splay on x. If x does not exist then something else will be at the root and we can respond accordingly.

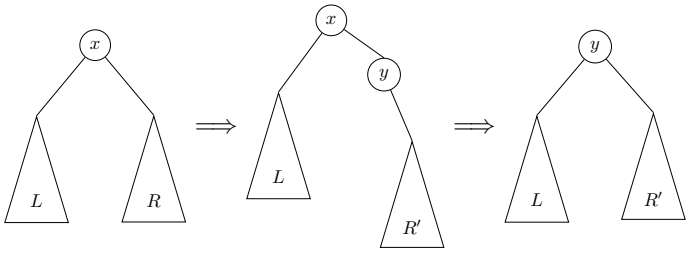
Otherwise x is at the root. Let L and R be its right and left subtrees.

If L is empty then just remove x and R is the new tree, whereas if R is empty then remove x and L is the new tree.

If neither L nor R is empty then we call splay on x but only in the subtree R. Since x is not there (because it's the parent of R) and because $R > x$ the result will be that the new root of R, call it y, will be the inorder successor of x. Consequently y will have no left subtree itself (because there is nothing greater than x and smaller than y) but it will have a (possibly empty) right subtree. We simply delete x and shift y up to its place.

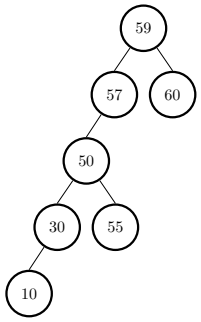
We could also have called splay on x in the subtree L with a symmetric argument.

Here is an illustration:

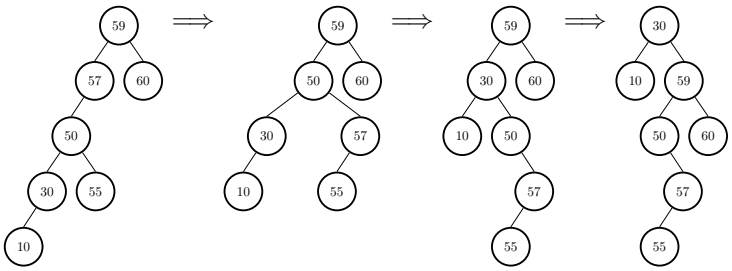


Note that we could have done a symmetric argument, you should consider what this would be.

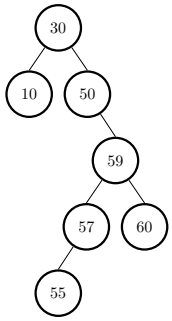
Example 7.1. Let's delete 30 from this tree:



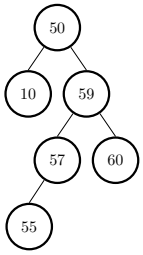
First we call splay on 30 which brings it to the root, in this case via a zig-zig and a zig:



Now we could either call splay on 30 in the left or right subtree. For a more interesting result let's call it on the right subtree. Since 30 is obviously not in the right subtree the splay operation finds 50 before it would drop out of the tree and it brings 50 to the root of the subtree via a zig:



Now we simply throw out 30 and use 50 as the new root:



7.3 Algorithm 2

An alternate approach to deleting a key is to delete it as with a standard binary search tree and then splay the parent of that key (if there is one) to the root.

7.4 Time Complexity

See in the later section.

8 Time Complexity

8.1 Introduction

We have repeatedly implied that splay trees "tend to be balanced" and so intuitively this suggests that search, insert, and delete should be average case $\mathcal{O}(\lg n)$, where n is the number of nodes, but as of now we have no formal statement or proof of this.

8.2 Average Case Amortized Analysis

The analysis of splay tree operations in the average case sense is done via amortized analysis.

Theorem 8.2.1. Essentially the idea is that if we perform M operations that

this will take $\mathcal{O}(M \lg n)$ time, and therefore the amortized cost of a single operation is $\mathcal{O}(\lg n)$.

Proof. This proof is not trivial. It is done via the potential method of amortized analysis. Versions of this exist here and there on the web, a partial proof is in Dave Mount's notes. We will not be covering this for now so I won't insert the proof here. $\mathcal{QED}$

8.3 Worst Case

Although in practice it is incredibly unlikely, there is nothing stopping a splay tree from turning into a linked list. This means that worst-case for search, insert, and delete is $\mathcal{O}(n)$.

Example 8.1. If we insert n, $n-1$, ... , 2, 1 the result will be a linked list. Try it for a small n.