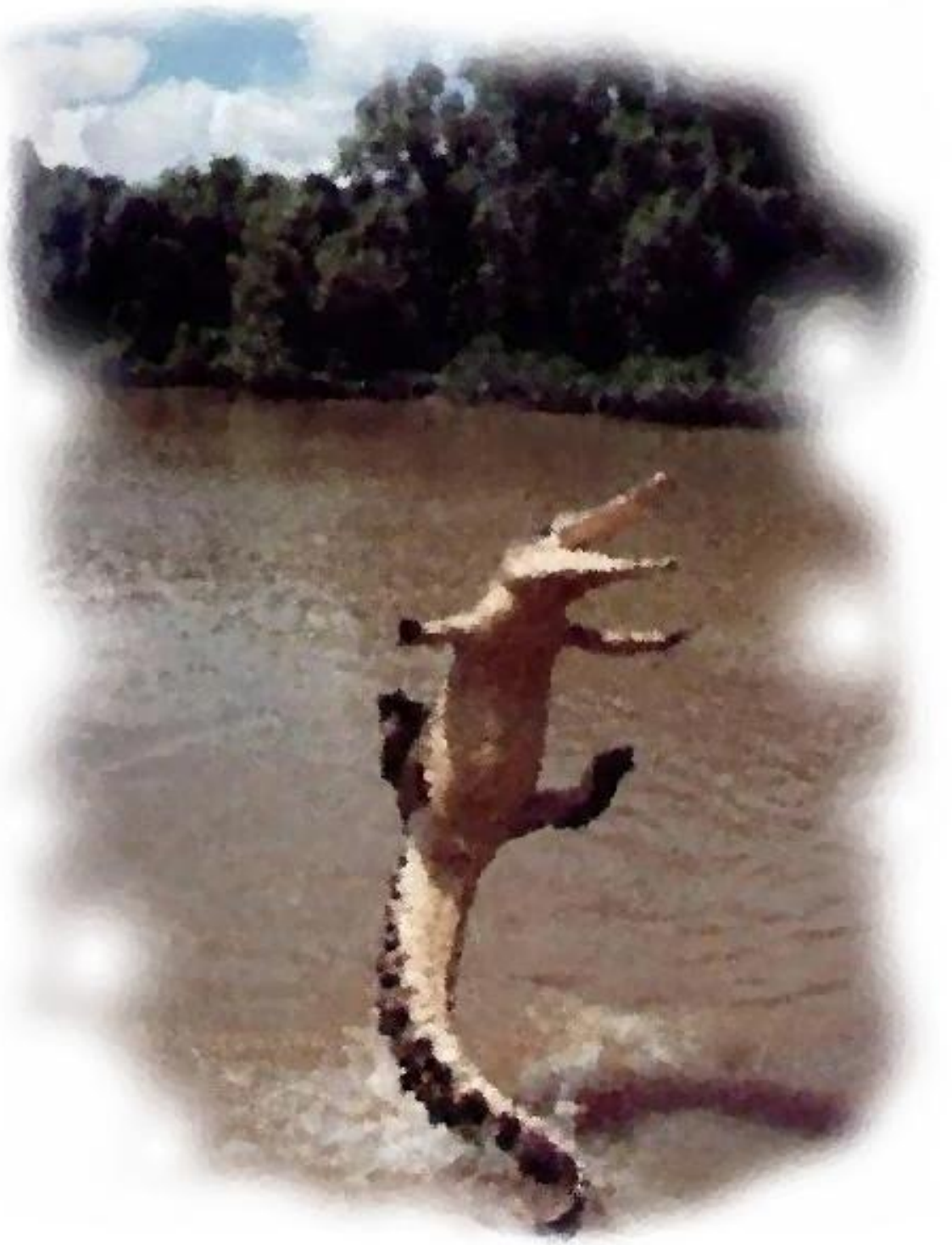
Mo' de Le Na Crocodile
As gators go, he quite docile



Mo' de Le na also first. Dis gator aces every quest

# Playing with Splaying


If you're here, it means you've conquered the BST.

All well and good with the BST, but what if you keep getting stuck with an unlucky sequence of inserts? (What might these be?)

Although a well-structured BST gives you worst-case O(log N) search, you're not guaranteed a well-structured BST if your numbers were inserted in an unfavorable sequence. You remain vulnerable to the O(N) beast, even if only slightly.

How can you insulate yourself against this worst-case possibility? By *insulating* I don't mean that you reduce the likelihood of the worst case from happening. I mean that you *eliminate* it. How can you *guarantee* that the worst case is never worse than logarithmic in the size of the tree?

One way is to keep your tree always *balanced*. You are by now, probably, familiar with one kind of a balancing act - AVL balancing. If not, this would be a good time to hit the reference material, experiment a bit and come back.

## Overview

Welcome back.

If you've tried to do AVL balancing, you would have found that you might have performed extra book-keeping in your tree manipulation algorithms. Even worse - you might have had to store extra information in each node (what is this extra info?).

What do you get for all that extra work?

Guaranteed log N search times.

Hrrrmmph! I can hear some people grumbling.

Good! As you may have discovered by now, the cool thing about many of these data structures is that we know many ways in which to deal with them depending on what makes sense to trade-off for what in a particular application.

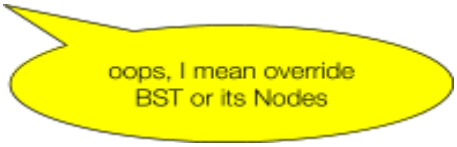
Suppose we use BSTs in an application where space is at a premium and you need your Nodes to be as thin as possible. Then it makes sense to ask *"I wonder if this particular application needs this function's runtime to be absolutely log N bounded or whether amortized log N times[1] will do"*

As it happens, there are many applications in which we can live with amortized log N access times if it means you get a simpler data structure requiring less book-keeping than always balanced trees (like AVL).

Like in real life, we get to say "*Hey! Just 'cuz I know how to balance don't mean I gotta keep balancing ALL of the time.*"

This lazy strategy of not trying to futz with any more of the BST than you absolutely have to gives us a simple and useful variety of BSTs - Splay trees. Like its balanced tree cousins, Splay trees will use left and right rotations (among other operations) to transform their internal tree representations at appropriate times.

But Splay trees don't have the headache of having to maintain the height of each node. In fact, you don't even have to make any structural changes to your tree. You don't have to ride the beast or its nose.




You can simply write a bunch of processing routines that operate on an existing BST. Much as you did with the Cormorant and matrices.

In this quest, you must implement a set of operations on a BST that, if used exclusively to operate on the tree, guarantee O(log N) amortized access time. You will do it by splaying your BST behind the scenes:

1. Create a class called `Tx` in a file called `Tree_Algorithms.h`
2. Make `Tx` a friend of your BST class
3. Implement the algorithms in this quest as static methods of `Tx`, operating on BSTs passed in by reference as their first parameters.

[1] Amortized means that you only consider the average time of M random calls to an operation in the limit. Although an individual call may end up taking O(N) time, the algorithm should guarantee that the sum of times for M calls is bounded by M log N. Note that the value of M matters. If M needs to be 1, then you have to suffer the overhead of continuous rebalancing (e.g. AVL). OTOH, techniques like splaying give you guaranteed average time behavior over a minimum of M calls. The smaller the value of M before the average is closer than some threshold to M log N, the better the algorithm (runtime-wise)

The algorithm for each miniquest is discussed at length in Loceff's modules, and also in the recommended text (Ch.12, *Advanced Data Structures and Implementations*, in my copy of Weiss). I'm sure you'll also find it in a zillion other places on the net if you look.

The basic idea in all of them is simple. I'll try and give you a slightly different perspective from the others. If you understand one, you'll understand them all.

## Overview of Splaying

Splaying is an exercise in spin-offs and mergers. When you splay a BST for a target X, you will essentially:

1. Dismantle the input tree and break it into three separate trees.
    a. One part is known to be less than X (Loceff and the book call this the *left* tree)
    b. One part is known to be greater than X. (the *right* tree)
    c. One part is known to either be rooted at X or not contain X (the middle tree)
2. You reassemble a new tree by using these three parts.

How do you dismantle the input tree?

As you barrel towards X (or where you think it might be) in your middle tree starting at the root, you make decisions about entire sub-trees which you can know for sure to not contain X. You also know if the elements in these subtrees are greater than X or less than it. You can leverage this knowledge to come up with the following strategy.

It uses a 2-step lookahead. By this, I mean that specific actions you choose will be determined by the next two nodes you may visit in your search for X in the middle tree.[2]

1. Start with 3 trees:
    a. A tree with all items < X (*left - empty at first)*
    b. A tree with all items > X (*right - empty at first*)
    c. A tree with items you don't yet know about (*middle - the entire input tree at first*)
2. Standing at the root of the middle tree and looking towards where X might be,
    a. if you find you have to travel in the same direction (left-left or right-right), select and perform the *zig-zig* dance (appropriately adjusted for direction). See Fig 1.
    b. If you find you have to travel first in one direction and then the other (left-right or right-left), select and perform the *zig-zag* dance (appropriately adjusted for direction). See Fig 2.

[2] Do you think it makes sense to look ahead more than 2 steps? Do you think we might find one of 8 as-good-or-better reconfigurations if we looked 3 steps ahead? What if the answer to the previous question was "yes"? (Actually the answer has got to be yes. why?)

These dances will remove entire chunks in the middle tree which you now know to be either < X or > X and attach them to the < X or > X trees as appropriate.[3]

3. Finally, use the three trees to reassemble a new BST. (BTW, how do you know for sure that this new BST is *improved* for future access? By *improved* I mean that it is restructured in a way that has resulted in reduced amortized access time. One way is to believe me because I say so. But I might be bluffing.)

## The Dances

Figure 1 shows the steps needed for the catchy little number called zig-zig. Make sure you do each step exactly once and in the right order. Cuz, otherwise, you'll trip over yourself.

Maybe you want to start with the easier zig-zag dance? You can find the steps in Figure 2.

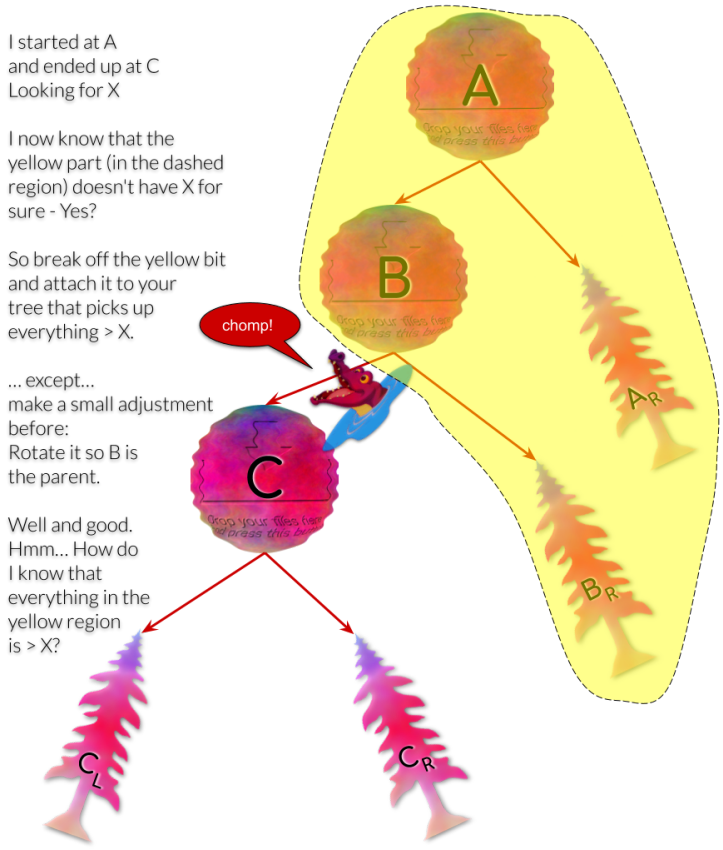



Figure 1. The zig-zig dance[4]

[3] Why these particular sequences of moves (involving rotations and all)? Can you choreograph any other move sequences that provide the same or better run-time guarantees than this does?
[4] If you did everything except rotation, what happens? Do you still end up with a tree rooted at X (or its neighbor)? What does rotation give you?

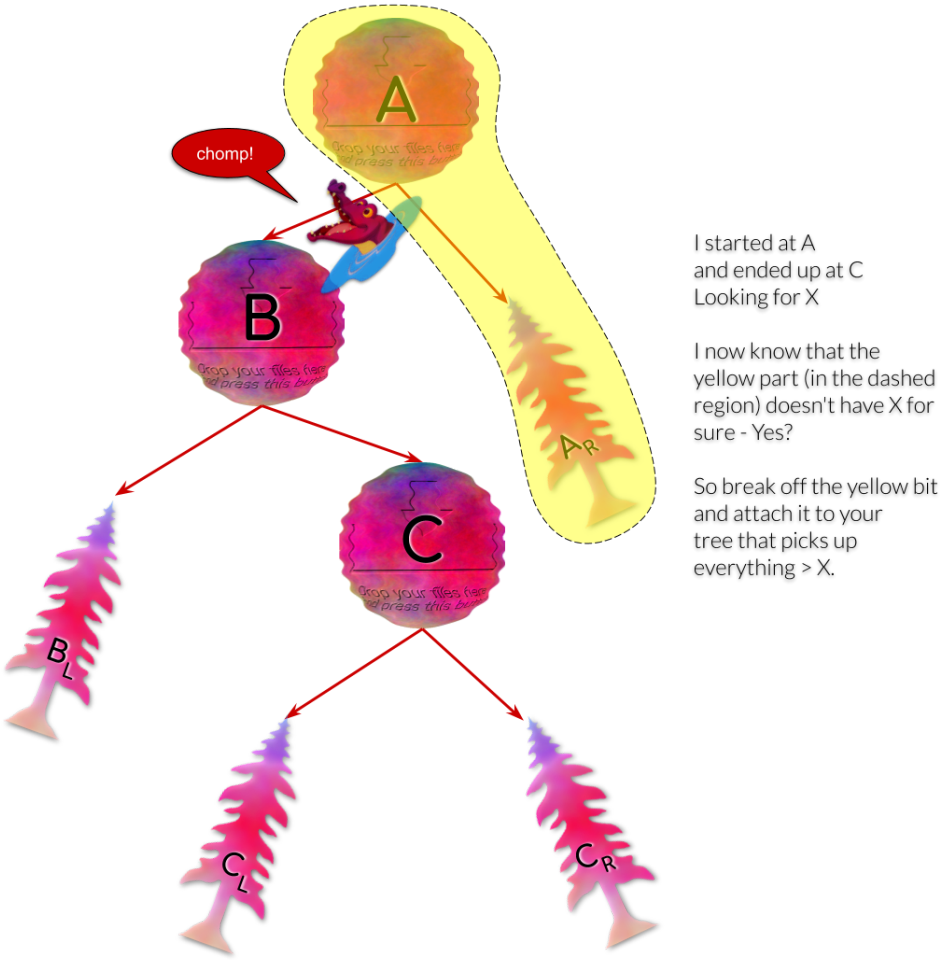



Figure 2. The zig zag dance

## Merger

After you zig and zag your way through the middle tree, throwing its branches left and right, you will eventually arrive at a node where you have either found your target, X, or you know for sure that you will not find it. Figure 3 shows how to handle this situation:

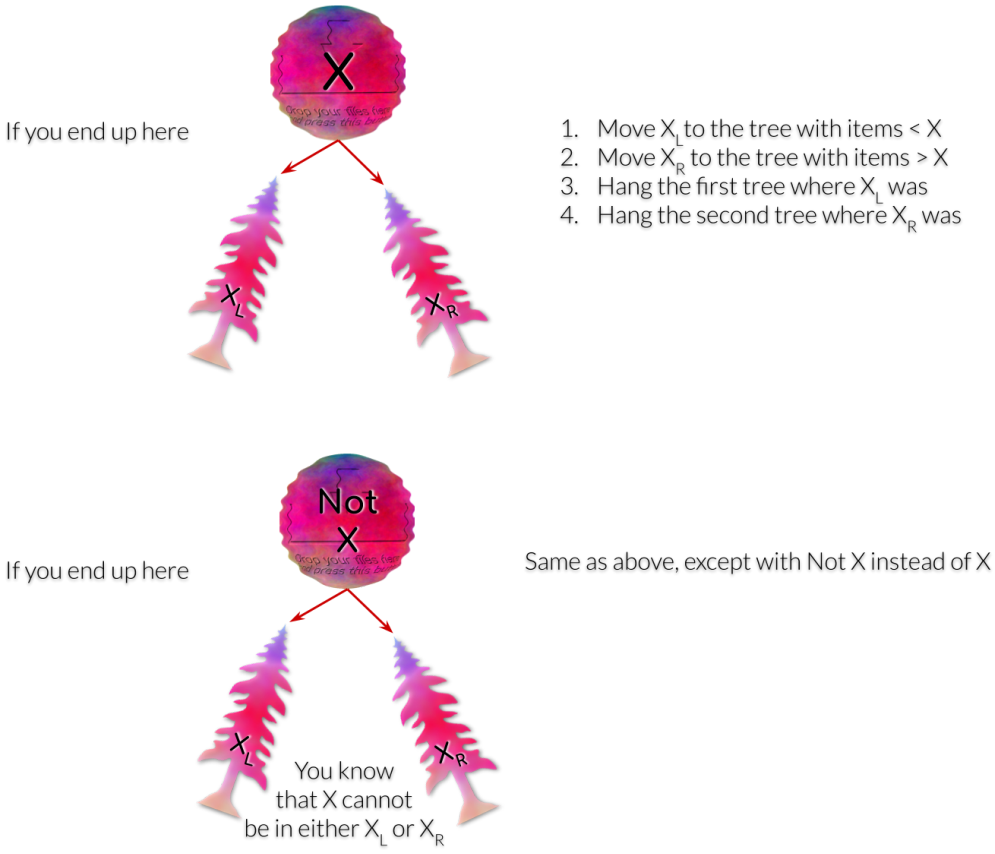



Figure 3. The final step. Merger.

The move sequences you need to execute as you search for your target can be found in multiple reference materials. Discuss the miniquest *moves* in our sub. Use pictures if you can. And also, ponder:

1. How can you be sure that your particular sequence of moves results in a tree whose worst case access time is < what than you started with?
2. What if it is not < but rather <= ? Is this a possibility? Or is that what you're trying to avoid?
3. What happens when you don't find the target in the tree?

Brainstorm your own moves, your own dance, and try to prove other desirable things that your choreography will yield (does not need to be restricted to running times).

# Starter code

I managed to get a fuzzy pic of the very top of the `Tree_Algorithms.h` file. And then the quest master said "*What? You already took almost 20 of my 160 lines!*"

So I couldn't risk going back for more. But I got an ok to put what I got up in Figure 4 here.

```
11
12 #include "BST.h"
13
14 class Tx {
15 private:
16     template <typename T> static void _splay(typename BST<T>::Node *&p, const T &x);
17     template <typename T> static void _rotate_with_left_child(typename BST<T>::Node *&p);
18     template <typename T> static void _rotate_with_right_child(typename BST<T>::Node *&p);
19
20 public:
21     template <typename T> static const T &splay_find(BST<T> &tree, const T &x);
22     template <typename T> static bool splay_contains (BST<T> &tree, const T &x);
23     template <typename T> static bool splay_insert(BST<T> &tree, const T &x);
24     template <typename T> static bool splay_remove(BST<T> &tree, const T &x);
25
26     friend class Tests; // Don't remove
27 };
28
```

Figure 4. Tx -. A regular class with static template methods (a handy packaging idea!)

# Moar Detail

Here are some possibly rewarding facts to know about the `Tx` class:

## Spin left and spin right

These take a node parameter by reference (this is important; why?) Upon return, the subtree rooted at the given node must be reconfigured as dictated by the moves of the corresponding rotation dance. If you want to review how to spin (which you presumably did a lot of in AVL trees), see Figure 5 for the choreography of the *Spin-with-your-left*. In the code, the corresponding method would be called `_rotate_with_left_child()`

little gangaram playing questball

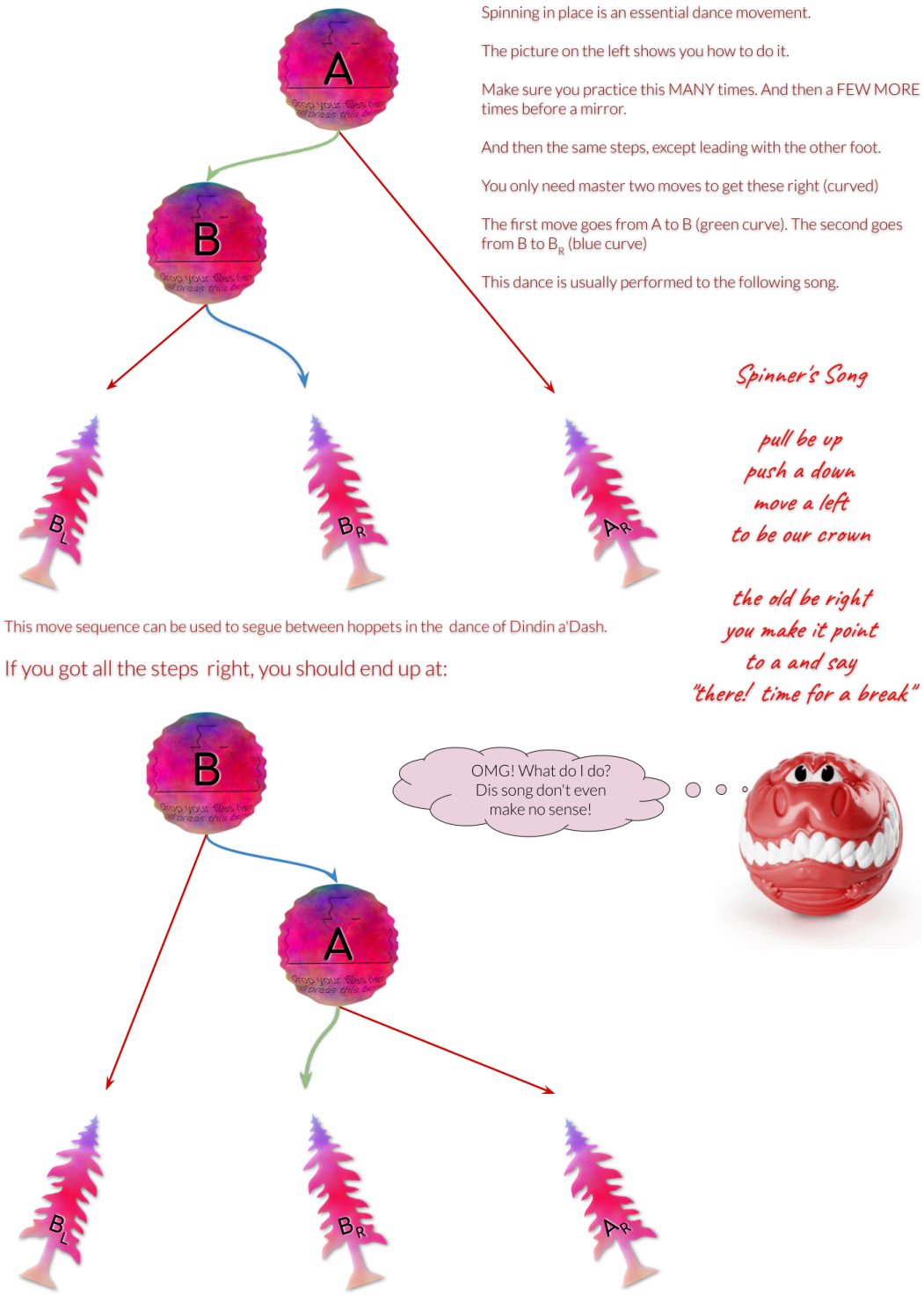



Figure 5. Moves for the *Spin-with-your-left* dance

## Splay

This is the biggie. When invoked it must splay the tree rooted at the given node for the given target value. Note that the node is passed in by reference as usual.

Your implementation must be the top-down approach described in this spec. Also, it must be iterative (not recursive).

## Find and Contain

Like before, `find()` should return a reference to the found item or throw an exception. But `contains()` should always quietly return with a bool. `contains()` should invoke `find()` and `find()` should make its decision by examining the root of the input tree after splaying it for the target.

## Insert

To insert a new value into the tree, first splay it on the given value. Now look at the root of the splayed tree.

If it is equal to the new value, it was already there in the tree. No need to do anything. Return false to say so.

If the root is not equal to the new value, then you can dismantle the splayed tree and stick its parts under a brand new node created to contain the given value. See Figure 6.

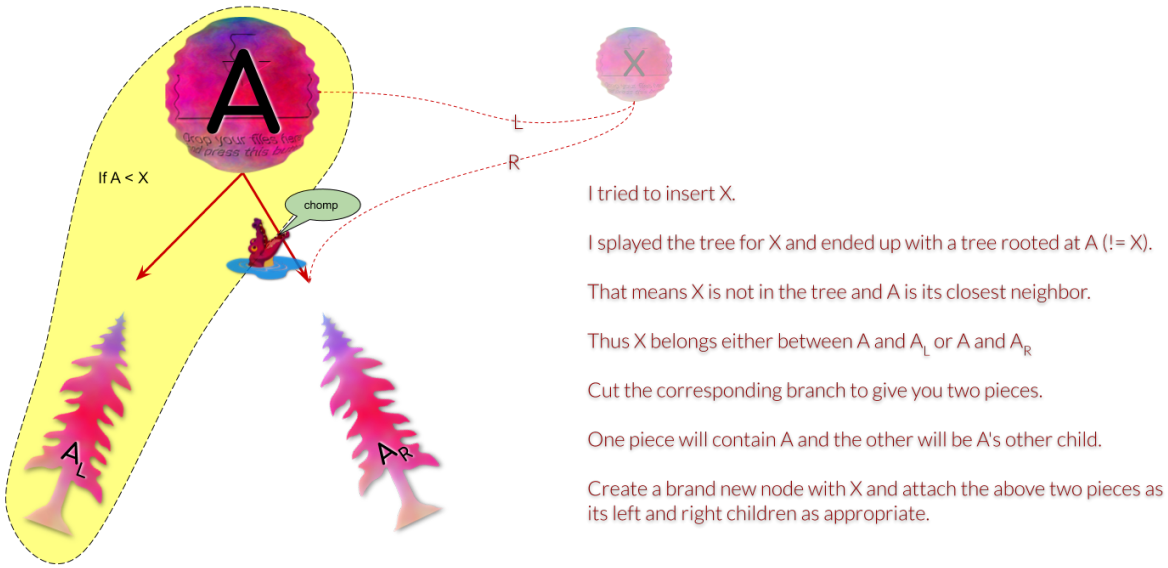



Figure 6. Inserting X whose closest neighbor in the tree is A

## Remove

First, splay the tree on the given value like you did for `insert()`.

If the splayed root is not equal to the given value, it was not in the tree to begin with. Return false to say so.

If the root is equal to the given value, it must be removed. How?

Simply splay one of its children (I'm afraid it has to be the left child to agree with my reference implementation) on the given value. Since this value is not present in the left sub-tree, it will bring its closest smaller neighbor up to the root *with an empty right child* (why smaller neighbor and why empty right?)

Now all you have to do is to break the right branch off the original and attach it to the left child's right. Y now becomes the new root. See Figure 7.

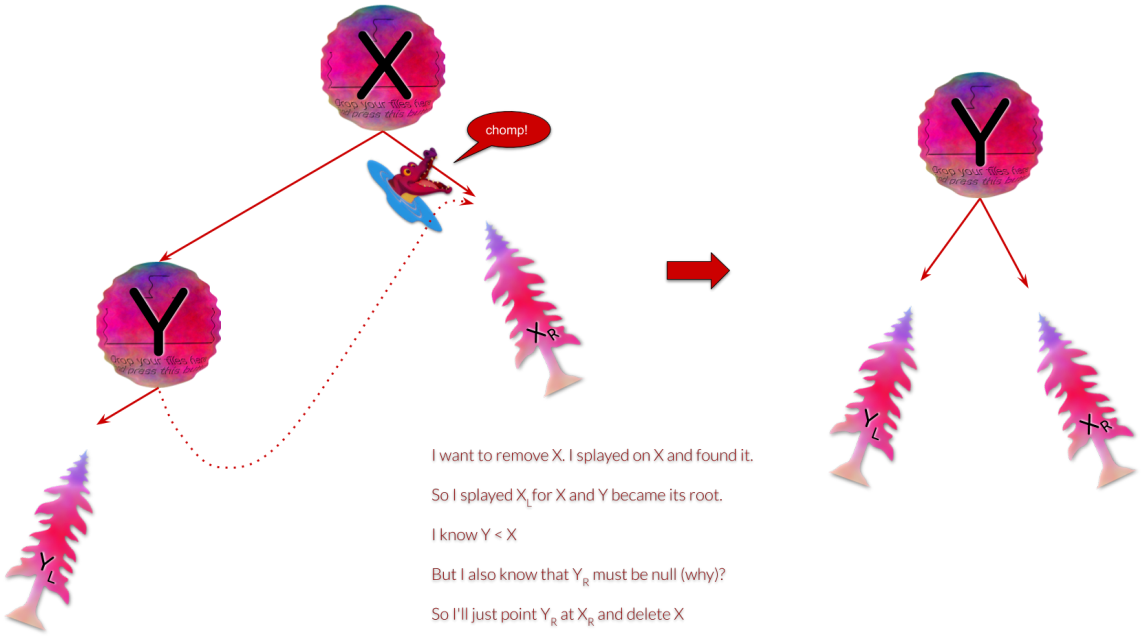
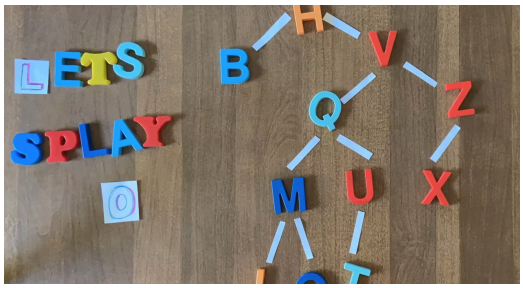



Figure 7. Remove.

## Icing (Edit Jun 1, 2021)

Thanks to https://reddit.com/u/lane_johnson, I'm able to share the following really cool video he put together. It's at the end of the spec for a reason. Understand the concept first, try and fail many times, and then watch this video for maximum enjoyment. You can meet Lane in the STEM center.




Lane's Splay Vid

# Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files[5] into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questing,

&

nonlinear gator

[5] `BST.h` and `Tree_Algorithms.h`