Leaf It Up To Binary Trees

Vaidehi Joshi · Follow
Published in basecs · 11 min read · Feb 20, 2017

Most things in software can be broken up into smaller parts. Large frameworks are really just small pieces of functionality that have been built up to create a heavyweight code infrastructure. Object-oriented programming is really just a bunch of classes that inherit from one another. And classical inheritance and the "class hierarchy" is really just a hierarchal tree structure. And trees are data structures that are really just a bunch of nodes and links that are connected to one another.

See? Everything is a bunch of little things put together. Where it gets *really* interesting, however, is when we start looking at how those little pieces function under the surface — that is to say, all of the different ways that they can be used to build larger abstractions.

Last week, we learned about tree data structures, which are usually abstracted away but are important pieces of how larger things — like the file system on our computers — actually work under the hood. There are a few different types of tree structures that are used in programming, but the most common (and, arguably, the most powerful) one is a **binary search tree**.

Binary search trees, sometimes abbreviated as BSTs, are tree data structures that adhere to a set of very specific rules. Just from their name, you *might* already be able to guess what those rules are. The reason these trees are so powerful and end up being so useful is *because of* these specific rules. But, before we gush over binary search trees any further, let's dig a little deeper into what they are, and what makes them different from any other tree out there!

Two trees within a tree

We're already familiar with the term **binary** since we've talked about it in the context of the binary (or base 2) number system. However, binary also has a simpler meaning: anything *relating to* or *composed of* **two things**. In the context of trees, this might seem a little odd at first. A tree can only have one single root — that's one of its defining characteristics! So how does one tree become two? How does a tree become…*binary*, exactly?

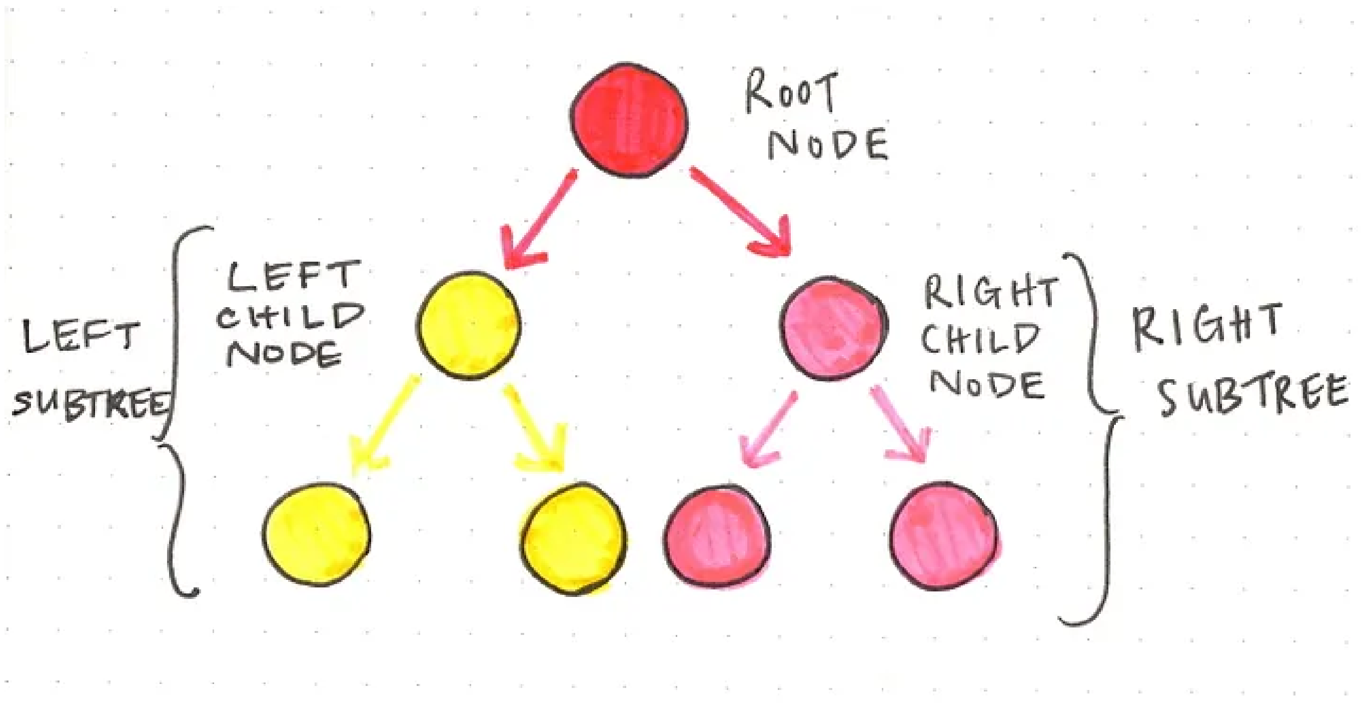
In order to understand what puts the "binary" in binary search tree, we have to think back to one other characteristic of the tree data structure: recursiveness. Trees are *recursive data structures*, which means that a single tree is made up of many others.

> The child node of a tree structure could also very well be the parent node to many other child nodes — which would effectively make it the root node of a mini subtree of the larger tree structure.

The recursive characteristic of a tree is crucial to how a binary search actually functions (not to mention, one of the reasons that it is so powerful).

Binary trees get their name from the way that they are structured; more specifically, the get their name from the way that their *subtrees* are structured. Every binary tree has a root node, just as we might expect. But after that, things get a bit more narrow and strict. A binary tree can only ever have two links, connecting to two nodes. This means that every parent node can only ever have two possible child nodes — and *never* any more than that.

But where does recursion come into the mix? Well, if every parent node, including the root node, can only ever have *two* child nodes, this means that the root node can only point to two subtrees. Thus, every binary tree will contain two subtrees within it: a **left subtree** and a **right subtree**. And this rule keeps applying as we go down the tree: both the left subtree and the right subtree are binary trees in and of themselves because they are recursively part of the larger tree. So, the left subtree's root will point to two more trees, which means that it contains *its very own* left subtree and right subtree!



Every binary tree will contain two subtrees within it: a left subtree and a right subtree.

The recursive aspect of a binary search tree is part of what makes it so powerful. The very fact that we know that one single BST can be split up and evenly divided into mini-trees within it will come in handy later when we start traversing down the tree!

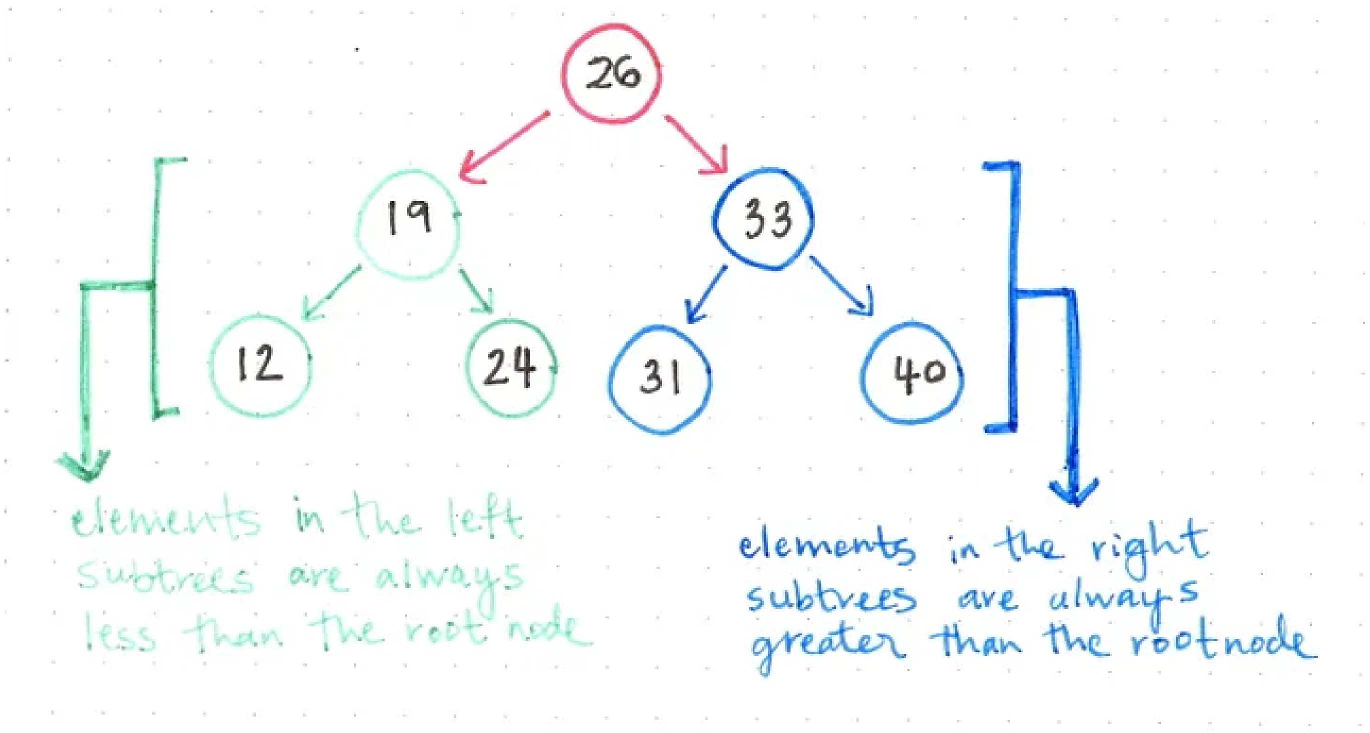
But first, let's take a look at one more important rule that makes binary trees so special.

To the left, to the left, all the small numbers in the tree to the left

If you haven't fully grasped the power of binary search trees yet, that's because I haven't shared their *most important* characteristic: a binary search tree's nodes *must* be sorted in a specific way. In fact, the way that a binary search tree is organized and sorted is what makes it "searchable".

In order for a BST to be searchable, all of the nodes to the left of the root node must be less than the value of the root node. You might be able to guess, then, that this must mean that all of the values to the *right* of the root node have to be greater than the root node.

If we look at this ordering not just in theory but actually in practice, a pattern starts to reveal itself. All of the subtrees to the left of a node will always be smaller in value than the subtrees to the right of a node — and of course, because of the recursive nature of trees, this applies not just to the main overarching tree structure, but to every single nested subtree as well.

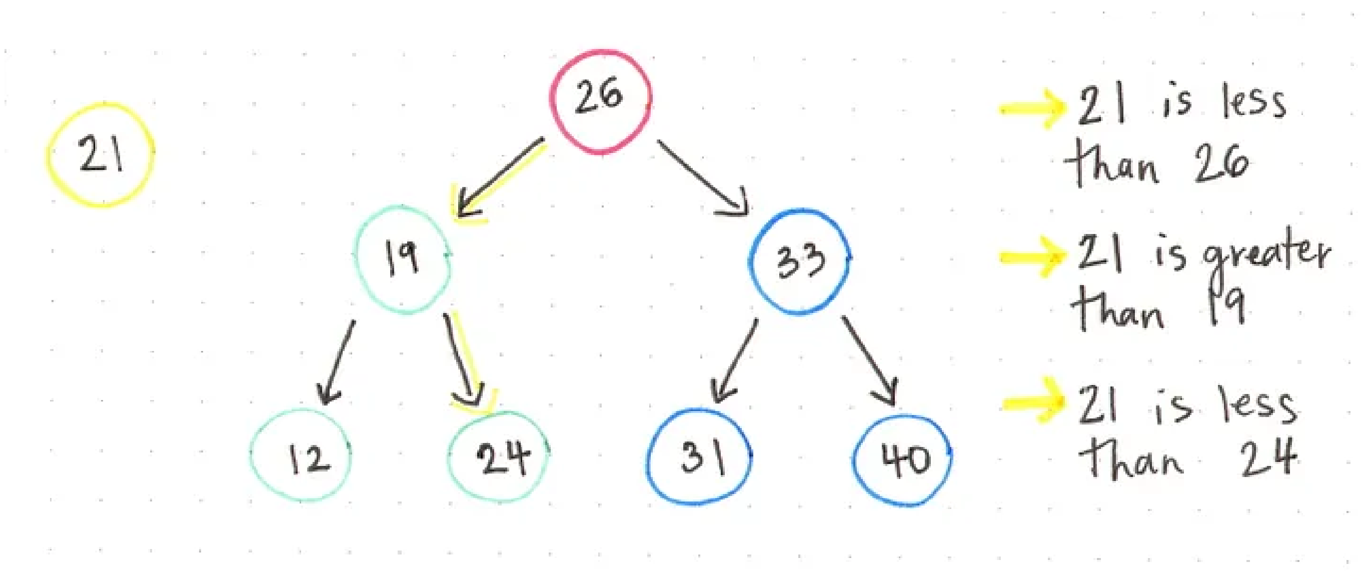

26
19
33
12
24
31
40
elements in the left subtrees are always less than the root node
elements in the right subtrees are always greater than the rootnode

Nodes in a binary search tree are organized and order by value

In the example tree above, all of the nodes to the left of the root node, which has a value of 26, are smaller than 26. But, if we look at the green subtree on the left, we'll notice that even though 24 is smaller than 26, it is still bigger than 19, and so it goes to the right of the node 19.

> Any tree can be a binary tree if each node has only two child nodes. It's the ordering of nodes that makes a binary tree searchable and, by extension, what makes it so powerful.

The explicit ordering of nodes is what makes a binary search tree so easy to, well, *search*. And this plays into all of the basic operations one might want to perform on a tree, too. For example, let's say we wanted to perform an **insert()** operation, which we would expect to be able to take a value and insert it into the correct position in the tree.



How would we insert 21 into this binary search tree?

We can pseudocode an **insert()** function pretty easily simply because of the rules of ordering of a BST:

1. We start at the root node, and compare that value of the root node (`26`) to the item we want.

2. Since `21` is *less than* `26`, we immediately determine that the item we want to insert is going to live somewhere within the left subtree of the larger binary search tree.
3. We look at the new “root” node: `19`. We know that the item we want to insert, `21`, is greater than `19`. So, we move to the right node of the node `19`, since we know the item we are inserting is larger and has to be on the right subtree.
4. Now we come to a leaf on the subtree: the node is `24`, which is bigger than `21`. We need to insert our item somewhere here, but we also need to make sure that the node with a value of `24` is in the correct place.
5. We set the node we’re inserting, `21`, to point it’s right pointer reference to the pre-existing node `24`, since `24` is greater than `21`. And our insert is done!

Half the elements and twice the fun

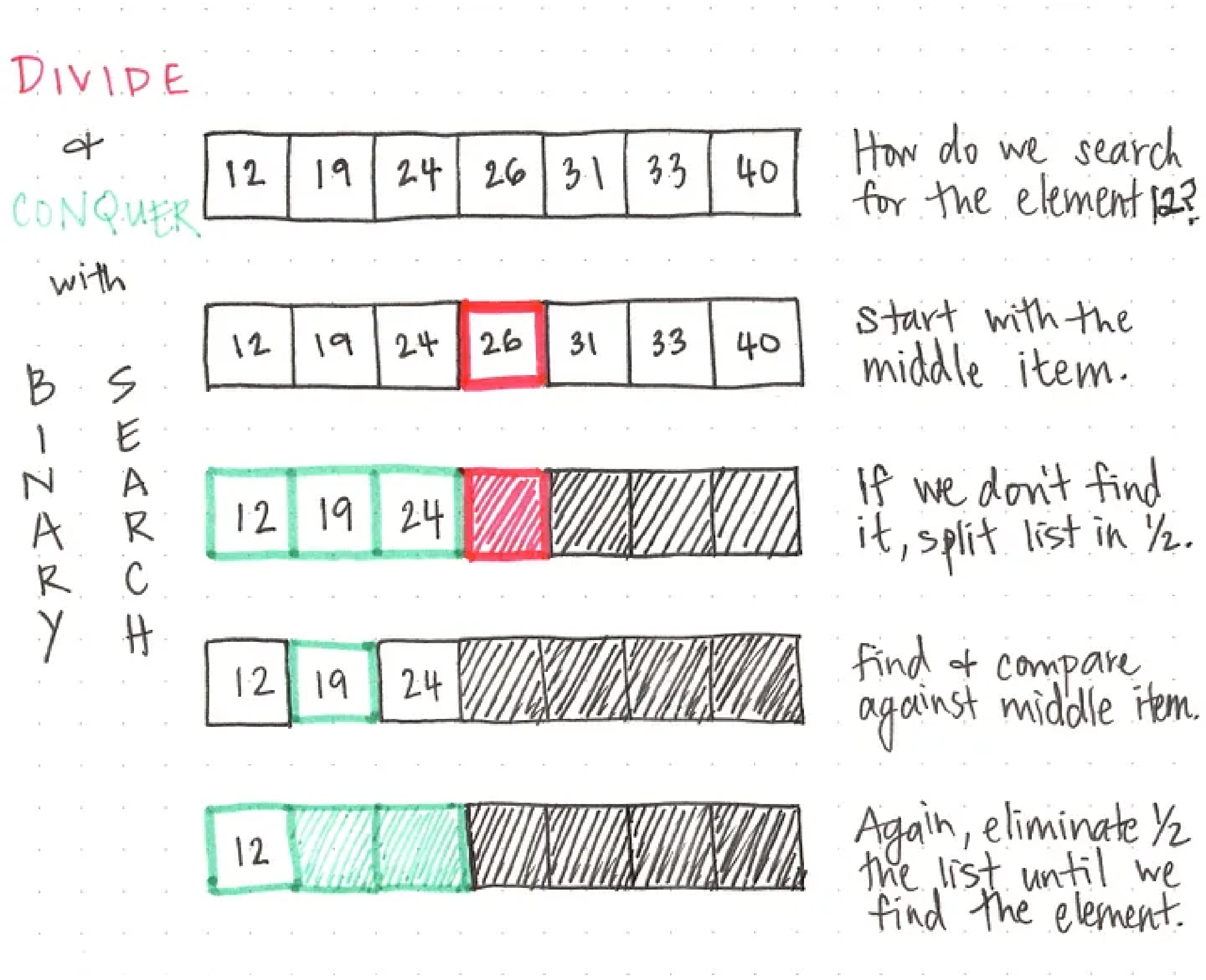
Binary search trees are really special in computer science. And the reason for this is simple: they allow you to leverage the power of the binary search algorithm. You might recall that not all binary trees are binary search trees — they have to be organized in a specific way in order for us to perform binary searches *on* them.

Hold up — what even *is* a binary search? Well, we already started dipping our toes into binary searches in the process of pseudo-coding that insert function earlier.

> A binary search is an algorithm that simplifies and speeds up searching through a sorted collection by

> dividing the search set into two groups and comparing an element to one that is larger or smaller than the one you're looking for.

That definition is a mouthful, but it's actually a lot simpler to understand and grasp by looking at it in action. So, let's make it easier on ourselves with a drawing! We'll work with the same binary search tree from earlier, which had a root node of `26`; instead of tree format though, let's reorganize it into a linear structure, like an array, to make it a little easier to look at.


DIVIDE
&
CONQUER
with
BINARY SEARCH
12 19 24 26 31 33 40
How do we search for the element 12?
12 19 24 26 31 33 40
Start with the middle item.
12 19 24
If we don't find it, split list in ½.
12 19 24
Find & compare against middle item.
12
Again, eliminate ½ the list until we find the element.

Divide and conquer with binary search

Here we have 7 elements (nodes), and we want to find just one: `12`. In most situations, we wouldn't be able to easily see the values of all of our node, but in this example, I wrote them all out so that it's easier to understand.

Okay — so how can we search for the node with a value of `12`?

Well, we already know that our elements are all sorted by size. So let's leverage that. We'll start from the middle element, which is `26`. Is `26` smaller or larger than the number we're looking for? Well, it's definitely larger. Which means we can be 100% certain that nothing to the right of this middle element could be home to the element that we're looking for, `12`. So, we can *eliminate* everything to the right of our array here.

Okay, so we have 3 elements to look through now! Let's choose the middle of those, which happens to be `19`. Is it bigger or smaller than `12`? Definitely bigger; so we can eliminate everything to the right of this node, since we know it'll be too big to be the right number.

Alright, one more number to remains, and that's the one we'll check! And of course, it's the node we've been looking for: `12`! Success!

If we take a look at this example again, we'll notice that with each check and comparison that we do, we actually *eliminate* ***half*** *of the remaining elements* that we need to check. Think about that for a second.

> In the process of narrowing down the search set, we also remove half of the search space. That's enormously powerful — and that's exactly what makes binary searches so powerful.

We reorganized our BST into an array, but a binary search on a tree and a binary search on an array function the same — just as long as all of the elements are *sorted*. That part is key. In an array, the elements to the left of the middle element are the left “subarray”, just like how in a binary search tree, the elements to the left of the root node make up the left “subtree”.

This algorithm is incredibly powerful when you think about the fact that we can have *tons* of nodes in a tree, but not have to search through all of them (the way that we would have to if we were searching through all of these elements one by one, or in a *linear search*). A binary search is far more efficient if our data is ordered and sorted so that we can easily cut our search space in two.

Hopefully, by seeing a binary search in action, you can start to see where it gets its name from — and why it is so perfectly named! We can search through a huge dataset, but be much smarter about it by *dividing our search space in half* with each comparison that we make.

Everyday binary searches

Abstractions are at their best when you can wrap your head around *when* they are used and *why* they are so important and what makes them so useful. Which begs the question: when on earth do binary searches and binary search trees actually get used in the larger world of computer science?

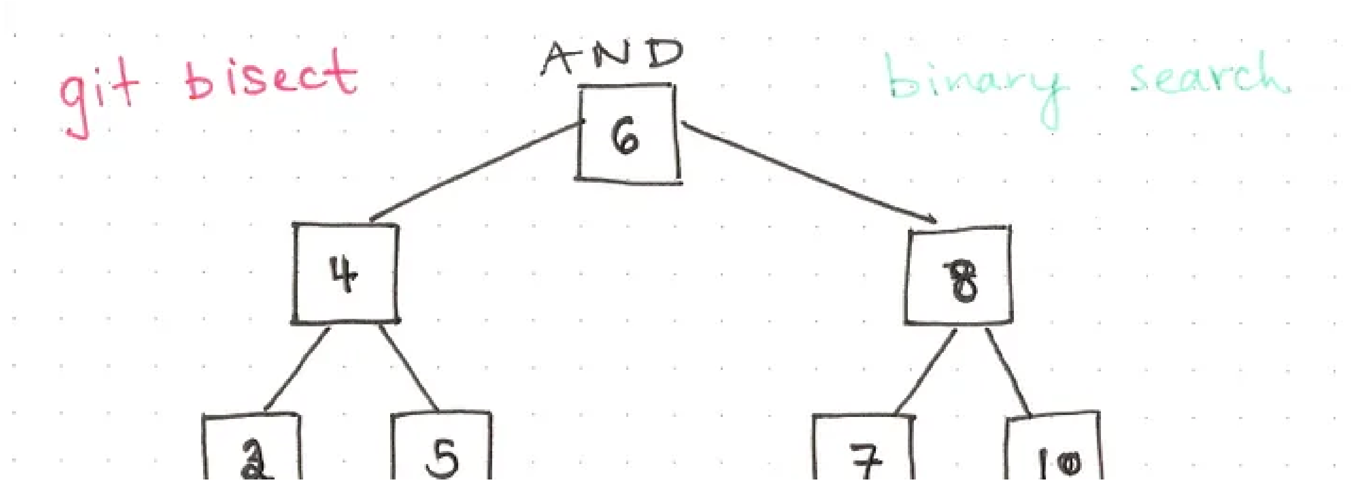
Well, the answer is: in a lot of places! To be a little more precise in my answer: in databases, for one. If you’ve ever seen the term **t-tree** or **b-tree** in when working with a database, these are both types of binary tree data structures. But wait — it gets better! A database uses indexing to search for the correct row to retrieve and return. How does it search for the correct index? You guessed it: by using binary search!

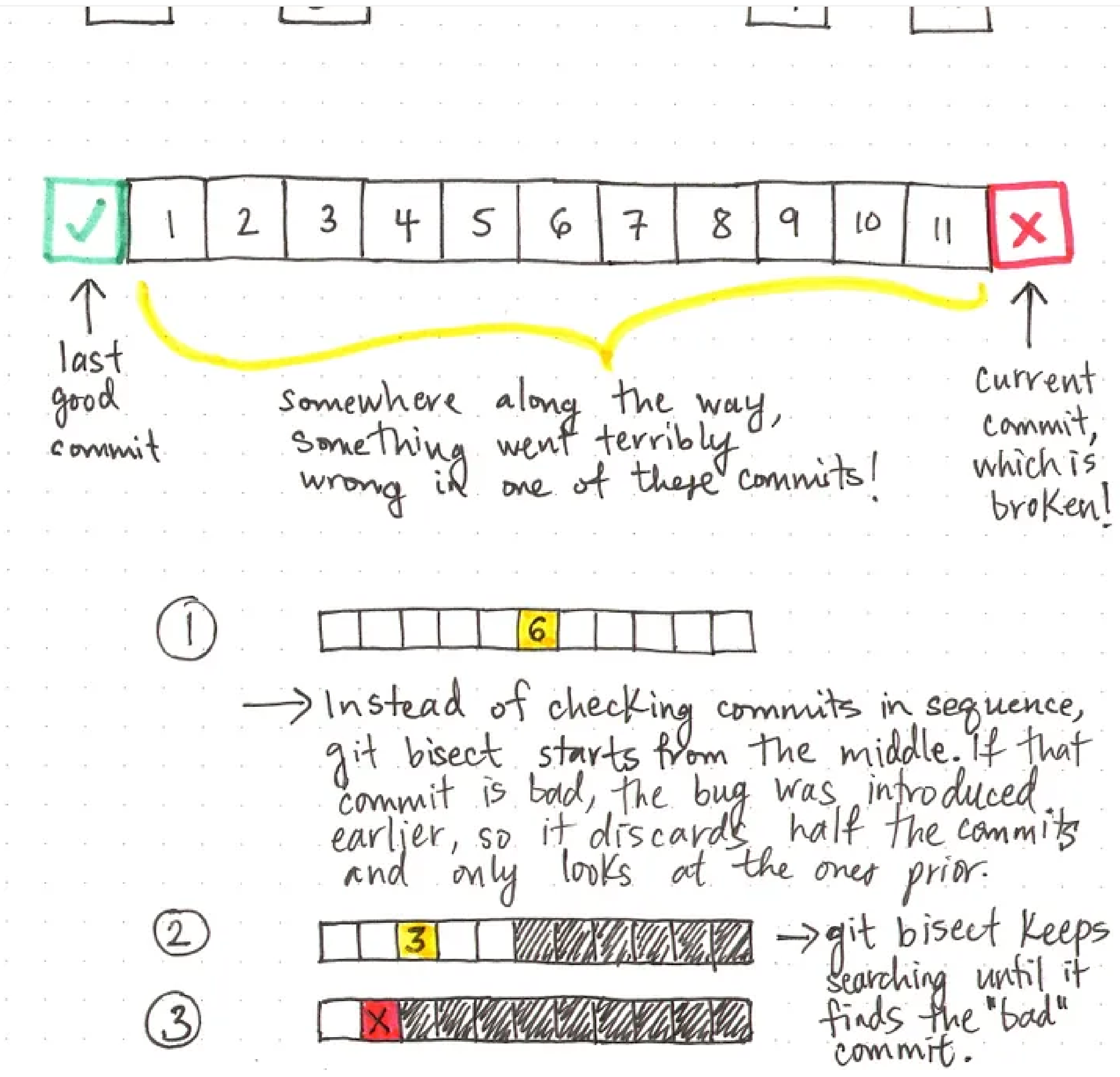
However, binary searches and binary search trees aren't just used in a lower level context; some of us have probably already interacted with them directly.

If you've ever been working on a project and realized that, somewhere along the way, you introduced a bug or broke a test, you might already be familiar with the awesome git command that is `git bisect`. According to the git documentation, this command lets you tell git to search through a list of all of your commits:

> *You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then* `git bisect` *picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.*

How does this magic even work!? Through the beauty of a binary search tree, of course!





Somewhere along the way, something went terribly wrong in one of these commits!

→ Instead of checking commits in sequence, git bisect starts from the middle. If that commit is bad, the bug was introduced earlier, so it discards half the commits and only looks at the ones prior.

→ git bisect keeps searching until it finds the "bad" commit.

The git bisect command uses binary search to find “bad” commits.

By telling git when the last “good” commit was, it searches through all of the commits from then to now. But it doesn’t search chronologically of course — instead, it starts from the middle commit, and if you confirm that the middle commit is also “bad”, then it will continue to the earlier commits before that one, and discard all the commits afterwards. It conducts a binary search, exactly like the ones that we’ve been dealing with today!

Amazing, right? Who knew that so many of us were using such a simple concept to debug things on such a high level?! I don’t know about you, but I, for one, am convinced: we’re all better off when we leaf the traversing up to the binary search trees.

Resources

1. Binary Trees, Professor Victor Adamchik
2. Difference between binary search and binary search tree?, StackOverflow
3. Data Structure and Algorithms Binary Search, TutorialsPoint
4. Binary Trees (Advanced), Professor Nick Parlante
5. Problem Solving With Algorithms and Data Structures, Brad Miller and David Ranum
6. A tale of debugging with git-bisect, Hassy Veldstra

Programming Computer Science Tech Algorithms Data Structures