# Design and B-Trees

**Gabe Johnson**

**Applied Data Structures & Algorithms**

**University of Colorado-Boulder**

This sequence involves building a B-Tree data structure, which is really tricky, and we're only making you do this because we're mean.

Actually, there's a head trip here. Since B-Trees are so difficult to implement correctly, you'll need to plan your approach deliberately. So the real topic at hand here is design, being able to properly frame the problem before you try to solve it.

I should say also that my background is in both computer science and design. Throughout this sequence I'm giving a sort of heavy handed account of design, and I'll say "do this", "don't do that", "here's what you should think about". Of course, as you gain experience you will figure out your own strategies. The advice in this sequence is good advice, but not the only advice.

If by the end of this you have an appreciation for design as something worth studying and practicing in its own right, then it will all be worth it. So, let's get started!

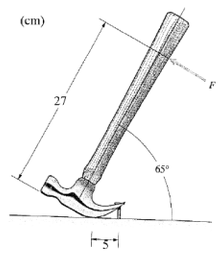
Episode 1

# Software Design

# About the Homework…

**Two Homework Assignments:**

1. **B-Tree Design Document**
2. **B-Tree Implementation**




There are two homework assignments here. The first one is to write a design document that outlines how you're going to implement B-Trees. And the second is the actually implement them.

We're purposefully giving you a task that is probably beyond what you think you can do. It's like pulling a nail out of a board. You'd be a little nuts to do it bare-handed. Realistically, you'd use a tool like a hammer or a prybar. Well, with the B-Tree assignment we're metaphorically asking you to pull the nail out. Your tool will be a deliberate, systematic design strategy that you'll develop yourself.

The design document is worth the same as the other assignments, 20 points. The B-Tree assignment is out of 20 points as well but you can actually get a total of 40 points if you implement everything correctly. Last time I did this with about 150 students, maybe five people got all 40 points.

# Design

**(Algorithm) Design**
**(Software) Design**
**(Interface) Design**
**(Implementation) Design**
**(Testing) Design**

This episode is about design. It is not a stretch whatsoever to say that design saves lives.

I'm using the word 'Design' here, which is kinda fraught because there are hundreds of different kinds of designers from interior designers to landscape designers to medical device designers, to other kinds of designers that are so cool they don't even have design in their title. Like Architects.

I say 'fraught' because the word itself means different things to different people. It's a big concept.

So I'll narrow this down a bit.

Mostly I'm talking about how we as computer scientists design algorithms.

And as how we as computer programmers go about writing software.

And if we're making something to be used directly by humans, there's interface design as well, so there’s something of ergonomics and cognition in there.

And if we're engineering, then you have to design your implementation and testing strategies as well.

You will get this in much greater detail later on in software engineering courses, but I think it is an important enough topic that you should start to think about it, if even superficially, at this point in the game.

# Design Saves Lives

This is the Therac-25 medical device. It is supposed to dose patients with a relatively low amount of radiation in most cases.

Most. Not all.

The Therac-25 is often propped up as an example of deeply flawed software design _and_ user interface design. For our purposes in this class, the software that controls this thing was not designed or developed in such a way that controlled software testing could be performed.

The safeties were supposedly done in software. But there were certain conditions where they didn't work. This caused severe radiation poisoning in some patients, and actually killed other patients.

So, don’t be like Therac-25.

# What could possibly go wrong?

who’s this for?
why do they need this thing?
what are the use cases?
how should it work?
how do we know if it works?
how do we know if it sets the building on fire?
how do we know if it is safe for the user?
what conditions should it fail in?
…
and so on, and so forth

Some questions to ask yourself.

First two questions, and the only two questions that you should absolutely always ask but most people don't are "who is this for" and "why do they care"? If you have the temerity to ask this, that's awesome! But don't have an answer, stop. Figure this out first.

Assuming you know that bit...

Ask more questions, what are the use cases? how should it work? how do we _know_ if it works?

Ask about the failure cases, especially safety concerns. Maybe the thing will have known failure cases, like if it is below freezing or we get bad input?

# “Requirements”

Gathering requirements for building a satellite?
please nail this down before launch

Gathering requirements for some misc social media app?
an iterative design/build/examine approach probably is best.

You don't need to have all of these things nailed down to start with, because you can always go back and fix things, but keep in mind that for many things is way way easier to change the design when you're still designing, than later when it is in production being used by real people.

When you have all that written out, and better yet, openly discussed among collaborators, you have a much better sense of what's involved, and where the hazards are.

I have "Requirements" in scare quotes, because sometimes you won't know really what the requirements are until you've built a prototype, or even until the thing is actually in production use.

There are design methodologies that demand you to have a complete, formal list of requirements from the start, and that makes sense if you're building something that is being launched into orbit. But if you're making an Uber for dog walking app, you might get more mileage out of just building something and seeing how people use it.

And this is what I meant earlier when I said that you'll need to design your overall strategy.

# Avoid Hubris

Many programmers think they are *teh best evah*!

**Many reasons to just dive in and see what happens**

**just don’t let hubris be one of them**

Please think about testing, and do it if it makes sense.
(It usually makes sense.)

Requirements are hard. Coming up with a list of requirements and testing strategies is a job in itself, so many programmers skip this step because they think they're just that great at hacking code, but they're wrong.

There are many reasons to skip the requirements and test plan step. Hubris should never be one of them.

You can do this in a really comprehensive way, or it can be slap-dash. I've never seen a programming task that didn't benefit at least a little from some up-front engaging-of-the-brain.

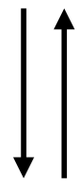
Episode 2

# Testing as a Design Strategy

There are loads of ways to design software, or anything really. As you get more experienced writing code, you'll see many strategies. One useful strategy is to use automated testing. This episode covers test development. You'll get into this much more deeply when you take a software engineering methods course.

# Requirements <—> Tests

**What's the requirement?**



**Automated test.**

**When not coding, maybe "get out of the building" at least metaphorically. Think about the users, their motivations, their use case. Observe. Try to avoid thinking about code or implementation details.**

**When writing tests you think critically about a narrow topic and might realize the requirement is wrong, incomplete, or something else.**

When you have your requirements, you've got a great starting point for writing tests. And I should say here that this is not a one-way road.

You might write some tests and eventually realize that there's something that you missed in the requirements, like there's some use context that you didn't think of. And it is great when you realize this, because at this stage, adding or changing requirements is way way cheaper than it will be later, when your spaceship is sitting on the launch pad and you realize your guidance code is working in inches but the rest of civilized humanity uses metric.

# Testing: Five Questions

1. What's it do?
2. What's the input?
3. What's the output?
4. What potential problems are there?
5. Do we have a faulty definition?

```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

Here's a list of five questions to get you started in thinking about writing tests. We'll work with this pretty basic requirement here. It's a function that accepts some integer array, does some math, and returns the sum of their squares.

# 1. What’s it do?

```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

What’s it do? This is easy enough. We know it is going to compute some squares, and add them together.

# 2. What’s the input?

```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

What’s the input?

We see from the function signature that it takes an array of integers, conveniently called _input_. Think about what could be wrong with this input. Empty? Very big? Negative numbers? Zeros? Some of these might be problems, others might not be. You have to engage your brain.

# 3. What's the output?

```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

What's the output?

We see from the function signature that it returns a single integer. So we know that inside this function definition we'll need a return statement, and we have to guarantee that it actually runs. We know that squares are non-negative, so returning a negative value would be a fail.

# 4. What potential problems are there?

```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

What could go wrong?

Say we get input that is very large. Maybe one of those numbers is so large that when we take its square, we don’t have enough room inside this computer’s int data type to store the result. Or, maybe we get that problem when we start adding the squares together.

# 5. Do we have a faulty definition?


```
//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input);
```

Now, do we even have the right definition here?

We don’t even know how much data is contained in the input array. In some languages we don’t need this info passed in explicitly (like Go and Python) but others we need to be given the input size. Maybe the person who gave you this definition messed up? Maybe they spend their days hacking Java and it didn’t occur to them that C++ is different. This is always a possibility.

I know your boss might have given this to you, but like the bumper sticker says... question authority!

# Writing Tests

```
void test_get_sum_of_squares(testing_framework* t) {
  // this is an automated test that runs at build or deploy time.
  //
  // create a bunch of different inputs, send them to get_sum_of_squares,
  // and see if we get the expected output.  the way you do this is highly
  // dependent on language and testing style. this is in the style of Go.
  //
  // ... <tests go here>
}


//
// This function returns the sum of squares of the input array.
//
int get_sum_of_squares(int[] input) {
    // run-time checks on input, output, perhaps
    // global context if there is one.
}
```

All of the observations we just made could potentially be turned into an automated test. These tests are for the benefit of you and your collaborators. They're run at build-time, as you're writing code, or some other time like when you check your code in, or when it is deployed to production. At any rate, these are tests that can be fired up at any point.

The exact strategy, even the syntax, for writing tests depends on which language you're using, which framework, and then other social guidelines maybe defined by your company or community.

Some observations could be the basis of sanity checks inside your code, like checking to see if a number is zero before dividing by it and creating a black hole. These are run-time tests and they usually deal with checking the input or contextual state for sanity.

# Write Tests First

**this helps clarify what the code should / shouldn’t do.**

**gives you tools for:**
**- finding bugs**
**- tracking progress**
**- letting you know if you screw something up later on**

If you write your tests first, not only does that help clarify in your mind what exactly your code should and shouldn't do, it also gives you a tool for finding bugs and tracking progress.

# B-Tree Unit Test Example

**Using the catch.hpp framework and our homework Makefile…**

```
// in btree_test.h
TEST_CASE("My B-Tree Test Rig", "[size]") {
  btree* t1 = make_empty_tree();// hand build an empty tree in other function
  REQUIRE(btree_size(t1) == 0); // trigger error if empty tree size isn't 0
  btree* t2 = make_three_node_tree(); // hand build 3-node tree in other function
  REQUIRE(btree_size(t2) == 3); // errors if 3-node tree size isn't 3
  // and so on
}


$ make
$ ./btree_test “[size]”
```

The homework doesn't actually call for a B-Tree size function, but there's nothing stopping you from writing your own. If you did, you might test it like this.

And here we're using the catch unit testing framework. Just edit the testing file, in our case btree_test.cpp, and add code! Then build and run it like like the slide shows. You do need the double quotes and square brackets, that's just a peculiarity of how catch works.

# Tests test your assumptions

**unit test ‘size’ failed!**

**does the size function report number of nodes?**

**oh wait**

**or does it report the number of keys?**

**or children?**

**or memory usage?**

**dang what am i even doing**

The previous slide was based on the assumption that the size of the b-tree is the number of nodes. But if the size function reports the number of keys instead, then we have a problem. So if you write test code and its assertions fail, the problem could be the B-Tree code, or it could be the assumptions encoded in your unit test.

Once faulty assumptions are recognized we can strengthen everything by updating our tests first.

# In Closing…

Design is as much about problem *framing* as it is about problem *solving*.

I'll just close this episode out by saying that designing is a critical skill that you'll need to build over the course of years. Be reflective on how you approach it, and try to develop your own strategies.

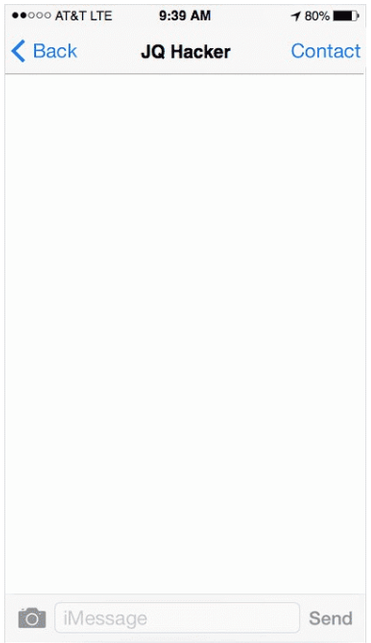
Design is as much about problem _framing_ as it is about problem _solving_.

Episode 3

# Operation Times

We've seem big oh notation, which gives an upper bound on resource usage as a function of input size. The most common resource is time. This episode touches on a different measure of time, namely wall clock time.

# Intuition, Again!




Computational Complexity analysis is one of the most important intellectual tools you'll learn. Don't just be able to get the right answer. Understand the intuition behind why this is interesting, and how it can save you a lot of mental duress. If you know in advance that an algorithm is fast, slow, intractable, then you can plan your life around it.

Consider the length of time it takes to perform basic operations using real hardware...

| | |
|---|---|
| execute typical instruction | 1/1,000,000,000 sec = 1 nanosec |
| fetch from L1 cache memory | 0.5 nanosec |
| branch misprediction | 5 nanosec |
| fetch from L2 cache memory | 7 nanosec |
| Mutex lock/unlock | 25 nanosec |
| fetch from main memory | 100 nanosec |
| send 2K bytes over 1Gbps network | 20,000 nanosec |
| read 1MB sequentially from memory | 250,000 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |
| read 1MB sequentially from disk | 20,000,000 nanosec |
| send packet US to Europe and back | 150 milliseconds = 150,000,000 nanosec |

source: http://norvig.com/21-days.html - totally recommended reading btw

This is real, wall-clock durations for a bunch of common computing operations. It is easy to think of computer stuff as just sort of being vaguely "fast", but when you quantify how long different things take, you can see that your design decisions about which operations your code does, and how often, can have a big practical effect on how your program performs.

# Awful O(n) code

```
function darken(imageFileName, w, h) {
    output = pixel[w][h] // 2d array of pixels
    for i := 0 to w {
        for j := 0 to h {
            pic = load(imageFileName) // WAT
            output[i][j] = pic.pixel(i, j) * 0.5
        }
    }
    return output
}
```

For example, here is a function that has a runtime complexity of O(n), where n is the number of pixels in an image. It is awful because it loads the image from disk on the inside of that double loop, and we just saw how long that takes.

# Not Awful O(n) code

```
function darken(imageFileName, w, h) {
    output = pixel[w][h] // 2d array of pixels
    pic = load(imageFileName) // Whew.
    for i := 0 to w {
        for j := 0 to h {
            output[i][j] = pic.pixel(i, j) * 0.5
        }
    }
    return output
}
```

This code has exactly the same runtime complexity, O(n), but the line that loads the image is moved to the outside so it only executes one time.

The before and after here shows that you can have a profound impact on the practical behavior of your code without changing the order of complexity.

# B-Tree Context

- Minimize height of tree
- Each ‘hop’ represents a long operation (e.g. disk access)
- Used in databases, file systems
- Algorithmic complexity basically same as red-black trees
- Practically, B-Trees are way better in their context

B-Trees are a data structure that is designed to minimize the height of the tree, because each time we need to access a child node's data, it corresponds to a really long disk operation.

B-Trees are used in databases, file systems, and I'm sure relatives of B-Trees are used in all sorts of other contexts where you can't fit all the data you need into memory.

Even though the algorithmic complexity of B-Trees is essentially the same as, say, a red-black tree, practical contexts mean that B-Trees are way better at a certain category of work, while red-black trees are way better at a different category.

So, just don't look at the big-oh of an algorithm and think you know the whole story. Wall clock time and computational runtime are not the same thing. big oh can hide terrible wall-clock times. For example, a constant time algorithm can take 5 minutes.

# When to optimize

**Best Case:** input has low variance & need good performance for typical case.

**Average Case:** input has large variance (e.g. no ‘typical’ case), just do your best.

**Worst Case:** rare or unexpected inputs must absolutely not fail.

We should probably say something here about optimization.

We’re often concerned about the best, the average (or typical), and worst-case complexity of an algorithm. We can optimize for each, based on whatever situation we have. We can roll our own algorithm or we can use something well-known.

Best case: optimize for situations where we have reasonable understanding of the data, and know the best case will be common.

Average case: optimize for the typical case when input is not particularly weird, but we don’t know what is coming. This is what most algorithms do.

Worst case: optimize worst-case scenario to prevent catastrophes: missile defense, air traffic control, real-time medical hardware, etc.

# Premature Optimization

**Please don't.**

**Requirement: render() < 16ms**

**Status: render() ~= 5ms**

**Question:**
Spend one month getting render routine to execute in 4ms?

**Answer:**
Depends.

But by default, it is not 'yes'.

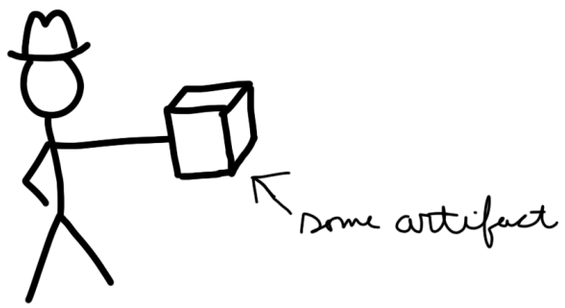
Engineers love to optimize things. I know I do! But you should be aware of the trap of premature optimization. Say you've got a graphic routine that has to update 60 times a second, or within about 16ms. Your current code runs in 5ms, but you're sure you could get it down to 4 if you spent the next month working the problem. Is that a useful tradeoff?

It might be! But before you do it, it makes sense to have some evidence and weigh the relative benefits of doing that versus something else.

Episode 4

# Indiana Jones and the Temple of the B-Tree

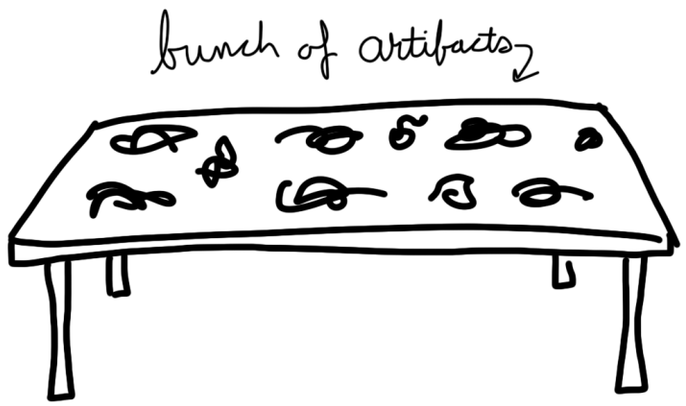
# Intro




Imagine you work at a museum. Indiana Jones brings things in from time to time, and it is your job to keep track of everything. He has been known to need access to these things in a mad rush. The problem is, there's a lot of stuff to keep track of.

And if Indiana can't get the thing he needs right now, the world may end in some sort of cosmic apocalypse.

# Small Piles of Stuff

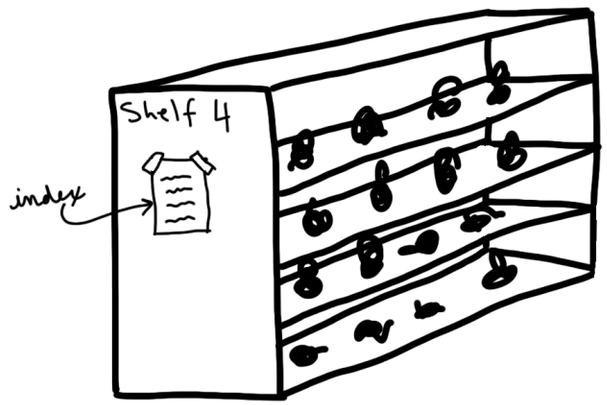



At first the museum collection is small, so you can put it all on the table next to your desk.

Over time the desk becomes cluttered, and you can’t spot things easily. You need an organizational system.

You need an index. And rather than putting things on a desk, you need storage shelving.

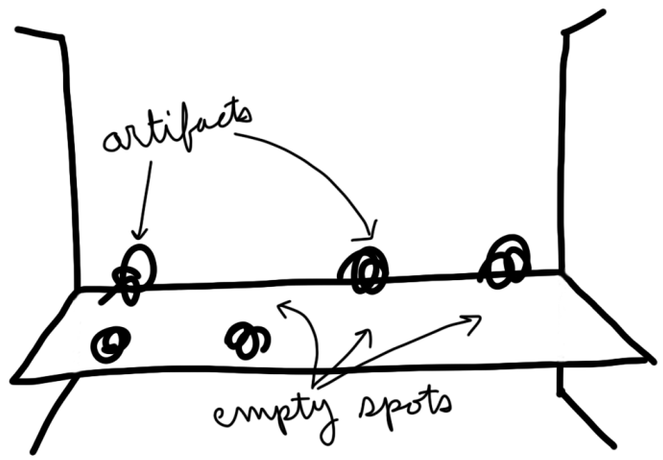
# Shelves of Stuff




You start out with a few shelving units, and keep a sheet of paper that has the artifact description next to which shelving unit it is on.

You keep another sheet of paper taped to each unit, which lists the artifact and which shelf it is on (lower shelf, middle, upper, etc.)

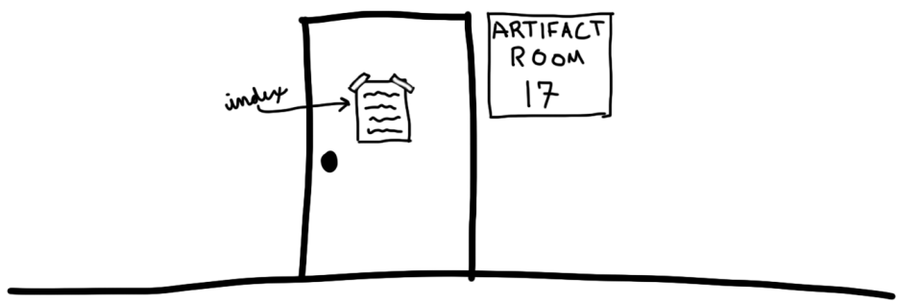
# Shelves Are Not Full




It is time-consuming to move things around on the shelves all the time, so your strategy is to try to keep the shelves at about the same level of fullness, and every time you have to add a new shelf, you make sure it is only half-full. That way when new items come in, you don’t have to do the expensive, time-consuming process of shuffling stuff around.

The shelf approach works for a while. Eventually Indiana Jones brings so much stuff to the museum that you need to spread out into different rooms.

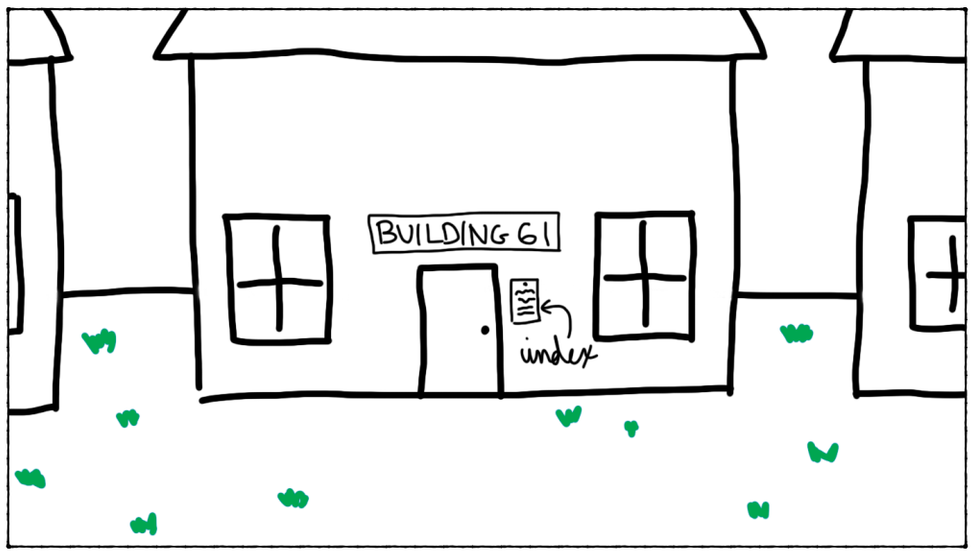
# Rooms of Stuff




You expand your storage operation into separate rooms, much to the dismay of the graduate students who were working there and now have to hide under the awning on the roof. Add another layer to your index strategy. And of course, things keep moving around, since, you know, you have to keep all the mystical spheres together.

Artifact J7-394 (Mystical Sphere): Room 1
Artifact E-032 (Diamond Brain): Room 3

# Buildings of Stuff

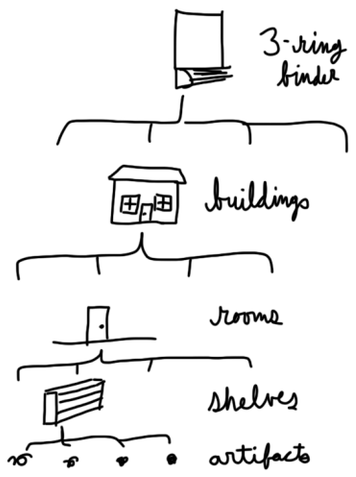


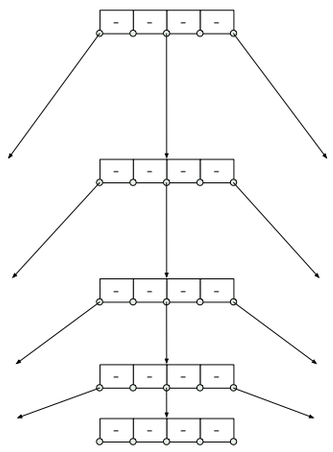

The sheer volume of crap our cowboy archaeologist brings now demands the use of separate buildings.

Every time you create a new tier of organization, you need to create another index-of-indexes.

Your main 3-ring binder tells you what building an item is in. In each building, there's another binder that tells you which room it is in. In each room, a sheet of paper tells you which shelving unit. And so on.

# The B-Tree Metaphor



The preceding silly story about Indiana Jones and the Monster Museum of Doom is actually about B-Trees.

Remember that table with the relative times that it takes to perform operations based on what kind of hardware we're using?

The one that looks like this...

| | |
|---|---|
| execute typical instruction | 1/1,000,000,000 sec = 1 nanosec |
| fetch from L1 cache memory | 0.5 nanosec |
| branch misprediction | 5 nanosec |
| fetch from L2 cache memory | 7 nanosec |
| Mutex lock/unlock | 25 nanosec |
| fetch from main memory | 100 nanosec |
| send 2K bytes over 1Gbps network | 20,000 nanosec |
| read 1MB sequentially from memory | 250,000 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |
| read 1MB sequentially from disk | 20,000,000 nanosec |
| send packet US to Europe and back | 150 milliseconds = 150,000,000 nanosec |

source: http://norvig.com/21-days.html - totally recommended reading btw

# Memory = Fast. Disk = Slow.

| | |
|---|---|
| fetch from main memory | 100 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |

**80,000x difference**

fetch from main memory -- 100 nanosec
fetch from new disk location (seek) -- 8,000,000 nanosec

If these numbers are to be believed (and I do), fetching data from disk is 80,000 times slower than fetching from memory.

If a memory fetch is analogous to looking up the location of our data in our 3-ring binder, then a disk fetch is analogous to trudging across town to some warehouse where Artifact J8-382 happens to be.

# B-Tree Uses

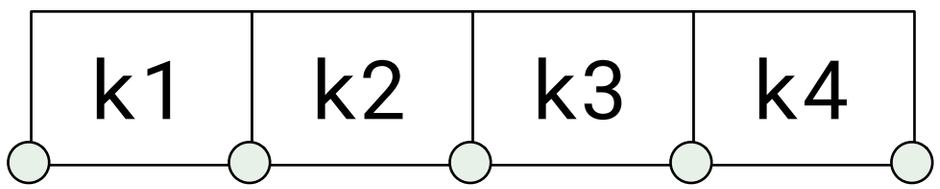
**Filesystems**

**Databases**

**Cloud computing**

We use B-Trees as a way to optimize our use of fast data stores while minimizing the interaction with slow ones. Classic examples are filesystems (the program that controls reads and writes of files with physical hardware) and databases (the same idea, but possibly spread out over multiple computers in a cluster).

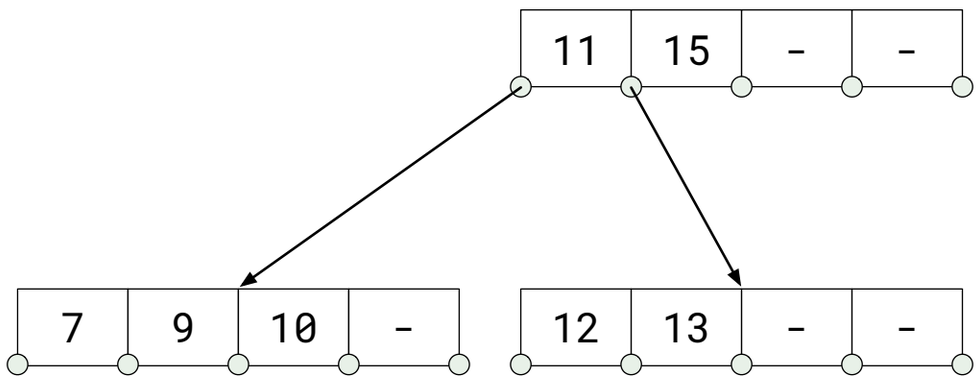
# Main Idea w/B-Trees




There are nodes, keys, and links.

A node has keys inside it. Here’s a node with four keys labeled k1 through k4.

# Keys and Links




A Key is a number that fills up a node from left to right. Notice how I draw links represented as circles that sit between key slots. That key represents a boundary. Everything less than 11 goes left, everything larger to the right. Duplicates can be allowed but for our purposes we won’t do dupes for the homework assignment.

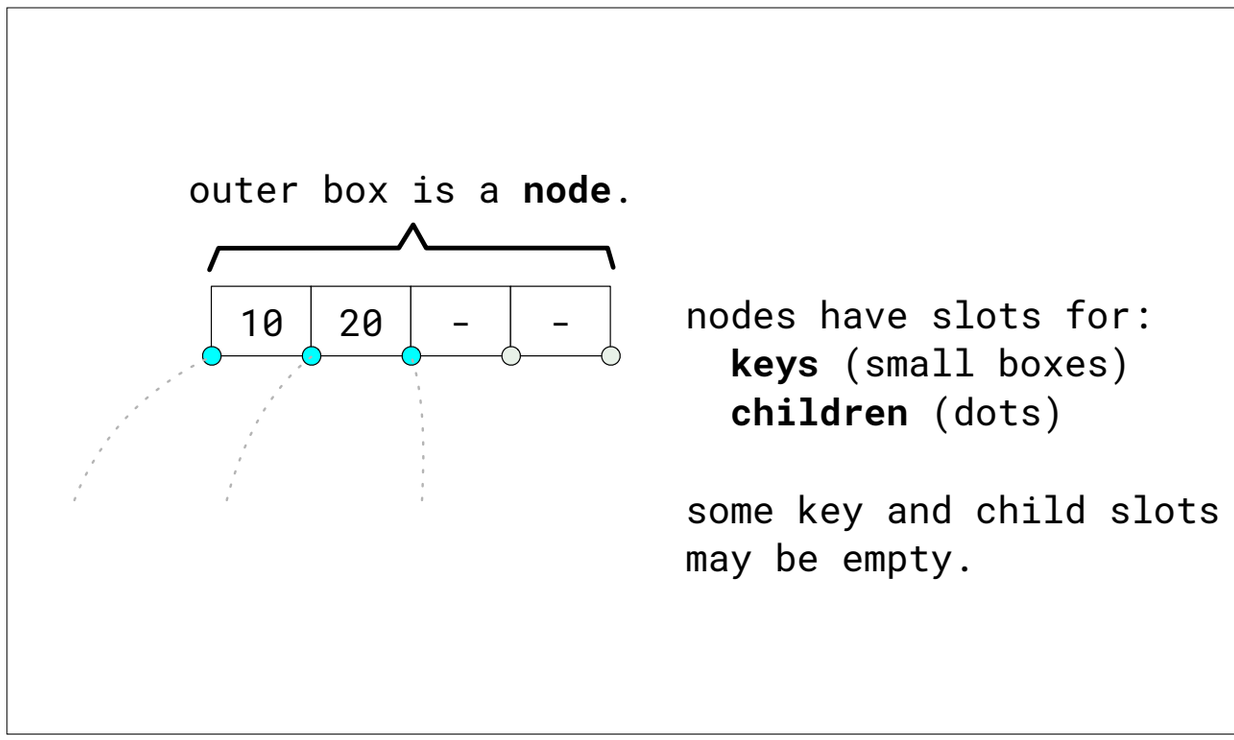
# Invariants Next

Now that we've had an extended into to what B-Trees are for, we'll get into the node and structure invariants in the next episode.

Episode 5

# B-Tree Invariants

Now we're going to go over the basic structure of a B-Tree, its nodes, and all the invariants. This is maybe a little bit longer of an episode because it doesn't make sense to break this up.

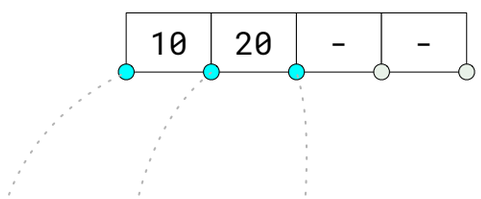



A B-Tree node has slots for __keys__ and __children__.

As pictured, some slots can be empty, but there are rules about how many can legally be empty.

Within a B-Tree, all the nodes have the same number of allowed slots. This is called the _Order_ of the B-Tree but unfortunately there's inconsistency in specifically what people mean by this.

min and max child
and key counts:




```
m          = max num children
ceil(m/2)  = min num children
num keys   = num children - 1

in this case…

m            = 5
ceil(5/2)    = 3
num children = 3
num keys     = 2
```

Sometimes the order refers to the max number of keys, other times it refers to the max number of children. For this class we'll standardize on the child count.

So if we have an order-M B-Tree, it means each node has at most m children. And each node, except for the root, has to have at least half that, rounding up. And the number of keys is always the number of current children minus one.

So for example, in the diagram here, m is 5 --- just count the dots at the bottom of the node. 5 divided by 2 is 2.5, so round up and we get 3 as the minimum number of children.

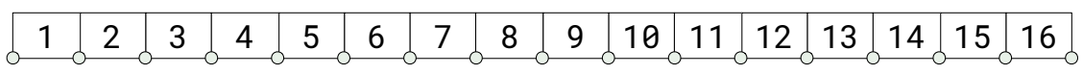
In this particular node, we happen to have two keys and three children. But it could also have three keys with four children, or four keys with five children. Did I get that right? This is a spot where it would be super easy to have an off-by-one error.

in other words…

there's a minimum number of keys (mustn't be too empty)
and a maximum number of keys (can be full)
and we always have one more child than we have keys.

if m = 5, we can have between 2 and 4 keys.
(that is, between 3 and 5 children)

if m = 17, we can have between 8 and 16 keys.
(that is, between 9 and 17 children)

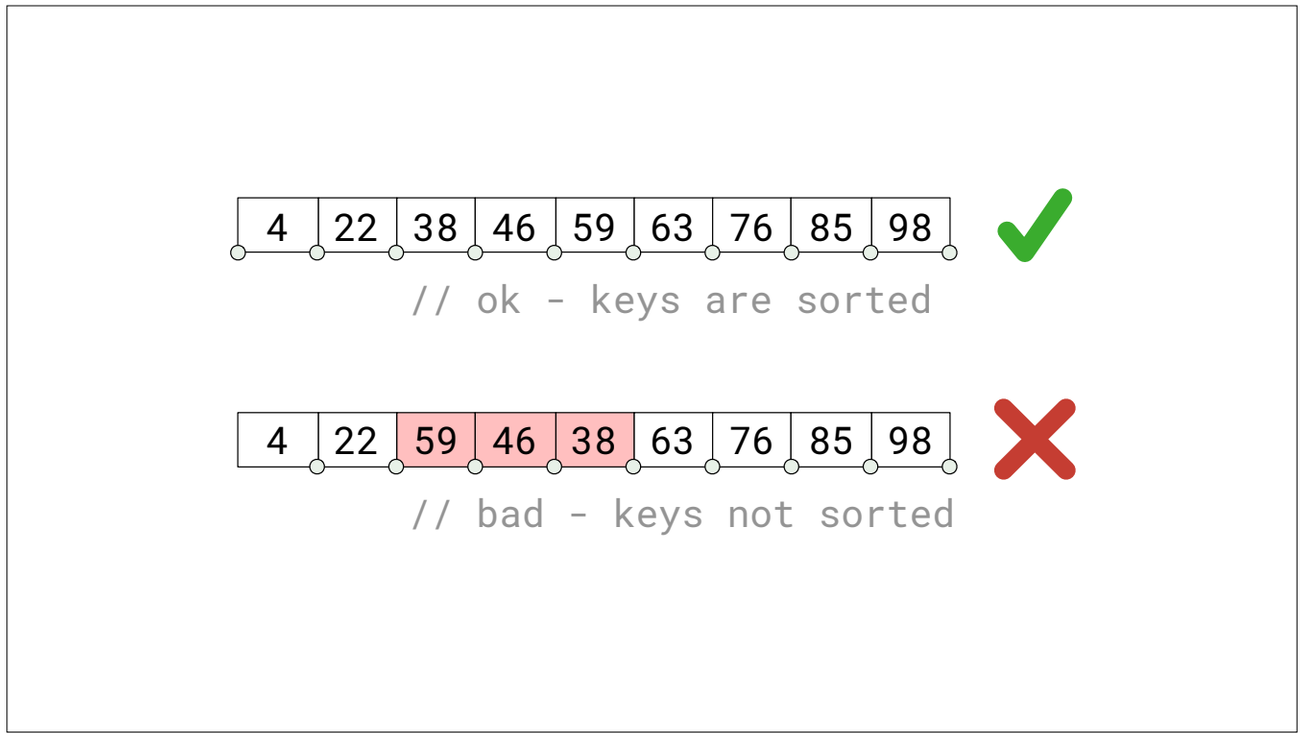



In other words, there's a minimum number of keys. The node shouldn't be too empty.

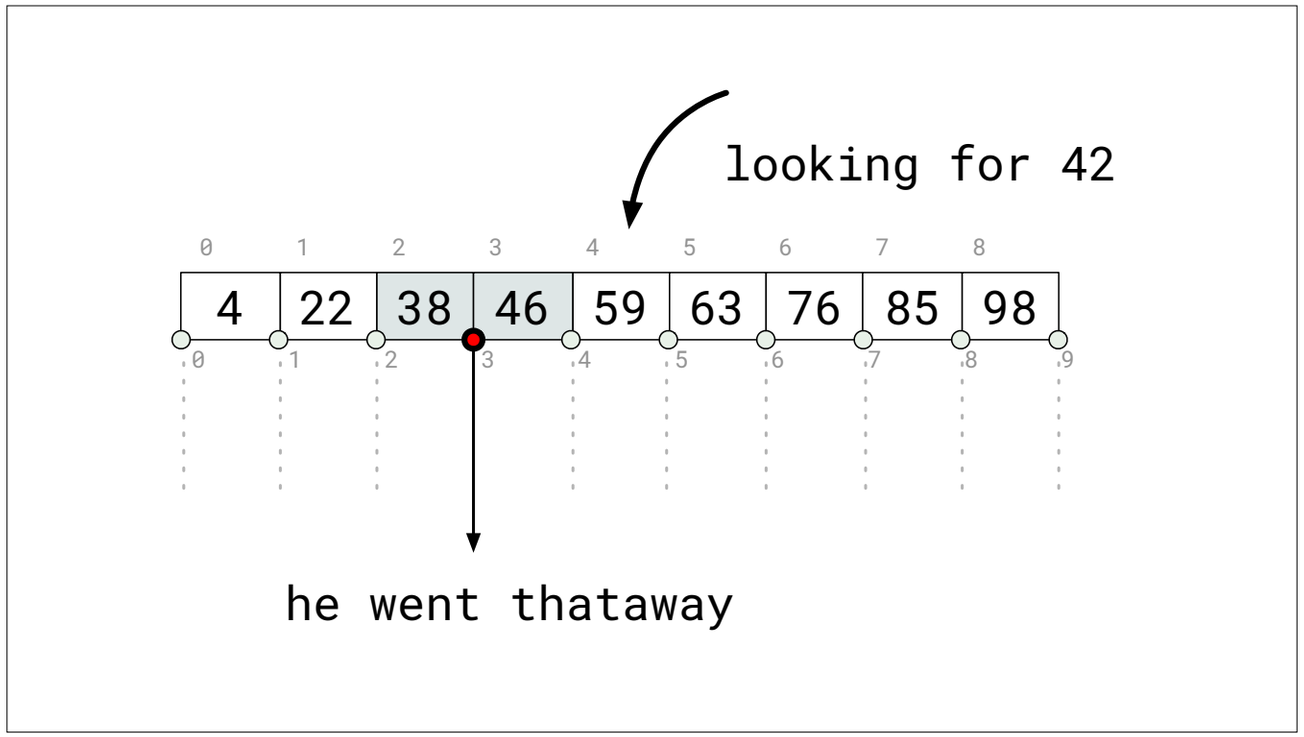
And there's a maximum number of keys, too. The node shouldn't be over-full.

Lots of these diagrams use an m=5 B-Tree. There we can between 2 and 4 keys, and 3 to 5 children.

If we had a weirdo m=17 tree, that's between 8 and 16 keys, and 9 and 17 children.



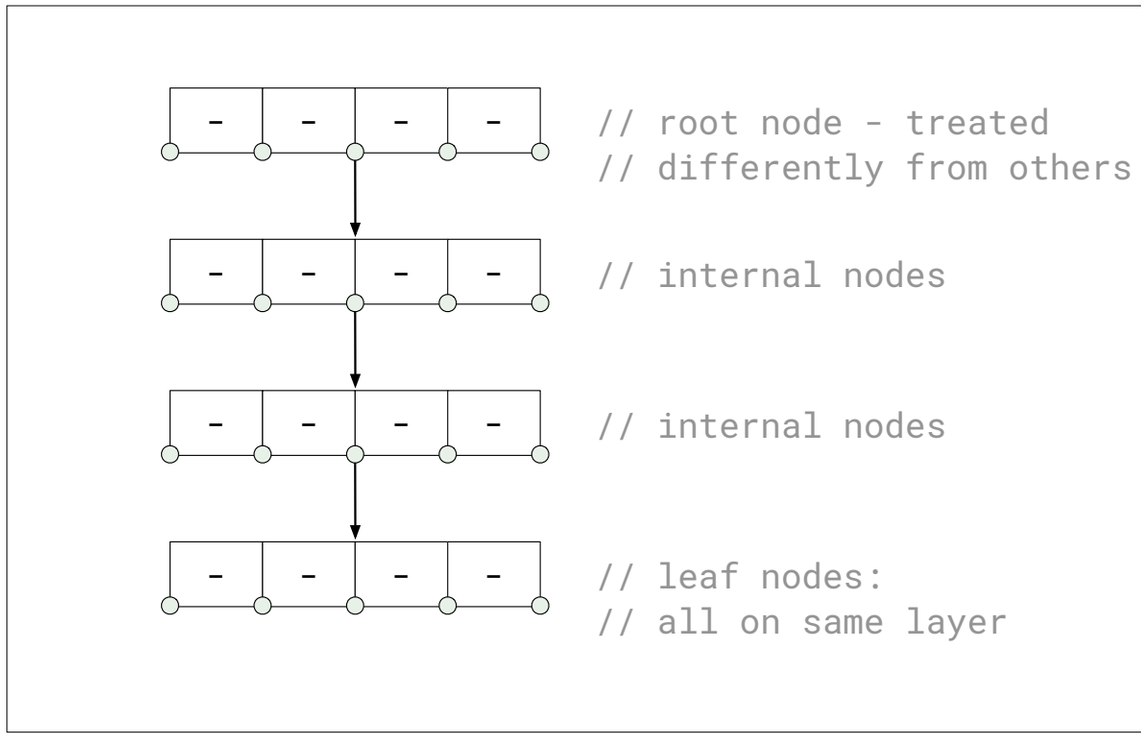

Within a node, the keys have to remain sorted. This is one common source of bugs. The reason for this is that each key serves as a separator, so we know which child link to follow when we're traversing the tree. If the keys aren't in order, there's no way to know which child link to follow.




So if we're looking for some specific key, it works like this. If we're looking for 42, first thing to check is if the key is here. In this case it isn't, but we have a bunch of child nodes to look in.

If your target value is smaller than the first key, follow the first child. Or if the target value is larger than the last key, follow the last child. Keys basically partition the child nodes. So if we're looking for key 42, we know that if it is in the B-Tree, it will be down the child link at index 3, between keys 38 and 46.

B-Trees have the same sorting semantics that binary search trees do. Lower values to the left, higher values to the right.




The root node is the one at the very top, and it is treated somewhat differently than all the others. We're a little looser with the invariants that pertain to it.

Between the root and the leaves we have internal nodes.

The leaves of a B-Tree are all at the same level. They don't have any children, otherwise, they wouldn't really be leaves, right?

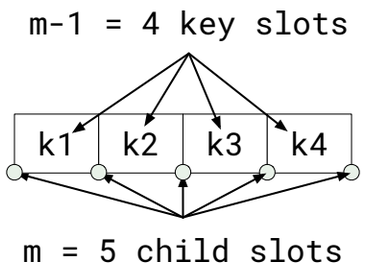
Given these three labels, we can talk about non-root, meaning internal and leaf nodes. We can also talk about non-leaf, meaning root and internal nodes.

# Invariants!

**The following invariants include those for a standard, vanilla B-Tree.**

**We also add a couple of our own for the homework assignment.**

**(1) Every node has at most m slots for children.**

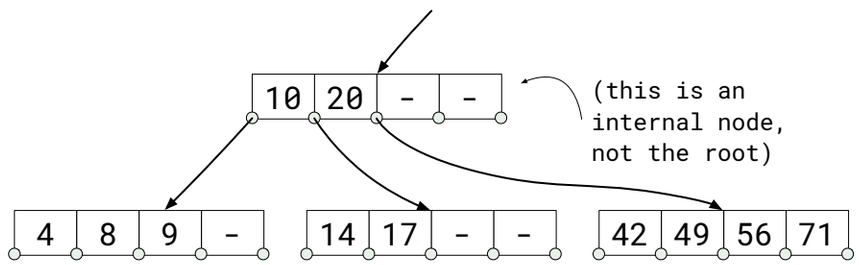



In this case, m=5. This means there are slots for five children (count the dots), and slots for 4 keys.

Every node has at most m slots for children. This is the 'order' of the B-tree.

If we have an order 5 tree, it means there are slots for five children. Just count the dots. And slots for four keys.

**(2) All internal nodes have at least ceil(m/2) children (doesn't apply to root and leaf nodes).**

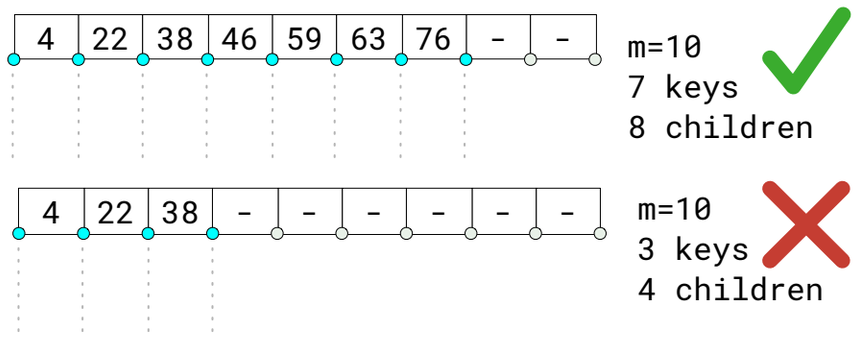



m is the number of allowed children. Here, m=5. ceil(5/2) = 3. Consequently, the minimum number of keys is 2.

All internal nodes have at least m/2 children (round up). This doesn't apply to root and leaf nodes.

So for an order 5 tree, 5/2 is 2.5, and we round up to give us 3, that's the minimum number of children. And it also means the minimum number of keys is two.

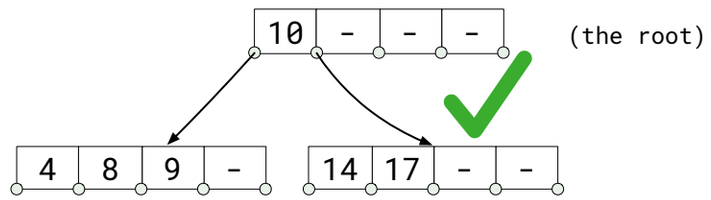
**(2) All internal nodes have at least ceil(m/2) children (doesn't apply to root and leaf nodes).**

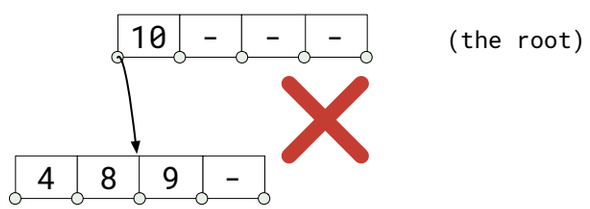



given m=10, requirements are:
5 children
4 keys

As another example of this invariant, if we have an order 10 tree, you can have between 5 and 10 children, and 4 to 9 keys. So this upper one is fine, but the lower one is under-full.

**(3) The root node has at least two children if it is not also a leaf.**



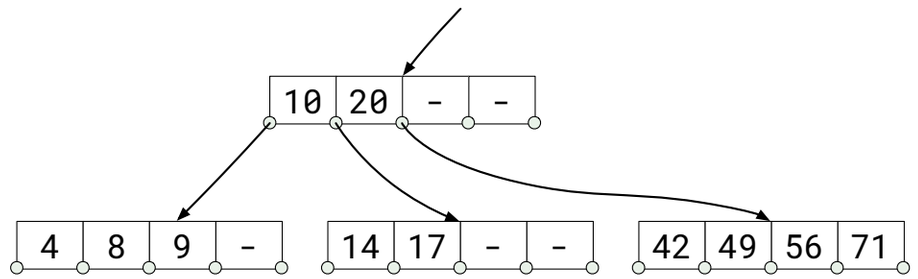




The root node has to have at least two children, assuming it isn't also a leaf. When you have a fresh B-Tree and start adding nodes to it, the root will also be a leaf.

The upper example is OK because the root isn't a leaf, and it has two children.

Lower is bad because the root isn't a leaf but it only has the one child.

**(4) A non-leaf node that currently has k keys must have k+1 children.**

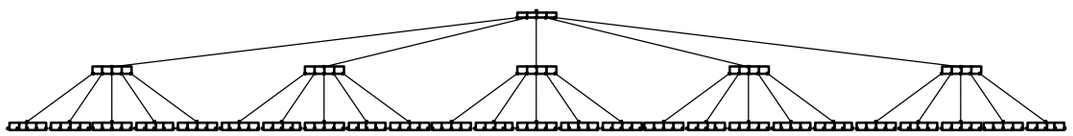



Conversely, if a non-leaf node has X children, it must have X-1 keys.

A non-leaf node that currently has some number of keys must have that number plus one children. This also means that if a non-leaf has some number of children, it has to have that number minus one keys.

All this is a really confusing way to say that there's always one more child than keys.

**(5) All leaves appear in the same level.**

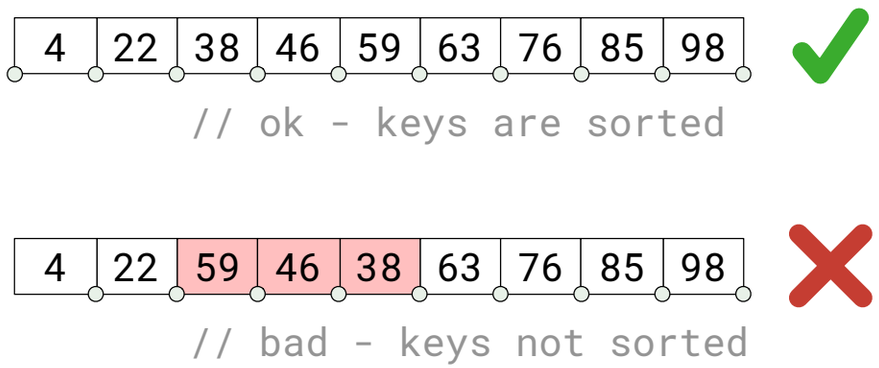


Here, the root is layer 1, internal nodes are on layer 2, and all leaf nodes are on layer 3. It would break the invariant if there was a leaf on layer 2 or layer 4.

All leaves appear in the same level.

This is an order-5 b-tree, with three levels. Root is layer 1, internal nodes are layer 2, and all the leaf nodes are on layer 3. It would break the invariante if there was a leaf on layer 2 or layer 4.

**(6) The keys within a node must be sorted in ascending order.**




The keys within a node must be sorted in ascending order.

This invariant is sometimes not even mentioned but I thought I'd include it here just so you're sure to keep it in mind. The homework testing stuff will check this.

# Next Two = Custom Invariants

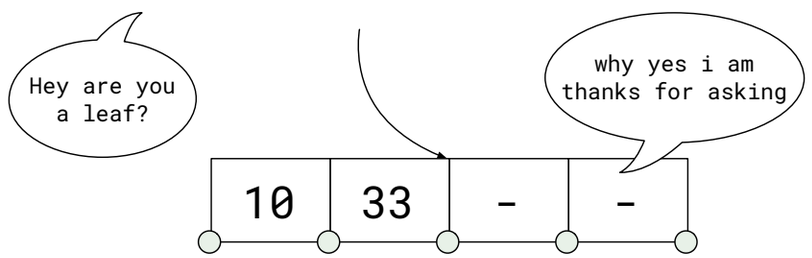
**The next two invariants are specific to our homework assignment.**

**A real B-Tree implementation would would likely not have these specific invariants, but they might have others.**

The next two invariants are specific to our homework assignment.

A real B-Tree implementation would would likely not have these specific invariants, but they might have others.

## (7*) A node's is_leaf property is true for leaves, false otherwise.

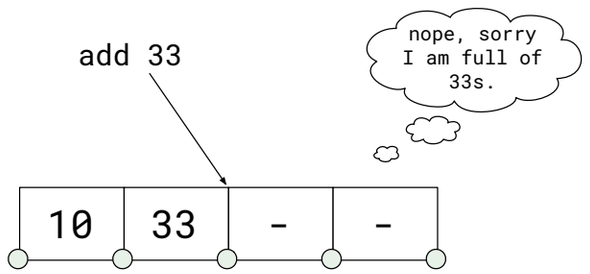



* this invariant is not a standard one;
it applies to the homework assignment.

A node's `is_leaf` property is true for leaves, false otherwise.

You'll see the b-tree node structure in the header file. Be aware that the root can be a leaf at the beginning, and whenever it becomes a non-leaf, be sure to change this flag.

## (8*) Keys are unique; inserting duplicates will silently fail.




* this invariant is not a standard one;
it applies to the homework assignment.

Keys are unique, and inserting duplicates will silently fail.

This is sometimes a real B-tree constraint, sometimes it isn't. It depends on the use case. And since our use case is homework, I thought I'd add this because it makes things easier to understand.

# B-Tree Invariants

**1.** Every node has at most m slots for children.
**2.** Every internal node has at least ceil(m/2) children.
**3.** The root node has at least two children if it is not also a leaf.
**4.** A non-leaf node that currently has k keys must have k+1 children.
**5.** All leaves appear in the same level.
**6.** The keys within a node must be sorted in ascending order.

For our homework assignment there are additional invariants:

**7.** A node's is_leaf property is true for leaves, false otherwise.
**8.** Keys are unique; inserting duplicates will silently fail.

I’ll just throw up this summary page here so they’re all in one spot. Get the slides, or make a screenshot so you have this list.

# Das Homework

**Test code validates your tree (and function) by checking all the invariants.**

**Implement your code however you like; the invariants just have to pass.**

The test code for the homework assignment will check to see if your B-Tree breaks any of these invariants. You can implement the functions however you'd like; all that matters is that they perform their function and that the B-Tree invariants are met after they complete.

Episode 6

# B-Tree: Find

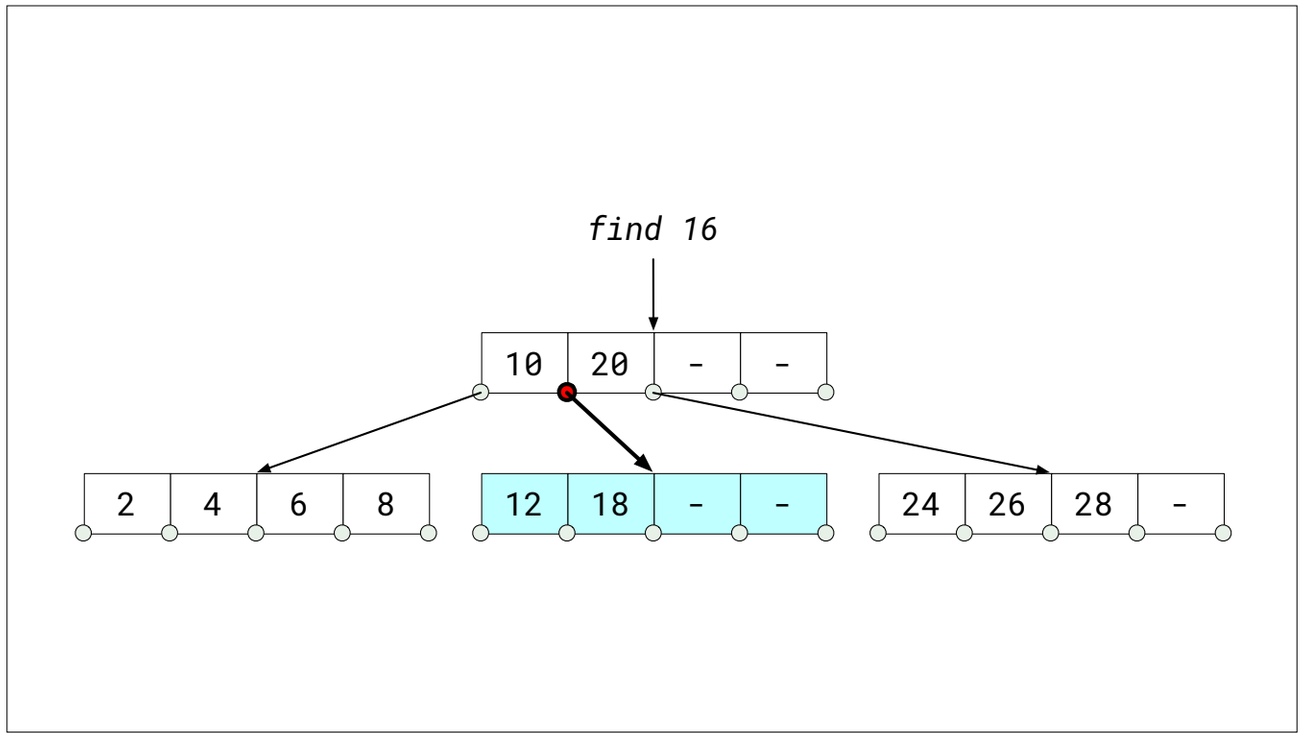
Now we're going to look at the B-Tree's find function, and the approach that people often take for implementing it.

# B-Tree Find Function

```
btree* find(btree*& root, int key);
```

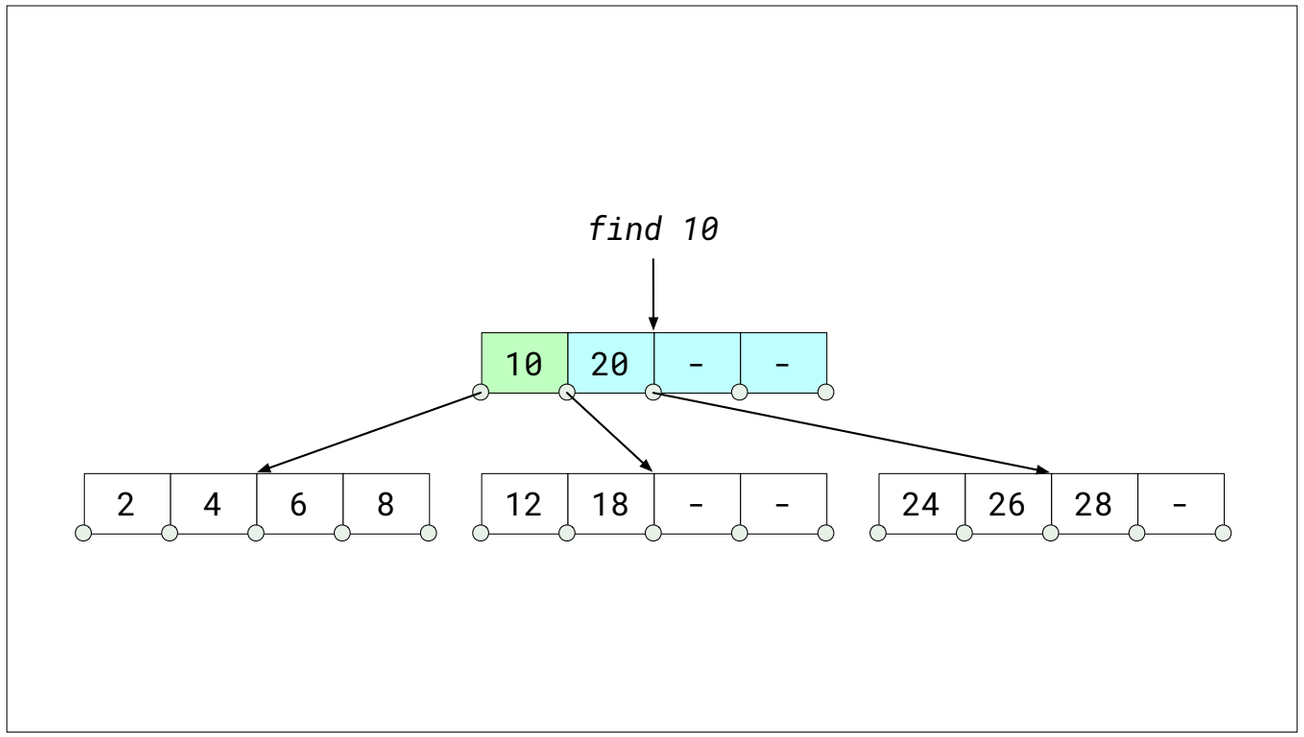
The find function has a signature like this.

It looks for the node that either contains this key, or would contain this key if we tried to insert it. Notice that we're using a pass-by-reference param, and it is a pointer. This means if we modify the value of root on the inside of our find function, that change is persistent outside of it as well.

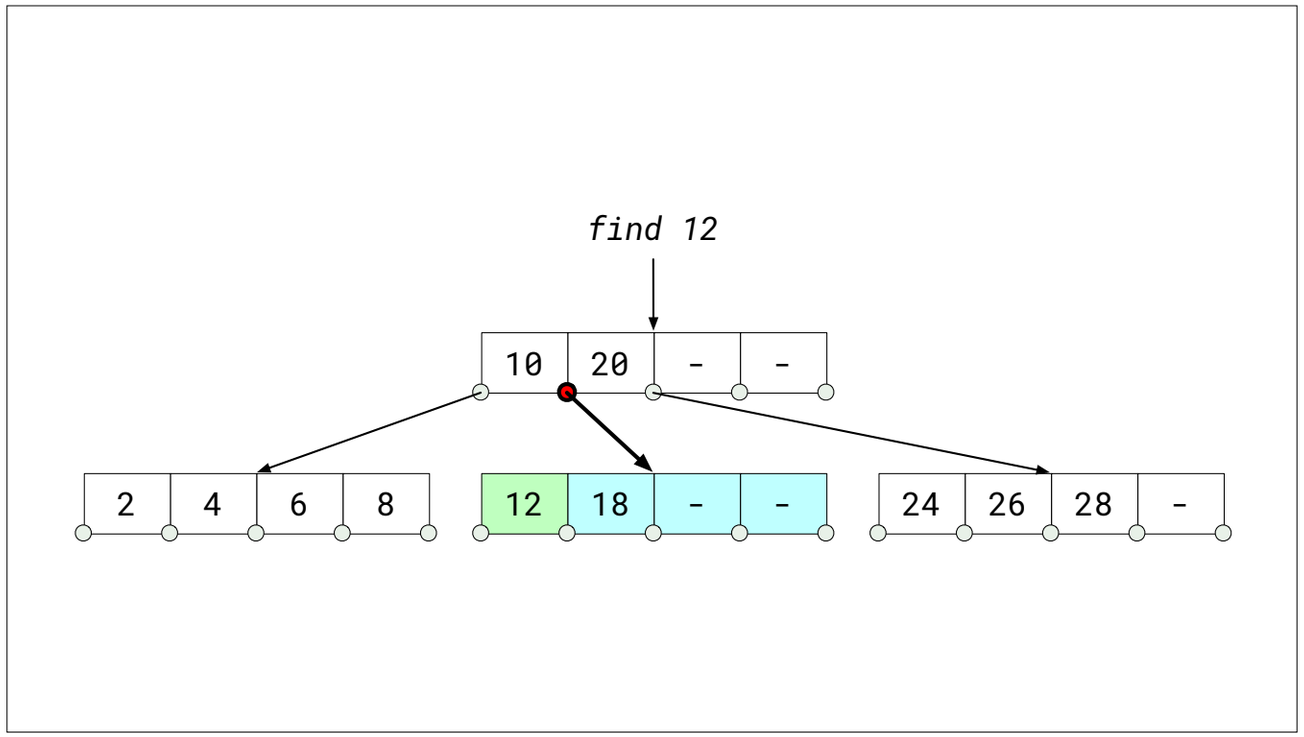



If we're looking for the key 16 in this tree, it first determines it isn't in the root node, then figures out which child to descend into. Well, the keys in the root are 10 and 20, and 16 is between those two, so we follow the child that's between 10 and 20.

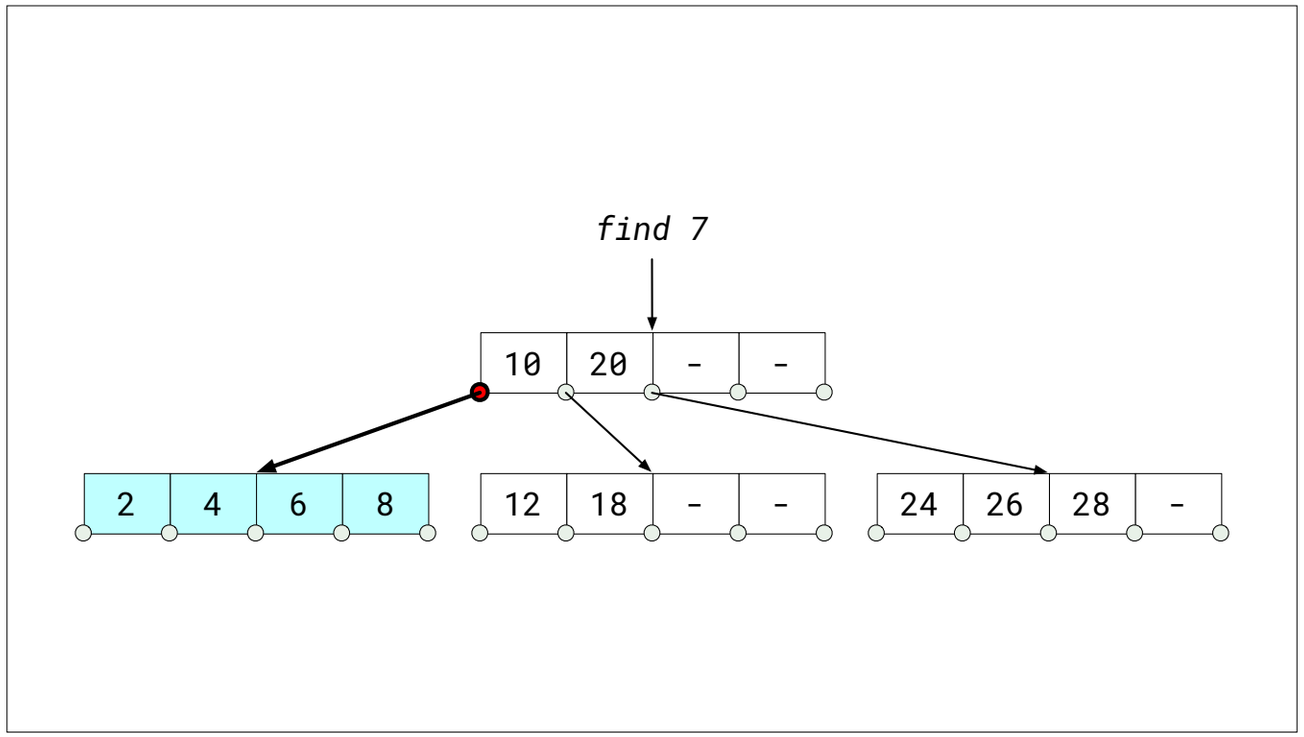
Then we get to the next node, and it is a leaf, so we just return it. This node doesn't happen to have that key, but that's just OK since we're just finding the node that either contains the key, or would if we wanted to insert it.




Now if we try to find a node with a key that's in the tree, we return it when we find the key. 10 is in the root, so here find will just return the root without diving into child links.




It works the same way for non-roots. 12 is in this leaf node, and this is the same return value as when we ran find with 16.




And a special caveat, if that 'would be container' node is full, that's OK. When we run the insert process, there's a process of moving things around and splitting/merging nodes.

Episode 7

# B-Tree: Demo

This episode is a live demo of how b-trees work, using this wonderful interactive thing that David Galles at the University of San Francisco made. There are a bunch of other interactive, visualization hacks up there for other data structures and algorithms, and I recommend checking those out!

**Live Demo: Interactive B-Tree Visualization**

https://www.cs.usfca.edu/~galles/visualization/BTree.html

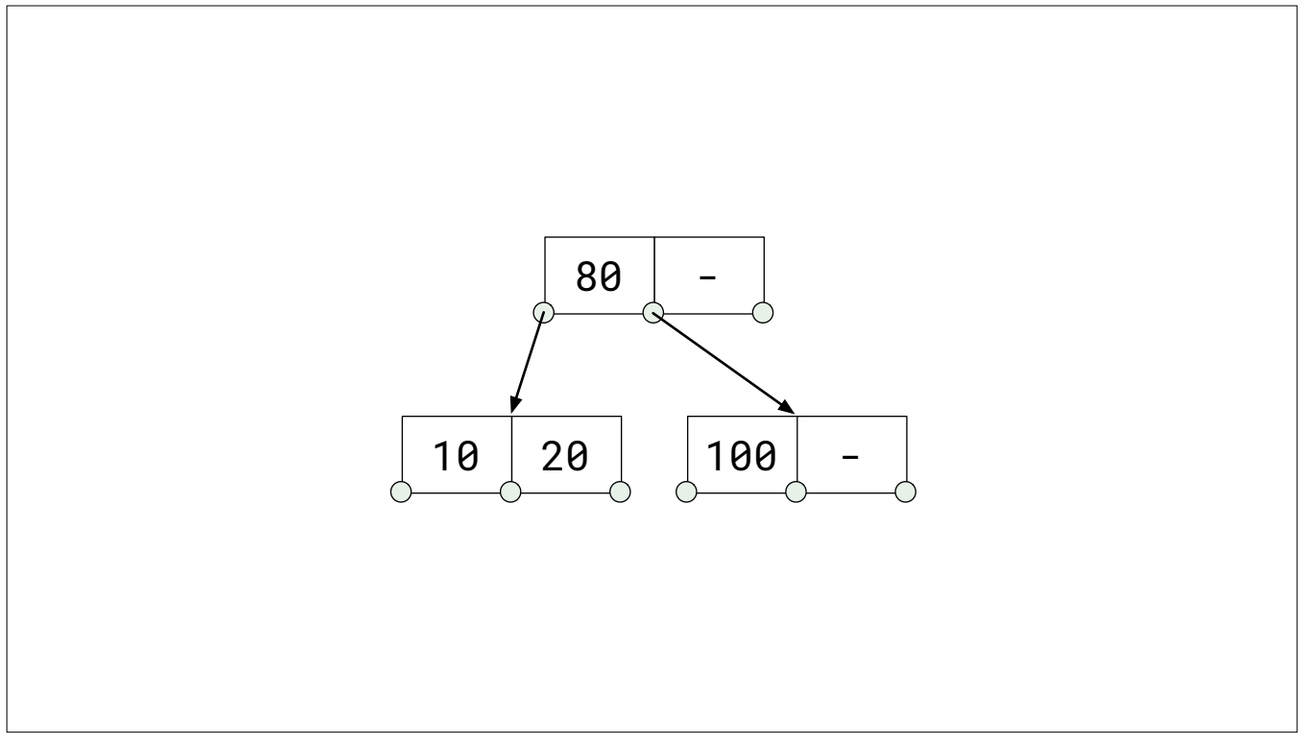
This is the URL. You'll probably want to just watch the demo first, and then play around with the visualizer. It should really help you when you design your strategy.

https://www.cs.usfca.edu/~galles/visualization/BTree.html

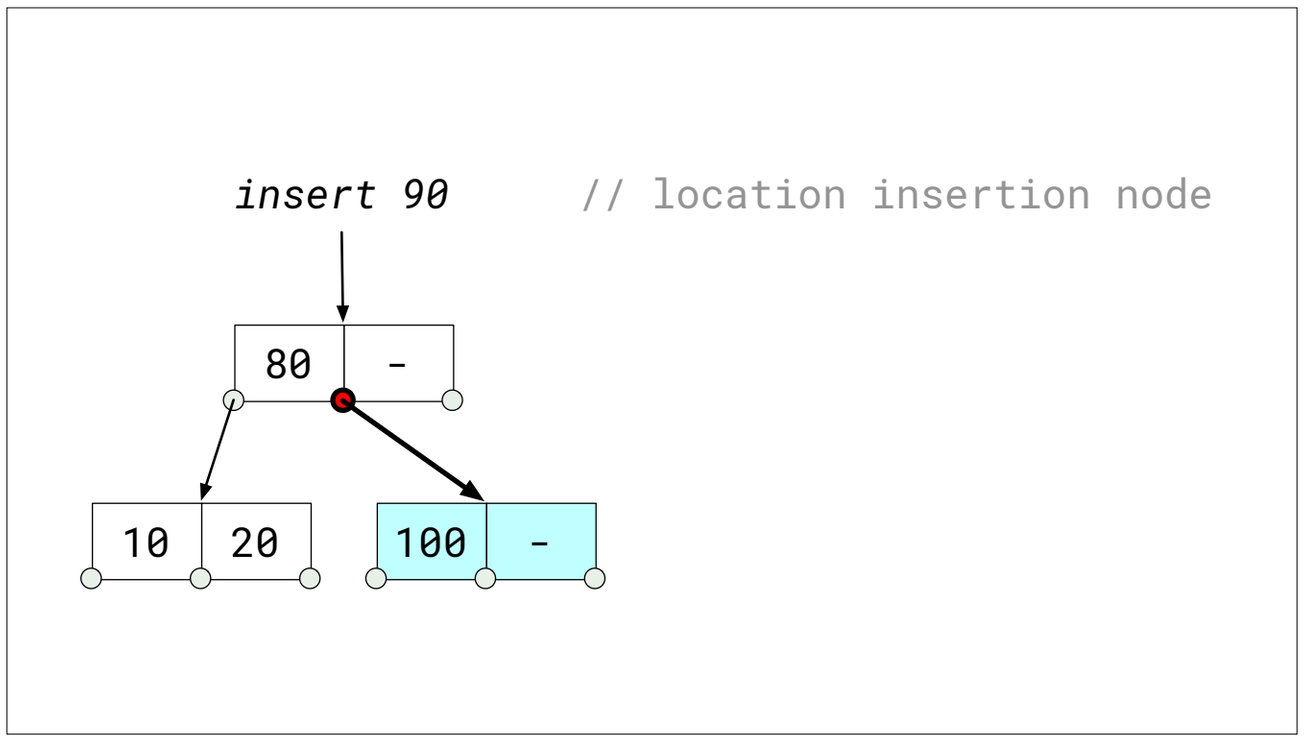
Episode 8

# B-Tree: Insert

This episode is another telling of the demo, basically. We're going to retrace the same steps but with hopefully more clarity with diagrams.

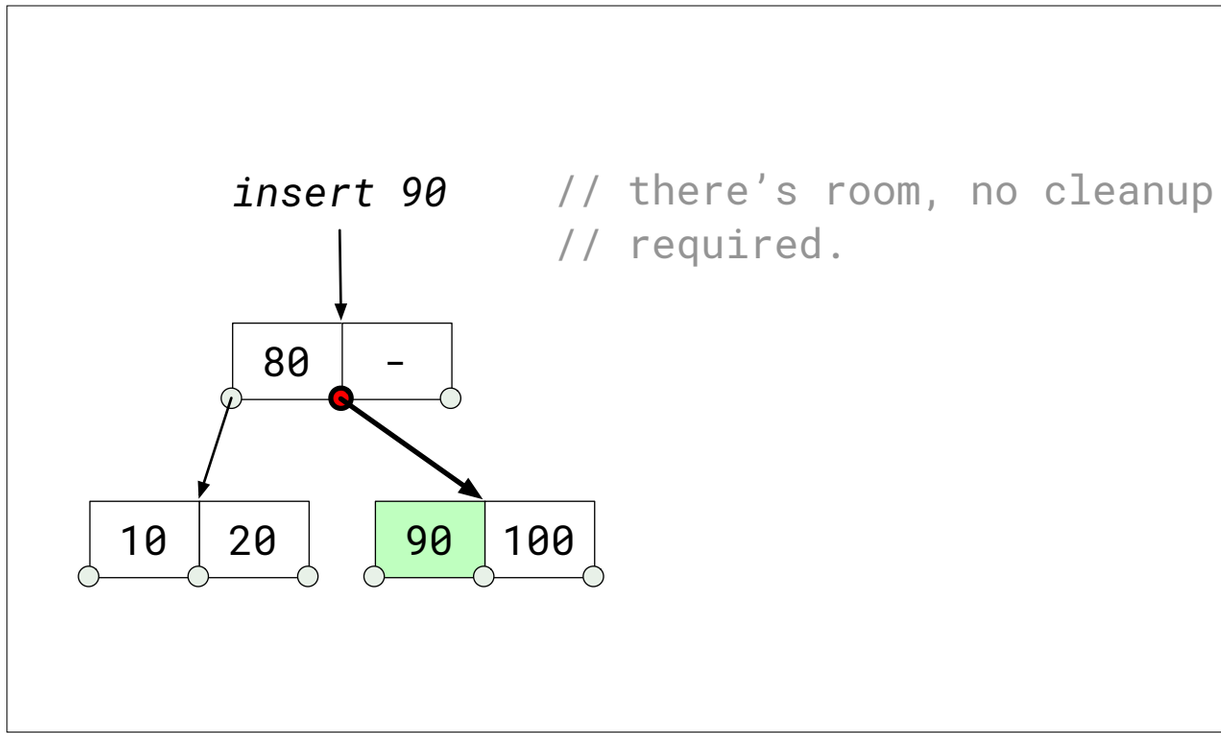



We start with this tree. Notice this is an order-3 b-tree. Three children, two keys max, one key minimum.

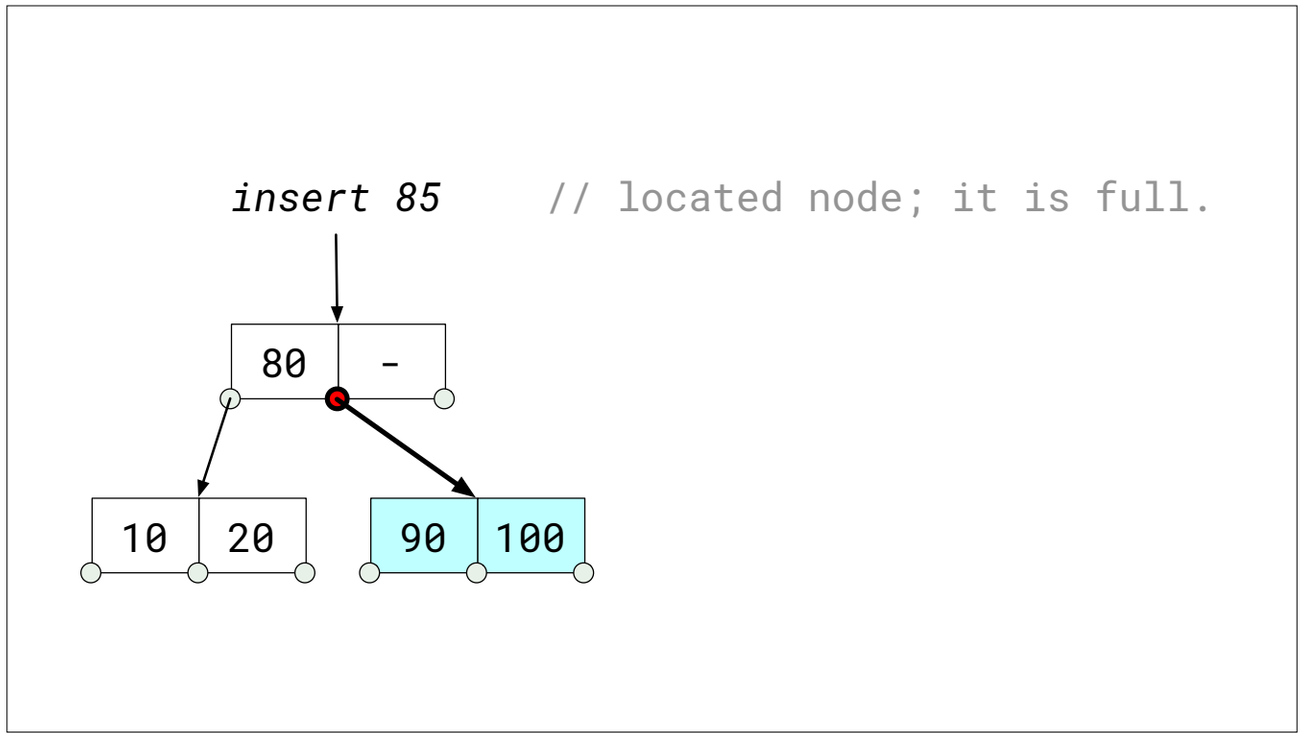



Let's insert 90. Finding the insertion node is one of the homework functions, by the way.

90 is larger than the only and last key, so we follow the last child link to bring us to that node with 100 in it.




We can add 90 to this node without over-filling it, so just put it there and call it a day.

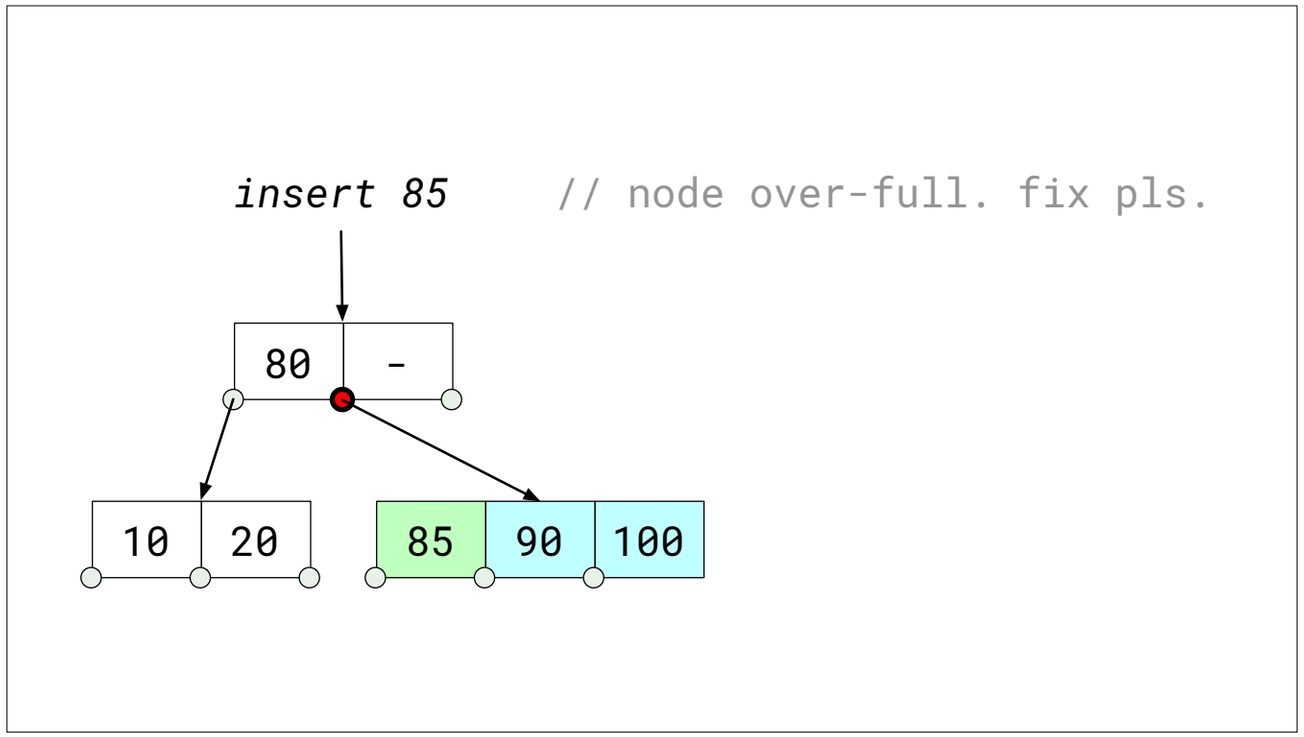



When we insert 85, we follow the links to the node where it would go, only to find that it is full. That's OK!

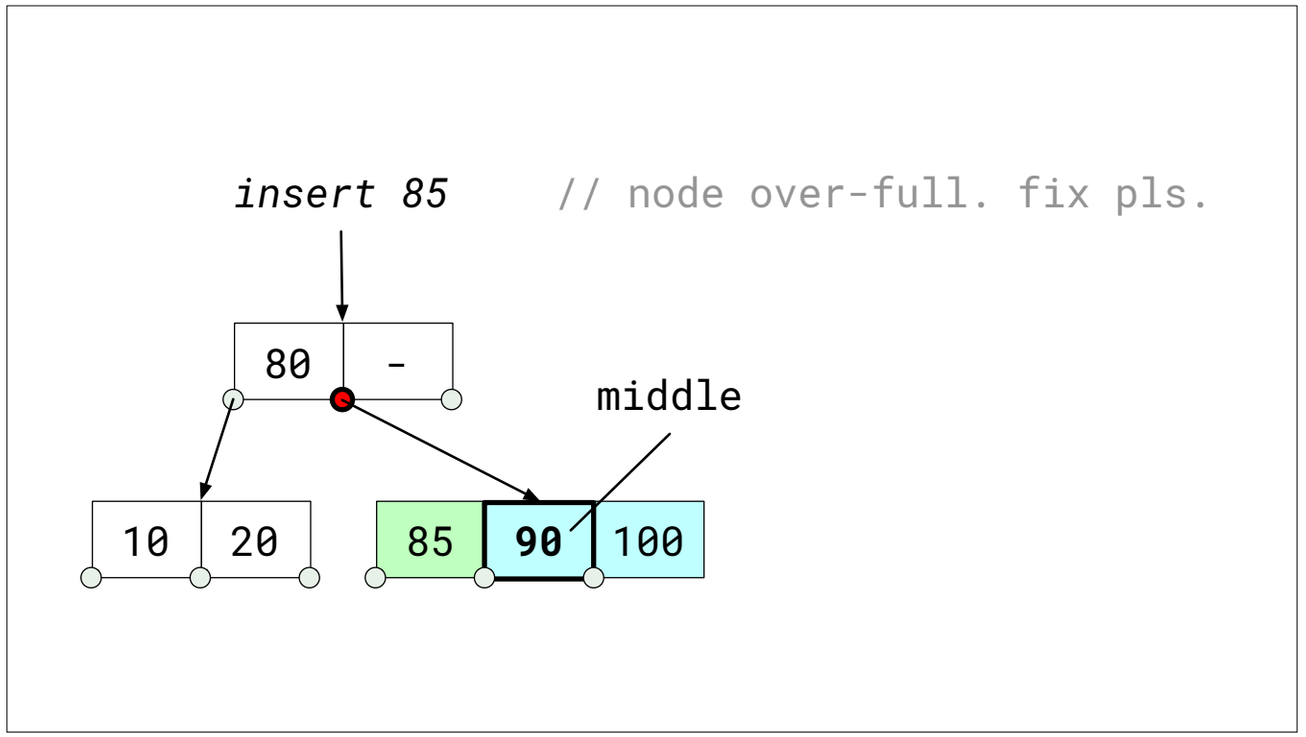
```
struct btree {
  bool is_leaf;
  int num_keys;
  int keys[BTREE_ORDER];
  btree* children[BTREE_ORDER + 1];
};
```

The node data structure we're using looks like this.

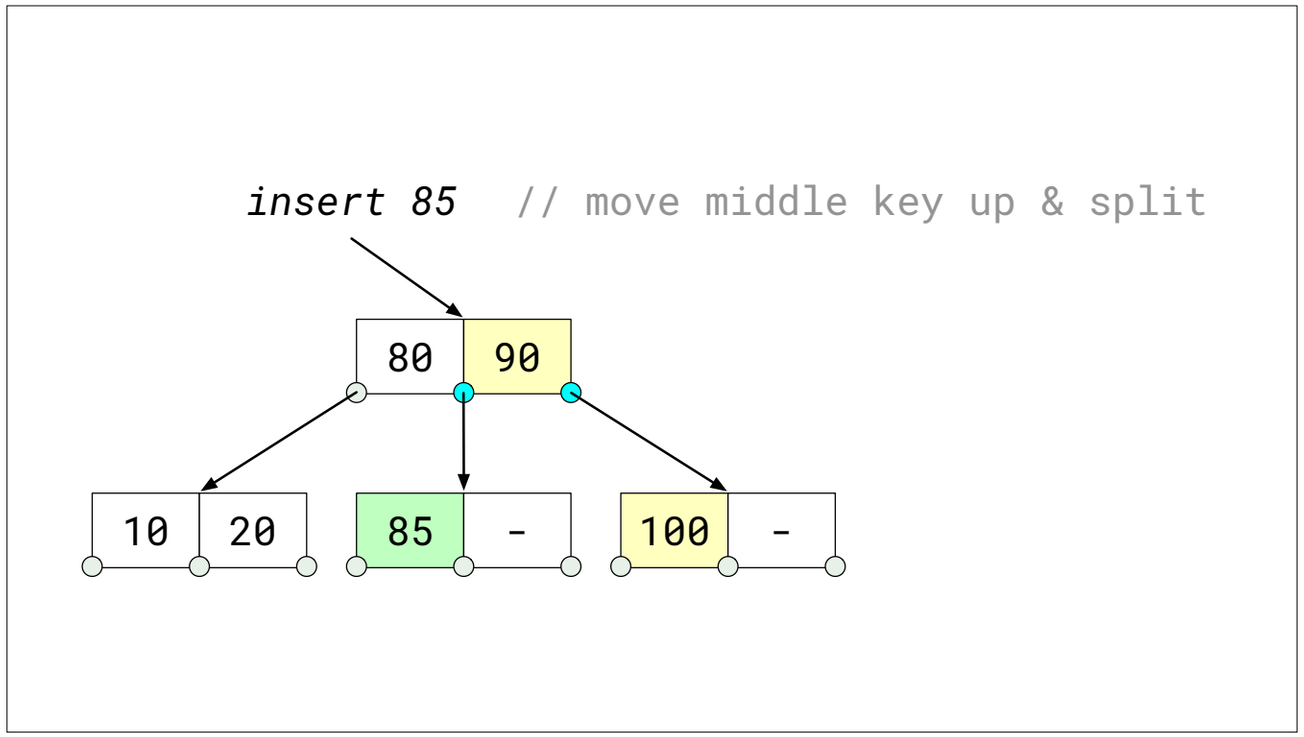
The keys and children arrays are both one bigger than they need to be to satisfy the invariant. Because of this, we can temporarily over-fill a node, and then immediately clean it up.




So over-fill the node by putting our new key in the right spot. Happens to be on the left here.



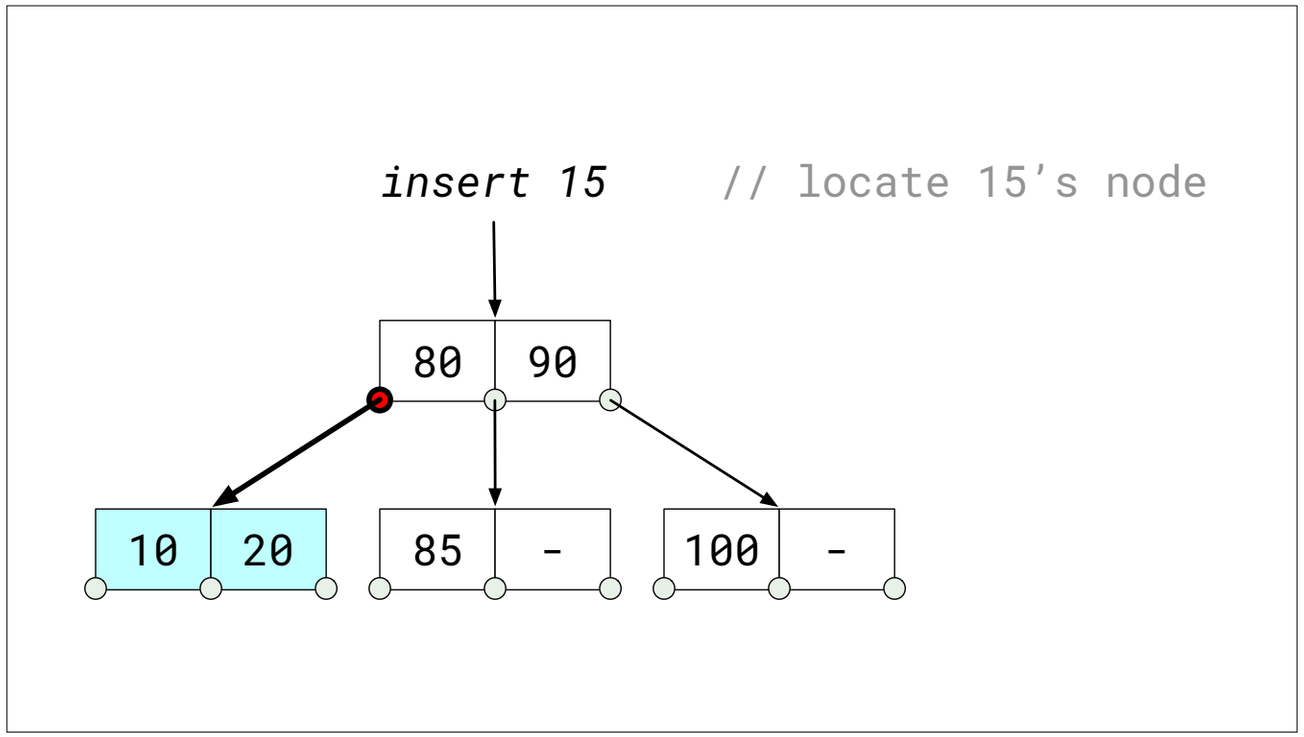

To clean up, we now have to identify the median node, the one in the middle, or if we've got an even number of keys, one of the two closest to the middle. Here it's going to be the 90.




The cleanup process involves splitting the node we are in. Keys less than our median go in a left node, and keys larger than the median go in the right node. The median itself moves up to the parent.

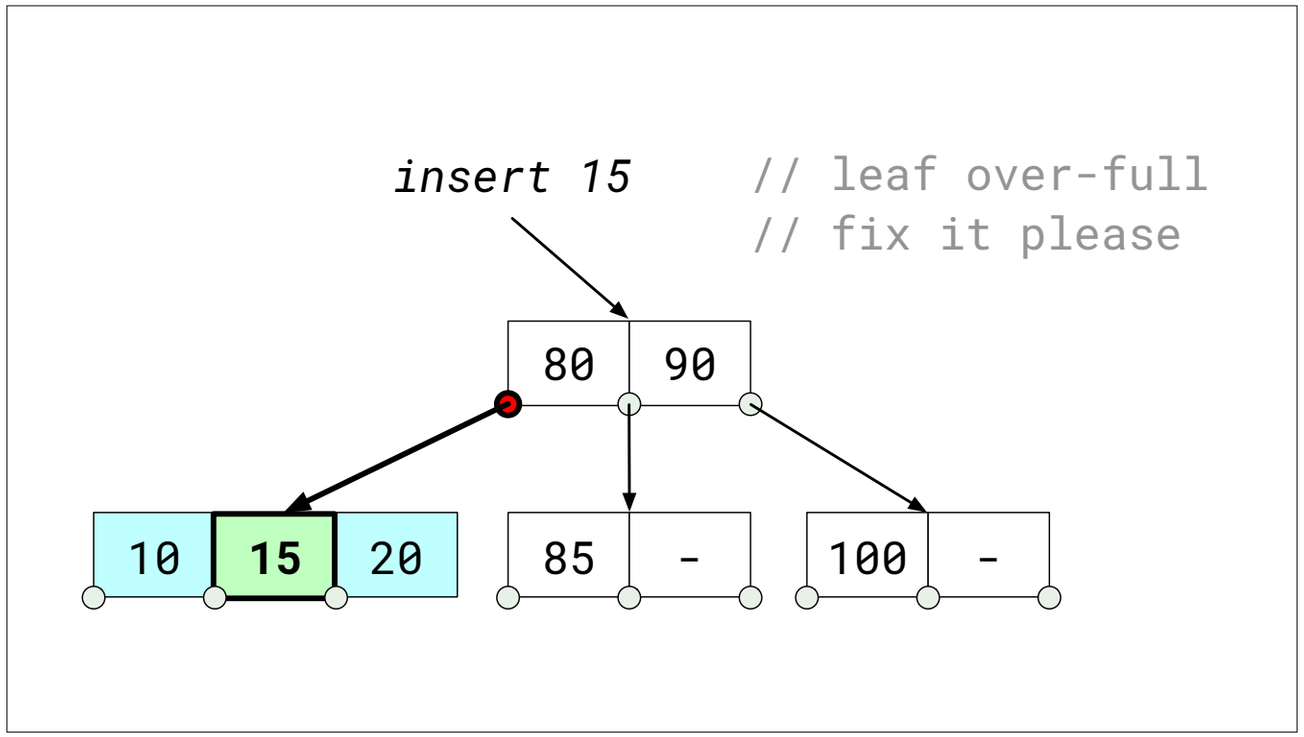
Because we have two nodes, we need two child links from the parent. And since we've given the parent a new key, our child links will just sit on either side of it. Really, that 90, the median we gave it, serves as a partition between the two halves of our split node.

And so now our b-tree invariants are all OK, and the 85 is inserted, so we're done.

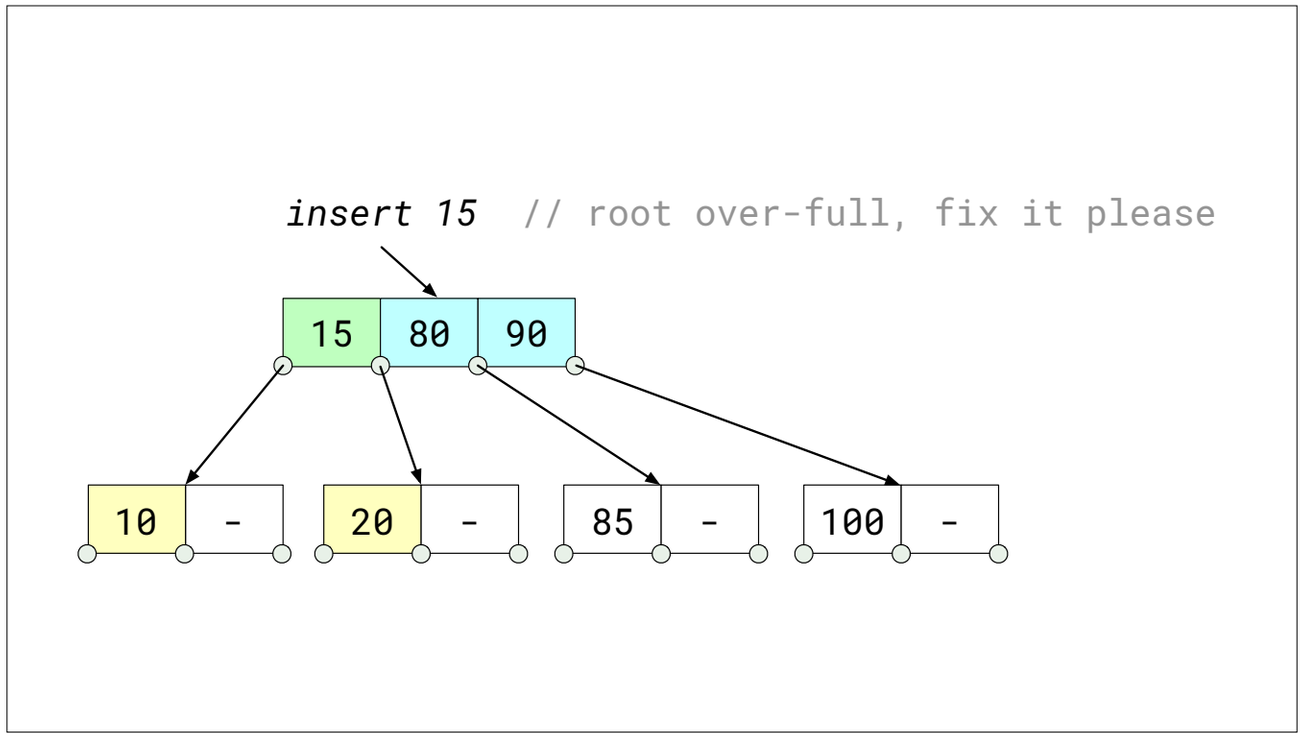



Let's insert 15, shall we? Do you feel like you've seen this one before?

OK, locate the node.




It is overfull, so we fix it just like we did in the last insertion. Identify the middle key, which is 15.

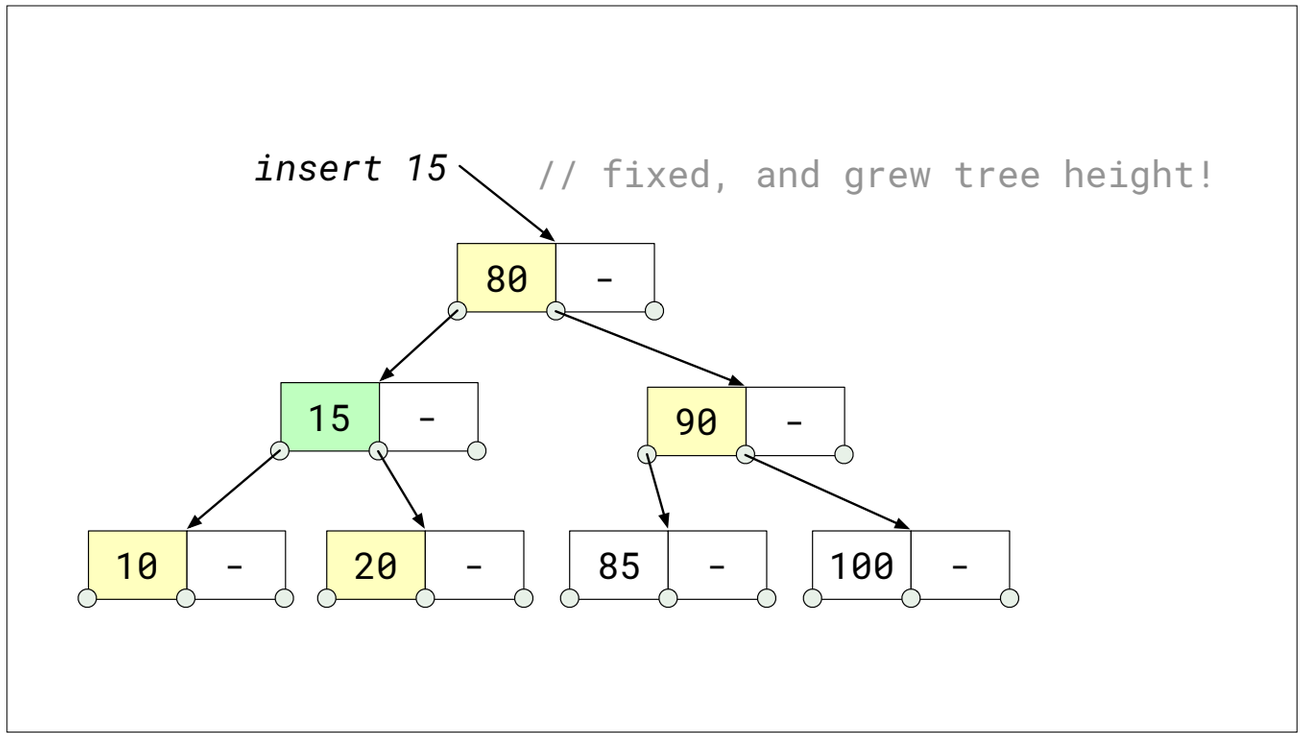



We split our overfull node, moving the median up to the root.

But oh no! Doing that causes the parent node to be over-full.

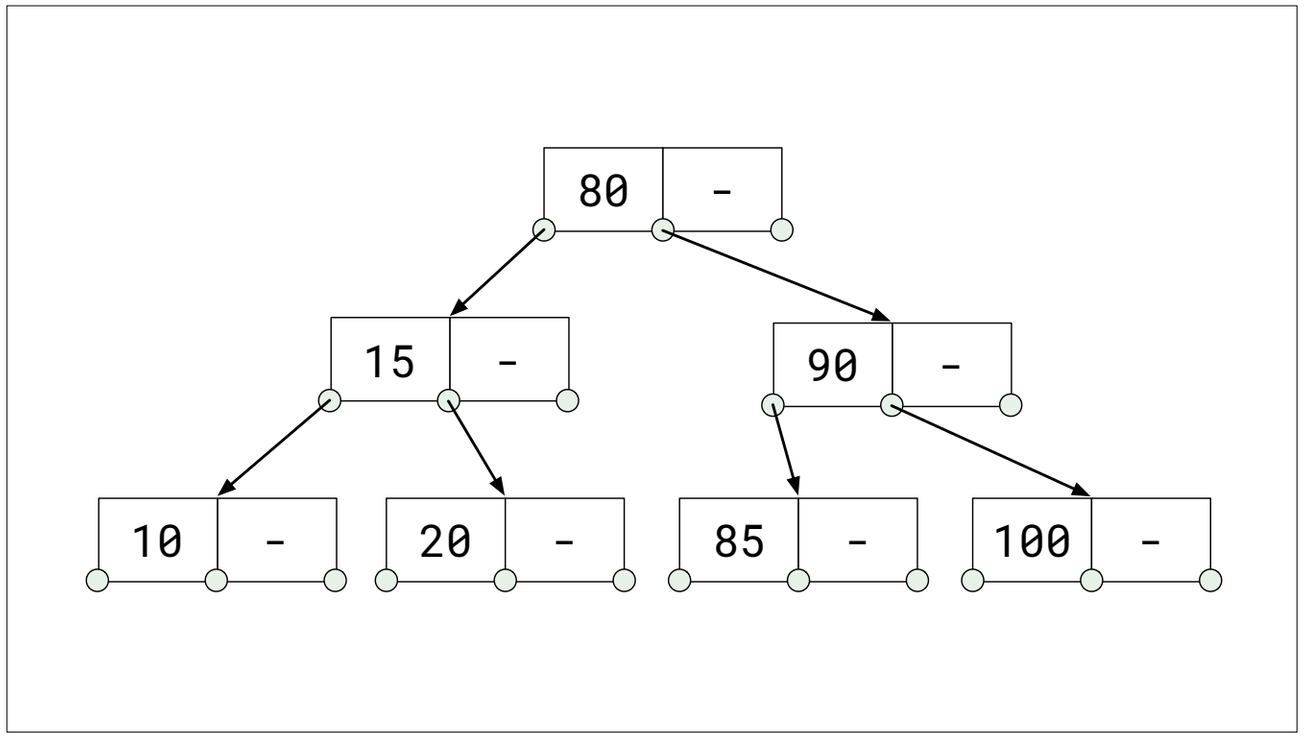
Happily, we already have the code for splitting and sending a median key upwards.

The only difference here is that there isn't a parent node, since it is the root that is bursting at the seams.




It's cool, we just make a new node, and put the median in it.

This is why the invariants treat the root node somewhat differently. It is OK for the root node to only have one key; it just needs to have two children.




And so, that was the same example from our little demo in the last episode. Next up is the remove function.

Episode 9

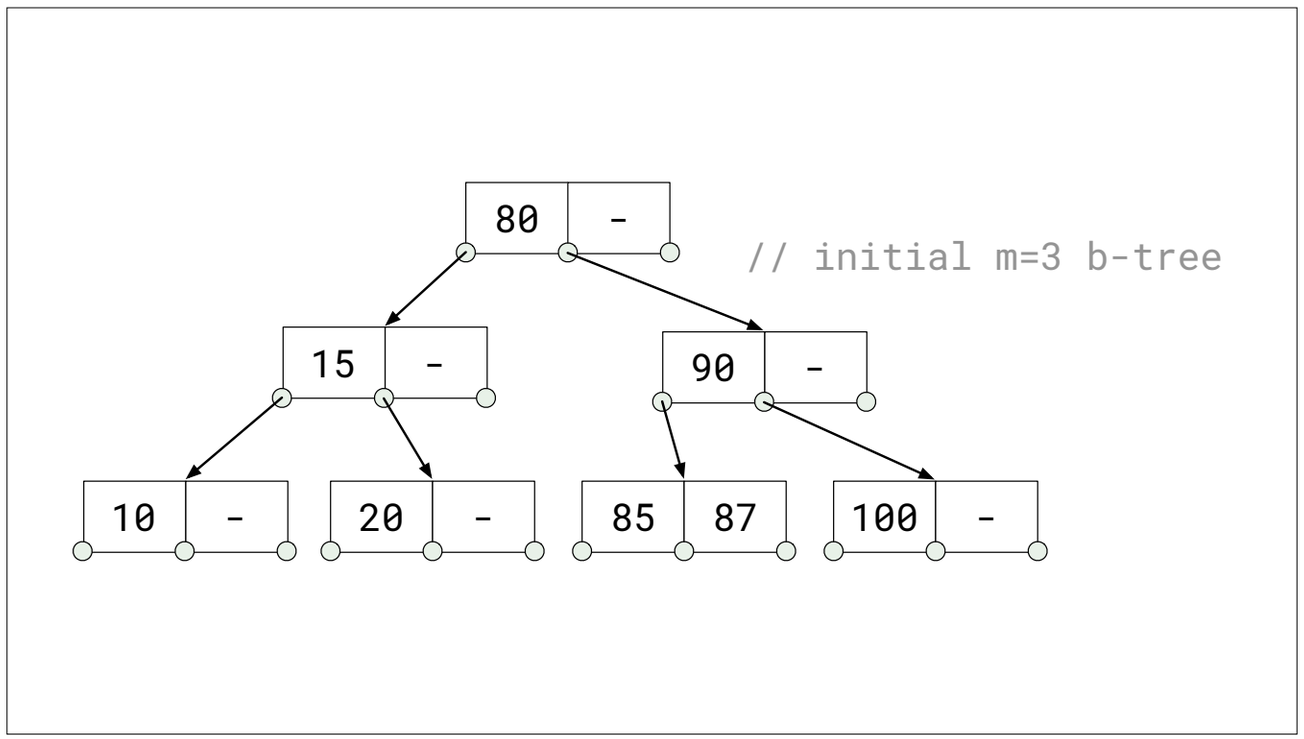
# B-Tree: Remove

Now we're going to go over the B-Tree remove operation. Like last time there's no pseudocode, since the whole idea is for you to figure out the specific steps, and where there might be helper functions to write and re-use. So, this is basically a bunch of diagrams showing what the process might look like.

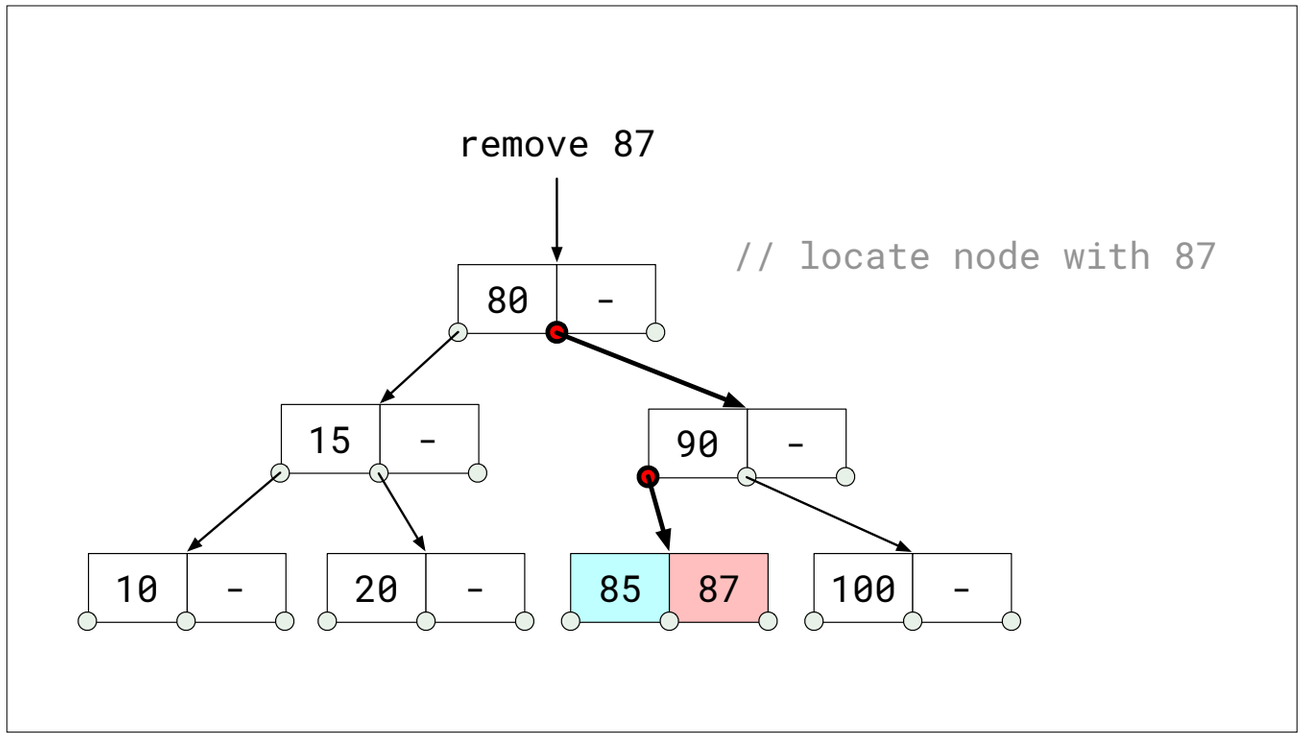
And I should point out that the invariants are all that matters. You can implement these functions however you'd like. You just need to make sure that keys are inserted and removed, and when your functions exit, the B-Tree's invariants are all OK.

# Remove: Easy Mode

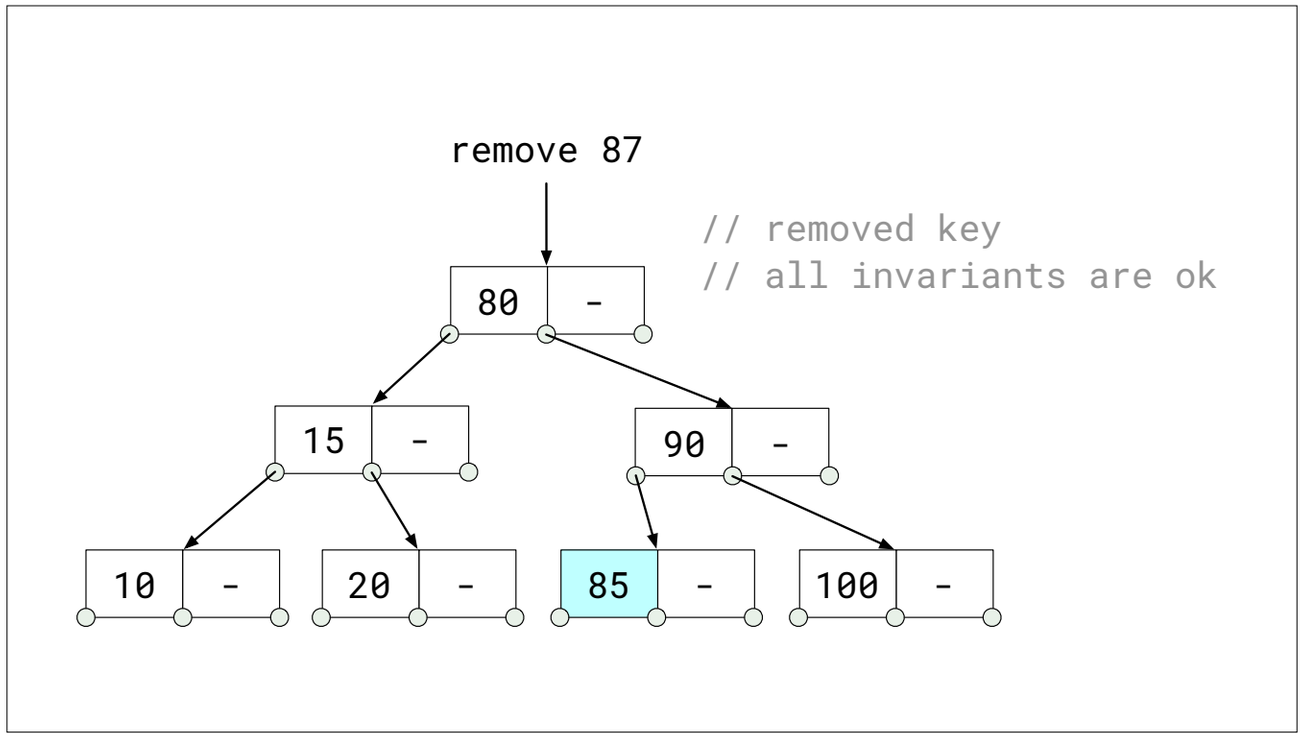
Starting with an easy removal. And by the way I'm trying to stick to the word 'remove' since 'delete' has a specific C++ meaning, which is freeing memory. You should remember to use _that_ too, when it is appropriate. Hint hint!




This is the initial B-tree, it is of order 3, meaning nodes can have one or two keys.



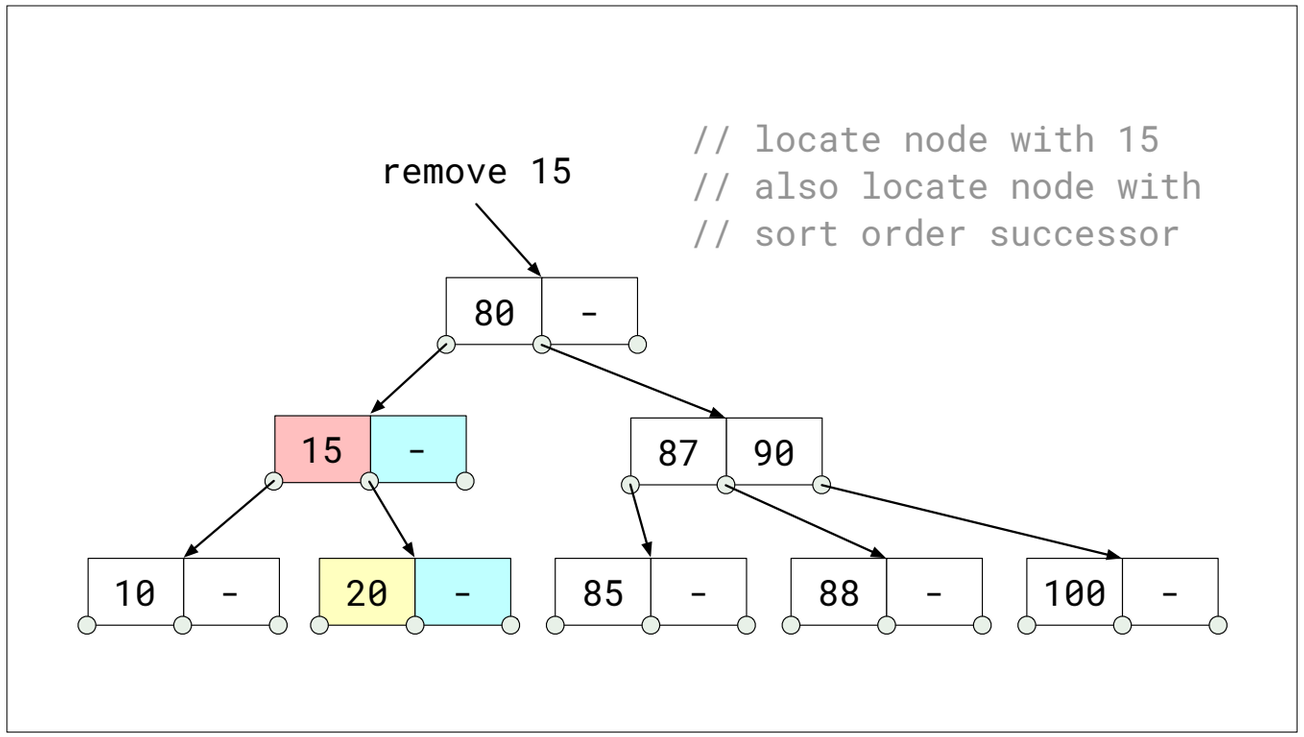

We're going to remove the key 87 from this tree. So first step is to locate the node that contains the key, if it is there at all.




Like I said, this is easy-mode. We could just remove 87 from that node, and all the B-Tree invariants check out. So, we're done.

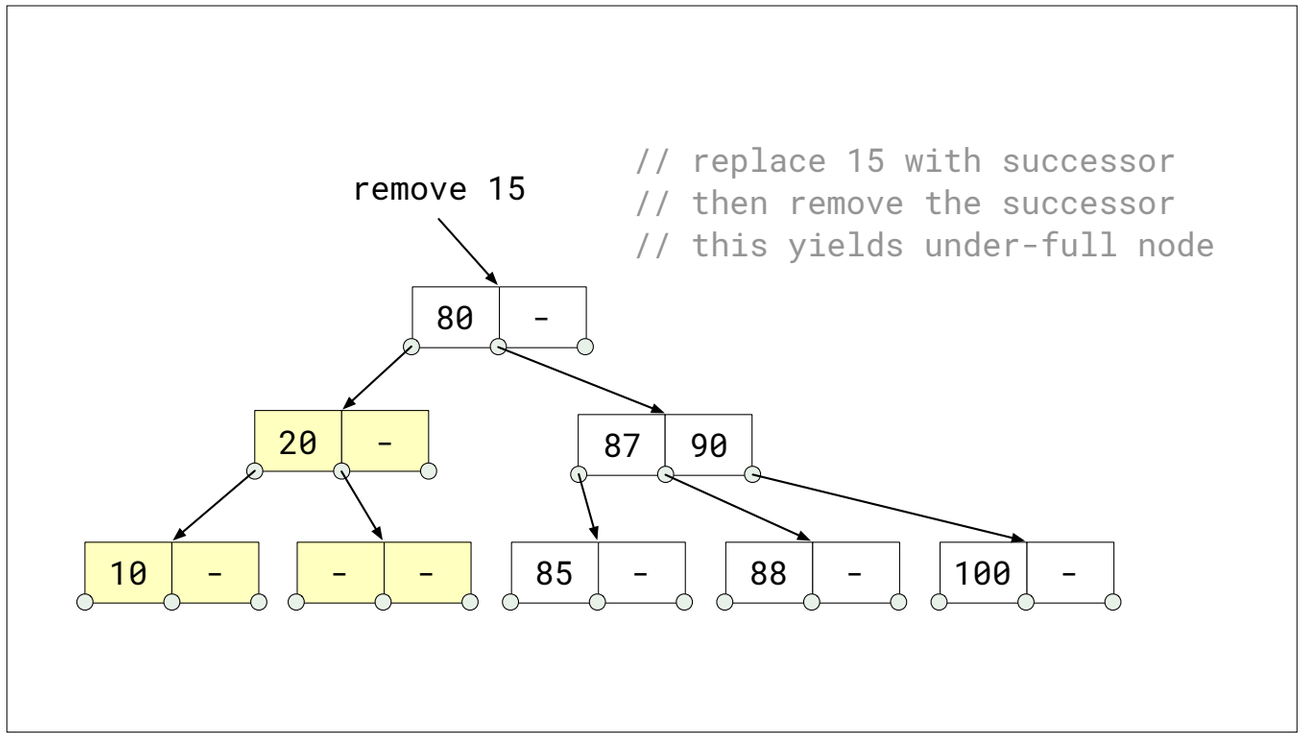
# Remove: Internal Key

Now let's try a little trickier remove process. Let's remove key 15.




When we locate the node with the key to remove, we should notice that it is an internal node. So we'll have to shuffle things around, since that key is currently used to partition the search space between its children.

Locate the sort order successor key, and the node that contains it.




Just like binary search tree removal, we use a cheap trick, which is to just swap in the successor key into the spot where the removed key was, and remove the successor key from the node it was in. This leaves us with broken invariants which we can then clean up.

We could also use the predecessor.

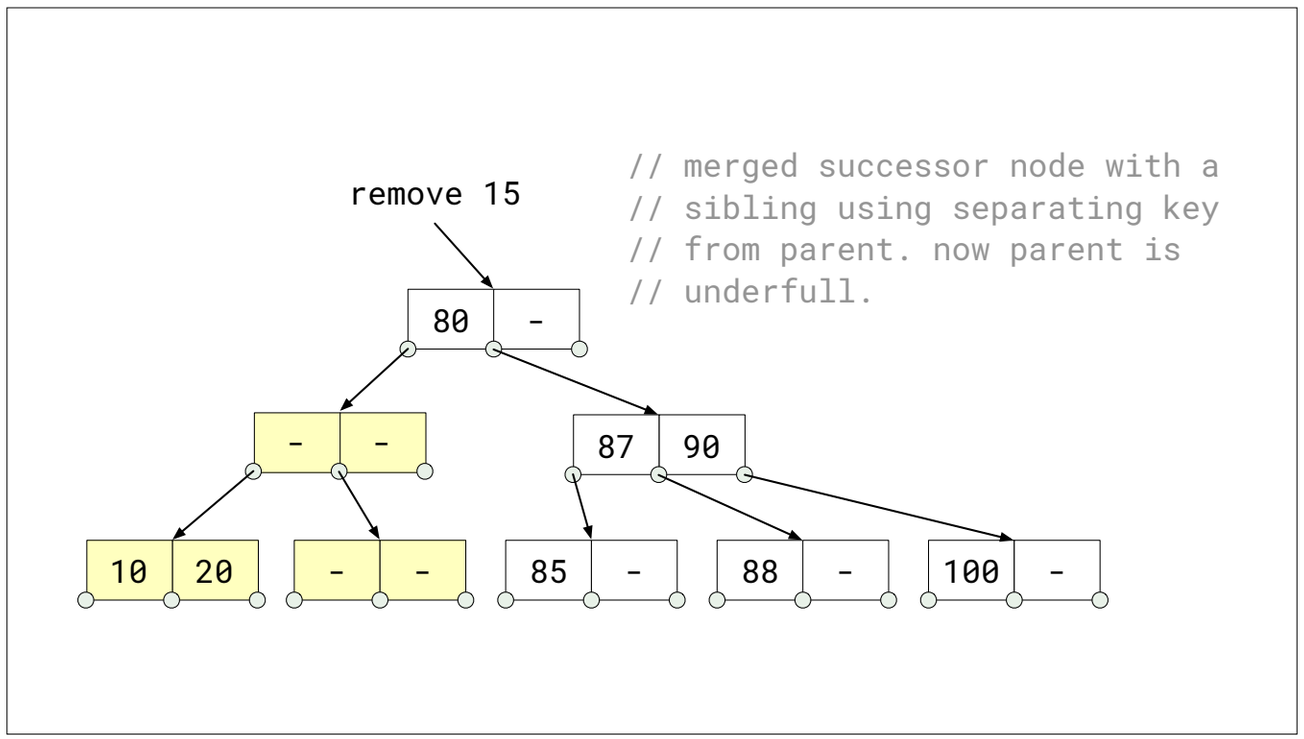
That lower right yellow node is the one that is under-full, because as you can see it has zero keys now, and that breaks the one-key minimum.

So we have to fix the broken invariant, but there's more than one way to do this.

One way is to merge our under-filled node with one of its siblings. It only has the one sibling, the one with the 10 in it.

Do this merging process in situations where the siblings are at their minimum capacity.

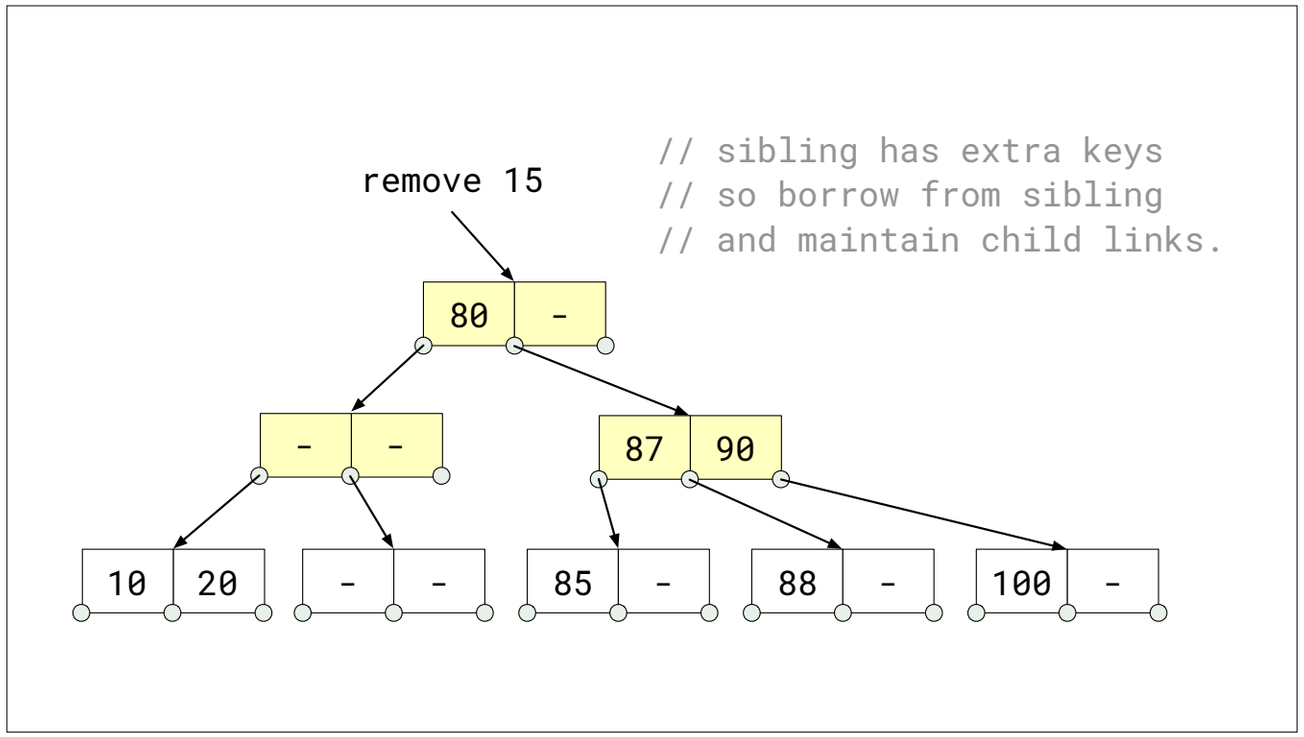
Remember the sort order of our keys has to be maintained. This means when we merge siblings, we have to involve the key from the parent that partitions their key values. So take all the keys in the left sibling, then the partitioning key from the parent, and all the keys from the right sibling. That leaves us with something that looks like this...




OK. It looks like we're making a hot mess here. But I promise this is actually progress!

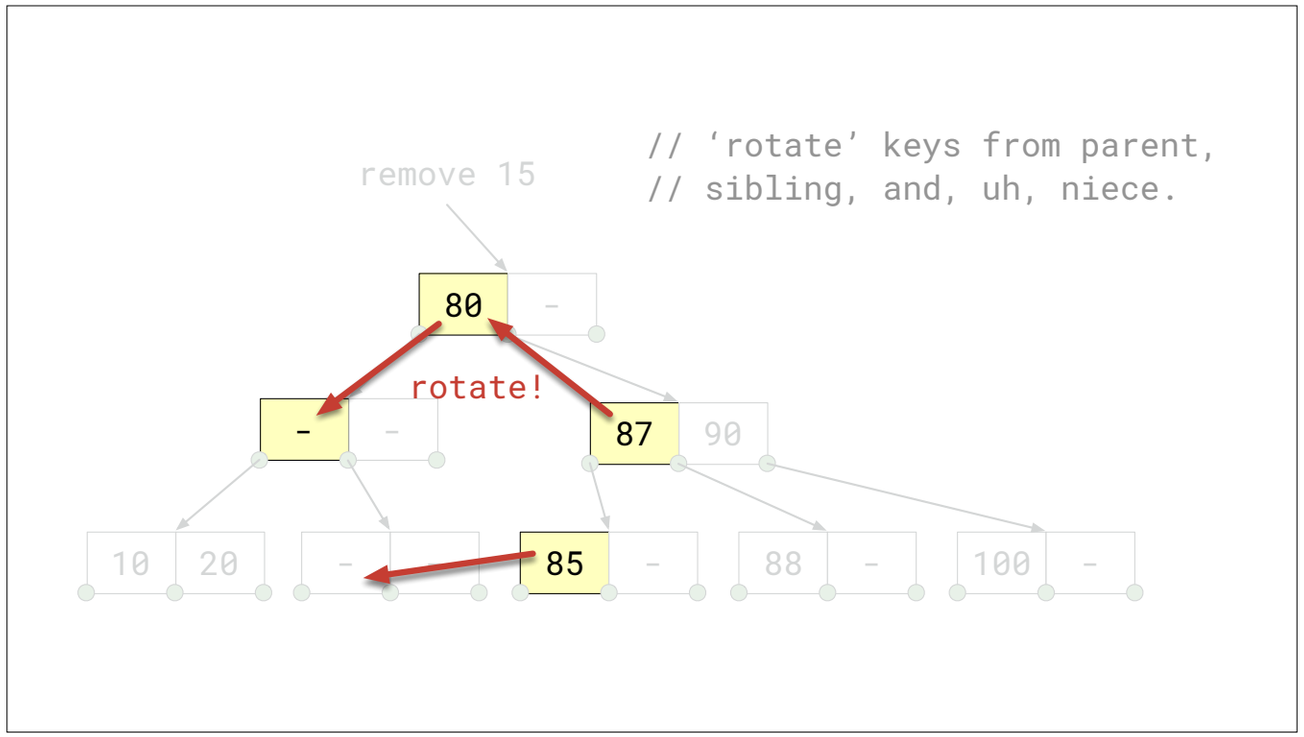
The parent node is now under-full. And we just solved the problem of under-full nodes, but at the leaf level.

Now we have an under-full node in an internal layer, and we can actually use the same process to fix this.



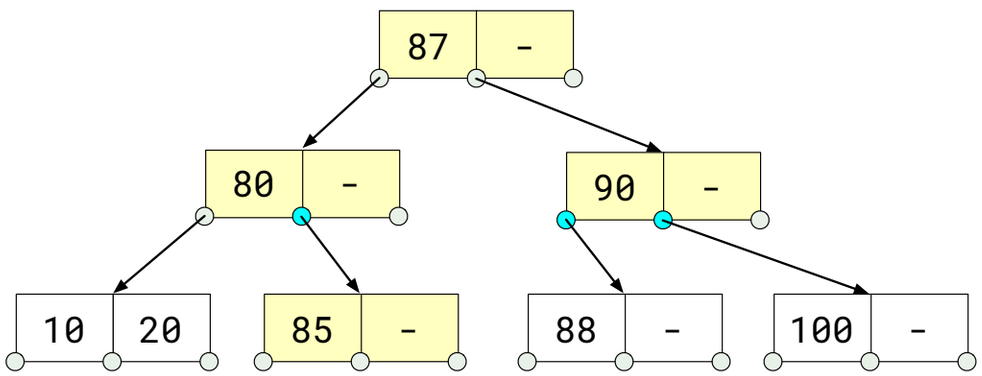

Highlighted are the under-full node, its sibling, and the parent that contains the partitioning key value between them. The last time we handled an under-full node, we did a merging process because our under-full node's sibling was at minimum capacity.

But this time, our sibling has an extra key. So rather than merging, we perform a rotation.




Here's what the rotation looks like. You pull a key from the sibling up to the parent, pull the partitioning key from the parent down into our under-full node. And then the child of our sibling, whatever you want to call that, gets adopted by the under-full node. The intuition here is that we need to maintain key sort order, and end up with a B-Tree without broken invariants.

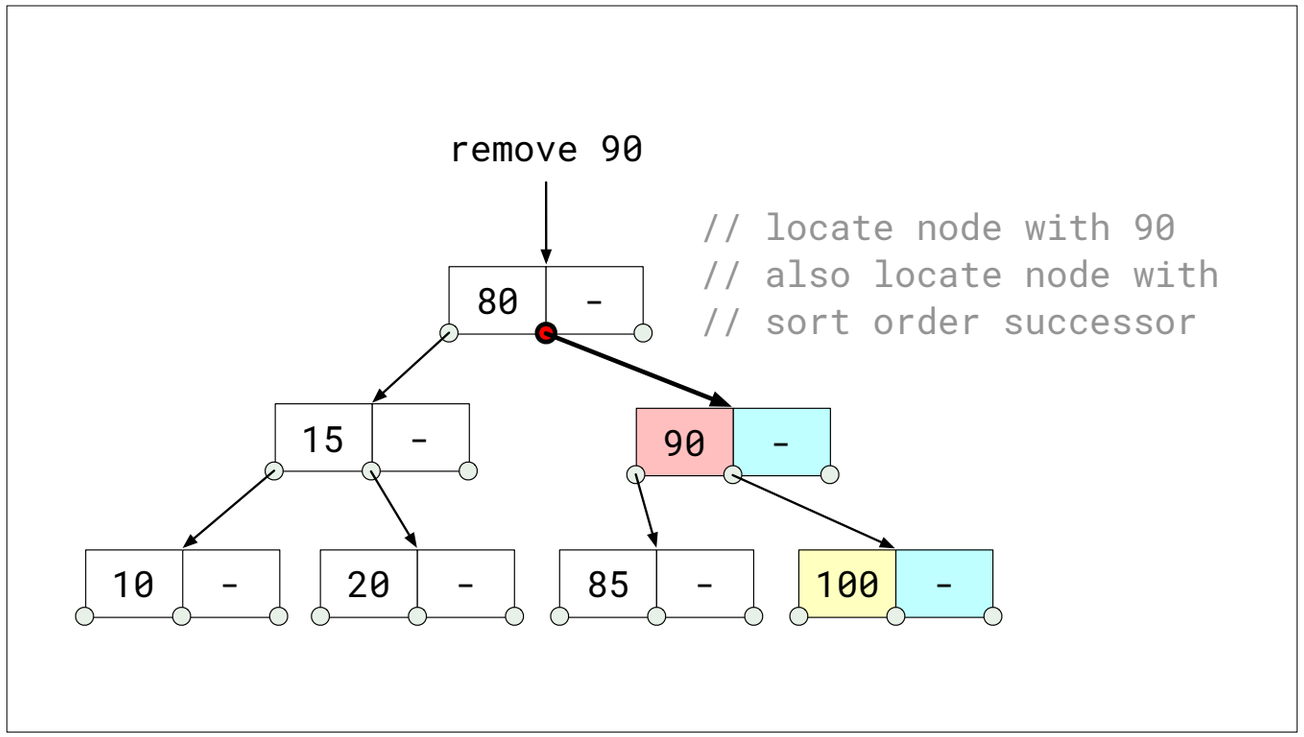
```
// target key 15 removed
// and invariants all ok
```




And here's the tree with the 15 removed. I've left the nodes and child links that we affected by this last step highlighted.

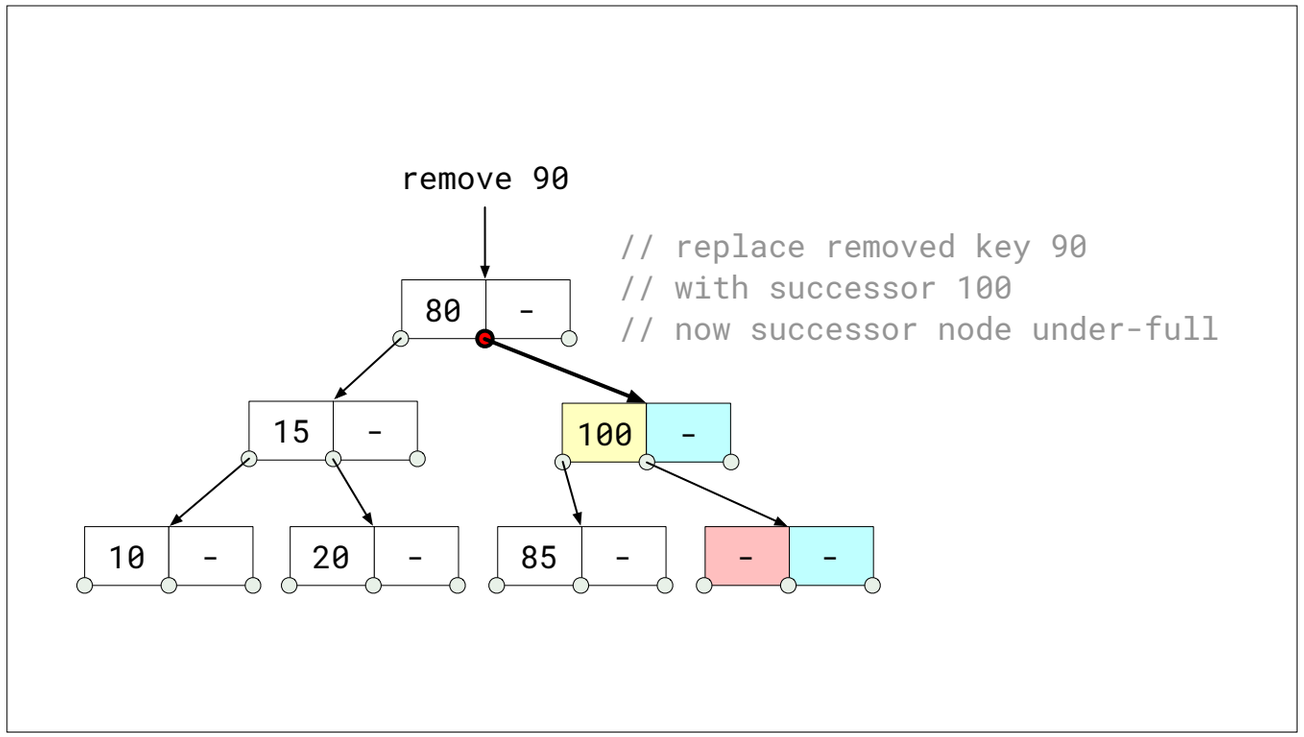
# Remove: Hard Mode

Allright! Now for hard mode!

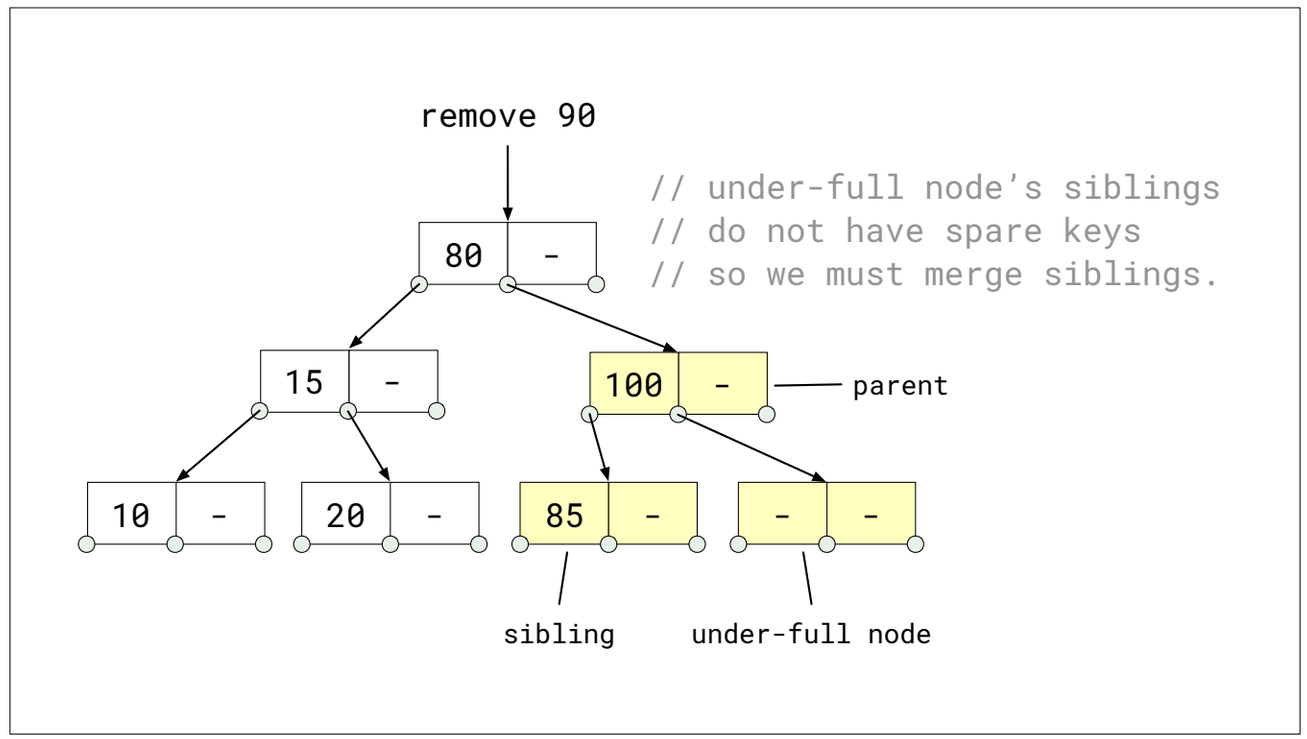



This is the tree we're doing surgery on now. Notice all nodes in this tree are at minimum capacity to start with, so whatever we remove, we'll probably be entering a world of pain.

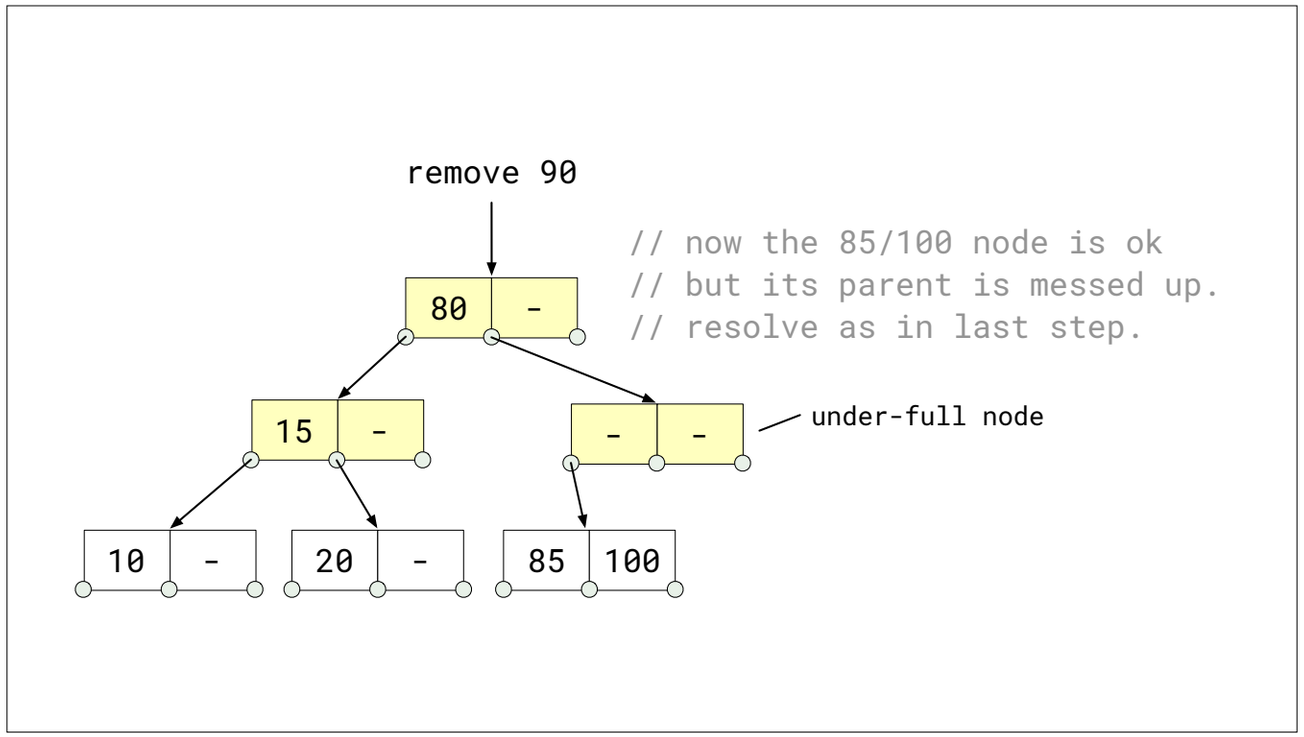
So like last time, find the node with the key we're removing, and also the node that contains its sort order successor.




We do the same trick as with binary search trees, moving the successor up to replace the doomed key. This results in an under-full node there on the bottom right.

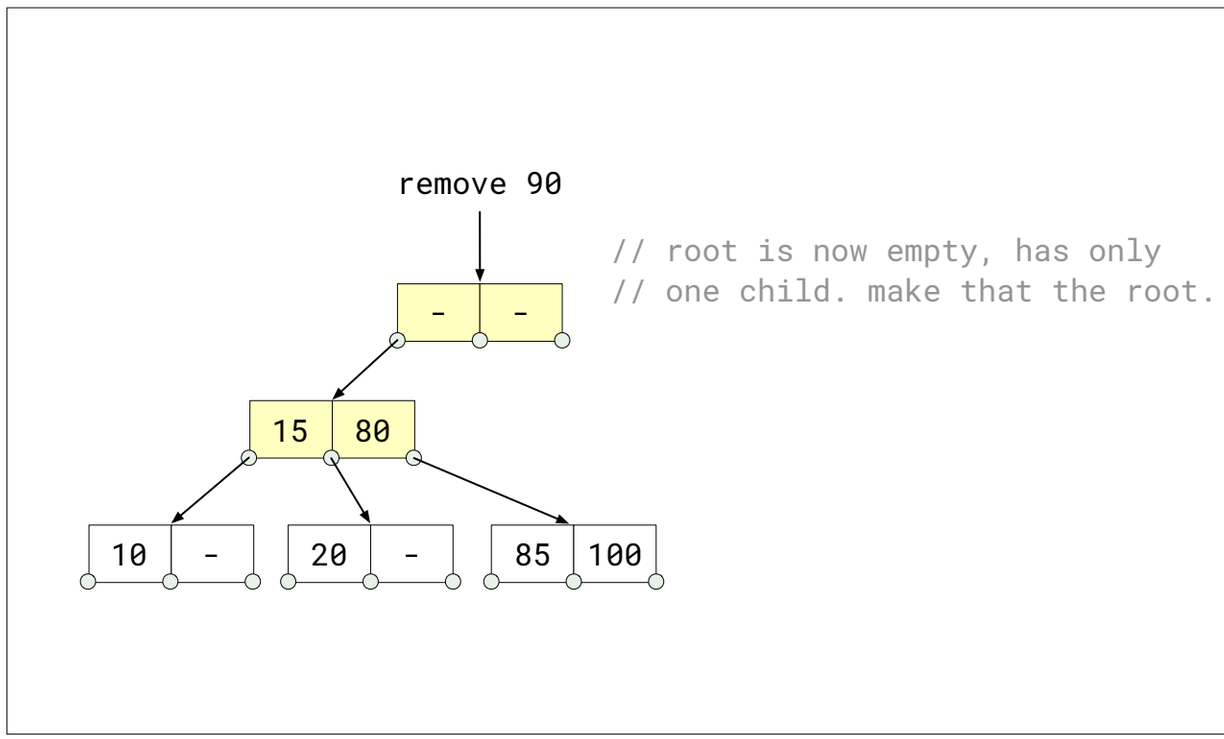



But the under-full node's sibling doesn't have any spare keys. We saw this in the last case, so we know what to do here. We have to merge nodes using the partitioning key from the parent.



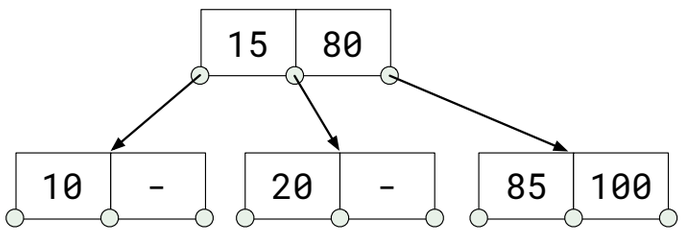

Once we do that, there's another under-full node, on the right there. Again, we've seen this before, so we merge with a sibling using the separating key from the parent.

So the new node will have 15, 80, and that's it since there are no keys in the under-full node.




Now the root node is empty. We can't have that, since there's an invariant that the root node has at least two children. So, we just update the root pointer so it points at the 15/80 node.

```
// after removing 90. all
// invariants are met.
```




Now that pesky 90 is gone, and the B-Tree's invariants are all met.