In this video, we'll build a simple B-tree data structure in Haskell, using a GADT to ensure that we maintain the B-tree's structural invariant. As in my previous video on Hole-driven Haskell, we'll use types to guide the development.

Although B-trees come in various sizes, I'll be restricting our attention to the smallest possible B-tree. I'll refer to this variant as a 2-3 B-tree, though others might use a different name for the same structure.

We can define a 2-3 B-tree informally by three rules. Firstly, a 2-3 B-tree comprises either a node with one element and two sub-trees, or a node with 2 elements and 3 sub-trees, or a leaf containing nothing at all. Secondly, every leaf must be equidistant from the root of a tree. And finally, data must be ordered left to right.

So what's the motivation for these seemingly arbitrary rules? Well, the third rule means that we can search for an item within the tree, without back-tracking. Left-to-right ordering allows us to narrow the search to at most one sub-tree, using just one or two comparisons with the elements in a node. The first and second rules together mean that each narrowing is by a factor between one half and one third. As a result, search takes log n time in the worst case. Finally, the first rule also gives us just enough flexibility to implement some useful operations, including insertion and deletion, also in log n time.

Now, to represent a 2-3 B-tree in Haskell, we can make a first approximation using a pair of mutually recursive data types that capture just the first rule. The type **N** describes the structure of internal nodes, while **T** describes the overall recursive structure of the B-tree. I could have written this as a single data type, but splitting it up like this will make things clearer later on.

Unfortunately, this definition makes it possible to construct trees like this one which breaks the second rule requiring all leaves to be equidistant from the root. We'd need something other than the types to ensure that search doesn't blow out to a linear-time algorithm.

We can do better than this, but before I can show you how, I need to introduce GADT-style definitions.

Consider the types of the constructors in our definition of **T**. For example, the **Br** constructor is a function that takes an argument of type **N a**, and constructs a **T a**. Whereas our ordinary data definition lists the types of the components of each constructor, from which the types of the constructors are inferred, a GADT-style definition gives the type of the constructor itself, and the types of the components are inferred.

Aside from syntax, the important difference between the two styles is that a GADT gives the result type of each constructor explicitly, whereas the ordinary definition always implies that all of the constructors' result types are just the same as the type being defined.

So what have we gained? Well, in a GADT, we're allowed to specialise type variables in the constructor result types. We can use this to encode the second rule, which we rephrase to say that sub-trees of an internal node must all have the same height.

Now we need to measure the heights of trees, and we'll do this using the standard unary definition of natural numbers, which says that a natural number is either zero or the successor of some other natural number. Although this is an ordinary data definition, we're actually interested in using **Z** and **S** as type constructors, and for that, we'll use GHC's

DataKinds extension. An important property of this definition is that **Z** and **S p** are distinct types for any **p**, and also **S p** and **S q** are distinct types if and only if **p** and **q** are distinct types.

We can now add a type index to our tree, to reflect its height. So, rather than describing trees of some element type, we now describe trees of a particular height and element type.

Naturally, we define a **LF** as a tree of height zero. And when an internal node has sub-trees at height **n**, we define the resulting **BR** to have height **S n**. We also need to annotate the type of an internal node **N** with the same height as its components, but this only requires an ordinary Haskell data type.

Now that we've captured the first two rules, we only need to remember to keep our data ordered. But before we head to the code, let's illustrate the insertion algorithm.

We'll start with a B-tree containing the elements one to three, and we'll insert the element 4. First, we search from the root of the tree, to find the insertion point. This will always be at the base of the tree.

In this case, there is room to grow the node at the insertion point, so we construct a new node containing the existing contents and the new element. Remember, these are immutable structures, so we perform operations by creating new structures that share parts of the old structures.

Working backwards, we reconstruct the search path, incorporating the new node, and parts of the old structure wherever possible.

If we no longer need the old version of the structure, it can be reclaimed by garbage collection.

Now, if we want to insert a 5, we can't extend the node at the insertion point, because it already has the maximum size. I'll call this an overflow. Instead, we create a pair of nodes containing 3 and 5, and push the in-between element, 4, up to the next level. We need this extra element at the next level up, because we have an extra sub-tree at this level, and sub-trees and elements are always interleaved. Then, to complete the insertion, we create a new node to combine the pushed element with the existing parent node, and garbage-collect the unused parts of the old version.

There are no surprises if we insert a 6 into this tree, though, at this point, I'll give up showing both the old and new versions, and just assume that you understand what it means to operate on immutable structures.

But now if we insert a 7, we get an overflow at the insertion point, which then causes an overflow at the parent node. In turn, this pushes the in-between element, 4, up to the next level. Since there is no parent, this creates a new level, and this how B-trees grow.

Ok, at last we're ready to look at some code. Here, you can see the data types we previously defined. I have a couple of helper functions, and I've also enabled a number of language extensions. **GADTs** is needed for our tree data type, and **DataKinds** allows us to use our **Nat** constructors as types. The others are to support the hole-driven style of development I described in my previous video.

I'm going to add some dummy types which we'll use to trick GHC into telling us the types of things in our environment. We'll use **Foo** to elaborate ordinary data types, and we'll use **M** and **P** for **Nat**-kinded types.

And I'll add smart constructors for the common cases where we want to construct a **Br** node containing a **T1** or **T2**.

Now, having tree-depth information in our types might help us get our implementation right, but it will cause problems for our clients, who aren't interested in such implementation details.

We can hide the depth-indices from our clients using an existential type. The GADT syntax works well for this. We define a type with a single constructor, which takes our depth-indexed **T**, and builds a non-indexed **Tree** to give to the client. Just as important, when we pattern-match on a **Tree**, we temporarily get access to our depth-indexed **T**.

Let's start with the type of **insert**. For any type **a**, **insert** takes an **a** and a **Tree** of **a**, and gives us back a **Tree** of **a**. For searching within a **Tree**, we also need an **Ord** relation on **a**.

Now, for the implementation, I'll delegate to a worker function **ins**, because I want to explicitly write its type, including the depth indices. If I deliberately write an incorrect type for **ins**, GHC tells me what type it expected, so I'll write that as the type of **ins**, adding an explicit **forall** binder for the depth index **n**.

Now, **T** is a recursive structure, so **ins** will be a recursive function. But see how the argument to **ins** carries its depth **n** in its type. If **ins** returns a **Tree**, that depth information is lost, and this means we won't be able to recombine the result of a recursive call with its sibling **T** structures. So this is no good.

Instead, let's make a new intermediate result type **Ins**, which does carry depth information. We'll need a function **finish** to convert this to a **Tree** at the top level.

Ok, so what does **Ins** look like? Remember that one of two things happens when we insert an element into a sub-tree. In the first case, the inserted element can be accommodated in slack space somewhere in the sub-tree, which means the result is just a new **T** at the same level. I'll call this case **Keep**. In the second case, the sub-tree overflows, pushing an element up to the parent, sandwiched between a pair of sub-trees.

Now we can easily write **finish**, by unwrapping each case and constructing the corresponding **Tree**, and note that the second case produces a tree one level higher than the first. Of course, we're also going to have to do a similar case analysis every time we receive an **Ins**, and it turns out that we'll always do this immediately after its construction. That's going to get tedious, so I think we'll get a more sensible implementation if we switch to continuation-passing style. This means passing functions that inline the result of the case analysis wherever we would otherwise construct an **Ins** value.

So we do this by converting **Keep** and **Push** to function types which take their respective components as arguments, and ultimately return the result **Tree**. But rather than returning **Tree** explicitly, I'll generalise to a type variable **t**, because polymorphic types help to constrain their implementations. The type variable will only be instantiated to **Tree** of **a** at the top level. Now when we call **ins**, we pass a **Keep** and a **Push** as alternative

continuations. The implementation of **ins** will need to call one of these to construct something of the type variable **t**.

For example, at the top level, instead of a call of **finish**, we pass continuations corresponding to the two cases of **finish**. For **Keep**, the continuation is just the **Tree** constructor, and for **Push**, we construct a new **T1** node inside a **Tree**.

That type-checks, so let's get rid of the stuff we're no longer using.

Now, there are two cases **ins** must handle, one for each constructor of **T**, so let's stub them out. An interesting thing happens here. Remember that the values of **T** have type indices that depend on which constructor was applied. So when we match on the constructor, the types of its components are refined. That's why I've delayed naming the continuation parameters, because I want to explicitly write their refined types.

Let's start with an easy case, the **LF** constructor. I'll delegate this to a subordinate definition **i**, so I can write its type. The depth type index gets refined by the pattern match, so let's use a dummy **Nat**, **M**, to provoke a type error, which tells us this should have been a **Z**. That's great, because it tells us a lot about what these continuations expect as arguments. We can also write assertions to check that we know how to expand the types of the continuations, like this.

Now, **i** needs to produce something of type variable **t**, and the only things we have that can do that are the **keep** and **push** continuations, so we'll need to call one or the other. Both continuations require arguments of type **T Z a**, and looking back at our definition for **T**, it's clear that the only thing that has that type is a **LF**.

So, we have two possibilities. We could **keep** a **LF**, but then we've clearly failed to **insert x**. Alternatively, we can **push** something sandwiched between a pair of leaves. GHC tells us that thing needs to be of type **a**, and of course, that would be the **x** that we're inserting.

This makes sense, because **push** is meant for the case where insertion causes overflow. And of course, a **LF** can never contain data, so it always overflows.

I'll just tidy this up a bit, and then we can move on to a more interesting case, the **Br** constructor. Again, we'll delegate, but we'll pass through the internal node so we can include it in the assertion of the refined type.

Because this case needs to work at any tree depth, except at the leaves, I'll use variables **p** and **m** for the depth indices of the internal node and continuations. I use separate variables because I'm not yet sure of the relationship between **p** and **m**. Now, although this type-checks, it's too general. We won't be able to call the continuations, because we can't construct arguments of the right types, if **p** and **m** are unconstrained.

So, to help us find the right constraint, I'll provoke a type error by attempting to constrain the variables to our dummy **Nats**, capital **P** and **M**. Essentially, this error tells us that we want **m** to be the successor of **p**. This makes sense, because the components of the internal node are a level below the node from which they came.

There are two cases for internal nodes, **T1** and **T2**. I'll focus on the **T2** case, which is the more interesting. As before, we can write assertions to expand the types of the continuations.

Now, to insert **x**, we compare it to **b** and **d**. I have a helper function, **select2** to expand this into the five cases. All five cases need to construct something of type variable **t**, and it's clear from the types of what's in scope that we'll have to call one of the continuations to do that.

In the case where **x** equals **b**, we want to replace the **T2** node with a new one which has **x** in place of **b**. The type of the new node is **N p a**. But to construct the required **t**, we'll need to call a continuation with a **T m a**, where **m** is the successor of **p**. We can get that by substituting our smart constructor, and then it's clear from the types that we need to use the **keep** continuation. This makes sense, because **keep** is intended for the case where there is no overflow.

The case where **x** equals **d** is much the same.

When **x** is less than **b**, we want to recursively insert into the left subtree, which is **a**. Remember, **ins** takes continuations to handle the result of the recursive insertion, so this is just a tail call. I'll name the continuations for the recursive call as **rkeep** and **rpush**, so I can ask GHC to tell me their types. And I'll also name the arguments to **rkeep** and **rpush**.

We're buried a little deep here, so let's take our bearings. We've been called to insert **x** into a **T2** node, and we've been given continuations **keep** and **push** to call with the result of the insertion. To perform the insertion, we've determined we need to recursively insert into **a**, and the recursive call requires us to provide new continuations **rkeep** and **rpush**. One of the continuations we provide will receive the result of the recursive call, either a **T p a** in the case of **rkeep**, or an **a** sandwiched between a pair of **T p a** in the case of **rpush**. In both cases, we'll need to use those results to construct suitable arguments to pass back to one of **keep** or **push**. Note that the arguments we pass to **keep** or **push** need to be **T m a**, which is a level higher than those we'll receive from **rkeep** or **rpush**.

So, **rkeep** handles the case where the recursive insertion does not overflow. In that case, our **T2** node also does not overflow, and we need only construct a new **T2** node, replacing **a** with the result of the recursive insertion, which is **k**. As we can see, the type of this new node is **T m a**, which is just what we need to pass to **keep** to obtain the result of type **t**.

Now, **rpush** handles the case where the recursive insertion overflows. The result consists of **p**, **q**, and **r,** which should replace **a** in the **T2** node. Since we also need to keep **b**, **c**, **d** and **e**, we clearly can't accommodate everything in a single node.

What can we do? We could construct two layers of **T1** nodes, but when we check the type, it's a level too high for either of the continuations. But the inner **T1** nodes are at the right level, so we have exactly what we need to pass to **push**, along with the in-between element. Again, this makes sense, because if the recursive insertion overflows, our **T2** node, which is already full, must also overflow.

Now if we tidy up, inlining **rkeep** and **rpush**, it's easy to follow the pattern to complete the other two sub-cases.

So now we've completed the **T2** case. The **T1** case is similar, so you can either attempt that yourself, or find the code via the companion article.

Now let's look at deletion. This is more complicated than insertion, and we'll need a couple of tricks to make our types work. In return, the types work even harder to help us get this right. But first, let's look at some informal illustrations of the algorithm.

We'll start with this 7-element tree, and we'll delete the element 3, which happens to be at the base of the tree. Remember, for insertion, we had to deal with overflow. In contrast, when we delete this element, we have an underflow, which we handle by pulling an element from the parent node. In turn, this causes an underflow in its parent node, which pulls from the root node, causing the tree to shrink by one level.

Now, just as we always inserted at the base of the tree, we also always delete from the base of the tree. So what do we do when the element we want to delete is not at the base of the tree?

In this case, we simply replace the element to be deleted, with its immediate predecessor or successor, which will always be at the base of the tree. It doesn't matter which, so I'll just choose the predecessor. This replacement maintains the left-to-right order, and allows the deletion to proceed from the base of the tree.

This means that deletion is a two-phase operation. We have a search phase which uses the usual comparisons to find the element to be deleted, and a replace phase, which locates its immediate predecessor. The latter simply traverses to the right-most element of the sub-tree to the left of the element to be deleted.

We perform the replacement, and then proceed with the deletion as before. Here, we can also see what happens when the sibling of an underflow is full. In this case, we get a rotation instead of a pull. If you like, you can think of a rotation as a pull, which is then followed by an overflow that occurs when the pulled element tries to combine with the full sibling.

Now, there are a couple of other cases that arise during deletion. However, one of the things I'm trying to demonstrate here, is that when we make good use of types, it's enough to have an understanding of the general shape of the algorithm. We don't need to fully catalog all the cases, because the types will show us the details.

Ok, so let's implement **delete**. The type of **delete** is the same as **insert**, and as before we'll delegate to a worker function, using continuation passing style. Remember, deletion occurs in two phases: **search** and **replace**, so we start with a tail-call to the **search** phase.

The type of the **search** worker is similar to the **ins** worker, except now we have to deal with underflow instead of overflow, so our second continuation will be a **Pull** instead of a **Push**. We don't quite know what **Pull** looks like, but it is a continuation, so it must be a function returning **t**. For now, we'll just assume that it takes an argument of some type **Shrunk n a**, which we'll elaborate later. At the top of the **Tree**, the **Keep** continuation will just be a **Tree** constructor, as before, and the **Pull** continuation will be some function, **shrink**, which takes a **Shrunk n a** to a **Tree a**.

This time, I won't bother to delegate any deeper than this, so there are three cases for **search** to handle. However, I'm going to ignore the **T2** case, since it doesn't add anything particularly interesting. You can either implement it yourself, or find the code in the companion article.

The **LF** case is easy. If we're still in the **search** phase when we hit the bottom of the tree, there is nothing we need to delete, so we just **keep** a **LF**. In fact, as we'll see later, the types actually prevent us from doing anything else.

In the **T1** case, we compare **x** with **b**, and split into three sub-cases, all of which need to construct something of type variable **t**. When **x** is less than **b**, we continue the **search** phase in the left sub-tree, which is **a**. If that doesn't underflow, we **keep** a new **T1**, which we construct by replacing **a** with the result of the recursive **search**. We'll figure out what to do with underflow later. And the case where **x** is greater than **b** is similar.

Now, when **x** is equal to **b**, we've found the item we need to **delete**, so we enter the **replace** phase.

Remember, to keep the tree ordered, we need to replace the deleted item with its immediate predecessor, so we can handle the deletion from the bottom of the tree. We find the predecessor by traversing to the right-most element of the left subtree, which means we'll start the **replace** phase with **a**.

The **replace** phase will also have alternative **Keep** and **Pull** continuations. The **Keep** continuation needs to know what replacement to use for **b**, so we'll call the replacement **r**, and include it as an additional argument to the continuation. The **Pull** continuation will also need an extra argument for the replacement, but we'll otherwise leave this to later.

So now in the type of **repl**, we need to add the extra argument to the types of the continuations, which we can do like this.

And now we run into a problem. When we try to implement the **LF** case, we're stuck. To construct a **t**, we need to call either the **Keep** or **Pull** continuation, and both of them require a replacement argument of type **a**. But we don't have anything of type **a**, except of course **x**, which is what we're supposed to **delete**!

To see where we went wrong, rewind to where I said that when the **search** phase found the item to **delete**, the **replace** phase would select the predecessor by finding the right-most element of the left sub-tree. But of course, that only makes sense if the left sub-tree is non-empty. If the left sub-tree is just a **LF**, then there is no need for a **replace** phase. We're already at the bottom of the tree, so we can just start the deletion from where we are.

However, to avoid an extra case analysis, we'll call **repl** anyway, and just pass through the result that **repl** should return if it finds a **LF**. In this case, if **a** and **c** are leaves, and we delete **b**, our **T1** node will underflow, so we'll need to **pull**, though we're not yet sure what that means.

Now, the result **repl** needs is of type **t**, so that's also the type of the extra argument. If you like, you can think of this as a nullary continuation. And it's this argument that **repl** returns if it finds a **LF**.

When **repl** finds a **T1** node, it seeks the rightmost element, immediately recursing into the right sub-tree, which is **c**. If the recursive call doesn't underflow, we **keep** a new **T1** node, in which we replace **c** with the result of the recursive **repl**. Note that the replacement for the deleted item is passed through as an invisible extra argument.

Again, we'll defer thinking about underflow. The third argument to the recursive call is for the case when **c** is a **LF**. In that case, **a** must also be a **LF**, and **b** must be the immediate predecessor of the deleted item. So we **pull**, though we're not yet sure what, but we do pass **b** as the second argument, to replace the deleted item.

So, at last, we're forced to figure out the **Shrunk** data type. Looking at the **Pull** continuations, it seems clear that a **Shrunk** value received from a recursive call must contain all the elements of the sub-tree recursed into, except for a single deleted element. If the recursive call had not underflowed, the **Keep** continuation would have been called, so we know that the **Shrunk** value is necessarily smaller than that sub-tree.

At the top of the tree, an underflow should result in a tree which is one level shorter than the original tree. Compare that with the overflow case in **insert**, which results in a tree one level higher. So, essentially, a **Shrunk n a** should just be a **T (n-1) a**. Now, we don't have a predecessor operation for our type-level **Nat** numbers. We could construct one, but as it turns out, we don't need to. In fact, it's best if **Shrunk Z a** is uninhabited, because this ensures that deletion from an empty sub-tree is handled by the **Keep** continuation. So, if we only need to consider **Shrunk (S n) a**, then it's contents are just **T n a**.

We can express this by defining **Shrunk** as another GADT with a single constructor **H** that takes a **T n a** to **Shrunk (S n) a**. To map this back to a **Tree**, we just rewrap the contents.

Note the type of the **shrink** mapping. It takes a **Shrunk n a**, not a **Shrunk (S n) a**. If we try the latter, the program is rejected, because **search** requires a **Pull** continuation for any **n**, not just **S n**. But how can this be, if we can only construct a **Shrunk (S n) a**? The point is that it's entirely possible to construct a function that takes a **Shrunk n a** for any **n**. However, we'll only be able to apply this function to those **Shrunk n a** that we can actually construct, which is just those **Shrunk (S n) a**. Remember that pattern matching on a GADT refines the type of that branch, so when we match the **H** constructor, the **n** gets refined to **S n** anyway. It's exactly this property of GADTs that ensures that our **Pull** continuation can only be applied in the case that the recursive call has actually shrunk the sub-tree. It's also what prevents us from using **pull** when search finds a **LF**.

So, now we can implement our **Pull** continuations, which need to combine the **Shrunk** value with the siblings of the sub-tree recursed into. In the case where **x** is less than **b**, we need to combine **p** with **b** and **c**. I'll call this merging, and note that it's asymmetric, since **p** and **c** have different types. Thus, we'll have left and right merges for the **T1** case, depending on whether **p** replaces the left or right sub-tree. For this case, we'll have a **mrgl** of **p**, **b** and **c**. And for the case when **x** is greater than **b**, we'll have a **mrgr** of **a**, **b** and **p**.

Now, when **x** equals **b**, we'll have another **mrgl**, since we start the **repl** phase in the left sub-tree. We use the replacement value **r** instead of **b**, because **b** is being deleted, so the values we merge are **p**, **r** and **c**.

So, what is the type of **mrgl**? It takes a **Shrunk p a**, an **a**, and a **T p a**, and as with all our continuations, returns a **t**. Here, **p** is one level below **n**, so we can express that in a constraint. And **mrgr** takes the same types in reverse order.

The cases that **mrgl** has to consider are those of its third argument, of type **T p a**. When the argument is a **T1** node, we have **a**, **c**, and **e**, which are **T** nodes at the level below **p**, interspersed with **b** and **d**, which are just values of type **a**. These will all fit in a single **T2** node, but do we **keep** or **pull**? The types can tell us. If we try to **keep**, we can see that

we're at the wrong level, even though GHC has renamed the type variable. This makes sense, because the components here are two levels below the **T1** node where we started. We dropped one level when we made the recursive call to **search**, and another when we pattern-matched on the arguments of **mrgl**. So, instead, we need to **pull**, and since **pull** expects a **Shrunk** value, we also need to wrap the **T2** node in an **H** constructor.

When **mrgl** finds a **T2** node, we have **a**, **c**, **e**, and **g**, which are **T** nodes at the level below **p**, interspersed with values **b**, **d**, and **f**. We can't fit those into a single node, so we need to split into two layers of **T1** nodes. This time, we've regained the two levels that we dropped, so it makes sense that **pull** is rejected, but **keep** is accepted.

Now, what about the case when **mrgl** finds a **LF**? This case actually makes no sense, because **mrgl** only occurs in the context of an underflow, and an underflow cannot originate in a row of leaves. Fortunately, the types actually prevent us from implementing this case. Remember that pattern-matching on a GADT refines type variables. Matching on the **H** constructor refines **p** to **S** of some **q**, while matching on **LF** refines **p** to **Z**. Of course, **p** cannot be both **S q** and **Z**, so this is a type error. Similarly, it would also be an error to attempt to call **mrgl** with values that would have matched this case.

And so we just leave this case out. Unfortunately, if we enable warnings, GHC will still complain about an incomplete pattern match, even though it complains harder if we attempt to complete it. At least for now, that's just something you have to live with if you work with GADTs in Haskell, though it might improve in a future release of GHC.

The implementation of **mrgr** is much the same as **mrgl**. We just have to reverse everything.

Now, **repl** also has a **Pull** continuation. Since **repl** recurses into the right sub-tree, this is a **mrgr** of **a**, **b**, and **p**. The type of this **mrgr** is similar to the type of **mrgr** in the **search** phase, except this **mrgr** takes an extra argument for the replacement value. However, despite the different type, the implementation is exactly the same, because in the context of **repl**, **keep** and **pull** also expect this extra argument. Obviously, we should factor out this common code, and that just requires us to pass the **keep** an **pull** continuations explicitly, rather than capturing them from the environment. To account for the different types, we just generalise the continuations' result type.

Now, all that remains is to figure out what to pass as the third continuation argument in each call to **repl**. Remember, this is the nullary continuation that is only used when **repl** finds a **LF**.

Within **search**, this occurs when we find the item to be deleted at the bottom of the tree, so we want to drop that item, and **pull** an empty tree, or in other words, **pull** an **H LF**. Although this is correct for the case we're interested in, it has the wrong type for all other cases, because **pull** expects a **Shrunk n a**, not always a **Shrunk (S Z) a**. So where can we find a value that is **LF** in the case we care about, and otherwise has the right type? Of course, this value is only used when **a** is a **LF**, and **a** always has the right type, so we can just use **a** instead of **LF**.

If you just raised an eyebrow, yes, I agree that this is a bit tricky, and usually we don't like tricky code, but the only alternative I've found so far is to put data into the leaves, and that causes the code to blow up by about a factor of two. I'm not sure which is worse, but I do

know which I can fit onto your screen. Please leave a comment if you can show me a better way!

Similarly, within **repl**, this case occurs exactly when we have just found the immediate predecessor, **b**, of the item being deleted. We're already passing **b** back as the replacement for the deleted item, and since that leaves us with nothing else, we need to **pull** a **LF**, but to satisfy the type, we use **a** instead.

To complete the implementation of **delete**, we would need to fill in the **T2** cases for both **search** and **repl**, but since that's just more of the same, I'll leave the video here. You can try to finish this yourself, or find my code via the link shown at the start of the video.

So, hopefully, you've been able to see that judicious use of types can not only guard against a larger class of errors, but can also help us to find the right implementation. And, for the most part, the types don't get in our way.

Note that our GADT only specifies the B-tree structure invariant. It doesn't specify the order invariant, and it certainly doesn't ensure that our functions actually perform insertion and deletion. We'll still need to test for those properties. Perhaps, in a future video, I'll look at ways to guarantee the latter properties using types.