ProgrammingAbstractions-Lecture22

Instructor (Julie Zelenski):How are we doing? Spring apparently is here; just like California, one day it's winter, one day it's spring. Hopefully it's here to say. Hopefully thought it's not interfering with you getting all of your work done? How many people have a PQ implementation working, at least one of them? That's a whole bunch. How many have two of them? Even better. Any advice from those of you who have gotten or two of those things doing their job that you want to offer anybody who's yet to reach that state of enlightenment? Okay, so that's coming in on Wednesday and then your final assignment will go out on Wednesday. It will be due on Friday of Dead Week, so a little bit over a week to do that.

In term of late day planning, if you're the type of person who just feels like you would be hopefully cheated if you didn't get to use your late days, your plan was to make sure you used them. Let it be known that you will be able to use at least one on the last one. So it will be due on that Friday. If you are taking a late day, it will be due on Monday of exam week, probably not a good thing to actually slip into exam week, but if you're really in a bind you can use one, but no more than one. So if you want to make sure you use yours up, you may have to use some now. Not that I am recommending it. I think that there's a good point of pride to get done and have both of your late days in the bag. It means that you didn't hit any crisis, which is good.

We're going to talk about binary search trees and implementing the map abstraction using a binary search tree today. And the reading that goes along with that is chapter 13, all sorts of tree stuff. Anything administratively? No? So we had talked about doing it, we did this last time, we did an unsorted vector implementation of the map, we just did a little struct of the key value pair. And then we went through that both add and get value are running at linear time in that form because of the need to search; search to find a value to override it, and search to find a value to pull out the existing match to it. And if we went to the sorted form of that, we could get get value to be able to capitalize on that binary search property and be able to quickly find the key if it exists in there or to determine it doesn't exist.

But the add still suffered from the shuffling problem. We can quickly find, using the binary search to find where the new element would need to go. But if it doesn't already go there, it doesn't already exist. We're going to have to shuffle to make that space. So we have still a linear factor in there that we're working on getting rid of. And then I know at the bottom that none of them have any built in overhead per element. There may be some capacity, excess capacity in the vector that we're absorbing, but there's no four, ten, eight-byte chunk that every element is taking as a cost.

So we started to talk about this at the end on Friday, and we're going to go with this and see where this takes us. We had the idea of the vector, and we know what vector constraints are, right? This idea of the contiguous memory, and we were going to think about the link list for a minute to see if it was going to buy us anything. It tends to have an inverse relationship, that once you know where you want to insert something it's easy

to do that rearrangement because it's just a little bit of pointer rewiring that needs to happen in that situation. But it's hard to find a position insert, even if the link list is in sorted order, knowing where the S's are, or the M's are, or the B's are is still a matter of starting from the beginning; that your only access in the singly linked list is to start at the beginning and work your way down.

And so I'm going to take you through a little bit of a thought exercise about well, what if we try to do something about that? Right now having a pointer coming into Bashful and then subsequent ones going in sorted order isn't really very helpful. But if we were willing to try to rearrange our pointers to give us the access that we have in the sorted array form; for example, if I had a pointer to the middlemost element, in this case the Grumpy that's midway down the list, if I had that pointer and from there I could say, is the element I'm looking for ahead of Grumpy or behind Grumpy. So having Grumpy split my input in half. If I had that, right, then that would actually get me sort of moving towards that direction that I wanted to get, which is using the binary search property to facilitate throwing away half the data and look into just the half remaining that's interesting.

Well, if I had a pointer to Grumpy, that doesn't quite solve my problem. A pointer to Grumpy could get me halfway down the list, but then the pointer's coming in this way. If I inverted the pointers leading away from it, that's also kind of helping to facilitate what I'm trying to do. So if I could point toward the middle and then kind of move away, and then if I actually kind of applied the same strategy not just at the topmost level, but at every subsequent division down, from the half, to the quarter to the eight, to the sixteenth. And instead of having a pointer to the front of the list, what I really maintained was a pointer to the middlemost element, and then that element maintained pointers to the middle of the upper quarter and the middle of the lower quarter, and all the way down. Then I could pull up this list into something that we call a binary search tree.

So the way I would be storing this is having a pointer to a node, which is expected to the median or the middle, or somewhere right along the values being stored. And that it has two pointers, not just a next or previous or something like that, that are related to all the elements that precede this one in the ordering, and all the elements that follow it are in these two pointers. They call these subtrees coming off the smaller trees over here with three elements on this side, and three elements on that side. And that each of those nodes has recursively the same formulation; that it has two subtree hanging off of it that contain, given any particular node, that there's this binary search ordering here that says everything that's in that left subtree precedes this node in ordering. And everything that's in that right subtree follows it. And that's true for every node throughout the entire arrangement here.

So this is called a tree, and this particular form is actually called the binary search tree. I'm going to throw a little bit of vocabulary at you, because I'm going to start using these words, and I just want to get them out there early so that you understand what those words mean when I start using them. Is that each of the cells in here, like the cells that are allocated for a link list, are individually heap allocated, we're going to have pointers that

allow us to get back to them. We typically call those nodes; in terms of tree I'll call it a cell or node in the list kind of interchangeably. I'm more likely to use node when I'm talking about tree.

And a tree is really just a pointer to a node. The empty tree would be a pointer whose value is null, so pointing at nothing. A singleton tree would be a pointer to a single node that has left and right subtrees that are both empty or both null. So Bashful by itself is a singleton, a pointer to Bashful.

When we talk about subtrees, it just kind of means looking at the top; Grumpy has two subtrees, a left and a right, that themselves are just smaller forms of the same recursive structure. And then we would say that Grumpy is the parent of Doc and Sleepy. So those are the two children nodes, and we'll call them the left child and the right child. And the node that's at the very top, the one that our external pointer is coming into and then kind of leads the way, is called the root node. The first node of the tree where we first start our work is going to be called the root. And these nodes down here at the bottom that have empty subtrees are called leaf nodes. So a node that has no further subtrees, so by itself has null left and right trees, is called a leaf.

So that actually means the tree is kind of upside down if you look at it. That Grumpy's the root and these things are the leaves shows you that apparently most computer scientists don't get out very much, because they haven't noticed that trees actually grow the other direction. But just kind of go with it, and you'll be speaking tree-speak in no time. Anybody have any questions about what we've got basically set up here?

So there is a whole family of things that come under the heading of tree in computer science. We're going to look in particular at this one embodiment, the binary search tree. But before I get deep down in the details of that, I just want to back up for a second and just talk about what trees in general mean, and what kind of things in general are represented in manipulative trees. So that you have a sense of what is the broad notion of tree as a computer scientist would see it, is just that a tree is a recursive branching structure where there are nodes, which have pointers to subtrees. There may be one of those subtrees. There may be ten of those subtrees. There may be two; depending on the nature of the tree you're working on there may be a constraint on exactly how many children they have or some flexibility in the number of children they have depending on what's being modeled here.

There's always a single root node. There's a node that kind of sits at the top of this hierarchy that leads down to these things, and that to form a tree there has to be a reachable path from the root to every single node. So there aren't some nodes that exist way out here, and there aren't two different ways to get to the same node in a tree; that if the midterm is kept in the folder called exams, which is in the folder called cs106, which is in my home folder. There's not some other way that you can get to the midterm; the midterm lives exactly that part in the tree, and there's no circularity or loops in the structure.

And so a lot of things do get modeled tree, the one I've drawn here is just the file system. Think about how your files are maintained on a computer, there tends to be a structure of an outermost folder, which has inner folders, which have inner folders. And along the way there are these files, which you can think of as leaf nodes that have no further subtrees underneath them. And you can talk about, say, what is the size of all the folders in my home directory? Well it would kind of be recursively defined as summing over all the folders within it, how much storage is being used by those; which when it gets down to a leaf node can say, how much space does this file take up and kind of add those together.

Things like your family trees, right, sort of modeling genealogy. Things like the decomposition tree in your program; main calling, read file calling, set up boggle board calling, give instructions. There is a tree structure to that, about what pieces use what pieces and descendants from there. The ones that we're going to look at in particularly are the ones that are in the family of the binary trees.

So a binary tree has two children, exactly two children. There's a left and a right pointer coming out of each node. One or both of those can be null, but there can never be more than two children. And there is sort of space for recording two children, as opposed to recording things which are trinary, which have three possibilities for children, or where there may be some four or eight or some other number there. And in particular it's the search tree that's going to help us get the work done that we want, which is there's not just any old arrangement of what's on the left, and what's on the right side. That the nodes are going to be organized in here to facilitate binary search by keeping one half over on the left and one half over on the right, and that we'll know which half something goes in by virtue of looking at the key.

So let's take a look at a simple example of a binary search tree with numbers. So if I have a struct node that keeps track of an integer value and has two pointers, the left and the right, that point away from it. And in this case my root node is 57, and looking at 57 I can tell you that every value in this tree that is less than 57 will be in the left subtree. There will be no values that are 56 or lower that are on the right side. And that will be true at every step down the tree. So if I'm looking for something that actually gives me that immediate access we were using the vector's sorting property for. Which is if I'm looking for the number 70, and I start at the root node of 57, then I know it's not in the left subtree, so assuming that about half the nodes are pointed to over on that side, I've now discarded half of the input even from consideration.

If I start at the 70, and I say okay well if 57 must be over here, then again recursively apply the same search property here. It's like if I'm looking for 70, is it going to be before or after 79? It's going to be before, so in fact I throw away anything that leads away on the right subtree of 79, and work my way one level down. So each step in the recursive search is represented by sort of taking a step down in that search tree; working my way down from the root to the leaf nodes. And I either will find the value I'm looking for along the way, or I will fall off the bottom of that tree when I haven't been able to find it.

And that will tell me; oh it couldn't have been in the tree, because the only place it could have been is the one that followed the binary search property.

And so if I look for 70, I say is it greater or less than 62? It's greater. Is it greater or less than 71? It's less then. And then I work off there I get to the empty tree. Then I say well if there was a 60, that's where it was going to be, and it wasn't there so there much not be a 70 in this tree. So it is exactly designed to support one kind of search, and one kind of search only, which is this efficient divide and conquer binary search, work its way down.

The other kinds of things, let's say, that trees are used for may not need this kind of support, so they may not go out of their way to do this. But the one we're actually very interested in is because of maps doing a lot of additions into a sorted facility and searches in that sort of facility. If we can bottle it as a binary search tree, we're able to get the best of both worlds. To have that super fast logarithmic search, as well as have the flexibility of the pointer based structure to help us do an insert.

So if I'm ready to put a 70 into this tree, the same path that led to me to where it should be tells us where to put that new node in place. And if all I need to do is to wire that in place; and if all I need to do to wire that in place is to kind of create a new node in the heap and put the pointers coming in and out of it the way that I would in a link list, then I should be able to do both searches and inserts in time proportional to the height of the tree, which for a relatively balanced tree would be logarithmic. So it sounds like we can get everything we wanted here with a little bit of work.

So let me tell you – just do a little bit of playing with trees, just to kind of get used to thinking about how we do stuff on them, and then I'm going to go forward with implementing map using a binary search tree. Trees are very, very recursive. So for those of you who still feel like recursion is a little bit hazy, this is another great area to strengthen your thinking about recursive. In fact, I think a lot of people find operating on trees very, very natural. Because trees are so recursive, the idea that you write recursive algorithms to them is almost like you do it without even thinking, you don't have to think hard. You're like, you typically need to do something to your current node, and then recursively operate on your left and your right. And it's so natural that you don't even have stop and think hard about what that means.

So your base case tends to be getting to an empty tree and having a pointer that points to null, and in the other cases have some node operate on your children. Some of the simplest things are just traversals. If I tree structure and I'd like to do something to every node, go through it and print them all, go through it and write them to a file, go through and add them all up to determine the average score, I need these things. I'm going to need to do some processing that visits every node of the tree once and exactly once, and recursion is exactly the way to do that.

And there's a little bit of some choices, and when you're handling a particular part of a tree, are you handling the current node and then recurring on the left and right subtrees? Are you recurring first and then handling the nodes. So some of those things that we saw

in terms of the link list that change the order of the processing, but they don't change whether you visit everybody. It's just a matter of which one you want to do before.

I'm going to look at a couple of the simple traversals here, and I'll point out why that mattered. If we look at in order, so we'll consider that an in order traversal is for you to operate on your left subtree recursively, handle the current node right here, and then operate on your right subtree recursively. The base case, which is kind of hidden in here, is if T is null, we get to an empty subtree we don't do anything. So only in the nonempty case are we actually processing and making recursive calls. So if I start with my node 57 being the root of my tree, and I say print that tree starting from the root. It will print everything out of the left subtree, then print the 57, and then print everything out of the right subtree.

And given that gets applied everywhere down, the effect will be 57 says first print my left subtree. And 37 says, well print my left subtree. And 15 says, well first print my left subtree. And then seven says, first print my left subtree. Well, that ones empty, and so nothing gets printed. As the recursion unwinds, it comes back to the last most call. The last most call was seven. It says, okay, print seven and now print seven's right subtree. Again that's an empty, so nothing gets printed there. It will come back and then do the same thing with 15. I've done everything to the left of 15; let's print 15 and its right. And doing that kind of all the way throughout the tree means that we will get the numbers out in sorted order; all right, five, 17, seven, 15, 32, 57, 59, 62, 71, 79, 93. An in order traversal in a binary search tree rediscovers, visits all the nodes from smallest to largest.

Student:[Inaudible]

Instructor (Julie Zelenski):So duplicates, you just have to have a strategy. It turns out in the map, the duplicates overwrite. So any time you get a new value you print there. Typically you just need to decide, do they go to the left or do they go to the right? So that you know when you're equal to if you wanted to find another one of these, where would it go? In a lot of situations you just don't have duplicates, so it's not actually a big deal. You can't throw them arbitrarily on either side because you would end up having to search both sides to find them. So you want to just know that's it's less than or equal to on the left and greater than on the right.

So that's pretty neat, right, that that means that this for purposes of being able to iterate for something, if you were to walk the structure in order, then you can produce them back out in sorted order, which tends to be a convenient way to print the output. There are a lot of situations where you do want that browsable list in alphanumerical order. And the tree makes that easy to do.

When it comes time to delete the tree, I'm going to need to go through and delete each of those nodes individually, the same way I would on a link list to delete the whole structure. The order of traversal that's going to be most convenient for doing that is going to be to do it post order. Which is to say, when I'm looking at the root node 57, I want to delete my entire left subtree recursively, delete my entire right subtree recursively, and

only after both of those pieces have been cleaned up, delete the node we're on; so deleting down below to the left, down below to the right and then delete this one. If I did it in the other order, if I tried to delete in between in the in order or in the pre order case where I did my node and then to my subtrees, I'd be reaching into parts of my memory I've deleted. So we do want to do it in that order, so recursion coming back to visit you again.

Let's talk about how to make map work as a tree. So if you think about what maps hold, maps hold a key, which is string type; a value, which is client some sort of template parameter there. Now that I've built a structure that raps those two things with a pair and adds to it, right these two additional pointers to the left and to the right subtrees that point back to a recursive struct, the one data member that the map will have at the top is just the root pointer. The pointer to the node at the top of the tree, and then from there we'll traverse pointers to find our way to other ones. And we will organize that tree for the binary search property.

So using the key as the discriminate of what goes left and right, we'll compare each new key to the key at the root and decide is it going to go in the left subtree root or the right subtree root? And so on down to the matching position when we're doing a find, or to the place to insert if we're ready to put a new node in. So get value and add will have very similar structures, in terms of searching that tree from the root down to the leaf to find that match along the way, or to report where to put a new one in.

So let's look at the actual code; it has my struct, it has the strike key, it has the value type node, two pointers there, and then the one data member for the map itself here is the pointer to the root node. I also put a couple of helper number functions that I'm going to need. Searching the tree and entering a new node in the tree are going to require a little helper, and we're going to see why that is. That basically that the add and get value public number functions are going to be just wrappers that turn and pass the code off to these recursive helpers. And if these cases, the additional information that each of these is tracking is what node we're at as we work our way down the tree.

But the initial call to add is just going to say, well here's a key to value go put it in the right place. And that while we're doing the recursion we need to know where we are in the tree. So we're adding that extra parameter. What a lot of recursive code needs is just a little more housekeeping, and the housekeeping is what part of the tree we're currently searching or trying to enter into.

So this is the outer call for get value. So get value is really just a wrapper, it's going to call tree search. It's going to ask the starting value for root. It's the beginning point for the search, looking for a particular key. And what tree searches are going to return is a pointer to a node. If that node was not found, if there was no node that has a string key that matches the one that we're looking for, then we'll return null. Otherwise, we'll return the node. And so it either will error when it didn't find it or pull out the value that was out of that node structure.

Well, this part looks pretty okay. The template, right, there's always just a little bit of goo up there, right, seeing kind of the template header on the top and the use of the val type, and all this other stuff. It's going to get a little bit worse here. So just kind of hopefully take a deep breath and say, okay that part actually doesn't scare me too much. We're going to look at what happens when I write the tree search code, where it really actually goes off and does the search down the tree. And there's going to be a little bit more machinations required on this.

So let's just talk about it first, just what the code does. And then I'm going to come back and talk about some of the syntax that's required to make this work correctly. So the idea of tree search is, given a pointer to a particular subtree, it could be the root of the original tree or some smaller subtree further down, and a key that we're looking for, we're going to return a match when we find that node. So looking at the step here, where if the key that we have matches the key of the node we're looking at, then this is the node. This is the one we wanted.

Otherwise, right, we look at two cases. If the key is less than this node's key, then we're just going to return the result from searching in the left subtree. Otherwise it must be greater than, and we're going to search in the right subtree. So this case, right, duplicates aren't allowed, so there will be exactly one match, if at all. And when we find it we return. The other case that needs to be handled as part of base case is if what if we ever get to an empty tree. So we're looking for 32 and the last node we just passed was 35. We needed to be the left of it, and it turns out there was nothing down there, we have an empty tree. We'll hit this case where T is null, and we'll return null. That means there was never a match to that. There's no other place in the tree where that 32 could have been.

The binary search property completely dictates the series of left and right turns. There's nowhere else to look. So that's actually, really solving a lot of our – cleaning up our work for us, streamlining where we need to go; because the tree is organized through the mid point and the quarter point and then the eighth point to narrow down to where we could go. So as it is, this code works correctly. So do you have any questions about how it's thinking about doing its work? And then I'm going to show you why the syntax is a little bit wrong though.

Okay, so let's talk about this return type coming out of the function. So let's go back to this for a second. That node is defined as a private member of the map class. So node is actually not a global type. It does not exist outside the map class. Its real name is map val type:: node. That's its full name, is that. Now, inside the implementation of map, which is to say inside the class map, or inside member functions that we're defining later, you'll not that we can just kind of drop the big full name. The kind of my name is mister such and such, and such and such; it's like okay, you can just call it node, the short name internal to the class because it's inside that scope. And it's able to say, okay in this scope is there something called node? Okay there is. Okay, that's the one.

Outside of that scope, though, if you were trying to use that outside of the class or some other places, it's going to have a little bit more trouble with this idea of node by itself. It's

going to need to see that full name. Let me show you on this piece of code, that node here happens to be in the slight quirk of the way sequel plus defines things, that that use of node is not considered in class scope yet. That the compiler leads from left to write, so we'll look at the first one. It says, template type name val type, and so what the compiler sees so far is, I'm about to see something that's templated on val type. So it's a pattern for which val type is a placeholder in.

And then it sees val type as the return value, and it says, okay, that's the placeholder. And then it sees this map {val type}:: and it says, okay, this is within the – the thing that I'm defining here is within scope of the map class. And so it sees the map {val type}:: and that as soon as it crosses those double colons, from that point forward it's inside the map scope. And so, when I use things like tree search, or I use the name node, it knows what I'm talking about. It says, oh there's a tree search in map class. There's a node struct in map class, it's all good.

In the case of this one I've seen template type and val type, and at this point it doesn't know anything yet. And so it sees me using node, and it say, node I've never heard of node. I'm in the middle of making some template based on val type, but I haven't heard anything about node. Node isn't something that exists in the global name space. You must have made some tragic error. What it wants you to say is to give it the full name of map {value type}:: node. That the real full this is my name, and otherwise it will be quite confused.

If that weren't enough, we'd like to think that we were done with placating the C++ compiler. But it turns out there's one other little thing that has to happen here, which is there is an obscure reason why, even though I've given it the full name, and I feel like I've done my duty in describing this thing, that the compiler still is a little bit confused by this usage. And because the C++ language is just immensely complicated, there are a lot of funny interactions, there are a couple of situations it can get in to, where because it's trying hard to read left to right and see the things. Sometimes it needs to know some things in advance that it doesn't have all the information for. And it turns out in this case there's actually some obscure situation where it might not realize this is really a type. It might think it was more of a global constant that we actually have to use the type name key word again here on this one construction.

It is basically a hint to the compiler that says, the next thing coming up really is a type name. So given that you can't tell that, you'd think it would be obvious; in this case it doesn't seem like there's a lot for it to be confused at, but there's actually some other constructions that we can't rule out. So the compiler is a little bit confused in this situation, and it wants you to tell it in advance, and so I'll tell you that the basic rules for when you're going to need to use this is exactly in this and only this situation. And I'll tell you what the things are that are required for type name to become part of your repertoire.

You have a return type on a member function that is defined as part of a class that is a template, and that type that you are using is defined within the scope of that template. So it's trying to use something that's within a template class scope outside of that scope that

requires this type name. And the only place where we see things used out of their scope is on the return value. So this is kind of the one configuration that causes you to have to do this. It's really just, you know, a little bit of a C++ quirk. It doesn't change anything about what the code's doing or anything interesting about binary search trees, but it will come up in any situation where you have this helper member function that wants to return something that's templated. You have to placate the compiler there.

The error message when you don't do it, actually turns out to be very good in GCC and very terrible in visual studio. GCC says type name required here, and visual studio says something completely mystical. So it is something to be just a little bit careful about when you're trying to write that kind of code. Question?

Student: The asterisk?

Instructor (Julie Zelenski):The asterisk is just because it returns a pointer. In all the situations, I'm always returning a pointer to the node back there. So here's the pointer to the match I found or node when I didn't find one. All right, that's a little bit of grimy code. Let's keep going.

So how is it that add works? Well, it starts like get value, that the strategy we're going to use for inserting into the tree is the really simple one that say, well don't rearrange anything that's there. For example, if I were trying to put the number 54 into this tree, you could say – well you could imagine putting it at the root and moving things around. Imagine that your goal is to put it in with a minimal amount of rearrangement. So leaving all the existing nodes connected the way they are, we're going to follow the binary search path of the get value code to find the place where 54 would have been if we left everything else in tact.

Looking at 57 you could say, well it has to be to the left of 57. Looking to the 32 you say, it has to be to the right of 32. Here's where we get some empty tree, so we say, well that's the place where we'll put it. We're going to tack it off, kind of hang it off one of the empty trees along the front tier, the bottom of that tree. And so, we'll just walk our way down to where we fall off the tree, and put the new node in place there. So if I want the node 58 in, I'll say it has to go to the right; it has to go to the left; it has to go to the left; it has to go to the left; it will go right down there.

If I want the node 100 in, it has to go right, right, right, right, right. That just kind of following the path, leaving all the current pointers and nodes in place, kind of finding a new place. And there's exactly one that given any number in a current configuration and the goal of not rearranging anything, there's exactly one place to put that new node in there that will keep the binary search property in tact. So it actually does a lot of decision making for us. We don't have to make any hard decisions about where to go. It just all this left, right, left, right based on less than and greater than.

Student: What if the values are equal?

Instructor (Julie Zelenski):If the values are equal you have to have a plan, and it could be that you're using all the less than or equal to or greater than or equal to. You just have to have a plan. In terms of our map it turns out to equal to all the rights. So we handle it differently, which we don't every put a duplicate into the tree. So the add call is going to basically just be a wrapper that calls tree enter, starting from the root with the key and the value that then goes and does that work, walks its way down to the bottom to find the right place to stick the new one in.

Here is a very dense little piece of code that will put a node into a tree using the strategy that we've just talked about of, don't rearrange any of the pointers, just walk your way down to where you find an empty tree. And once you find that empty tree, replace the empty tree with a singleton node with the new key and value that you have. So let's not look at the base case first. Let's actually look at the recursive cases, because I think that they're a little bit easier to think about. Is if we every find a match, so we have a key coming in, and it matches the current node, then we just overwrite. So that's the way the map handles insertion of a duplicate value.

Now, in some other situation you might actually insert one. But in our case it turns out we never want a duplicate. We always want to take the existing value and replace it with a new one. If it matches, we're done. Otherwise, we need to decide whether we go left or right. Based on the key we're attempting to insert or the current node value, it either will go in the left of the subtree, left of the right, in which case we make exactly one recursive call either to the left or to the right depending on the relationship of key to the current key's value.

And then the one base case that everything eventually gets down to, that's going to create a new node, is when you have an empty tree, then that says that you have followed the path that has lead off the tree and fallen off the bottom. It says replace that with this new singleton node. So we make a new node out in the heap. We're assigning the key based on this. We're assigning the value based on the parameter coming in. And then we set its left and right null to do a new leaf node, right, it has left and right null subtrees. And the way this got wired in the tree, you don't see who is it that's now pointing to this tree, this new singleton node that I placed into here? You don't typically see that wire that's coming into it. Who is pointing to it is actually being done by T being passed by reference.

The node right above it, the left or the right of what's going to become the parent of the new node being entered, was passed by reference. It previously was null, and it got reassigned to point to this new node. So the kind of wiring in of that node is happening by the pass by reference of the pointer, which is a tricky thing to kind of watch and get your head around. Let's look at the code operating. Do you have a question?

Student: Yes. [Inaudible]. Can you explain that little step? Instructor:

Well this step is just something in the case of the map, if we every find an existing entity that has that key, and we overwrite. So in the map you said previously, Bob's phone number is this, and then you install a new phone number on Bob. If it finds a map to Bob

saying Bob was previously entered, it just replaces his value. So no nodes are created, no pointers are rearranged; it just commandeers that node and changes its value. That's the behaviors specified by map.

So if we have this right now, we've got some handful of fruits that are installed in our tree with some value. It's doesn't matter what the value is, so I didn't even bother to fill it in to confuse you. So this is what I have, papaya is my root node, and then the next level down has banana and peach, and the pear and what not. So root is pointing to papaya right now. When I make my call to insert a new value, let's say that the one that I want to put in the tree now is orange, the very first call to tree enter looks like this, T is the reference parameter that's coming in, and it refers to root. So the very first call to tree enter says please insert orange and its value into the tree who is pointed to by root.

So it actually has that by reference. It's going to need that by reference because it's actually planning on updating that when it finally installs the new node. So the first call to tree enter says, okay well is orange less than or greater than papaya. Orange precedes papaya in the alphabet, so okay, well call tree enter the second time passing the left subtree of the current value of tee. And so T was a reference to root right then, now T becomes a reference to the left subtree coming out of papaya. So it says, now okay, for the tree that we're talking about here, which is the one that points to the banana and below. That's the new root of that subtree is banana, it says, does orange go to the left or right of the root node banana?

And it goes to the right. So the third stack frame of tree enter now says, well the reference pointer is now looking at the right subtree coming out of banana. And it says, well does melon go to the left or the right of that? It goes to the right, so now on this call, right, the current value of T is going to be null. It is a reference, so a synonym for what was the right field coming out of the melon node, and in this is the case where it says, if orange was in the tree, then this is where it would be hanging off of. It would be coming off of the right subtree of melon. That part is null, and so that's our place. That's the place where we're going to take out the existing null, wipe over it, and put in a new pointer to the node orange that's out here.

And so that's how the thing got wired in. It's actually kind of subtle to see how that happened, because you don’t ever see a direct descendant, like T is left equals this, equals that new node; or T is right equals that new node. It happened by virtue of passing T right by reference into the calls further down the stack. That eventually, right when it hit the base case, it took what was previously a pointer to null and overwrote it with a pointer to this new cell, that then got attached into the tree.

Let's take a look at this piece of code. That by reference there turns out to be really, really important. If that T was just being passed as an ordinary node star, nodes would just get lost, dropped on the floor like crazy. For example, consider the very first time when you're passing in root and root is empty. If root is null, it would say, well if root is null and then it would change this local parameter T; T is some new node of these things. But if it really wasn't making permanent changes to the root, really wasn't making root pointer

of that new node, root would continue pointing to null, and that node would just get orphaned. And every subsequent one, the same thing would happen.

So in order to have this be a persistent change, the pointer that's coming in, either the root in the first case or T left or T right in subsequent calls, needed to have the ability to be permanently modified when we attach that new subtree.

That's really very tricky code, so I would be happy to try to answer questions or go through something to try to get your head around what it's supposed to be doing. If you could ask a question I could help you. Or you could just nod your head and say, I'm just totally fine.

Student: Could you use the tree search algorithm that you programmed before?

Instructor (Julie Zelenski):They're very close, right, and where did I put it? It's on this slide. The problem with this one right now is it does really stop and return the null. So I could actually unify them, but it would take a little more work, and at some point you say, wow I could unify this at the expense of making this other piece of code a little more complicated. So in this case I chose to keep them separate. It's not impossible to unify them, but you end up having to point to where the node is or a null where you would insert a new node. Tree search doesn't care about, but tree enter does. It sort of would modify it to fit the needs of both at the expense of making it a little more awkward on both sides. Question here.

Student:[Inaudible] the number example. If you were inserting 54 where you said, if you called print, wouldn't it print out of order?

Instructor (Julie Zelenski):So 54 would go left of 57, right of 32, so it'd go over here. But the way in order traversals work is says, print everything in my left subtree, print everything in my left subtree, left subtree. So it would kind of print seven as it came back five, 32, and then it would print 54 before it got to the 57. So an in order traversal of a binary search tree, as long as the binary search tree is properly maintained, will print them in sorted order. It's almost like, don't even think too hard about it, if everything in my left subtree is smaller, and everything in my right subtree is greater than me, if I print all of those I have to come out in sorted order. If the property is correctly maintained in the tree, then that by definition is a sorted way to pass through all the numbers.

So if at any point an in order traversal is hitting numbers out of order, it would mean that the tree wasn't properly organized. So let's talk about how that works, there are a couple of other operations on the tree, things like doing the size, which would count the number of members. Which you could do by just doing a traversal, or you might do by just caching a number that you updated as you added and moved nodes. And the contains keys, that's basically just a tree search that determines whether it found a node or not. The rest of the implementation is not interest. There's also deleting, but I didn't show you that.

The space use, how does this relate to the other known implementations that we have for math? It adds certainly some space overhead. We're going to have two pointers, a left and a right off every entry. So every entry is guaranteed to have an excess 8-byte capacity relative to the data being stored. It does actually add those tightly to the allocation. So like a link list, it allocates nodes on a per entry basis, so there's actually not a lot of reserve capacity that's waiting around unused the way it might been in a vector situation. And the performance of it is that it's based on the height of the tree, that both the add and the get value are doing a traversal starting from the root and working their way down to the bottom of the tree, either finding what it wanted along the way or falling off the bottom of the tree when it wasn't found or when it needs to be inserting a new node.

That height of the tree, you're expecting it to be logarithmic. And so if I look at, for example, the values, I've got seven values here, two, eight, 14, 15, 20, and 21. Those seven, eight values can be inserted in a tree to get very different results. There's not one binary search tree that represents those values uniquely. Depending on what order you manage to insert them you'll have different things. For example, given the way we're doing it, whatever node is inserted first will be the root, and we never move the root. We leave the root along. So if you look at this as a historical artifact and say, well they entered a bunch of numbers, and this is the tree I got. I can tell you for sure that 15 was the very first number that they entered. There's no other possibility, 15 got entered first.

The next number that got inserted was one of eight or 20, I don't know which. I can tell you, for example, that eight got inserted before 14. In terms of a level-by-level arrangement, the ones on level one were always inserted before the ones on level two. But one possibility I said here was well maybe 15 went in first, then eight, and then two, and then 20, and then 21, and then 14, and then 18. There are some other rearrangements that are also possible that would create the exact same tree. And there are other variations that would produce a different, but still valid, binary search tree.

So with these seven nodes, right, this tree is height three, and is what's considered perfectly balanced. Half the nodes are on the left and the right of each tree all the way through it, leading to that just beautifully balanced result we're hoping for where both our operations, both add and get value, will be logarithmic. So if I have 1,000 nodes inserted, that height tree should be about ten. And if everything kind of went well and we got kind of went well, and we got an even split all the way around, then it would take ten comparisons to find something in that tree or determine it wasn't there' and also to insert, which was the thing we were having trouble with at the vector case, that because that search leads us right to the place where it needs to go, and the inserting because of the flexibility of the pointer base structure is very easy, we can make both that insert and add operate logarithmically.

Let's think about some cases that aren't so good. I take those same seven numbers and I insert them not quite so perfectly, and I get a not quite so even result. So on this side I can tell you that 20 was the first one inserted. It was the root and the root doesn't move, but then subsequent ones was inserted an eight, and then a 21, and then an 18, and then a 14, and then a 15, and then it went back and got that two, the one over there. Another

possibility for how it got inserted 18, 14, 15, and so on. These are not quite as balanced as the last one, but they're not terribly unbalanced. They're like one or two more than the perfectly balanced case. So at logarithmic plus a little constant still doesn't look that bad.

Okay, how bad can it really get? It can get really bad. I can take those seven numbers and insert them in the worst case, and produce a completely degenerate tree where, for example, if I insert them in sorted order or in reverse sorted order, I really have a link list. I don't have a tree at all. It goes two, 18, 14, 15, 20, 21, that's still a valid binary search tree. If you print it in order, it still comes out in sorted order. It still can be searched using a binary search strategy. It's just because the way it's been divided up here, it's not buying us a lot. So if I'm looking for the number 22, I'll say, well is it to the left or the right of two? It's to the right. Is it to the left or right of eight? Oh, it's to the right. Is it to the left or right, you know like all the way down, so finding 22 looking at every single one of the current entries in the tree, before I could conclude, as I fell off the bottom, that it wasn't there.

Similarly kind of over there; it didn't buy us a lot. It has a lot to do with how we inserted it as to how good a balance we're going to get. And there are some cases that will produce just drastically unbalanced things that require linear searches, totally linear to walk that way down to find something.

There's a couple of other degenerate trees, just to mention it's not only the sorted or reverse sorted. There's also these other kind of saw tooth inputs, they're called, where you just happen to at any given point to have inserted the max or the min of the elements that remain, so it keeps kind of dividing the input into one and N-1. So having the largest and the smallest, and the largest and the smallest, and then subsequent smallest down there can also produce a linear line. It's not quite all just down the left or down the right, but it does mean that every single node has an empty left or right subtree is what's considered the degenerate case here. And both have the same property where half of the pointers off of each cell are going nowhere.

There is an interesting relationship here between these worst-case inputs for transversion and quick sort, and they're actually kind of exactly one to one. That what made quick sort deteriorate was the element of the input, the max or the min of what remain. And that's exactly the same case for transversion, max or min of what remains means that you are not dividing your input into two partitions of roughly equal size. You're dividing them into the one, N-1, which is getting you nowhere.

What do you get to do about it? You have to decide first of all if it's a problem worth solving. So that's what the first one say. Sometimes you just say that it will sometimes happen in some inputs that I consider rare enough that it will degenerate. I don't think they're important enough or common enough to make a big deal out of. I don't think that applies here, because it turns out that getting your data in sorted order is probably not unlikely. That somebody might be refilling a map with data they read from a file that they wrote out in sorted order last time. So saying that if it is coming in sorted, then I'm probably going to tolerate that, is probably not going to work.

And there's two main strategies then, if you've agreed to take action about it, about what you're going to do. The first one is to wait until the problem gets bad enough that you decide to do something about it. So you don't do any rebalancing as you go. You just let stuff go, left, right, left, right, whatever's happening. And you try to keep track of the height of your tree, and you realize when the height of your tree seems sufficiently out of whack with your expected logarithmic height, to do something about it. Then you just fix the whole tree. You can clean the whole thing up. Pull them all out into a vector, rearrange them, put them back in, so the middlemost node is the root all the way down. And kind of create the perfectly balanced tree, and then keep going and wait for it to get out of whack, and fix it again; kind of wait and see.

The other strategy, and this happens to be a more common one in practice, is you just constantly do a little bit of fixing up all the time. Whenever you notice that the tree is looking just a little bit lopsided, you let it get a little bit; usually you'll let it be out of balance by one. So you might have height four on that side and height three on that side. But as soon as it starts to get to be three and five, you do a little shuffle to bring them back to both four. And it involves just a little bit of work all the time to where you just don't let the tree turn into a real mess. But there's a little bit of extra fixing up being done all the time.

And so the particular kind of tree that's discussed in the text, which you can read about, but we won't cover as part of our core material, is called the ADL tree. It's interesting to read about it to see what does it take, what kind of process is required to monitor this and do something about it. And so if we have that in place, we can get it to where the binary search tree part of this is logarithmic on both inserting new values and searching for values, which is that Holy Grail of efficiency boundary. You can say, yeah logarithmic is something we can tolerate by virtue of adding 8 bytes of pointer, potentially some balance factor, and some fanciness to guarantee that performance all over, but gets us somewhere that will scale very, very well for thousands and millions of entries, right, logarithmic is still a very, very fast function.

So it creates a map that could really actually go the distance in scale. So, that's your talk about binary search trees. We'll come back and talk about hashing in graphs and stuff. I will see you on Wednesday.

[End of Audio]

Duration: 51 minutes