

wk7

2-wk3



wk7

# 0wk7-Overview

SUMMARY: This week we start our discussion of data structures, a huge (and hugely useful) topic. This week covers heaps and balanced binary search trees. My primary goals here are to teach you the operations that these data structures support (along with their running times), and to develop your intuition about which data structures are useful for which sorts of problems.

THE VIDEOS: Part XII begins with a short overview video in which I explain my approach to teaching data structures in this course. The next two videos discuss heaps and some of their applications (required), and some details about how they are typically implemented (optional, recommended for hardcore programmer/computer science types). Part XIII discusses balanced binary search trees --- the supported operations and canonical uses (required) and a glimpse at what's "under the hood" (optional).

THE HOMEWORK: Problem Set #3 should solidify your understanding of heaps and search trees. In the programming assignment you'll implement a slick data structure application covered in the lectures (median maintenance).

SUGGESTED READINGS FOR WEEK 3:

CLRS Chapter 6,11,12,13

DPV Section 4.5

SW Section 3.3, 3.4

# 12-1-Data Structures - Overview

In this video, I'm gonna tell you a little bit about my approach toward teaching data structures in this course. So knowing when and how to use basic data structures is an essential skill for the serious programmer. Data structures are used in pretty much every major piece of software. So let me remind you of what's the point, the raison d'etre of the data structure? Its job is to organize data in a way that you can access it quickly and usefully. There's many, many examples of data structures and hopefully you've seen a few of them and perhaps even used a few of them in your own programs. And they range from very simple examples like lists, stacks and queues to more intricate but still very useful ones like heaps, search trees, hash tables. Relatives thereof like balloon filters. Union find structures and so on. So why do we have such a laundry list of data structures? Why is there such a bewildering assortment? It's because different data structures support different sets of operations and are therefore well suited for different types of tasks. Let me remind you of a concrete example that we saw back when we were discussing graph search. In particular, breadth first search and depth first search. So we discussed how implementing breadth first search the right data structure to use was a queue. This is something that supports fast, meaning constant time insertion to the back, And constant time deletion from the front. Depth first search by contrast is a different algorithm with different needs. And because of its recursive nature, a stack is much more appropriate for depth first search. That's because it supports constant time deletion from the front, But constant time insertion to the front. So the last in first out support of a stack is good for depth first search. The first in first out operations of a queue work for breadth first search, Now because different data structures are suitable for different types of tasks you should learn the pros and cons of the basic ones. Generally speaking the fewer operations th at a data structure supports the faster the operations will be. And the smaller the space overhead require by the data structure. Thus, as programmers, it's important that you think carefully about the needs of your application. What are the operations that you need a data structure to export? And then you should choose the right data structure, meaning the one that supports all of the operations you need. But ideally no superfluous ones. Let me suggest four levels of data structure knowledge that someone might have, So level zero is the level of ignorance, for someone who has never heard of a data structure, And is unaware of the fact that organizing your data can produce fundamentally better software. For example, fundamentally faster algorithms, Level one I think of as being cocktail party level awareness. Now obviously, here I'm talking only about the nerdiest of cocktail parties. But nonetheless, this would be someone who could at least hold a conversation about basic data structures. They've heard of things like heaps and binary search trees, they're perhaps aware of some of the basic operations, but this person would be shaky using them in their own program or say in a technical interview context. Now, with level two, we're starting to get somewhere. So here I would put someone who has solid literacy about data structures. They're comfortable using them as a client in their own programs, and they have a good sense of which data structures are appropriate for which types of tasks. Now level three, the final level, is the hardcore programmers and computer scientists. And these are people who are not content to just be a client of data structures, and use them in their own programs, but they actually have an understanding of the guts of these data structures. How they are coded up, how they're implemented, not merely how they are used. Now my guess is that, really a large number of you will wind up using data structures in your own programs, and therefore, learning about what are the operations of different data structures and what are they good for, will be a quite empowering skill for you as a programmer. On the other hand, I'll bet that very few of you will wind up having to implement your own data structures from scratch, as opposed to just using as a client the data structures that already come with the various standard programming libraries. So with this in mind I'm going to focus my teaching on taking up to level two. My discussion gonna focus on the operations reported by various data structures and some of canonical applications. So through this I hope I'll develop your intuition for what kinds of data structures are suitable for what kinds of tasks. Time permitting, however, I also want to include some optional material for those of you wanting to take it to the next level and learn some about the guts of these data structure, the canonical limitations of how you code them up.

# 12-2-Heaps - Operations and Applications

So in this video, we'll start talking about the heap data structure. So in this video I want to be very clear on what are the operations supported by heap, what running time guarantees you can expect from [inaudible] limitations and I want you to get a feel for what kinds of problems they're useful for. In a separate video, we'll take a peek under the hood and talk a little bit about how heaps actually get implemented. But for now, let's just focus on how to use them as a client. So the number one thing you should remember about a given data structure is what operations it supports, and what is the running time you can expect from those operations. So basically, a heap supports two operations. There's some bells and whistles you can throw on. But the two things you gotta now is insertion and extract min. And so the first thing I have to say about a heap is that it's a container for a bunch of objects. And each of these objects should have a key, like a number so that for any given objects you can compare their keys and say one key is bigger than the other key. So for example, maybe the objects are employee records and the key is social security numbers, maybe the objects are the edges of a network and the keys are something like the length or the weight of an edge, maybe each object indicates an event and the key is the time at which that event is meant to occur. Now the number one thing you should remember about a given data structure is, first of all what are the operations that it supports? And second of all, what is the running time you can expect from those operations? For a heap, essentially there's two basic operations. Insert and extract the object that has the minimum key value. So in our discussion of heaps, we're going to allow ties that are pretty much equal to easy with or without ties. So, when you extract men from a heap they may have duplicate key values then there's no specification about which one you get. You just get one of the objects that has a tie for the minimum key value. Now, of course, there's no special reason that I chose to extract the minimum rather than the maximum. You either you can have a second notion of a heap, which is a max heap, which always returns the object of the maximum key value. Or if all you have at your disposal is one of these extract min-type heaps, you can just, negate the sign of all of the key values before you insert them, And then extract min will actually extract, the max key value. So, just to be clear, I'm not proposing here a data structure that supports simultaneously an extract-min operation and an extract-max operation. If you wanted both of those operations, there'd be data structures that would give it to you; probably a binary search tree is the first thing you'd want to consider. So, I'm just saying, you can have a heap of one off two flavors. Either the heap supports extract-min and not extract-max or the heap will support extract-max and not extract-min. So I mentioned that you should remember not just the supported operations of a data structure, but what is the running time of those operations. Now, for the heap, the way it's canonically implemented, the running time you should expect is logarithmic in the number of items in the heap. And its log base two, with quite good constants. So when you think about heaps, you should absolutely remember these two operations. Optionally, there's a couple other things about heaps that are, might be worth remembering Some additional operations that they can support. So the first is an operation called heapify. Like a lot of the other stuff about heaps, it has a few other names as well. But I'm going to call it heapify, one standard name. And the point of heapify is to initialize a heap in linear time. Now, if you have N things and you want to put them all in a heap, obviously you could just invoke insert once per each object. If you have N objects, it seems like that would take N times log N time, log N for each of the N inserts. But there's a slick way to do them in a batch, which takes only linear time. So tha t's the heapify operation, And another operation which can be implemented, although there are some subtleties. Is you can delete not just the minimum, but you can delete an ar-, arbitrary element from the middle of a heap, again, in logarithmic time. I mention this here primarily cuz we're gonna use this operation when we use heaps to speed up Dijkstra's Algorithm. So that's the gist of a heap. You maintain objects that have keys you can insert in logarithmic time, and you can find the one with the minimum key in logarithmic time. So let's turn to applications, I'll give you several. But before I dive into any one application let me just say; what's the general principle? What should [inaudible] you to think that maybe you want to use a heap data structure in some task? So the most common reason to use a heap is if you notice that your program is doing repeated minimum computations. Especially via exhaustive search, Most of the applications that we go through will have this flavor. It will be, there will be a naive program which does a bunch of repeated minimums using just brute force search and we'll see that a very simple application of a heap will allow us to speed it up tremendously. So let's start by returning to the mother of all computational problems, sorting and unsorted array. Now, a sorting algorithm which is sort of so obvious and suboptimal that I didn't even really bother to talk about it at any other point in the course is selection-sort. What do you do? In selection sort, you do a scan through the unsorted array. You find the minimum element; you put that in the first position. You scan through the other N-1 elements; you find the minimum among them. You put that in the second position. You scan through the remaining N-2 unsorted elements. You find the minimum; you put that in the third position, and so on. So evidently, this [inaudible] sorting algorithm does a linear number of linear scans through this array. So this is definitely a quadratic time algorithm. That's why I didn't bother to tell you about it earlier. So this certainly fits the bill as being a bunch of repeated minimum computations. Or for each computation, we're doing exhaustive search. So this, we should just, a light bulb should go off, and say, aha! Can we do better using a heap data structure? And we can, and the sorting algorithm that we get is called heap sort. And given a heap data structure, this sorting algorithm is totally trivial. We just insert all of the elements from the array into the heap. Then we extract the minimum one by one. From the first extraction, we get the minimum of all N elements. The second extraction gives us the minimum of the remaining N-1 elements, and so on. So when we extract min one by one, we can just populate a sorted array from left to right. Boom, we're done. What is the running time of heap sort? Well, we insert each element once and we extract each element once so that's 2n heap operations and what I promised you is that you can count on heaps being implemented so that every operation takes logarithmic time. So we have a linear number of logarithmic time operations for running time of n log n. So let's take a step back and appreciate what just happened. We took the least imaginative sorting algorithm possible. Selection sort, which is evidently quadratic Time. We recognize the pattern of repeated minimum computations. We swapped in the Heap Data structure and boom we get an NlogN sorting algorithm, which is just two trivial lines. And remember, N log N is a pretty good running time for a sorting algorithm. This is exactly the running time we had for merge sort; this was exactly the average running time we got for randomized quick sort. Moreover, Heap Sort is a comparison based sorting algorithm. We don't use any data about the key elements we just used as a totally [inaudible] set. And, as some of you may have seen in an optional video, there does not exist a comparison-based sorting algorithm with running time better than N log N. So for the question, can we do better? The answer is no, if we use a comparison based sorting algorithm like heap sort. So that's pretty amazing, all we do is swap in a Heap and a running time drops from really quite unsatisfactory quadratic to the optimal N log N. Moreover, HeapSort is a pretty practical sorting algorithm: when you run this it's gonna go really fast. Is it as good as quick sort? Hm, maybe not quite but its close it's getting into the same [inaudible]. So let's talk of another application which frankly in some sense is almost trivial but this is also a canonical way in which heaps are used. And in this application it will be natural to call a heap by a synonymous name, a priority queue. So what I want you to think about for this example is that you've been tasked with writing software that performs a simulation of the physical world. So you might pretend, for example, that you're helping write a video game which is for basketball. Now why would a heap come up in a simulation context? Well, the objects in this application are going to be events records. So an event might be for example that the ball will reach the hoop at a particular time and that would be because a player shot it a couple of seconds ago. When if for example the ball hits the rim, that could trigger another event to be scheduled for the near future which is that a couple players are going to vie for the rebound. That event in turn could trigger the scheduling of another event, which is one of these players? commits, an over the back foul on the other one and knocks them to the ground. That in turn could trigger another event which is the player that got knocked on the ground gets up and argues that a foul call, and so on. So when you have event records like this, there's a very natural key, which is just the timestamp, the time at which this event in the future is scheduled to occur. Now clearly a problem which has to get solved over and over and over again in this kind of simulation is you have to figure out what's the next event that's going to occur. You have to know what other events to schedule; you have to update the screen and so on. So that's a minimum computation. So a very silly thing you could do is just maintain an unordered list of all of the events that have ever been scheduled and do a linear path through them and compute the minimum. But you're gonna be computing minimums over and over and over again, so again that light bulb should go on. And you could say maybe a heap is just what I need for this problem. And indeed it is. So, if you're storing these event records in a heap. With the key being the time stamps then when you extract them in the hands for you on a silver platter using logarithmic time exactly which algorithm is going to occur next. So let's move on to a less obvious application of heaps, which is a problem I'm going to call median maintenance. The way this is gonna work is that you and I are gonna play a little game. So on my side, what I'm going to do is I'm going to pass you index cards, one at a time, where there's a number written on each index card. Your responsibility is to tell me at each time step the median of the number that I've passed you so far. So, after I've given you the first eleven numbers you should tell me as quickly as possible the sixth smallest after I've given you thirteen numbers you should tell me the seventh smallest and so on. Moreover, we know how to compute the median in linear time but the last thing I want is for you to be doing a linear time computation every single time step. [inaudible] I only give you one new number? Do you really have to do linear time just to re-compute the median? If I just gave you one new number. So to make sure that you don't run a linear time selection algorithm every time I give you one new number, I'm going to put a budget on the amount of time that you can use on each time step to tell me the median. And it's going to be logarithmic in the number of numbers I've passed you so far. So I encourage you to pause the video at this point and spend some time thinking about how you would solve this problem. Alright, so hopefully you've thoug ht about this problem a little bit. So let me give you a hint. What if you use two heaps, do you see a good way to solve this problem then. Alright, so let me show you a solution to this problem that makes use of two heaps. The first heap we'll call H low. This equal supports extract max. Remember we discussed that a heap, you could pick whether it supports extract min or extract max. You don't get both, but you can get either one, it doesn't matter. And then we'll have another heap H high which supports extract min. And the key idea is to maintain the invariant that the smallest half of the numbers that you've seen so far are all in the low heap. And the largest half of the numbers that you've seen so far are all in the high heap. So, for example, after you've seen the first ten elements, the smallest five of them should reside in H low, and the biggest five of them should reside in H high. After you've seen twenty elements then the bottom ten, the smallest ten, should, should reside in H low, and the largest ten should reside in H high. If you've seen an odd number, either one can be bigger, it doesn't matter. So if you have 21 you have the smallest ten in the one and the biggest eleven in the other, or vice-versa. It's not, not important. Now given this key idea of splitting the elements in half, according to the two heaps. You need two realizations, which I'll leave for you to check. So first of all, you have to prove you can actually maintain this invariant with only O of log I work in step I. Second of all, you have to realize this invariant allows you to solve the desired problem. So let me just quickly talk through both of these points, and then you can think about it in more detail, on your own time. So let's start with the first one. How can we maintain this invariant, using only log I work and time step I, and this is a little tricky. So let's suppose we've already processed the first twenty numbers, and the smallest ten of them we've all worked hard to, to put only in H low. And the biggest ten of th ''em we've worked hard to put only in H high. Now, here's a preliminary observation. What's true, so what do we know about the maximum element in h low? Well these are the tenth smallest overall and the maximum then is the biggest of the tenth smallest. So that would be a tenth order statistic, so the tenth order overall. Now what about in the, the hi key? What s its minimum value? Well those are the biggest ten values. So the minimum of, of the ten biggest values would be the eleventh order statistic. Okay, so the maximum of H low is the tenth order statistic. The minimum of H high Is the [inaudible] statistic, they're right next to each other; these are in fact the two medians Right now So When this new element comes in, the twenty-first element comes in, we need to know which heap to insert it into and well it just, if it's smaller than the tenth order statistic then it's still gonna be in the bottom, then it's in the bottom half of the elements and needs to go in the low heap. If it's bigger than the eleventh order statistic, if it's bigger than the minimum value of the high heap then that's where it belongs, in the high heap. If it's wedged in between the tenth and eleventh order of statistics, it doesn't matter. We can put it in either one. This is the new median anyways. Now, we're not done yet with this first point, because there's a problem with potential imbalance. So imagine that the twenty-first element comes up and it's less than the maximum of the low heap, so we stick it in the low heap and now that has a population of eleven. And now imagine the twenty-second number comes up and that again is less than the maximum element in the low heap, so again we have to insert it in the low heap. Now we have twelve elements in the low heap, but we only have ten in the right heap. So we don't have a 50. 50, 50 split of the numbers but we could easily re-balance we just extract the max from the low heap and we insert it into the high heap. And boom. Now they both have eleven, and the low heap has the smallest el even, and the high heap has the biggest eleven. So that's how you maintain, the invariant that you have this 50/50 split in terms of the small and the high, and between the two heaps. You check Where it lies with respect to the max of the low heap and the mid of the high heap. You put it in the appropriate place. And whenever you need to do some re-balancing, you do some re-balancing. Now, this uses only a constant number of heap operations when a new number shows up. So that's log I work. So now given this discussion, it's easy to see the second point given that this invariant is true at each time step. How do we compute the median? Well, it's going to be either the maximum of the low heap and/or the minimum of the high heap depending on whether I is even or odd. If it's even, both of those are medians. If I is odd, then it's just whichever heap has one more element than the other one. So the final application we'll talk about in detail in a different video. A video concerned with the running time of Dijkstra's shortest path algorithm. But I do wanna mention it here as well just to reiterate the point of how careful use of data structures can speed up algorithms. Especially when you're doing things like minimum computations in an inner loop. So Dijkstra's shortest path algorithm, hopefully, many of you have watched that video at this point. But basically, what it does is it has a central wild loop. And so it operates once per vertex of the graph. And at least naively, it seems like what each iteration of the wild loop does is an exhaustive search through the edges of the graph, computing a minimum. So if we think about the work performed in this naive implementation, it's exactly in the wheel-house of a heap, right. So what we do in each of these loop iterations is do an exhaustive search computing a minimum. You see repeated minimum computations, a light bulb should go off and you should think maybe a heap can help. And a heap can help in Dijkstra's algorithm. The details are a bit subtle, and they're discussed i n a separate video, but the upshot is, we get a tremendous improvement in the running time. So we're calling that M denotes the number of edges. And N denotes the number of vertices of a graph. With a careful deployment of heaps in Dijkstra's algorithm, the run time drops from this really rather large polynomial. The product of the number of vertices and the number of edges. Down to something which is almost linear time. Anyway, o of m log n. Where m is the number of edges and n is the number of vertices. So the linear time here would be o of m. The liner of the number of edges we're picking up an extra log factor but still this is basically as good as sorting. So this is a fantastically fast shortest path algorithm. Certainly, way, way better that what you get if you don't use heaps and do just repeated exhaustive searches for the minimum. So that, that's wraps up our discussion of what I think you really want to know about heaps. Namely, what are the key operations that it supports? What is the running time you can expect from those operations? What are the types of problems that the data structure will yield speed ups for? And a suite of applications. For those of you that want to take it to the next level and see a little bit about the guts of the implementation, there is a separate optional video that talks a bit about that.

# 12-3-Heaps - Implementation Details [Advanced - Optional]

So, in this video, we're gonna take it at least partway to the next level for the heap data structure. That is, we'll discuss some of the implementation details. I.e., how would you code a, a heat data structure from scratch? So remember what the point of a heap is. It's a container, and it contains objects, and each of these objects, in addition to possibly lots of other data, should have some kind of key. And we should be able to compare the keys of different objects; you know, Social Security numbers for different employers, edge weights for different edges in a network, or time stamps on different events, and so on. Now remember, for any data structure the number one thing you should remember is what are the operations that it supports, i.e., what is the data structure good for. And also, what is the running time you can count on of those operations. So something we promised in a previous video. [inaudible] Indicate the implementation of the miss video is these two primary operations that a heap exports. First of all, you can insert stuff into it, and it takes [inaudible] logarithmic in the number of objects that the heap is storing. And second of all, you can extract an object. That has the minimum key value, and again we're going to allow duplicates in our, in our heaps, so if there's multiple objects, then I'll have a minimum, a common minimum key value, the heap will return one of those. It's unspecified, which one. As I mentioned earlier, you can also dress up heaps with additional operations, like you can do batch inserts, and you can do linear time rather than log in time. You can delete from the middle of the heap. I'm not gonna discuss those in the video; I'm just going to focus on how you'd implement inserts and extract [inaudible]. If you wanna know how heaps really work, it's important to keep in mind simultaneously Two different views of a heap One, as a tree And one, as a, array. So we're gonna start on this slide with the tree view. Conceptually, this will be useful to explain how the heap ope rations are implemented. A conceptual will think of a heap not just as any old tree, but as a tree that's rooted. It'll be binary. Meaning that, each node will have zero, one or two children nodes And third, it will be as complete as possible. So let me draw for your amusement an as complete as possible binary tree that has nine nodes. So if the tree had, had only seven nodes it would have been obvious what, is complete as possible means. It would have meant we just would have had three completely filled in levels. If it had, had fifteen nodes it would have had four completely filled in levels. If you're in between, these two numbers that are powers of two minus one, well we're going to call a complete to the tree as just in the bottom level you fill in the leaves from left to right. So here the two extra leaves on the fourth level are both, pushed as far to the left as possible. So, in our minds, this is how we visualize heaps. Let me next define the heat property. This imposes an ordering on how the different objects are arranged in this tree structure. The heap property dictates that at every single node of this tree it doesn't matter if it's the root if it's a leaf if it's an internal node whatever. At every node X the key of the object stored in X should be no more than the keys of Xs children. Now X may have zero children if it's a leaf it may have one child or it may have two children whatever those cases zero one or two children all those children keys should be at least that of key at X. For example, here is a heap with seven nodes. Notice that I am allowing duplicates. There are three different objects that have the key value four, in this heap. Another thing to realize is that while the heap property imposes useful structure on how the objects can be arranged it in no way uniquely pins down their structure. So this exact same set of seven keys could be arranged differently and it would still be a heap. The important thing is that, in any heap, the root has to have a minimum value key. Just like in the se two organizations of these seven keys, the root is always a four, the minimum key value. So that's a good sign, given that one of the main operations we're supposed to. Quickly implement, is to extract the minimum value. So at least we know where it's going to be, it' gonna be at the root of a heap. So while in or minds we think of heaps as organized in a tree fashion, we don't literally implement them as trees. So in something like search trees, you actually have pointers at each node and you can traverse pointers to go from a [inaudible] to the children from the children to the parents, yada, yada, yada. Turns out it's much more efficient in a heap to just directly implement it as an array. Let me show you by example how a tree like we had on the last slide maps naturally onto an array representation. So let's look at a slightly bigger heap, one that has nine elements. So let me draw an array with nine positions. Labeled one, two, three all the way up to nine And the way we're going to map this tree which is in our mind to this array implementation is really very natural. We're just going to group the nodes of this tree by their level. So, the root is gonna be the only node at level zero. Then the children of the roots are level one, their children constitute level two, and then we have a partial level three, which is just these last two notes here. And now we just stick these notes into the array, one level at a time. So the roots winds up in the privileged first position, so that's going to be the, the first, the object which is the first copy of the four. Then we put in the level one object, so that's the second copy of the four and the eight, and then we put in level two Which has our third four along with the two nines. And then we have the last two notes from level three rounding out the penultimate and final position of the array. And you might be wondering how it is we seem to be having our cake and eating it, too. On the one hand we have this nice, logical tree structure. On the other hand we have this array implementation and we're not wasting any space on the usual pointers you would have in a tree to traverse between parents and children. So where's the free lunch coming from? Well the reason is that because we're able to keep this binary tree as balanced as possible, we don't actually need pointers to figure out who is whose parent and who is whose child. We can just read that off directly From the positions in the array. So, let me be a little bit more specific. If you have a node in the fifth position, I'm assuming I here is not one, right? So the, the root doesn't have any, does not have a parent But any other, any other objects in position I does have a parent And what the position that is, depends on, in a simple way on whether I is even or odd. So if I is even, then the parent is just [inaudible] the position of I/2, and if I is odd, then it's going to be I/2. Okay, that's a fraction. So we take the floor that is we round down to the nearest integer. If I is odd, So for example, the objects in positions two and three have as their parent the object in position one, and those in four and five have the one in position two as his parent. Six and seven have as their parents the node in, the object in position three and so on And of course we can invert this function we can equally easily determine who the children are of a given node so if we have an object in position I. Then you notice that the children of I are gonna be at the position 2i and 2i+1. Of course those may be empty so if you have a leaf of course that doesn't have any children and then maybe one node that has only one child. But in the common case of an internal node it's gonna be two children that are gonna be in positions 2i and 2i+1. So rather than traversing pointers it's very easy to just go from a node to its parent to either one of its children just by doing these appropriate trivial calculations with respects to its position. So this slide illustrates, some of the. Lower level reasons that heaps are quite a popular data struct ure So the first one, just in terms of storage. We don't need any overhead at all. We are just. We have these objects; we're storing them directly In an array, with no extra space. Second of all, not only do we not have to have a space for pointers But we don't even have to do any traversing. All we do are these really simple. Divide by two or multiply two operations And using bit shifting tricks. Those can also be implemented extremely quickly. So, the next two slides let me indicate, at a high level, how you would implement the two exported operations, namely insertion and extract [inaudible] in time algorithmic in the size of the heap And rather than give you any pseudo code, I'm just going to show you how these work by example. I think it will be obvious how they extended the general case. I think just based on this discussion, you'll be able to code up your own versions, of insert and extract [inaudible], if you so desire. So let's redraw the 9-node heap that we had on the previous slide and again, I'm gonna keep drawing it as a tree and I'm gonna keep talking about it as a tree but always keep in mind the way it's really implemented is in terms of these array and when I talk about the parent of a node, again, what that means is you go to the appropriate position given the position of the node from where you started. So let's suppose we have an existing heap, like this blue heap here And we're called upon to insert a new object. Let's say with a key value K. Now remember heaps are always suppose to be perfectly balanced binary trees. So if we want to maintain the property that this tree is perfectly balanced is pretty only one place we can try to put the new key K and that's as the next leaf. That is it's going to be the new right most leaf on the bottom level. Or in terms of the array implementation we just stick it in the first non empty slot in the array And if we keep track of the array size we're getting constant time of course know where to put the new key. Now whether or not we can get away with this depends on what the actual key value K is But, you know, for starters we can say, what if we insert a key that's seven? Well, then we get to say, whew, we're done. So, the reason we're done is cuz we have not violated the heap property. It is still the case that every node has key no bigger than that of its children. In particular, this third copy of a four, it picked up a new child, but its key seven was bigger than its key four. So, you can imagine that maybe we get lucky with another insert. Maybe the next insertion is a ten And again, we put that in the next available spot in the last level And that becomes the second child of the third copy of the four And again we have no violation of the heap property. No worries still got a heap And in these lucky events insertion is even taking constant time. Really all we're doing is putting elements at the end of an array and not doing any rearranging. So where it gets interesting is when we do an insertion that violates the heap property. So let's supposed we do yet another insertion, and the left child of this twelve, it becomes a five. Now we got a problem. So now we have a as perfectly as possible balanced binary tree, but the key property is not satisfied. In particular, it's violated at the node twelve. It has one child. The key of that child is less than its own key. That's no good. So is there some way we can restore the heap property? Well, a natural idea is just to swap the positions of the five and the twelve, and that's something that of course can be done in constant time, cuz again from a node, we can go to the parent or the child in constant time just with a suitable trivial computation. So we say okay, for starters we put the five at the end, but that's no good. So we're gonna swap the five with the twelve And now we see we're not out of the woods. No longer is there a heap violation at the node twelve. That's been fixed. We've made it a leaf But we've Pushed up the heap violations. So now instead it's the eight that has a problem. The eight used to h ave two children, with keys twelve and nine that was fine. Now the eight has two children with keys five and nine. The five is less than the eight, that's a violation of the heap property But again, that's the only violation of the heap property. There's no other node you could have screwed up, because eight was the only person whose children we messed around with. Alright So now we just it again. Let's try [inaudible] again, locally fix the heap violation by swapping the five with the 8s And now we see we've restored order. The only place where there could possibly be a violation of the heap property is at the root. The root, when we did this swap, the only person whose children we really messed with was the root four, and fortunately its new child has key five, which is bigger than it. One subtle point that you might be thinking that in addition to screwing up at the root node, that messing around with his children, maybe we could have screwed up the twill by messing around with its parent. Alright, its parent used to be five, and now its parent is eight. So is there some possibility that his parent would all of a sudden have a key bigger than it But if you think about it, this eight and this twelve, they were a parent-child relationship in the original heap Right? So back in the blue heap, the twelve was under the 8s. Now the twelve is under the eight yet again. Since we have the heap property for that pair before, we certainly have it now. So in general, as you push up this five up the tree, there's only going to be one possible edge that could be out of order And that's between where the five currently resides and whatever its parent is. So when the 5's parent was twelve that was a violation When 5's parent was eight that was a violation But now that we've pushed it up two levels and 5's parents is four, that's not a violation because four is less than five. So in general, step two of insertion is, you do this swap, which it's called a lot of different things. I'm gonna call it bubble up because that's how I learned it more years ago than I care to admit But also this is called, sometimes sift up, happily up, and so on. So now told you to just how to implement insertions by repeated bubbling ups in a heap data structure And this is really how it works, there is nothing I haven't told you but you know, I'm not going to really fill in all the details but I'll encourage you to do that on your own time, if it's something that interests you And the two main things that you should check is first of all, is bubbling up process is gonna stop and when it stops, it stops with the heap property restored. The second thing needs to be checked in this, I think is easier to see is that we do have the desire one time log [inaudible] make in the number of elements in the heap. The key observations areas that because this is a perfectly balanced binary tree. We know exactly how many levels there are. So this is basically log based two event levels where n is the number of items in the heap And what is the running time of this insertion procedure while you only do a constant amount of work at each level, just doing the swap and comparison and then in the worst case, you'll have to swap at every single level and there is a lot [inaudible] number of levels. So that's insertion. Let's now talk about how do we implement the extract [inaudible] operation and again I'm gonna do this by example and it's gonna be by repeating the [inaudible] of a bubble [inaudible] procedure. So the Extract main operation is responsible for removing from the heap an object with minimum key value and handing it back to the client on a silver platter. So it pretty much have to whip out the root. Remember the minimum is guaranteed to be at the root. So that's how we have to begin to extract the subroutine is just we pluck out the root and hand it back to the client. So this removal of course leaves a gaping hole in our tree structure and that's no good. One of the [inaudible] responsible for maintaining is that we always have an as perfectly balanced as possib le binary tree And if you are missing a root you certainly don't have an almost perfect. Binary tree So, what are we going to do about it? How do we fill this hole? Well, there's pretty much only one node that could fill this hole without causing other problems with the tree structure, and that is the very last node. So the rightmost leaf at the bottom level one that simple fix is to swap that up and have that take the place of the original root. So in this case, the thirteen is going to get a massive promotion And get teleported all the way to be the new root of this tree. So now we've resolved our structural challenges. We now again have a, as perfectly balanced as possible, binary tree But of course now we've totally screwed up the heap property, right. So the heap property says that at every node, including the root, the key value at that node has to be less than both of the children, and now it's all messed up. Right, so at the root, the key value's actually bigger than both of the children. And matters that are little bit more tricky than they were with insertion, right, when we inserted at the bottom because every note has a unique parent. If you wanna push a note upward in the tree, there's sort of only one place you can go, right, all you can do is swap with your parent, unless you're going to try to do something really crazy But if you want to do something local, pretty much you only have a unique parent you got to swap with. Now when you're trying to push notes down to the rightful position in the tree, there is two different swaps you could do, one for the left child, one for the right child and the decision that we make matters To see that concretely, let's think about this example. There's this thirteen at the root, which is totally not where it should be, and there's the two children. The four and the eight, and we could try swapping it with either one. So suppose we swap it in a misguided way with the right child, with the eight. So now the eight becomes the root, and the thirteen gets pushed dow n a level. So on the one hand; we made some progress because now at least we don't have a violation between the thirteen and the eight. On the other hand, we still have violations involving the thirteen. The thirteen is still violated with respect to the twelve and nine And moreover, we've created a new problem between the eight and the four, right? So now that the eight is the root, that's still bigger than its left child, this four. So it's not even clear we made any progress at all when we swapped the thirteen with the eight, okay? So that was a bad idea And if you think about it would made it a bad idea, the stupid thing was to swap it with the larger child. That doesn't make any sense. We really want to swap it with the smaller child. Remember, every node should have a key bigger than both of its children. So if we're going to swap up either the four or the eight, one of those is going to become the parent of the other. The parent is supposed to be smaller, so evidently we should take the smaller of the two children and swap the thirteen with that. So we should swap the thirteen with the figure. Not with E And now we observe a phenomenon very much analogous to what we saw in insert. When we were bubbling up during insertion, it wasn't necessarily that we fixed violations of the heat property right away But we would fix one And then introduce another one that was higher up in the tree And we had confidence that eventually we could just, push this violation up to the root of the tree And squash it, just like we're trying to win a game Of Whack a Mole. Here, it's the opposite. It's just in the opposite direction. So we swap the thirteen with the four. It's true we've created one new violation of the heap property. That's again involving the thirteen with its children nine and four. But we haven't created any new ones. We've pushed the heap violation further down the tree And hopefully again, like in Whack a Mole. We'll squash it At the bottom. So after swapping the thirteen and the four, now we just gotta do t he same thing. We say, okay, we're not done. We still don't have a heap. This thirteen is bigger than both of its children But now, with our accumulated wisdom, we know we should definitely swap the thirteen with the four. We're not gonna try swapping with the nine, that's for sure. So we move the four up here And we, the thirteen takes the 4's old place. Boom! Now we're done. So now we have no violations remaining. The thirteen in its new position has no children so there's no way it can have any violations, and the four because it was the smaller child that's gonna be bigger than the 9's so we haven't introduced a huge violation there, and again we have these consecutive 4's but we know that's not gonna be a problem because those were consecutive 4's in the original heap as well. So you won't be surprised to hear that this procedure by which you push something down, by swapping it with its smaller children, is called bubble down, and extract men is nothing more than taking more than, taking this last leaf, promoting it to the top of the tree, and bubbling down until the heap violation has been fixed. So again on a conceptual level that's all of the ingredients necessary for a complete from scratch implementation of extracting the minimum from a heap and as before, I'll leave it for you to check the details. So first of all you should check that in fact this bubble down has to at some point halt And when it halts you do have a bona fide heap. The heap property is definitely restored And second of all the running time is, is logarithmic. Here the running time analysis is exactly the same as before so we already have observed that the heights of a heap because it's perfectly balanced is essentially the log base two of the number of elements in the heap and in bubbling down all you do is a constant amount of work per level. All you have to do is a couple comparisons and swap. So, that's a peek at what's under the hood in the heap data structure. A little bit about the guts of its elementation. So after having seen this, I hope you feel like a little bit more hard-core of a programmer, a little bit more hard-core of a computer scientist.

# 13-1-Balanced Search Trees - Operations and Applications

In this sequence of videos, we'll discuss our last but not least data structure namely the Balanced Binary Search Tree. Like our discussion of other data structures we'll begin with the what. That is we'll take the client's perspective and we'll ask what operations are supported by this data structure, what can you actually use it for? Then we'll move on to the how and the why. We'll peer under the hood of the data structure and look at how it's actually implemented and then understanding the implementation to understand why the operations have the running times that they do. So what is a Balanced Binary Search Tree good for? Well, I recommend thinking about it as a dynamic version of a sorted array. That is, if you have data store in a Balanced Binary Search Tree, you can do pretty much anything on the data that you could if it was just the static sorted array. But in addition, the data structure can accommodate insertions and deletions. You can accommodate a dynamic set of data that you're storing overtime. So to motivate the operations that a Balanced Binary Search Tree supports, let's just start with the sorted array and look at some of the things you can easily do with data that happens to be stored in such a way. So let's think about an array that has numerical data although, generally as we've said, in data structures is usually associated other data that's what you actually care about and the numbers are just some unique identifier for each of the records. So these might be an employee ID number, social security numbers, packet ID numbers and network contacts, etcetera. So what are some things that are easy to do given that your data is stored as a sorted array, most a bunch of things? First of all, you can search and recall that searching in a sorted array is generally done using binary search so this is how we used to look up phone numbers when we have physical phone books. You'd start in the middle of the phone book, if the name you were looking for was less than the midpoint, you recurse on the left hand side, otherwise you'd recurse on the right hand side. As we discussed back in the Master Method Lectures long ago, this is going to run in logarithmic time. Roughly speaking, every time you recurse, you've thrown out half of the array so you're guaranteed to terminate within a logarithmic number of iterations so binary search is logarithmic search time. Something else we discussed in previous lectures is the selection problem. So previously, we discussed this in much harder context of unsorted arrays. Remember, the selection problem in addition to array you're given in order statistic. So, if your order statistic that your target is seventeen, that means you're looking for the seventeenth smallest number that's stored in the array. So in previous lectures, we worked very hard to get a linear time algorithm for this problem in unsorted arrays. Now, in a sorted array, you want to know the seventeenth smallest element in the array. Pretty easy problem, just return whatever element happens to be in the seventeenth position of the array since the array is sorted, that's where it is so no problem. It's already sorted constant time, you can solve the selection problem. Of course, two special cases of the selection problem are finding the minimum element of the array. That's just if the order statistic problem with i = 1and the maximum element, that's just i = n. So this just corresponds to returning the element that's in the first position and the last position of the array respectively. Well let's do some more brainstorming. What other operations could we implement on a sorted array? Well here's a couple more. So there are operations called the Predecessor and Successor operations. And so the way these work is, you start with one element. So, say you start with a pointer to the 23, and you want to know where in this array is the next smallest element. That's the predecessor query and the successor operation returns the next largest element in the array. So the predecessor of the 23 is the seventeen, the successor of the 23 would be the 30. And again in a sorted array, these are trivial, right? You just know that predecessors just one position back in the array, the successor is one position forward. So given a pointer to the 23, you can return to 17 or the 30 in constant time. What else? Well, how about the rank operation? So we haven't discussed this operation in the past. So what rank is, this has for how many key stored in the data structure are less than or equal to a given key. So for example, the rank of 23 would be equal to 6. Because 6 of the 8 elements in the array are less than or equal to 23. And if you think about it, implementing the rank operation is really no harder than implementing search. All you do is search for the given key and wherever it is search terminates in the array. You just look at the position in the array and boom, that's the rank of that element. So for example, if you do a binary search for 23 and then when you terminates, you discover it is, they're in position number six then you know the rank is six. If you do an unsuccessful search, say you search for 21, well then you get stuck in between the 17 and the 23, and at that point you can conclude that the rank of 21 in this array is five. Let me just wrap up the list with the final operation which is trivial to implement in the sorted array. Namely, you can output or print say the stored keys in sorted order let's say from smallest to largest. And naturally, all you do here is a single scan from left to right through the array, outputting whatever element you see next. The time required is constant per element or linear overall. So that's a quite impressive list of supported operations. Could you really be so greedy as to want still more from our data structure? Well yeah, certainly. We definitely want more than just what we have on the slide. The reason being, these are operations that operate on a static data set which is not changing overtime. But the world in general is dynamic. For example, if you are running a company and keeping track of the employees, sometimes you get new employees, sometimes employees leave. That is one of the data structure that not only supports these kinds of operations but also, insertions and deletions. Now of course it's not that it's impossible to implement insert or delete in a sorted array, it's just that they're going to run way too slow. In general, you have to copy over a linear amount of stuff on an insertion or deletion if you want to maintain the sorted array property. So this linear time performance when insertion and deletion is unacceptable unless you barely ever do those operations. So, the raison d'etre of the Balanced Binary Search Tree is to implement this exact same set of operations just as rich as that's supported by a sorted array but in addition, insertions and deletions. Now, a few of these operations won't be quite as fast or we have to give up a little bit instead of constant time, the one in logarithmic time and we still got logarithmic time for all of these operations, linear time for outputting the elements in sort of order plus, we'll be able to insert and delete in logarithmic time so let me just spell that out in a little more detail. So, a Balanced Binary Search Tree will act like a sorted array plus, it will have fast, meaning logarithmic time inserts and deletes. So let's go ahead and spell out all of those operations. So search is going to run in O(log n) time, just like before. Select runs in constant time in a sorted array and here it's going to take logarithmic, so we'll give up a little bit on the selection problem but we'll still be able to do it quite quickly. Even on the special cases of finding the minimum or finding the maximum in our, in our data structure, we're going to need logarithmic time in general. Same thing for finding predecessors and successors they're not, they're no longer constant time, they go with logarithmic. Rank took as logarithmic time and the, even the sorted array version and that will remain logarithmic here. As we'll see, we lose essentially nothing over the sorted array, if we want to output the key values in sorted order say from smallest to largest. And crucially, we have two more fast operations compared to the sorted array of data structure. We can insert stuff so if you hire a new employee, you can insert them into your data structure. If an employee decides to leave, you can remove them from the data structure. You do not have to spend linear time like you did for sort of array, you only have to spend the logarithmic time whereas always n is the number of keys being stored in the data structure. So the key takeaway here is that, if you have data and it has keys which come from a totally ordered set like, say numeric keys, then a Balanced Binary Search Tree supports a very rich collection of operations. So if you anticipate doing a lot of different processing using the ordering information of all of these keys, then you really might want to consider a Balanced Binary Search Tree to maintain them. Well then, keep in mind though is that we have seen a couple of other data structures which don't do quite as much as balanced binary search trees but what they do, they do better. We already, we just discussed in the last slide of the sorted array. So, if you have a static data set, you don't need inserts and deletes. Well then by all means, don't bother with Balanced Binary Search Tree that use a sorted array because it will do everything super fast. But, we also sought through dynamic data structures which don't do as much but do it, but what they do, they do very well. So, we saw a heap, so what the heap is good for is it's just as dynamic as a search tree. It allows insertions and deletions both in logarithmic time. And in addition, it keeps track of the minimum element or the maximum element. Remember in a heap, you can choose whether you want to keep track of the minimum or keep track of the maximum but unlike in a search tree, a heap does not simultaneously keep track of the minimum and the maximum. So if you just need those three operations, insertions, deletions and remembering the smallest, and this would be the case for example in a priority queue or scheduling application as discussed in the heap videos. Then, a Binary Search Tree is over kill. You might want to consider a heap instead. In fact, the benefits of a heap don't show up in the big O notation here both have logarithmic operation time but the constant factors both in space and time are going to be faster with a heap then with a Balanced Binary Search Tree. The other dynamic data structure that we discussed is a hash table. And what hash tables are really, really good at is handling insertions and searches, that is look ups. Some, sometimes, depending on the implementation also handle deletions really well also. So, if you don't actually need to remember things like minima, maxima or remember ordering information on the keys, you just have to remember what's there and what's not. Then the data structure of choice is definitely the hash table, not the balance binary search tree. Again, the Balance Binary Search Tree would be fine and we'd give you logarithmic look up time but it's kind of over kill for the problem. All you need is fast look ups. A hash table recall will give you constant time look ups. So that will be a noticeable win over the Balanced Binary Search Tree. But if you want a very rich set of operations for processing your data. Then, the Balanced Binary Search Tree could be the optimal data structure for your needs.

# 13-2-Binary Search Tree Basics, Part I

So, in this video, we'll go over the basics behind implementing binary search trees. We are not going to focus on the balance aspect in this video that will be discussed in later videos and we are going to talk about things which are true for binary search trees on general, balanced or otherwise. But let's just recall, you know, why are we doing this, you know, what is the raison d'锚tre of this data structure, the balance version of the binary search tree and basically, its a dynamic version of a sorted array. So, that's pretty much everything you can do on a sorted array, maybe in slightly more expensive time. They are still really fast but in addition to this dynamic, it accommodates insertions and deletions. So, remember, if you want to keep a sorted array data structure, every time you insert, every time you delete, you're probably going to wind up paying a linear factor which is way too expensive in most applications. By contrast with the search tree, a balanced version, you can insert and delete a logarithmic time in the number of keys in the tree. And moreover, you can do stuff like search in logarithmic time, no more expensive than binary search on a sorted array and also you can sort of say the selection problem in the special cases, the minimum or maximum. Okay, it's not constant time like in a sorted array but still logarithmic pretty good and in addition, you can print out all of the keys from smallest to largest and in linear time, constant time per element just like you could with the linear scan through a sorted array. So, that's what they're good for. Everything a sorted array can do more or less plus insertions and deletions everything in logarithmic time. So, how are search trees organized? And again, what I'm going to say in the rest of this video is true both for balanced and unbalanced search trees. We're going to worry about the balancing aspect in the later videos. Alright, so, let me tell you the key ingredients in a binary search tree. Let me also just draw a simple cartoon example in the upper right part of the slide. So, this one to one correspondence between nodes of the tree and keys that are being stored. And as usual in our data structure discussions we're going to act as if the only thing that we care about, the only thing that exists at each node is this key when generally, this associated data that you really care about. So, each node in the tree will generally contain both the key plus a pointer to some data structure that has more information. Maybe the key is the employee ID number, and then there's a pointer to lots of other information about that employee. Now, in addition to the nodes, you have to have links amongst the nodes and there's a lot of different ways to do the exact implementation of the pointers that connect the node of the tree together but the video I'm just going to keep is straightforward as possible and we're just going to assume that in each node, there's three pointers. One to a left child, another one to the right child and then the third pointer which points to the parent. Now, of course, some of these pointers can be null and in fact in the five node binary search tree I've drawn on the right for each of the five nodes, at least one of these three pointers is null. So, for example, for the node with key one it has a null left child pointer, there was no left child. It's the right child pointer going to point to the node with key two and the parent pointer was going to a node that has key three. Similarly three is going to have a null parent pointer and the root node in this case, three is a unique node but has a null parent pointer. Here the node with key value three, of course, has a left child pointer points to one and has a right child pointer that points to five. Now, here is the most fundamental property of search trees. Let's just go ahead and call it the Search Tree Property. So, the search tree property asserts the following condition at every single node of the search tree. If the node has some key value then all of the keys stored in the left subtree should be less than that key. And similarly, all of the keys stored in the right subtree should be bigger than that key. So, if we have some node who's stored key value is x and this is somewhere, you know, say deep in the middle of the tree so upward we think of as being toward the root. And then if we think about all the nodes that are reachable, after following the left child pointer from x, that's the left subtree. And similarly, the right subtree being everything reachable via the right child pointer from x, it should be the case that all keys in the left subtree are less than x and all keys in the right subtree are bigger than x. And again, I want to emphasize this property holds not just to the root but at every single node in the tree. I've defined the search to a property assuming that all of the keys are distinct, that's why I wrote strictly less than in the left sub tree and strictly bigger than in the right subtree. But search trees can easily accommodate duplicate keys as well. We just have to have some convention about how you handle ties. So, for example, you could say that everything in the left subtree is less than or equal to the key at that node and then everything in the right subtree should be strictly bigger than that node. That works fine as well. So, if this is the first time you've ever heard of the search tree property, maybe at first blush it seems a little arbitrary. It seems like I pulled it out of thin air but actually, you would have reversed engineer this property if you sat down and thought about what property would make search really easy in a data structure. The point is, the search tree property tells you exactly where to look for some given key. So, looking ahead a little bit, stealing my fire from a slide to come, suppose you were looking for say, a key 23, and you started the root and the root is seventeen. The point of the search tree property is you know where 23 has to be. If the root is seventeen, you're looking to 23, if it's in the tree, no way is it in the left subtree, it's got to be in the right subtree. So, you can just follow the right child pointer and forget about the left subtree for the rest of the search. This is very much in the spirit of binary search where you start in the middle of the array and again, you compare what you're looking for to what's in the middle and either way, you can recurse on one of the two sides forgetting forevermore about the other half of the array and that's exactly the point of the search tree property. We're going to have to search from root on down, the search tree property guarantees we have a unique direction to go next and we never have to worry about any of the stuff that we don't see. We can also draw a very loose analogy with our discussion of heaps and may recall heaps were also logically, we thought of them as a tree even though they are implemented as an array. And heaps have some heap property and if you go back to review the heap property, you'll find that this is not the same thing as the search three property. These are two different properties and that's going to trying to make different things easy. Back when we talk about heaps, the property was that this is for the extract min version. Parents always have to be smaller than their children. That's different than the search tree property which says stuff to the left, that's smaller than you, stuff to the right is bigger than you. And heaps, we have the heap property so that identifying the minimum value was trivial. It was guaranteed to be at the root. Heaps are designed so that you can find the minimum easily. Search trees are, are defined so that you can search easily that's why, you have this different search tree property. If you want to get smaller, you go left. If you want to get bigger you go right. One point that's important to understand early, and this will be particularly relevant once we did, once we try to enforce balancing in our subsequent videos is that, for a given set of keys, you can have a lot of different search trees. On the previous slide , I drew one search tree containing the key values one, two, three, four, five. Let me redraw that exact same search tree here. If you stare to this tree a little while you'll agree that in fact that every single node of this tree, all of the things in the left subtree are smaller, all of the things in the right subtree are bigger. However, let me show you another valid binary search tree with the exact same sets of keys. So, in the second search three, the root is five, the maximum value. And everybody has no right children, only the left children are populated and that goes five, four, three, two, one in descending order. If you check here again, it has the property that at every node, everything in the left subtree is smaller. Everything in the right subtree, in this case, empty, is bigger. So, extrapolating from these two cartoon examples, we surmised that for a given set of n keys, search trees that contain these keys could vary in height anywhere from the best case scenario of a perfectly balance binary tree which just going to have logarithmic height to the worst case of one of these link list like chain which is going to be linear in the number of keys n. And so just to remind you the height of a search tree which is also sometimes called the depth is just the longest number of hops it ever take to get to from a root to a leaf. So, in the first search tree, here the height is two and then the second search tree, the height is four. If the search tree is perfectly arranged with the number of nodes essentially doubling at every level, then the depth is you're going to run out of nodes around the depth of log2n. And in general, if you have a chain of n keys that that's going to be n - 1 but we'll just call it n amongst friends. So, now that we understand the basic structure of binary search trees, we can actually talk about how to implement all of the operations that they support. So, as we go through most of the supported operations one at a time, I'm just going to give you a really high level description. It should be enough for you to code up on implementation if you want or as usual, if you want more details or actual working code, you can check on the web or in one of the number of good programming or algorithms textbooks. So, let's start with really the primary operation which is search. Searching, we've really already discussed how it's done when we discuss the search tree property. Again, the search tree property makes it obvious how to look for something in a search tree. Pretty much you just follow your nose you have no other choice. So, you started the root, it's the obvious place to start. If you're lucky, the root is what you are looking for and then you stop and then you return to root. More likely, the root is either bigger than or less than the key that you're looking for. Now, if the key is smaller, the key you are looking for is smaller than the key of the root, where you're going to look? Well, the search tree property says, if it's in the tree, it's got to be in the left subtree so you follow the left sub child pointer. If the key you're looking for is bigger than the key at the root, where is it got to be? Got to be in the right subtree. So, you're just going to recurse on the right subtree. So, in this example, if you're searching for, say the key two, obviously you're going to go left from the root. If you're searching for the key four, obviously you're going to go right from the root. So, how can the search terminate? Well, it can terminate in one of two ways. First of all, you might find what you're looking for so in this example, if you search for four, you're going to traverse to right child pointer then a left child pointer and then boom, you're at the four and you return successfully. The other way the search can terminate is with a null pointer. So, in this example, suppose you were looking for a node with key six, what would happen? Well, you start at the root, three is too small so you go to the right. You get to five, five is still too small cuz you're looking for six so you try to go right but the right child pointer is null. And that means six is not in the tree. If there was anywhere in the tree, it had to be to the right of the three, it had to be to the right of the five but you tried it and you ran on the pointers so the six isn't there. And you return correctly with an unsuccessful search. Next, let's discuss the insert operation which really is just a simple piggy backing on the search that we just described. So, for simplicity the first think about the case where there are no duplicate keys. The first thing to do on this insertion is search for the key k. Now, because there are no duplicates, this search will not succeed. This key k is not yet in the tree. So, for example, in the picture on the right, we might think about trying to insert the key six. What's going to happen when we search for six, we follow a right child pointer. We go from three to five and then we try to spot another one and make it stuck. There's a null pointer going to the right of five. Then when this unsuccessful search terminates at a null pointer, we just rewire that pointer to point to a node with this new key k. So, if you want to permit duplicates from the data structure, you got to tweak the code and insert a little bit but really barely at all. You just need some convention for handling the case when you do in counter the key that we are about to insert. So, for example, if the current note has the key equal to the one you're inserting, you could have the convention that you always continue on the left subtree and then you continue the search as usual again, eventually terminating at a null pointer and you stick the new inserted node you rewire to null pointer to point to it. One good exercise for you to think through which I'm not going to say more about here is that when you insert a new node, you retain the search tree property. That is if you start with the search tree, you start within tree where at every node stuff to the left is smaller, the stuff to the right is bigger. You insert something and you follow this procedure. You will still have the search tree property after this new node has been inserted. That's something for you to think through.

# 13-3-Binary Search Tree Basics, Part II

So what I want to do next is test your understanding about the search and insertion procedures by asking you about their running time. So of the following four parameters of a search tree that contains n different keys, which one governs the worst case time of a search or insertion. So the correct answer is the third one. So, the heights of a search tree governs the worst case time of the search or of an insertion. Notice that means merely knowing the number of keys n is not enough to deduce what the worst case search time is. You also have to know something about the structure of the tree. So, to see that, just let's think about the two examples that we've been running so far. One of which is nice and balanced. And the other of which, which contains exactly the same five keys is super unbalanced, It's this crazy linked list in effect. So, in any search tree, the worst case time to do is search or insertion is proportional to the largest number of pointers left to right child pointer that you might have to follow to get from the root all the way to a null pointer. Of course in a successful search, you're going to terminate before you encounter a null pointer but in the worst case, you want insertion you go all the way to a null pointer. Now on the tree on the left you're going to follow at most 3 such pointers. So for example, if you're searching for 2.5. You're going to follow a left pointer followed by a right pointer. By another pointer and that one is going to be null. So we're going to follow three pointers. On the other hand, in the right tree, you might follow as many as five pointers before that fifth pointer is null. For example, if you search for the key zero, you're going to traverse five left pointers in a row and then you're finally going to encounter the null at the end. So, it is not constant time certainly, you have to get to the bottom of the tree. It's going to be from proportional to logarithmic, logarithm in the number of keys if you have a nicely balanced binary search tree like this one on the left. It's going to be proportional to the number of keys n as in the fourth answer if you have a really lousy search tree like this one on the right and in general. Search time or the insertion time is going to be proportional to the height. The largest number of hops we need to take to get from the root to the leaf of the tree. Let's move on to some more operations that search tree support but that, for example, the dynamics data structures of heaps and hash tables do not. So let's start with the minimum and the maximum. So, by contrast and a heap remember, you can choose one or the two. You can either find the minimum, usually you find the maximum easily but not both. And the search tree is really easy to find, either the min or the max. So, let's start with the minimum. One way to think of it is that you do a search for negative infinity in the search tree. So, you started the root. And you just keep following left child pointers until you run out, until you hit a null. And whatever the last key that you visit has to be the smallest key of the tree, right? Because, think about it, suppose you started the root. Supposed that the root was not the minimum, then where is the minimum got to be, It's got to be in the left sub-tree so you follow the left child pointer and then you just repeat the argument. If you haven't already found the minimum, where it's got to be with respect to current place, it's got to be in the left sub tree and you just iterate until you can't go to the left any further. So for example, in our running search tree. You'll notice that if we just keep following left child pointers, we'll start at the three, we'll go to the one, we'll try to go left from the one. We'll hit a null pointer and we'll return one and one is indeed the minimum key in this tree. Now, given that we've gone over how to compute the minimum, no prizes to guess how we compute the maximum. Of course, if we want to compute the maximum instead of following left child pointers we follow right child pointers by symmetric reasoning as guaranteed upon the largest key in the tree. It's like searching for the key plus infinity. Alright. So what about computing the predecessor? So remember this means you're given key in the tree, in the element of the tree and you want to find the next smallest element so for example the predecessor of the three is two. The predecessor of the two in this tree is the one. The predecessor of the five is the four. The predecessor of the four is the three. So, here I'll be a little hand wavy just in the interest of getting through all of the operations in reasonable amount of time but let me just point out that there is one really easy case and then there is one slightly trickier case. So the easy case. Is when the node with the key k has a non-empty left sub tree. If that's the case, then what you want is simply the biggest element in this node left sub tree. So, I'll leave it for you to prove formally that this is indeed the correct way to compute predecessors for keys that do have a non-empty left sub tree, let's just verify in our example by going through the trees that have a left sub tree and checking this is in fact what we want. Now, if you look at it, there's actually only two nodes that have a non-empty left sub tree. The three has a non-empty left sub tree and indeed the largest key in the left sub tree three is the two and that is the predecessor of the three so that worked out fine. And then the other node with a non-empty left subtree is the five and it's left subtree is simply the element four of course the maximum of that tree is also four. And then you'll notice that is indeed the predecessor of five in this entire search tree. So, the trickier case is what happens if you know the key with no left subtree at all. Okay. So, what are you going to do if you not in the easy case, Well, given at this node with key k, you only have three pointers and by assumption, the left one is null so that's not going to get you anywhere, now, the right childpointer if you think about it is totally pointless for computing the predecessor. Remember, the predecessor is going to be a key less than the given key k. The right subtree by definition of a search tree only has keys that are bigger than k. So, it stands for reason to find the predecessor we got to follow the parent pointer. Maybe in fact more than one parent pointer so to motivate exactly how we're going to follow parent pointers, let's look at a couple of examples in our favorite search tree here on the right. So, let's start with a node two. So, we know we got to follow a parent pointer. When we follow to this parent pointer, we get to one and boom, one in fact is two's predecessor in this tree so that was really easy to computer two's predecessor. It seemed that all we have to do is follow the parent pointer. So, for another example though which think about the node four. Now, four when we follow which parent pointer, we get to five and. Five is not 4's predecessor, it's 4's successor. What we wanted a key that is less than where we started, we follow the parent pointer and it was bigger. But, if we follow one more parent pointer, then we get to the three. So, from the two we needed to follow one parent pointer, from the four we needed to follow two parent pointers. But the point is, you just need to follow parent pointers until you get to a node with key smaller than your own. And at that point you can stop and that's guaranteed to be the predecessor. So, hopefully, you would find this intuitive. I should say, I have definitely not formally proved that this works and that is a good exercise for those of you that want to have a deeper understanding of search trees and this magical search tree property and all of the structure that it grants you. The other thing I should mention is another way to interpret the, the terminating criteria. So what I've said is you stop your search of parent pointers as soon as you get to through smaller than yours If you think it about a little bit, you'll realize you'll get to a key smaller than yours, the very first time you take a left turn. So the very first time that you go from a right child to it's parent. Look at the example, when we started from two, we took a left turn, right? We went from upper link going leftward To it's a right child of one, and that's when we got to the predecessor in just one step. By contrast when we started from the four, our first step was to the right. So, we got to a node that was bigger than where we started for five is four's left child which is going to be smaller than five. But the first time we took a left turn on the next step, we got to a node that is not only smaller than five but actually smaller from four, smaller from the starting point. So, in fact, you're going to see a key smaller than your starting point at very first time, you take a left turn, the very first time you go from a node to a parent and in fact, that node is that parent's right child. So this is another statement which I think is intuitive but which formally is not totally obvious. And again I encourage you to think carefully about why these two descriptions of the terminating criteria are exactly the same so it doesn't matter if you stop when you first find a key smaller than your starting point. It doesn't matter if you first stop when you follow a parent pointer that goes from a node that's the right child of a node. Either way you're going to stop at exactly the same time so I encourage you to think about why those are the exact same stopping condition. A couple of other details if you start from the unique node that has no predecessor at all, you'll never going to trigger this terminating condition so for example if you start from the node one in the search tree, not only is the left subtree empty which says you're suppose to start traversing parent pointers but then when you traverse a parent pointer, you only go to the right. You never turn left and that's because there is no predecessor so that's how you detect that you're at the minimum of a search tree. And then of course if you wanted to be the successor of the key instead of the predecessor, obviously you just flip left and right through out this entire description. So that's the high level explanation of all of these different ordering operations, minimum and maximum predecessor and successor work in a search tree. Let me ask you the same question I asked you when we talked about search in insertion. How long that these operations take in the worst case? Well, the answer is the same as it was before. It's proportional to the height of the tree and the explanation is exactly the same as it was before. So to understand the dependence on the height was just focused on the maximum operation that has the state within the question. The other three operations, the running time is proportional to the height in the worst case for exactly the same reasons. So, what is the max operation do when you started the root and you just follow the right child pointers until you run out them so you hit null. So, you know, that the running time is going to be no worse in the longest such paths. It's particular path from the root to essentially a leaf. So instead we're going to have a running time more than the height of the tree, on the other hand for all you know. The path from the root to the maximum key might well be the longest one in the tree. It might be the path that actually determines the height of the search tree. So, for example in our running unbalanced example, that would be a bad tree for the minimum operation If you look for the minimum in this tree, then you have to traverse every single pointer from five all the way down to one. Of course there's an analogious bad search tree for the maximum operation where the one is the root and the five is all the way down to the left. Another thing you can do is search trees which mimics what you can do with sorted arrays is you can print out all of the keys in the sorted order in linear time with constant time per element. Obviously, in the sorted array this is trivial. Use your for loop start ing at the beginning at the array pointing up the keys one at a time and there's a very elegant recursive implementation for doing the exact same thing in a search tree. And this is known as an in order traversal of binary search tree. So as always you begin at the beginning namely at the root of the search tree. And a little bit of notation of which call, all of the search tree that starts at r's left child t sub l and the search tree routed at r's right child t Sub r. In our running example of course the root is three t sub l with correspondent in the search tree comprising only the elements one and two, t sub r would correspond to the sub-tree comprising only the elements five and four. Now, remember we want to print out the keys in increasing order. So in particular, the first key we want to print out is the smallest of them all. So it's something we definitely don't want to do is we don't want to first print out the key at the root. For example in our search tree example, the root's key is three, we don't want to print that out first. We want to print out the one first. So where is the minimum lie? Well, by the search tree property, it's got to lie in the left subtree t sub l, So we're just going to recurse on t Sub l. So by the magic of recursion or if you prefer induction, what re-cursing on t sub l is going to accomplish is we're going to print out all of the keys in t sub l in increasing order from smallest to largest. Now that's pretty cool because t sub l contains exactly the keys that are smaller than the key of the root. Remember that's the search tree property. Everything bigger than the root's key has to be in the left sub tree. Everything bigger than the root's key have to be in its right sub tree. So in our concrete example of this first recursive call is we're going to print the keys one and then two. And now, if you think about it it's the perfect time to print out the key at the root, right? we want to print out all the keys in increasing order we've already done everything less than the root's key Where re-cursing and on the right hand side will take you everything bigger in it so in between the two recursive calls, this is why it's called an in order traversal, that's when we want to print out. R's key. And clearly this works in our concrete example, the first recursive call print out one and two, it's the perfect time to print out three and then a recursive call of print out four and five. And more generally, the recursive call on there right subtree will print out all of the keys bigger than the roots key and increasing order again by the magic of recursion or induction So, the fact that the pseudo-code is correct. The fact that the so-called in-order traversal indeed print out the keys in increasing order. This is a fairly straightforward proof by induction. It's very much in the spirit or the proofs by induction, correctness of divide and conquer algorithms that we've discussed earlier in the course. So what about the running time of an in order traversal? The claim is that the running time of this procedure is linear. It's O of n where n is the number of keys in the search tree. And the reason is, there's exactly one recursive call for each node of the tree and constant work is done in each of those recursive calls. And a little more detail, so what is the in order] traversal do, It will print out the keys in increasing. In particular it prints out each key exactly once. Each recursive call prints out exactly one key's value. So there's exactly n recursive calls and all of the recursive call does is print one thing. So n recursive calls constant time for each that gives us a running time of O(n) overall. In most data structures, deletion is the most difficult operation and in search trees. There are no exception. So let's get into it and talk about how deletion works, there are three different cases. So the first order of business is to locate the node that has the key k, locate the node that we want to get rid off. Right so for starters, maybe we're trying to delete the key two from our running example search tree. So the first thing we need to do is figure out where it is. So, there are three possibilities for the number of children that a node in a search tree might have and might have zero children that might have one child it might have two children, corresponding to those three cases that the deletion pseudo-code will also have three cases. So, let's start with the happy case where there's only zero children like in this case where deleting the key 2 from the search tree. Then of course, we can, without any reservations just delete the node directly from the search tree, Nothing can go wrong, there's no children depending on that node. Then there is the medium difficult case. This is where. The node containing k has one child. An example here would be, if we wanted to delete five from the search tree so the medium case is also not too bad. All you got to do is splice out the node that you want to delete. That creates a hole in the tree but then that node, deleted node's unique child assumes the previous position of the deleted node. I can make a nerdy joke about Shakespeare right here but I'll refrain. For example, in our five node search tree if we wanted to, let's say we haven't actually deleted two out of this one, if we wanted to delete the five. The five when we take it out of the tree that would leave a hole but then we just replace the position previously held by five by it's unique child four. And if you think about it that works just fine in the sense of that preserves the search tree property. Remember the search tree property says that everything in say, a right subtree has to be bigger than everything in the nodes key, so we've made four the new right child of three but four and any children that it might have were always part of 3's right subtree so all that stuff has got to be bigger than three so there's no problem putting four and possibly all of its descendants. as the right child of three. The search tree property is in fact retained. So, the final difficult case then is when the node being deleted has both of its children, has two children. So, in our running example with five nodes, this would only transpire if you wanted to delete the root, you want to delete the key three from the tree. The problem, of course, is that, you know, you can try ripping out this node from the tree but then, there's this hole and it's not clear that it's going to work to promote either child. Into that spot. You might stare at our example search tree and try to understand what would happen if you try to bring one up to be the root or if you try to bring five up to be the root. Problems would happen, that's what would happen. This is an interesting contrast to when we faced the same issue with heaps. Because the heap property in some sense is perhaps less stringent, there we didn't have an issue. When we wanted to delete something with two children, we just promoted the smaller of the two children assuming we wanted to export and extract them in operation. Here, we're going to have to work a little harder. In fact this is going to be really neat trick. We're going to do something that reduces the case of two children to the previously solved cases of zero or one children. So here's a very sneaky way we identify a node to which we can apply either the case zero or the case one operation. What we're going to do is we're going to. Start from k and we're going to compute k's predecessor. Remember, this is the next smallest key in the tree. So, for example, the predecessor of the key three is two. That's the next smallest key in the tree. In general, let's call case predecessor l. Now, this might seem complicated. We're trying to implement one tree operation and with deletion and all of a sudden we're invoking a different tree operation predecessor which we covered a couple of slides ago. And to some extent you're right you know, delete, this is a nontrivial operation. But, it's not quite as bad as you think for the following reason. When we compute this predecessor, we're actually in the easy case of the predecessor operation conceptually . Remember how do you get a predecessor, well it depends. What does it depend on? It depends on whether you got a non-empty left sub tree or not. If you don't have a non-empty left sub tree, that's how you got to those things and follow a parent pointers upward until you find a key which is smaller than what you've started. But. If you've got a left sub tree, then it's easy. You just find the maximum of the left sub tree and that's got to be the predecessor and remember, finding maximum are easy. All you have to do is follow right child pointers until you can't anymore. Now, what's cool is because we only bother with this predecessor computation in the case where case k's node has both children. We only have to do it in the case where it has a non-empty left subtree. So really when we say compute k's predecessor l. All you got to do is follow k's left child. That's not null because it has both children. And then, follow right child pointers until you can't anymore and that's the predecessor. Now, here's the fairly brilliant parts of the way you do implement deletion in the search tree which is you swap these two keys, k and l. So for example in our running search tree, instead of this three at the root we would put a two there and instead of this two at the leaf, it would put a three there. And the first time you see this, it just strikes you as a little crazy, maybe even cheating or just simply disregarding the roles of, rules of search trees. And actually, it is like check out what happen to our example search tree. We swap the three and the two and this is not a search tree anymore, right? So, we have this three which is in two left sub tree and a three is bigger than the two and that is not allowed. That is violation of the search tree property. Oops. So, how can we get away with this and we get away with this is we're going to delete three anyway. So, we're going to wind up with the search tree at the end of the day. So we may have messed up the search tree property a little bit but we've swapped k in the position where its really easy to get rid of. Well how did we compute case predecessor l? Ultimately that was the result of a maximum computation which involves following right child pointers until you get stuck and l was the place we got stuck. What's the meaning to get stuck? It means l's right child pointer is null. It does not have two children. In particular it does not have a right child. Once we swap k in the l's old position, k now does not have a right child. It may or may not have a left child and the example on the right it does not have a left child either in this new position but in general it might have a left child. But, it definitely doesn't have a right child. Because that was a position at which a maximum computation got stuck. And if we want to delete a node that has only zero or one child, well that we know how to do. That we covered in the last slide. Either you just delete it, that's what we do in the running example here. Or in the case where k's new node does have a left child, you would do the splice out operation. So you would rip out the node that contains k and that the unique child of that node would assume the previous position of that node. Now an exercise which I'm not going to do here but I strongly encourage you think through in the privacy of your own home, is that , in fact, this deletion operation retains the search tree property. So roughly speaking, when you do the swap, you can violate the search tree property as we see in this example but all of the violations involved the node you're about to delete so once you delete that node, there's no other violations of the search property so bingo, you're left with the search tree. The running time this time no get, no prizes for guessing what it is because it's basically just one of these predecessor computations plus some pointer rewiring just like the predecessor and search is going to be governed by the height of the tree. So let me just say a little bit about the final two operations mentioned earlier, select and rank. Remember select is just a selection problem. I'll give you an order statistic like seventeen and I want you to return the seventeenth smallest key in the tree. Rank is I give you a key value and I want to know how many keys in the tree are less than or equal to that value. So, to implement these operations efficiently, we actually need one small new idea which is to augment binary search trees with additional information at each node. So, now the search tree will contain not just a key but also information about the tree itself. So, this idea is often called augmenting your data structure and perhaps the most canonical augmentation of the search tree like these is to keep track in each node, not just to the key value but also over the population of tree nodes in the sub tree that is rooted there. So let's call this size of x. Which is the number of tree nodes in the subtree rooted at x. So to make sure you know what I mean, let me just tell you what the size field should be for each of the five nodes in our running search tree example. So again example, we're thinking about how many nodes are in the subtree rooted given node. Or equivalently, following child pointers from that node how many different tree nodes can you reach? So from the root of course, you can reach everybody. Everybody's in the tree rooted at the root so the size there is five. By contrast, you start at the node one, well, you can get to the one or you can follow the right child pointer to get to the two. So at the one. The size would be two and the node with the key value five for the same reason, the size would be two. At the two leaves, the subtree where the leaf is just the leaf itself so there, the size would be one. There's an easy way to compute the size of a given node once you know the size of its two sub trees. So, if the given node in the search tree has children y and z, then, how many nodes are there in the sub tree rooted x, well, there's those that are rooted at y. There are those in the left sub tree, there are those that are reachable from z that is there are the children that are also children of z and then there's x itself. Now in general, whenever you augment a data structure, and this is something we'll talk about again when we discuss red black trees, you've got to pay the piper. So, the extra data that you maintain it might be useful for speeding up certain operations. But whenever you have operations that modify the tree, specifically insertion and deletion, you have to take care to keep that extra data valid, keep it maintained. Now, in the case of the subtree sizes, there are quite straightforward to maintain under insertion and deletion without affecting the running time of insertion and deletion very much but that's something you should always think about offline. For example, when you perform an insertion remember how that works. You do as, essentially a search. You follow left and right child pointers down to the bottom of the tree until you get a null pointer then that's where you stick a new node. Now what you have to do is you have to trace back up that path, all of the ancestors of the new node you just inserted and increment their subtree sizes by one. So let's wrap up this video by showing you how to implement the selection procedure given an nth order statistic in a search tree that's been augmented so that at every node you know the size of a subtree rooted at that node. Well of course as always you start at the beginning which in the search tree is the root. And let's say the root has a sub-children y and z. Y or z could be null, that's no problem. We just think of the size of a null node as being zero. Now, what's the search tree property? It says, every, these keys that are less than the keys sorted x are precisely the one that are in the left sub tree of x. The keys in the tree, they are bigger than the key to x or precisely the ones that you're going to find in x's right sub tree. So, supposed we're asked to find the seventeenth order statistic in the search three. Seventeenth smallest key that's stored in the tree, Where is it going to be? Where should we look? Well, it's going to depend on the structure of the tree and in fact it's going to depend on the subtree sizes. This is exactly. We're keeping track of them so we can quickly make decisions about how to navigate through the tree. So for a simple example, suppose that x's left subtree contains say 25 keys. So remember y know locally exactly what the population of the subtree is so in constant time from x, we can figure out how many keys are in y subtree let's say its 25. Now, by the defining property of search trees, these are the 25 smallest keys anywhere in the tree. Right, x is bigger than all of them. Everything in x's right subtree is bigger than all of them. So, the 25 smallest order statistics are all in the subtree rooted to y, clearly that where we should recurse. Clearly that's where the answer lies so in recursing the subtree root of y and then we are again looking for the seventeenth order statistic in this new smaller search tree. On the other supposed when we started x and we look, we ask why. How, how many nodes are there in your subtree. Maybe y locally have stored the number twelve. So there's only twelve things in x's left subtree. Well, okay, x itself is bigger than all of them so that's going to, x is going to be the thirteenth biggest order statistic. It's going to be the thirteenth biggest element in the tree. Everything else is in the right sub tree. So, in particular, the seventeenth order statistic is going to be in the right sub tree so we're going to recurse in the rght sub tree. Now, what are we looking for, we're not looking for the seventeenth order statistic anymore. The twelve smallest things all in x's sub tree, x itself is the thirteenth smallest so we are looking for the fourth smallest of what remains. So, the recursion is very much along the lines of what we did in the divide and conquer selection algorithms earlier in the course. So to fill in some more details, let's let a denote the subtree size at y. And if it happens that x has no left child, we'll, the point would be a to be zero. So the super lucky case is when there's exactly i - 1 nodes in the left subtree. That means the root here, x is itself the ith order statistic remember it's bigger than everything In it's left subtree it's smaller than everything in its right subtree. But, in the general case we're going to be recursing either on the left subtree or in the right subtree. We recurse on the left subtree when its population is large enough that we guarantee it and compasses the ith order statistic. And that happens exactly when it sides is at least i. That's because the left subtree has the smallest keys that are anywhere in the search tree. And in the final case when the left subtree is so small that the only does it not contain the ith order statistic but also x is too small to be an ith order statistic then we recurse in the right subtree knowing that we have thrown away a + 1, the a + 1 smallest key values anywhere in the original tree. So, correctness of this procedure is pretty much exactly the same as the inducted correctness for the selection algorithms we've discussed earlier in effect to the root of the search tree is acting as a pivot element with everything in the left sub tree being less than the root everything in the right sub tree being greater than the element in the root so that's why the recursion is correct. As far as the running time, I hope it's evident from the pseudo code that we do constant time each time they recurse. How many times can we recurse when we keep moving down the tree that maximum number of times we can move down the tree is proportional to the height of the tree. So, it was again is proportional to the height. So, that's the select operation, There is an analogous way to write the rank operation. Remember, this is where you're given the key value and you want to count up the number of stored keys that are less than or equal to that target value, Again, you use this augmented search trees and again, you can get running time porportional to the height and I encourage you to think through

# 13-4-Red-Black Trees

So, in this video, we'll graduate beyond the domain of just vanilla binary search trees, like we've been talking about before, and we'll start talking about balanced binary search trees. These are the search trees you'd really want to use when you want to have real time guarantees on your operation time. Cuz they're search trees which are guaranteed to stay balanced, which means the height is guaranteed to stay logarithmic, which means all of the operations search trees support that we know and love, will also be a logarithmic in the number of keys that they're storing. So, let's just quickly recap. What is the basic structure tree property? It should be the case that at every single node of your search tree, if you go to the left, you'll only see keys that are smaller than where you started and if you go to the right you only see keys that are bigger than where you started. And a really important observation, which is that, given a set of keys, there's going to be lot and lots of legitimate, valid, binary search trees with those keys. So, we've been having these running examples where the keys one, two, three, four, five. On the one hand, you can have a nice and balanced search tree that has height only two, with the keys one through five. On the other hand, you can also have these crazy chains, basically devolved to link lists where the heights for, and elements could be as high as N - 1. So, in general, you could have an exponential difference in the height. It can be as small, in the best case, as logarithmic and as big, in the worst case, as linear. So, this obviously motivates search trees that have the additional property that you never have to worry about their height. You know they're going to be well balanced. You know they're going to have height logarithmic. You're never worried about them having this really lousy linear height. Remember, why it's so important to have a small height? It's because the running time of all of the operations of search trees depends on the height. You want to do search, you want to insertions, you want to find predecessors or whatever, the height is going to be what governs the running time of all those properties. So, the high level idea behind balanced search trees is really exactly what you think, which is that, you know, because the height can't be any better than logarithmic in the number of things you're storing, that's because the trees are binary so the number of nodes can only double each level so you need a logarithmic number of levels to accommodate everything that you are storing. But it's got to be logarithmic, lets make sure it stays logarithmic all the time, even as we do insertions and deletions. If we can do that, then we get a very rich collection of supported operations all running in logarithmic time. As usual, n denotes, the number of keys being stored in the tree. There are many, many, many different balanced search trees. They're not super, most of them are not super different from each other. I'm going to talk about one of the more popular ones which are called Red Black Trees. So, these were invented back in the '70s. These were not the first balanced binary search tree data structures, that honor belongs to AVL trees, which again are not very different from red black trees, though the invariants are slightly different. Another thing you might want to look up and read about is a very cool data structure called splay trees, due to Sleator and Tarjan, These, unlike red black trees and AVL trees, which only are modified on insertions and deletions, which, if you think about it, is sort of what you'd expect. Splay trees modify themselves, even when you're doing look ups, even when you're doing searches. So, they're sometimes called self-adjusting trees for that reason. And it's super simple, but they still have kind of amazing guarantees. And then finally, going beyond the, just the binary tree paradigm many of you might want to look up examples of B trees or also B+ trees. These are very relevant for implementing databases. Here what the idea is, in a given node you're going to have not just one key but many keys and from a node, you have multiple branches that you can take depending where you're searching for falls with respect to the multiple keys that are at that node. The motivation in a database context for going beyond the binary paradigm, is to have a better match up with the memory hierarchy. So, that's also very important, although a little bit out of the scope here. That said, what we discuss about red-black trees, much of the intuition will translate to all of these other balance tree data structures, if you ever find yourself in a position where you need to learn more about them. So, red black trees are just the same as binary search trees, except they also always maintain a number of additional invariants. And so, what I'm going to focus on in this video is, first of all, what the invariants are, and then how the invariants guarantee that the height will be logarithmic. Time permitting, at some point, there will be optional videos more about the guts, more about the implementations of red black trees namely how do you maintain these invariants under insertions and deletions. That's quite a bit more complicated, so that's appropriate for, for optional material. But understanding what the invariants are and what role they play in controlling the height is very accessible, and it's something I think every programmer should know. So, there, I'm going to write down four invariants and really, the bite comes from the second two, okay, from the third and the fourth invariant. The first two invariants you know, are just really cosmetic. So, the first one we're going to store one bit of information additionally at each node, beyond just the key and we're going call this bit as indicating whether it's a red or a black node. You might be wondering, you know, why red black? Well, I asked my colleague, Leo Guibas about that a few years ago. And he told me that when he and Professor Sedgewick were writing up this article the journals were, just had access to a certain kind of new printing technology that allowed very limited color in the printed copies of the journals. And so, they were eager to use it, and so they named the data structure red black, so they could have these nice red and black pictures in the journal article. Unfortunately, there was then some snafu, and at the end of the day, that technology wasn't actually available, so it wasn't actually printed the way they were envisioning it but the name has stuck. So, that's the rather idiosyncratic reason why these data structures got the name that they did, red black trees. So, secondly we're going to maintain the invariant that the roots of the search tree is always black, it can never be red. Okay. So, with the superficial pair of invariants out of the way, let's go to the two main ones. So, first of all, we're never going to allow two reds in a row. By which, I mean, if you have a red node in the search tree, then its children must be black. If you think about for a second, you realize this also implies that if a notice red, and it has a parent, then that parent has to be a black node. So, in that sense, there are no two red nodes in a row anywhere in the tree. And the final invariant which is also rather severe is that every path you might take from a root to a null pointer, passes through exactly the same number of black nodes. So, to be clear on what I mean by a root null path, what you should think about is an unsuccessful search, right? So, what happens in an unsuccessful search, you start at the root depending on whether you need to go smaller or bigger, you go left or right respectably. You keep going left right as appropriate until eventually you hit a null pointer. So, I want you to think about the process that which you start at the root and then, eventually, fall off the end of the tree. In doing so, you traverse some number of nodes. Some of those nodes will be black some of those nodes will be red. And I want you to keep track of the number of black nodes and the constraints that a red black tree, by definition, must satisfy, is that no matter what path you take through the tree starting from the root terminating at a null pointer, the number of black nodes traversed, has to be exactly the same. It cannot depend on the path, it has to be exactly the same on every single root null path. Let's move on to some examples. So, here's a claim. And this is meant to, kind of, whet your appetite for the idea that red black trees must be pretty balanced. They have to have height, basically logarithmic. So, remember, what's the most unbalanced search tree? Well, that's these chains. So, the claim is, even a chain with three nodes can not be a red black tree. So, what's the proof? Well, consider such a search tree. So, maybe, with the key values one, two and three. So, the question that we're asking is, is there a way to color the node, these three nodes, red and black so that all four of the invariants are satisfied. So, we need to color each red or black. Remember, variant two says, the root, the one has to be black. So, we have four possibilities for how to use the color two and three. But really, because of the third invariant, we only have three possibilities. We can't color two and three both red, cuz then we'd have two reds in a row. So, we can either make two red, three black, two black, three red, or both two and three black. And all of the cases are the same. Just to give one example, suppose that we colored the node two, red, and one and three are black. The claim is invariant four has been broken and invariant four is going to be broken no matter how we try to color two and three red and black. What is invariant four says? It says, really on any unsuccessful search, you pass through the same number of black nodes. And so, one unsuccessful search would be, you search for zero. And if you search for a zero, you go to the root, you immediately go left to hit a null pointer. So, you see exactly one black node. Namely one. On the other hand, suppose you searched for four, then you'd start at the root, and you'd go right, and you go to two, you'd go right, and you go to three, you'd go right again, and only then will you get a null pointer. And on that, unsuccessful search, you'd encounter two black nodes, both the one and the three. So, it's a violation of the fourth invariant, therefore, this would not be a red black tree. I'll leave that for you to check, that no matter how you try to code two and three red or black, you're going to break one of the invariants. If they're both red, you'd break the third invariant. If at most one is red, you'd break the fourth invariant. So, that's a non-example of a red-black tree. So, let's look at an example of a red-black tree. One, a search tree where you can actually color the nodes red or black so that all four invariants are maintained. So, one search tree which is very easy to make red black is a perfectly balanced one. So, for example, let's consider this three nodes search tree has the keys three, five, and seven and let's suppose the five is the root. So, it has one child on each side, the three and the seven. So, can this be made a red black tree? So, remember what that question really means. It's asking can we color theses three nodes some combination of red and black so that all four of the invariants are satisfied? If you think about it a little bit, you realize, yeah, you can definitely color these nodes red or black to make and satisfy for the invariants. In particular, suppose we color all three of the nodes, black. We've satisfied variant number one, we've colored all the nodes. We've satisfied variant number two, and particularly, the root is black. We've satisfied invariant number three. There's no reds at all, so there's certainly no two reds in a row. And, if you think about it, we've satisfied invariant four because this tree is perfectly balanced. No matter what you unsuccessfully search for, you're going to encounter two black nodes. If you search for, say, one, you're going to encounter three and five. If you search for, say, six, you're going to encounter five and seven. So, all root null paths have exactly two black nodes and variant number four is also satisfied. So, that's great. But, of course, the whole point of having a binary search tree data structure is you want to be dynamic. You want to accommodate insertions and deletions. Every time you have an insertion or a deletion into a red black tree, you get a new node. Let's say, an insertion, you get a new node, you have to color it something. And now, all of a sudden, you got to worry about breaking one of these four invariants. So, let me just show you some easy cases where you can accommodate insertions without too much work. Time permitting we will include some optional videos with the notion of rotations which do more fundamental restructuring of search trees so that they can maintain the four invariants, and stay nearly perfectly balanced. So, if we have this red black tree where everything's black, and we insert, say, six, that's going to get inserted down here. Now, if we try to color it black, it's no longer going to be a red black tree. And that's because, if we do an unsuccessful search now for, say, 5.5, we're going to encounter three black nodes, where if we do an unsuccessful search for one, we only encounter two black nodes. So, that's not going to work. But the way we can fix it is instead of coloring the six black, we color it red. And now, this six is basically invisible to invariant number four. It doesn't show up in any root null paths. So, because you have two black nodes in all roots in all paths before, before the six was there, that's still true now that you have this red six. So, all four invariants are satisfied once you insert the six and color it red. If we then insert, say, an eight, we can pull exactly the same trick, we can call it an eight red. Again, it doesn't participate in invariant four at all so we haven't broken it. Moreover, we still don't have two reds in a row, so we haven't broken invariant number three either. So, this is yet another red black tree. In fact, this is not the unique way to color the nodes of this search tree, so that it satisfies all four of the invariants. If we, instead, recolor six and eight black, but at the same time, recolor the node seven, red, we're also golden. Clearly, the first three invariants are all satisfied. But also, in pushing the red upward, consolidating the red at six and eight, and putting it at seven instead, we haven't changed the number of black nodes on any given path. Any black, any path that previously went through six, went through seven, anything that went through eight, went through seven so there's exactly the same number of red and black nodes on each such path as there was before. So, all paths still have equal number of black nodes and invariant four remains satisfied. As I said, I've shown you here only simple examples, where you don't have to do much work on an insertion to retain the red black properties. In general, if you keep inserting more and more stuff and certainly if you do the deletions, you have to work much harder to maintain those four invariants. Time permitting, we'll cover just a taste of it in some optional videos. So, what's the point of these seemingly arbitrary four invariants of a red black tree? Well, the whole point is that if you satisfy these four invariants in your search tree, then your height is going to be small. And because your height's going to be small, all your operations are going to be fast. So, let me give you a proof that if a search tree satisfies the four invariants, then it has super small height. In fact, no more than double the absolute minimum that we conceivably have, almost two times log base two of N. So, the formal claim, is that every red-black tree with N nodes, has height O of log N, were precisely in those two times log base two of N + 1. So, here's the proof. And what's clear about this proof is it's very obvious the role played by this invariants three and four. Essentially, what the invariants guarantee is that, a red black tree has to look like a perfectly balanced tree with at most a sort of factor two inflation. So, let's see exactly what I mean. So, let's begin with an observation. And this, this has nothing to do with red black trees. Forget about the colors for a moment, and just think about the structure of binary trees. And let's suppose we have a lower bound on how long root null paths are in the tree. So, for some parameter k, and go ahead and think of k as, like, ten if you want. Suppose we have a tree where if you start from the root, and no matter how it is you navigate left and right, child pointers until you terminate in a null pointer. No matter how you do it, you have no choice but to see at least k nodes along the way. If that hypothesis is satisfied, then if you think about it, the top of this tree has to be totally filled in. So, the top of this tree has to include a perfectly balanced search tree, binary tree of depth k - 1. So, let me draw a picture here of the case of k = three. So, if no matter how you go from the root to a null pointer, you have to see at least three nodes along the way. That means the top three levels of this tree have to be full. So, you have to have the root. It has to have both of its children. It has to have all four of its grandchildren. The proof of this observation is by contradiction. If, in fact, you were missing some nodes in any of these top k levels. We'll that would give you a way of hitting a null pointer seeing less then k nodes. So, what's the point is, the point is this gives us a lower bound on the population of a search tree as a function of the lengths of its root null paths. So, the size N of the tree must include at least the number of nodes in a perfectly balanced tree of depth k - 1 which is 2^k - 1, So, for example, when k = 3, it's 2^3 (two cubed) - 1, or 7 that's just a basic fact about trees, nothing about red black trees. So, let's now combine that with a red black tree invariant to see why red black trees have to have small height. So again, to recap where we got to on the previous slide. The size N, the number of nodes in a tree, is at least 2^k - 1, where k is the fewest number of nodes you will ever see on a root null path. So, let's rewrite this a little bit and let's actually say, instead of having a lower bound on N in terms of k, let's have an upper bound on k in terms of N. So, the length of every root null path, the minimum length of every root null path is bounded above by log base two of quantity N + 1. This is just adding one to both sides and taking the logarithm base two. So, what does this buy us? Well, now, let's start thinking about red black trees. So now, red black tree with N nodes. What does this say? This says that the number of nodes, forget about red or black, just the number of nodes on some root null path has to be the most log base two of N + 1. In the best case, all of those are black. Maybe some of them are red, but in the, in, the maximum case, all of them are black. So, we can write in a red black tree with N nodes, there is a root null path with at most log base two of N + 1, black nodes. This is an even weaker statement than what we just proved. We proved that it have some, somehow must have at most log based two, n + 1 total nodes. So, certainly, that path has the most log base two of N + 1 black nodes. Now, let's, now let's apply the two knockout punches of our two invariants. Alright, so fundamentally, what is the fourth invariant telling us? It's telling us that if we look at a path in our red black tree, we go from the root, we think about, let's say, that's an unsuccessful search, we go down to a null pointer. It says, if we think of the red nodes as invisible, if we don't count them in our tally, then we're only going to see log, basically a logarithmic number of nodes. But when we care about the height of the red black tree, of course, we care about all of the nodes, the red nodes and the black nodes. So, so far we know, that if we only count black nodes then we're good, We only have log base two of N + 1 nodes that we need to count. So, here's where the third invariant comes in. It says, well actually, black nodes are a majority of nodes in the tree. In a strong sense, there are no two reds in a row, on any path. So, if we know the number of black nodes is small, then because you can't have two reds in a row, the number of total nodes on the path is at most twice as large. In the worst case, you have a black route, then red, then black, then red, then black, then red, then black, et cetera. At the worst case, the number of red nodes is equal to the number of black nodes, which doubles the length of the path once you start counting the red nodes as well. And this is exactly what it means for a tree to have a logarithmic depth. So, this, in fact, proves the claim, if the search trees satisfies the invariants one through four, in particular if there's no two reds in a row and all root null paths have an equal number of black nodes, then, knowing nothing else about this search tree, it's got to be almost balanced. It's perfectly balanced up to a factor of two. And again, the point then is that operations in a search tree and the search trees are going to run in logarithmic time, because the height is what governs the running time of those operations. Now, in some sense, I've only told you the easy part which is if it just so happens that your search tree satisfies these four invariants, then you're good. The height is guaranteed to be small so the operations are guaranteed to be fast. Clearly that's exactly what you want from this data structure. But for the poor soul who has to actually implement this data structure, the hard work is maintaining these invariants even as the data structure changes. Remember, the point here is to be dynamic, to accommodate insertions and deletions. And searches and deletions can disrupt these four invariants and then one has to actually change the code to make sure they're satisfied again, so that the tree stays balanced, has low height, even under arbitrary sequences of insertions and deletions. So, we're not going to cover that in this video. It can be done, without significantly slowing down any of the operations. It's pretty tricky, takes some nice ideas. There's a couple well-known algorithms textbooks that cover those details. Or if you look at open source and limitations of balanced search trees, you can look at code that does that implementations. But, because it can be done in a practical way and because Red Black Tree supports such an original array of operations, that's why you will find them used in a number practical applications. That's why balanced search trees should be part of your programmer tool box.

# 13-5-Rotations [Advanced - Optional]

In this video and the next we are going to take you to the next level and here under the hood into implementations of balanced pioneer research trees. Now frankly when any great details of all balance binary research tree implementations get pretty complicated and if you really want to understand them at a fine grain level there is no substitute for reading an advanced logarithms textbook that includes coverage of the topic and/or studying open source Implementations of these data structures. I don't see the point of regurgitating all of those details here. What I do see the point in doing, is giving you the gist of some of the key ideas in these implementations. In this first video I want to focus on a key primitive, that of rotations which is common to All balanced by the [INAUDIBLE] of limitations. Whether is red, black trees, EVL trees, B or B+ trees, whatever all of them use rotations. In the next video we'll talk a little bit more about the details of red, black trees in particular. So, what is the points behind these magically rotation operations? Well the goal is to do just constant work, just re-wire a few pointers, and yet locally re-balance a search tree, without violating the search tree properties/g Property. So there are two flavors of rotations, left rotations and right rotations. In either case, when you invoke a rotation it's on a parent child pair, in a search tree. If it's a right child of the parent, then you use a left rotation. We'll discuss that on this slide. And a right rotation is, in some sense, an inverse operation which you use when you have a left child of the parent. So what's the generic picture look like when you have a node x in a search tree and it has some right child y? Well x, in general, might have some parents. x might also have some left subtree. Let's call that left subtree of x, A. It could, of course, be And then Y has it's 2 subtrees, lets call the left subtree of Y B and the right subtree C. It's going to be very important that rotations preserve the search tree property so to see why that's true lets just be clear on exactly which elements are bigger then which other elements in this picture. So first of all Y being a right child of X, Y's going to be bigger than x. Now all of the keys which line the subtree a because they're to the left of x these keys are going to be even smaller than x. By the same token anything in the subtree capital c, that's to the right of y so all that stuff's going to be even bigger than y. What about sub tree capital B? What about the nodes in there? Well, on the one hand, all of these are in x's right sub tree, right? To get to any node in B, you go through x and then you follow the right child to y. So that means everything is B is bigger than x, yet, at the same time, it's all in the left sub tree of y so these are all Things smaller than y. Summarizing all of the nodes in b have keys strictly in between x and y. So now that we're clear on how all of these search keys in the different parts of this picture relate to each other, I can describe the point of a left rotation. Fundamentally the goal is to invert the relationship. Between the nodes x and y. Currently x is the parent and y is the child. We want to rewire a few pointers so that y is now the parent and x is the child. Now what's cool is given that is our goal is pretty much a unique way to put all these pieces back together to accomplish it. So lets just follow our nose. So remember the goal Y should be the parent and X should be the child. Well, X is less then And why and there's nothing we can do about that. So if x is going to be a child of y, its got to the left child. So your first question might be well what happened that x is parent. So x use to have some parent lets call it p and x's new parent is y. Similarly y used to have parent x and there's a question what should y new parent be? Well y is just going to inherit x's old parent p. SO this change has no bearing for the search tree property. Either of this collection of nodes was P's left sub tree, in that case all these nodes were less than P, or this sub tree was P's right sub tree which in that case all of these are bigger than P, but P can really care less , which of X or Y is it's direct descendant. Now lets move on to thinking about how...what we should do with the sub-trees A, B and C. So, we have 3 sub-trees we need to re-assemble into this picture, and fortunately we have 3 slots available. X has both of its child pointers available and Y has its right child available. So what Can we put where? Well, a, is the sub tree of stuff which is less than both x and y. So, that should sensibly be x's left child. That's exactly as it was before. By the same token, capital C is the stuff bigger than both x and y, so that should be, y, the bigger nodes child, right child just as As before. And what's more interesting is what happens to subtree capital B. So B used to be Y's left child but that can't happen anymore, 'cause now X is Y's left child. So, the only hope is to slot capital B into the only space we have for it, X's right child. Fortunately for us this actually works, sliding B into the only open space we have for it. X's right child does indeed preserve the switch tree property. Recall we notice that every key in capital B is strictly between X and Y, therefore it better be and X's right sub tree and it better be in Y's right sub tree, but it is, that's exactly where we put it. So that's a left rotation, but if you understand a left rotation then you understand a right rotation as well. because a right rotation is just the inverse operation. So that's when you take a parent child pair, where the child is the left child of the parent, and now you again want to invert the relationship. You want to make the old child the new parent and the old parent the new child. And once again given this goal there's really a unique way to reassemble the components of this picture so that the goal's accomplished, so that y is now the parent of x. So what are the laudable properties of rotations? Well first of all I hope it's clear that they can be implemented. In a constant time or you were doing a rewiring a constant number of pointers. Further more as have discussed they preserve the search tree property. So these nice properties are what make rotations the ubiquitous primitive common to all balanced search tree implementations. So this of course, is not the whole story. In a complete specification of a balanced search tree implementation, you have to say exactly when and how you deploy these rotations. You'll get a small taste of that in the next video but if you really want to understand it in more depth, I again encourage you to check out either a comprehensive Data structures textbook. Check out a number of balance search tree demonstrations, which are readily available on the web. Or have a peek at an open source implementation of one of these data structures.

# 13-6-Insertion in a Red-Black Tree [Advanced]

For this final video on binary search trees I want to talk a little bit about implementation, implementation details for the red black tree data structure in particular the insertion operation. As I've said in the past it really doesn't make sense for me to spell off all of the gory details about how this is implemented. If you want to understand them in full detail. Detail You should check out various demonstrations readily available on the web, or a comprehensive textbook, or an open source implementation. Red black trees, you'll recall satisfy four invariants and the final two invariants in particular ensure that the red black tree Always has logarithmic height and therefore all of the supported operations run in logarithmic time. The problem is we've got to pay the piper. Whenever we have a operation that modifies the data structure, it potentially destroys one or more of the invariants, and we have to then restore that invariant. Without doing too much work. Now amongst all of the supported operations there are only two that modify the data structure insertion and deletions. So from thirty thousand feet the approach to implementing insert and delete is to just implement them as if it's a normal binary search tree as if we didn't have to worry about these invariants and then if an invariant is broken we try to fix it with minimal work and two tools that we have our disposal to try to restore an invariant are first of all. Recoloring, flipping the color of nodes from to black and second of all left and right rotations as covered in the previous video. My plan is to discuss the insertion operation not in full detail but I'll tell you about all of the key ideas. Now deletion you got to remember that even in a regular binary search tree deletion is not that trivial and in a red black tree its down right painful. So, that I'm not going to discuss onto for you to text books or online resources to learn more about deletion. So here's how insert is going to work. So suppose we have some new node with the key x. And we're inserting it into a red black tree. So we first just forget about the invariance, and we insert it as usual. And remember, that's easy. all we do is follow left and right shot pointers, until we fall off the end of the tree until we get to a null pointer, and we install this new node with key x, where we fell off the tree. That makes x a leaf in this binary search tree. Let's let y denote x's parent, after it gets inserted. Now in a red-black tree every node has a color. It's either red or black. So we have a decision to make. We just added this new node with key x and we gotta make Get either red or black. And we're sort of between a rock and a hard place, whichever color we make it we have the potential of destroying one of the invariants. Specifically, suppose we color it red. Well remember what the third invariant says, it says you cannot have two reds in a row. So if Y, X's new parent is already red, then when we color X red, we have 2 reds in a row. And we've broken invariant number 3. On the other hand, if we color this new node, X, black, we've introduced a new black node to certain root null paths in this tree. And remember, the 4th invariant insists, that all the root null paths have exactly the same number of black. Notes, so by adding a black note to some but not all of the paths, we're in general, going to destroy that invariant, if we color x black. So what we're going to do is, we're going to choose the lesser of two evils, and in this context the lesser of the two evils is to color x red. Again, we might destroy the third invariant, we'll just deal with the consequences later. So why you ask, is coloring x red and destroying the third invariant, the lesser of two evils? Well, intuitively, it's because this invariant violation is local. The flaw in our not quite red black tree is small and manageable, it's just a single double red and we know exactly The word is it's x and y. So.this sort of more hope in squashing it with minimal work. I can't trust if we coated x black then we violated this much more global type of property involving all of the route in all paths and that's a much more intimidating violation to try to fix. Then just as local one of having a double red between x and it's parent. Indeed some of the time we'll just get lucky and it will just so happen that x is parent y is colored black and then we're golden. This new node x that's colored red, it doesn't create a double red, there's no other violations of the other invariants and so boom, we've got a new red black tree and we can stop. So, the tricky case then is when x's parent y is also red in this case we do not have a red, black tree we have a double red and we have to do some more work to restore the third invariant. So suppose y is red. What do we then know? Well remember, before we inserted x, we had a red black tree, all 4 of the invariants were satisfied. So therefore Y, by virtue of being red, it could not have been the root. It has to have a parent. Let's call that parent W. Moreover by the third invariant there was no double red in this tree before we inserted X so by virtue of Y being red, it's parent W must have been black. So, now the insertion operation branches into 2 different cases and it depends on the color, on the status of w's other child. So in the first case we're going to assume that w's other child that is not y but the other child of w exists in its colored red. In the second case, we're going to treat when w either doesn't have a second child. Y is its only child or when its other child is colored black. So let's recap where things stand. So we just inserted this new node, and it has the key x. And our algorithm colored this node red. So x is definitely red. Now, if it's parent y was black, we already halted. So we've already dealt with that case. So now, we're assuming that y. X's parent is also red, that's what's interesting. Now by virtue of y being red, we know that y's parent, that is x's grandparent w, has to be colored black. And, for case two of insertion, we are assuming that w has a second child, call it z, and that z is colored red. So, how are we going to quash this double red problem? We again, we have 2 tools at our disposal. One is to re-color nodes. The second is to do rotations. So for case 1, we're only going to actually have to do re-coloring. We're not even going to have to bust out per rotations. In particular what we're going to do is, we're going to recolor z and y black and we're going to recolor w red. So, in some sense we take the reds that are at z and y and we consolidate them at w. The important property of this recovering is that it does not break the fourth invariant, remember the forth invariant says that no matter which path you take from the root to a no pointer you see exactly the same number of black nodes. So why is invariance still true after this recoloring, well for any path from a route to a no pointer which doesn't go through the vertex w its relevant. None of these nodes are on that path, so the number of black dots is exactly the same. So think about a path which does go through w. Well if it goes through w to get to a no pointer has to go through exactly one of z or y. So before we did the recoloring this path picked up a black node via w and it did not pick up a black node via z or y both of those were red. Now any such path does not pick up a black node w that's now red but it does pick up exactly one black node either z or y. So, for every single path in the tree, the number of black nodes it contains is exactly the same before or after this recoloring, therefore since the fourth invariant held previously, it also holds after this recoloring. The other great thing is that it seems like we've made progress on restoring the third invariant. The property that we don't want any double-reds at all in the entire tree. Remember, before we did this recoloring, we only had a single double-red. It involved x and y. We just recoded y from red to black. So certainly we no longer have a double reded walling x and y and that was the only one in the tree. So are we done, do we now have a bonafied red black tree? Well the answer depends, and it depends on the core of W's parent. So remember W just got recolored from black to red. So there's now a possibility that W being this new red node participates in some new double red violation . Now w's children, z and y, are black. So those certainly can't be double reds. But w also has some parent, and if w's parent is red, then we get a double red involving w and its parent. Of course, if w's parent was black, then we're good to go. We don't get a double red by recoloring double. W red, so we have no w reds in the tree, and we can just stop. Summarizing, this recoloring preserves the fourth invariant, and either it restores the third invariant, or if it fails to restore the third invariant, at least it propagates the double red violation upward into the tree, closer to To the root.. We're perfectly happy with the progress represented by propagating the double red upward. Why? Well, before we inserted this new object x, we had a red black tree. And we know red black trees have logarithmic height. So the number of times that you can propagate this double red upward is bounded above by the height of the tree, which is only logarithmic. So we can only visit case 1 a logarithmic number of times before this W is propagated all the way to the top of the tree, all the way of the root. So we are not quite done, the one final detail is what happens when this recoloring procedure actually recolors the root. So, you could for example look at this green picture on the right side and ask, well what if w is actually the root of this red black tree and we just recolored it red? Now notice in that situation where the, we are dealing with the root of the tree we're not going to have a double red problem. So invariant three is indeed restored when we get to the top of the tree, but we have a violation of invariant number two which states that the root must always be black. Well if we find ourselves in this situation, there's actually a super simple fix which is this red root, we just recolor it black. Now clearly that's not going to introduce any new double reds. The worry instead is that it breaks invariant four. But, the special property of the root for text is that it A lies exactly once on every route on all path. So if we flip the color of the roof from red to black it increases the number of black nodes on every single routinal path by exactly 1. So if they all have the same number of black nodes before, they'll have the same number of black nodes now, after the recoloring. That completes case 1 of how insertion works. Let's move on to case 2. So case 2 gets triggered when we have a double red and the deeper node of this double red pair, call it X, its uncle, that is if it has grandparent W, parent Y and W's other child, other than Y either. Doesn't exist or if it exists it's labeled it's colored black. That is case 2. I want to emphasize you might find yourself in case 2 right away when you insert this new object x it might be there immediately it has some uncle which is covered x or it might be that if already visited case 1 a bunch of times propagating this double red up the tree and now at some Point. The deeper red node X has a black uncle. Either way, as soon as that happens, you trigger case 2. Well it turns out, case 2 is great in the sense that, with nearly constant work, you can restore in variant number 3 and get rid of the double red without breaking any of the other invariants. You do have to put to use both of the tools we have available in general. Both recolorings and rotations, left and right rotations, as we discussed in the previous video. But, if you do just a constant number of each, recolorings and rotations, you can get all four of the invariants simultaneously. There are unfortunately a couple of sub cases depending on exactly the relationships between x, y, z, and w. For that reason I'm not going to spell out all the details here, check out a textbook if you're interested, or, even better, work it out for yourself. Now that I've told you that two to three rotations plus some recolorings is always sufficient in case two to restore all of the In variance, follow your nose and figure out how it can be done. So let's summarize everything that we've said about how insertion works in a red black tree. So, you have your new node with key x, you insert it as usual. So you make it a leaf, you tentatively color it red. If it's parent is black, your done. You have a red black tree, and you can stop. In general, the interesting case is this new. And you know that x's parent is red. That gives you a double-red of violation of invariant three. Now, what happens is you visit this case 1, propagating this double red upward imagery. This upward propagation process can terminate in one of three ways. First of all, you might get lucky and at some point the double-red doesn't propagate, you do the recoloring in case 1. And it just so happens you don't get a new double red. At that point you have a red black tree and you can stop. The second thing that can happen is the double-red propagation can make it all the way to the root of the tree, then you can just recolor the root black and you can stop with all of the invariants satisfied. Alternatively at some point when you're doing this upward propagation you might find yourself in case 2 as was discussed on this slide. Where the lower red node on the double red pair x has a black or non-existent uncle, Z. In that case, with constant time, you can restore all of the Fourier theories. So the work done overall is dominated by the number of double red propagations you might have to do, that's bounded by the height of this tree and that's bounded by O of log n. So in all of the cases you restore all 4 invariants, you do only a logarithmic amount of work, so that gives you a logarithmic insertion operation for red black trees, as promised.

# zwk7-prog-MedianMaint

Download the following text file:

Median.txt

The goal of this problem is to implement the "Median Maintenance" algorithm (covered in the Week 3 lecture on heap applications). The text file contains a list of the integers from 1 to 10000 in unsorted order; you should treat this as a stream of numbers, arriving one by one. Letting xi denote the ith number of the file, the kth median mk is defined as the median of the numbers x1,…,xk. (So, if k is odd, then mk is ((k+1)/2)th smallest number among x1,…,xk; if k is even, then mk is the (k/2)th smallest number among x1,…,xk.)

In the box below you should type the sum of these 10000 medians, modulo 10000 (i.e., only the last 4 digits). That is, you should compute (m1+m2+m3+?+m10000)mod10000.

OPTIONAL EXERCISE: Compare the performance achieved by heap-based and search-tree-based implementations of the algorithm.