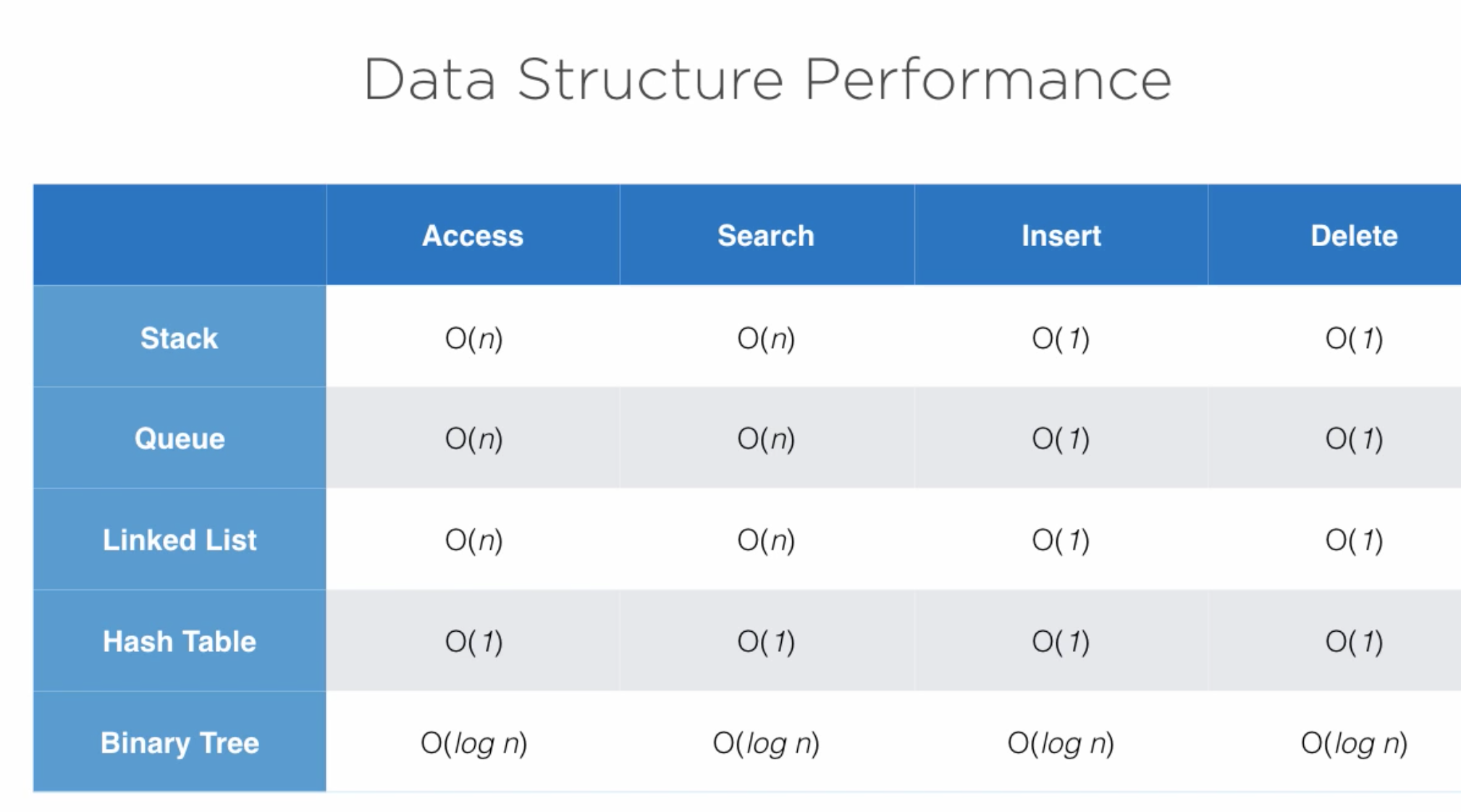


Averege Case:



Course Overview

Course Overview

Hey, my name is Dan Bunker, and welcome to my course on Implementing and Understanding Data Structures in Java. I'm a software developer and consultant based in the Salk Lake City, Utah, area. Data Structures are used all the time in programming. No matter what kind of program you are writing, you'll be utilizing a data structure of some sort. Learning these structures inside and out will provide a strong programing foundation you can rely on for years to come. In this course, we're going to be building five different data structures from scratch using the Java programing language. Even though the examples are in Java, the concepts apply to other languages as well. The major data structures that we're going to cover include queues, stacks, lists, hashes, and trees. By the end of this course, you'll know the strengths and weaknesses of each of these data structures, the performance costs, and how to actually implement them. To get the most out of this course, some familiarity with Java and the Spring Tool Suite editor will help, since I use those in the coding demos. But they certainly aren't required since the concepts apply to most languages. I hope you'll join me on this journey to learn more about data structures and improve your programing knowledge with my course on Implementing and Understanding Data Structures in Java at Pluralsight.

Getting Started

Introduction

Hey, this is Dan Bunker with Pluralsight and this course is about building and understanding data structures using the Java programming language. You may be wondering why you'd want to build many of these core data structures used everyday programming when Java provides many of these data structures as part of the core Java SDK. I hope that this course answers this question and shows you why data structure knowledge is so important and how it builds a strong foundation for your programming skillset. Even though I'm going to show examples using Java, the information in this course is applicable for any programming language and can easily be transferred to that language's syntax. If you're in the need of a data structure refresher or you've never taken the time to study these structures from the ground up, then this is the course for you, so let's go ahead and get started.

Data Structure Line Up

In this course we're going to build five common data structures from the ground up. Each of these data structures have strengths and weaknesses. By going through the motions of building them from scratch, you'll get the internal workings and knowledge of why they are strong in some aspects and weak in others. Here the data structures that we're going to build. A stack, a queue, a linked list, a hash table and a binary tree. Many colleges and universities require a data structure class in their computer science program. Many employers will also ask about them during interviews. This course will give you this knowledge in a condensed and packed bundle. If you take the time to build each of these data structures, you will become a better programmer. Let's start from the beginning and talk about what makes a data structure a data structure.

What Are Data Structures?

So, what exactly is a data structure? Data structures are used to organize data so that the data can be used efficiently. In software programming, data structures also often hold similar information that has some kind of relationship to the other information inside of that data structure. Let's take a look at some real-world examples. Since we're often asked to model real-world examples in our code, here's a classic real-world data structure that we should all be familiar with, a freeway or a highway. Each car or truck represents a chunk of information. The vehicle combined with the driver and a license plate creates a unique piece of data that exists on the freeway's data structure. As vehicles enter or leave the freeway there is an order and sequence to each vehicle. Now, think about a stretch of road that often has traffic jams or heavy traffic. What causes the traffic? Simply too many cars, not enough lanes, maybe there's too many off-ramps or on-ramps in one area. There's many reasons that can cause heavy traffic. A well-designed and thought-out freeway can help solve many of these problems and likewise in your code, a well-designed and thought-out data structure can make your data flow nicely and keep program performance top notch. Now, let's take a look at some data structures that we use in everyday programming. The common string data type when initialized to a string value is a data structure. What kind of data structure is this? If you take a look at the Java string class implementation you'd see that the string value is held as an array of characters. This is a basic list or array data type. I'll cover lists in more detail in the linked list module of this course. Another data structure that you use every time you do some programming is the memory heap space. You may have never looked into how memory is allocated and managed by Java in the JVM or any other language but this diagram shows a simple example of how objects are placed on the heap and how there's a relationship between them. When you instantiate an object such as a person using the new keyword, that person has attributes that also get allocated and placed on the heap interlink together. This creates a tree-like data structure as well as a linked list structure where each object is linked to another object. So, we've really only scratched the surface of where data structures are used in everyday programming. For a developer to understand what a data structure is and how to use them appropriately is key to building a strong and lasting software knowledge and foundation. I've shown you a real-life example and a few programming examples of data structures. Next, let's take a look at what characteristics and attributes make up a proper data structure.

Data Structure Characteristics

Even though all data structures have differences in how they work, they all have common characteristics that help us identify what they are well-suited for. When you are determining what data structures should be used to help you with your software problem, contemplate each of the following characteristics in regards to your needs. First, all data structures house and hold some kind of data. A structure is really just a synonym for a home. The data that is housed in the data structure could be something as simple as a number or as complex as a strand of DNA. The important thing to remember is that the data structure doesn't really care what the data is that it holds. The data structure is not defined by the data, it's just a place to keep and organize the data. The next key-defining characteristic is how data gets in and out of that structure. Can data be inserted or removed wherever on the data structure? Does the data have rules on how it enters or exits? These in-out characteristics place restrictions on how the data structure is used and are important to know. Many in-out data structure characteristics have acronyms that can describe how the data enters and leaves the data structure. FIFO and LIFO are examples of this. FIFO stands for first in, first out. This is a lot like a line at the store or a queued data structure. As you line up to buy your groceries is how you ultimately exit the line. You typically don't see a store line where the last customer in line gets served first. That's what a LIFO stands for, last in, first out. An example of a LIFO data structure is a stack which we'll be covering in the next module of this course. The next characteristic of a data structure is how all of the data inside of the structure relates to all of the other data inside of that structure. The place or position the data occupies in the structure could be something as simple as being the second in line at the store, or the data could have parents, children and siblings in a tree-like structure like you'd see in a family tree chart. One of the last and often most important characteristics is how well the data structure performs when doing certain operations like inserts, sorts and so on. In fact, performance is one of the main deciding factors when choosing a data structure for your needs, so I'll discuss that in more detail coming up on the next slide.

Big O Notation

Understanding the performance implications of a data structure is one of the harder things you'll have to deal with in this course and in working with data structures in general. Computer science defines performance by something called Big O notation. These are examples of Big O notation. This notation describes a data structure's function complexity. It's math based and might be a little foreign if you've never taken the time to work with Big O notation. The capital O is a notation that describes the order of the function or how the complexity of the function grows as the number of elements used in the function grow. The part in parentheses contains the actual math function that specifies the complexity. Think of the invariable as the number of actual elements in the data structure. As the value of N gets larger, what's going to happen to the performance? The easiest way to start to understand Big O notation is typically by looking at a graph of various Big O functions. The X axis or numbers at the bottom of this graph represent the number of items in your data structure or the invariable. The Y axis or the numbers on the left side of the graph represent how long it takes to do some kind of operation with all of those items. The yellow line is N squared and you can see that as the number of elements grows, the performance hit rises dramatically, whereas the green line represents a logarithmic function and as the number of elements grow, the performance hit rises just a little. This graph just shows three examples of Big O notation. There are more. Hopefully you can see that by assigning a Big O value to a data structure you can begin to understand how that data structure will perform as the data structure elements increase.

Common Big O Algorithms

This table shows six types of Big O notation complexities. They are ordered from most performant at the top to least performant at the bottom. The names of these Big O notations are math-based and if you've done any kind of math at an algebraic or higher level, they should sound familiar. So, I'm going to briefly cover each one. The first is constant time. That means that no matter how many elements you have in your data structure, the algorithm always performs the same numbers of operations. An example of code that would be considered constant would be a method or function that just simply returns true. Logarithmic time slowly rises as the number of elements used in the algorithm increase. Data structures that are logarithmic are very performant. A binary search implementation is an example of a logarithmic function. Linear time is seen quite a bit in everyday code. Any kind of for loop or while loop that starts at zero and iterates over a list of the elements until you reach the end is considered linear. If you only have 10 items, a loop is pretty performant but if you have 10 billion, then the loop might take some time to work through. Quadratic time is where algorithms start to go bad performance wise. As your number of elements increase, the performance hit increases much, much worse than linear or logarithmic. You'll often see quadratic complexity in everyday code where you have a loop with inside of a loop. So, let's be honest, we've all done this before and perhaps you haven't thought about the performance hit that this causes but now that I've told you, you'll probably cringe the next time you see this or try to write code like this. Next up is exponential time. This makes quadratic time look pretty good. So, why would you ever write code that performs this badly? At some point in your programming career, you'll encounter the concept of recursion. It sounds really neat and powerful and it is but it generally comes at a bad performance cost. If you recurse over your data structure elements and then loop with inside that recursive function, you've just created an exponential algorithm. Don't feel bad though because there's still worse ways to slow down your code. Factorial is about as bad as it can get and again, you can utilize recursion to obtain this performance nightmare. A real-life example would be looping over all of the elements in your data structure and then with inside of that loop, you call the recursive function over again but with the N minus one elements and this will happen over and over until you reach zero or some other condition. These are just a handful of common Big O notations. Again, there are more out there but if you stick to these six, you'll probably be good for most things that you'll encounter. As a side note, if you're ever bored and want to test out your Big O analysis skills, watch the movie Inception where they have dreams within dreams. If you can figure out the performance hit in Big O notation of that movie, you'll be set and data structures should start to feel easier for you. Up next, let's take a look at how the data structures in this course stack up on the Big O scale.

Data Structure Performance

Data structures have different operations associated with them. The four basic ways that we can interact with the data structure is by accessing an element, searching for an element, inserting an element or deleting an element. The data structures that we're going to build in this course are a stack, queue, linked list, hash table and binary tree and in this grade you can see that the Big O notation isn't always the same for each operation on a particular data structure. Some structures are better suited at accessing an element, while others do better at searching for an element. This great comparison can really help you pick the right structure for your data needs. The Big O algorithms that I'm showing here are for the average case scenario. The Big O notation algorithms can change for worst case scenarios. The last thing that I'm going to show you related to performance on a data structure is a little sample app that I wrote to help understand Big O performance cost. You can find this app on my GitHub profile in the repositories called big-o-performance and you can see the URL here. If you want to clone this little app and take it down to your environment, you can figure out how to do that at the read me page below or I'm just going to show you what this app does so you can get an idea of what Big O performance is doing based off of your element size. So, here's the app as it's running and in the middle you'll see that we have some sample graphs that show what each Big O complexity does visually. So, we have constant, logarithmic, linear, quadratic, exponential and factorial but you can also enter the number of elements for a dataset over here in this little text box and the values for those will be updated below. So, you can see if you have zero elements in your data structure, all of the performance hits are pretty much the same because we don't have any data, right? It's pretty simple. If we change this to 10, you'll notice that some of the data is starting to vary quite dramatically. Constant time remains where it's at, logarithmic time goes up a little bit, linear is linear but notice where factorial is. Once you hit 10 elements, factorial is extremely poor performance. If we change this to 100, things really start to get crazy over the linear side, you know, quadratic, we're squaring that, but exponential and factorial at this point are off the chart and honestly, 100 elements in a data structure isn't that big. I'm sure you've worked with data structures with more elements. So, I'm going to go ahead and bump it up to 10,000 and here you'll start to see something interesting. Logarithmic time has barely gone up, so logarithmic is an excellent performance Big O notation. Linear and quadratic are fairly expensive now but you'll notice that exponential and factorial have essentially become intractable or they take forever, it's infinity, it's so big. So, I know Big O notation isn't the most fun to talk about and there's a lot of math behind it but hopefully this will show you that certain things work better for certain data and certain data structures. So, if you find yourself working with a data structure with a lot of elements in it, you really need to make sure you stay away from anything worse than linear if you want to make performance a priority in your application.

Summary

We've covered a lot of theoretical ground in this module and knowing the data structure characteristics and foundations are really key at building a strong and deep understanding of why you would want to use one data structure over another. Let's recap the important takeaways from this module. The first question I've attempted to answer via examples and a definition is what a data structure is. Almost every program or piece of running code out there utilizes a data structure of some sort. Whether that's in the background using the language heap and stack or using your own data structures, they are there and you will be a better programmer by understanding them and what they're doing. Just remember, a data structure is simply a way of holding a related group of items or elements. Next, we looked at various features and characteristics that all data structures have in common and here are the basics. Every data structure houses or holds pieces of data in some fashion. How the data enters and leaves the structure has rules like first in, last out, or first in, first out. These rules define a major part of the data structure and how it ultimately behaves. How the data is ordered or sequenced in the structure is also an important consideration. Next, we took some time to go over everyone's favorite subject of Big O notation. Big O is used to identify how well a data structure performs as the number of items or elements grow. It is mathematically based and can be intimidating but once you get the basics of what it means and how it's applied, it's really not that hard. After all, it's not rocket science, it's just theoretical analytical math. Last, we looked at how different Big O notation algorithms perform. We then applied that to the various operations you use on a data structure to try and get the Big O performance concepts down to a more practical understanding level. And that's it. We didn't do any coding in this module and that's definitely going to change starting in the next module and continuing on throughout the rest of this course. The lecture part of this course is over so go ahead and get ready to dig in because we're going to get to work from here on out.

Stacks

Introduction

Hey! This is Dan Bunker with Pluralsight and at this point you should have a basic understanding of what a data structure is and how performance is a key indicator of a data structure's strengths and weaknesses. In this module we begin our detailed study of data structures by building a stack. Stacks are used by most languages to store the code flow of function calls as the code executes. Even if you aren't familiar with the stack concept you have been using them just by using programming languages. Before we start coding the stack it will help to understand its attributes and its performance characteristics. So let's get started by going over the stack features and the performance costs.

What Is a Stack?

So what makes a stack a stack? As the name implies, it represents a stack or ordered pile of some kind of info or data. An easy real life example of a stack is a deck of cards. Each card represents an element or a piece of data. As you place a card on top of the deck it becomes the last card to enter the data structure. In this case the queen of hearts was placed on the pile last. If this is a discard pile, say, in a game of Uno or Go Fish, a player can either discard another card on top of the queen of hearts or take the queen of hearts off of the deck. Players can't grab randomly from the middle of the pile or from the bottom. So in these card stacks, there is an order to how data enters and leaves the data structure. The last card placed on the pile is the first card to leave. As I mentioned in the last module, this is known as a LIFO, or last in, first out. Stacks use some unique terms when we are talking and discussing the stack data structure. The first term is known as a push. When you push something onto a stack, you place the data at the top of the stack just like this queen of hearts. When you want to take the data off the top of the stack, you pop it. If I were to pop the queen of hearts off the stack, that would mean that I would take that card off of the deck. What about the big O or performance characteristics on a stack? If I push something on there, all I need to do is just set the next element onto the top of the stack. It doesn't change regardless of how many items are in the stack. So push is considered constant time. A pop is always just taking the last element off of the stack and that doesn't change regardless of the number of items either so pop is also constant time. What if I want to find the king of hearts in this deck of cards? I have to start by pulling or popping each card off the top of the deck until I find the king. We don't know where it is. It could be the next card down or it could be all the way at the bottom. So by looking through each element in the stack to find another particular element I end up with linear performance cause we're having to go through each one, and that grows as the size of the deck or stack grows. If I didn't want to access the king of hearts and do something with that card, I first need to search for it and then I can pop it off the stack and do something with it. This makes accessing an element in a stack linear as well. But think about what had to happen to get the king out of the deck. We had to pop all of the elements off of the stack above the king. So to access any random element in your structure we'll alter the overall data structures elements. If you need to keep those other items in your structure, then you should consider using some other kind of data structure to represent your data. So now that we know what the basics of a stack are let's go ahead and start building one.

Setup

I'm going to briefly cover what you'll need to code the demos used in this course. Since I'm going to show all of my examples using the Java programming language, let's start with what it will take to do some coding in Java. The easiest way to run Java code is to use an IDE. I've been using Eclipse for so long that I'm currently a fan of using Spring Tool Suite, or STS. This is a customized Eclipse editor for the Spring environment. It will work fine for this course but you can also use any plain Eclipse version as well. You can download STS at spring.io/tools. You certainly can choose a different editor if you're more familiar with something else. IntelliJ by JetBrains or NetBeans are two of the popular choices used in the Java community. You could also use a plain text editor and compile from the command line if you are comfortable with that. Using one of the IDEs listed here will include Java and handle your compilation and other standard project needs. If you're not familiar with Java, I'd recommend sticking with Spring Tool Suite so you can easily follow along in these demos. If you are a Java veteran, use what you know and you are comfortable with. So this is Spring STS, just a basic vanilla install and nothing has been configured on it. So I'm going to start off by creating a project that I can put all of the data structures that will get built into this course. The current version of Spring Tool Suite at the time this course is published is Version 3.8.3. Since we'll be doing a lot of plain or just vanilla Java coding, you can use just about any version of STS or Eclipse because the coding samples will be quite simple as far as frameworks and other things are concerned. It'll just be pure Java code. So to create a project, I'm simply going to right-click in the Explorer pane and create a new Java project. I'm going to call the project data-structures and then click on Finish. If Spring STS asks if you want to associate the project with the Java perspective, go ahead and say Yes. At this point we have a basic project and we can go ahead and start coding in Java and building a stack.

Demo: Creating the Stack Class

All of the data structures that we'll be building in this course will begin with the very basics. And once we get the basic structure built we'll go ahead and discuss some options to make the code more robust, user friendly and follow best practices. In this demo we are going to create a stack class. This will be a plain old Java object and we'll call it basic stack. We also need to set up the underlying data structure that will house and hold the stack data. Since we are building this structure from scratch we'll stick to the basic Java data types for now. To start off, the first thing we need to do is create a class for our stack. We can do this pretty easily in the IDE by right-clicking on the Source folder and selecting new class. I'm going to call my stack a basic stack and leave it in the default package. If you prefer to add it into a package or alter the name, that's completely up to you, I'm just keeping this simple using the default package. All I would ask is just don't name your class stack because we'll use that later and you don't want to have a name clash. I can go ahead and click Finish now, and we'll have our basic Java class ready to use. So that completes our first task in this demo by creating our stack class. It was easy, right? The next thing we need to do is create a generic for our class so that we can store any kind of data with inside of our basic stack. To do that, all I need to do is add a slight suffix onto our basic stack class definition by giving it a generic type of x. If you're not familiar with generics, it's probably worth doing a Java primary to understand what this is doing. This is simply saying that we can type this class to anything generically right now but specifically later on. Next we need to add a private variable that we can use to hold the stack data. We need to define this before we can start working on our push and pop and search methods. Since this is a basic stack, I'm going to stick with primitive Java data types, and that simply means that I'm not going to use something like an array list or another complex collection to store the stack data but simply use a primitive array. And that declaration will look like this. You'll notice that I declared it private and I'm using the generic x type. I then have my primitive array with the brackets and I've named the internal variable as data. You want to make sure that you keep the scope private because we don't want any outside classes to use the underlying data structure directly. Since we don't want it used, we don't also want to add any getters or setters for it. This array is going to be handled internally only by the basic stack class. So at this point we're almost done with our objectives for this demo. Last thing we need to do is initialize the array to some size so that we can actually use it. A good choice or a place to do this would be in a class constructor. So I'm going to go ahead and add the following constructor. Inside of our basic stack constructor I'm going to go ahead and initialize our underlying data structure. So this line goes ahead and initializes our data structure to a new object array of 1000 and it sets it and casts it back to our generic x type. As far as that 1000 is concerned I just picked this number randomly and I'll talk more about this in a minute. But by hard coding this to 1000, it means that we can hold 1000 items in our stack. That should suffice for the most of our stack needs like the deck of cards, but it's definitely not the best pattern to hard code or initialize something to a hard number like that. So for now that's it for this first demo. Next let's go ahead and take a look at adding our accessor methods like the push and pop.

Demo: Implementing the Stack Methods

At this point we have our basic stack data structure in place, and we now need to add accessors so other code can actually utilize the stack. The first thing that we need to provide access to on our stack is the ability to insert or delete data on the stack. As we've talked before, the terminology for that is push or pop. Then we can add the ability to see if an item is in the stack by adding a contains method, and from there we can easily create a way to access data buried in the stack if needed. For this demo I'm going to start by defining the push and pop method signatures. Here's what I want the push method to look like. The parameter for this method is our generically typed new item that we want to put on the top of the stack. The pop method will look something like this. The returned generic item will be taken off the top of the stack. Now that we have the signatures all that's left is to simply implement these two methods. In order to make our push and pops perform in big O constant time, we need to add a class variable to track the stack location in the underlying array. Since primitive arrays in Java are integer based I'm going to add this class variable. Again, make sure to keep this variable private and don't add any getters or setters for it since we'll only be using this internally in our basic stack class. We'll want to initialize this to zero so I'm going to do that in the constructor by adding this line. The whole point of the stack pointer is to simply increment or decrement it based off of the size of the stack. So you can think of it as an internal pointer on our data structure. So in the push method we can take our incoming item and add it to the underlying data structure array. All this method is doing is taking the new item that's passed in and assigning it to the underlying data structure in one of its 1000 slots. The stack pointer determines which slot it gets. We start with zero but we'll notice that the stack pointer is at plus plus and it will increment it as it puts the new item into a data structure slot. One thing to note, the plus plus in Java is a post increment. It means that the new item gets assigned to the previous value and then the stack pointer gets incremented. So to start off the new item will go into position zero and then stack pointer will be incremented to one so that the next time the new item is associated or another item is pushed onto the stack, the stack pointer will be at one, it'll get placed in there, and then get incremented to two, and so on. For the pop method all we simply need to do is the reverse of the push. We're going to decrement the stack pointer and return the item. So this line of code is doing the decrement before the stack pointer variable. It means that we are going to do a pre-decrement. It's going to happen before we grab the data out of the underlying data structure. So if the current stack pointer is at one, it's going to decrement it back to zero, it's going to pull that value out of the data structure zero slot and return it. Keep in mind though that we are not actually deleting the data from the slot, we're simply returning it and decrementing the stack pointer. So as we move the stack up and down, we're simply overwriting or blowing away any data that's currently in any of those slots with the new values. Before we move off of the pop method, you'll want to think about what happens if the stack pointer is currently at zero and we try to pop something off of it. We'll get a negative stack pointer and that doesn't necessarily work well with Java primitive arrays. So we need to add some guard code in there. Before we decrement we'll go ahead and check to see if the stack pointer is currently at zero and if it is, we'll go ahead and throw an exception saying Hey, there's no more items on the stack, you can't pop anything off. For my implementation I've chosen to throw an exception here, but you could easily make this so that it returns null or does some other kind of behavior based off of if the stack pointer is currently at zero. So next up is the contains and access methods. The definitions of these methods will look like this. The contains signature simply says Hey, do I have the current item in the stack and if so, go ahead return true or false based off of where it's in there or not. The access method is similar, but instead of returning true or false we simply return the item. To implement the contains, we simply need to search through the data in the stack and determine if the object exists. So think about how you can do that and take a stab at implementing it before moving on. So here's how I've chosen to implement the contains method. I'm going to start off by declaring a local Boolean that tells me whether I've found the item or not and I'll initialize it to false since I have not found it yet. This variable will be the variable that I want to return so I'll do that at the end of the method. Now all that's left to do is to simply loop through all of the items in our stack and see if we have found the actual passed-in item. So I've chosen just to use a classic for loop where I initialize the variable to zero, I loop through all of my items and then I do a comparison check on the data slot with the current item passed-in. If I've found it, I'll set the variable to true. I'll break so that we don't continue looping through things, and then return the found item. Obviously, if I haven't found it, it's going to loop through all the items in the stack and then simply return false. And this is a basic linear function. As you loop through all items, that's just one after the other and that's linear performance. For the access method we need to start at the top of the stack and pop off the items until the passed-in item is found. So again take a moment before continuing on to try and implement this on your own before I show you my current implementation. So here is how I've chosen to implement the access method. So while we have items on the stack, I'm going to go ahead and grab each one by doing a pop. So this is actually a destructive operation because I'm popping items off the stack as I try to find them. When I pop the item off, I set it equal to a local variable temp item and then I do a comparison to see if the items are equal. If they are equal, I'll go ahead and return that item. So if the item is only two slots down, you'll find it fairly quick. If it's 999 slots down, it's going to go all the way through all the items until it finds it. And again, to wrap up this method, I'm going to add some guard or check code. In case we don't find the item we need to do something, and I'm going to go ahead and throw an exception in that case again. So if for some reason we make it down to the bottom here, we didn't find the item, I'll go ahead and throw an illegal argument exception saying that we could not find this item on the stack. You may choose to return null or do some other kind of condition when this happens, and that's up to you, and this is what you need to figure out and work with as you build your own data structures or APIs. Finally, there's one more method that I'm going to add, and I'll probably add this to each of my data structures in this course because it's very helpful and it is used a lot, and that is a size method. So here's what a size method will look like for the stack class. All we need to do is just simply return the current position of the stack pointer and that will give us the overall size of the stack. And with that we are ready to test and work with the stack. The basic stack features and functionality is built, and that's all it took. So up next we'll figure out how to work and test with this stack.

Demo: Testing the Stack

Now that we have our basic stack class built it's time to test it out. Since I introduced the stack concept with the deck of cards, we can use a simple main class to test out and work with the basic stack class. I've created some sample helper Java classes that you can use for this course on a GitHub repository. These are simple helper classes just to make testing all of the data structures that we built in this course easy and quick. To access the repository, simply go to github.com/dlbunker/ps-data-structures-helpers. There's a couple of ways that you can use these main classes. You can either clone this repository onto your local hard drive and then copy the classes in appropriately, or you can simply navigate on the website to the class that we're working with and then cut and paste or clone or download that specific class into your IDE that we're working with. So just to keep this simple, I'm simply going to just navigate to the class file that we need and pull it directly from the website. So all the sources located in source/java, so I'll go ahead and go in there, then select the CardStackApp.Java class. If you click on the Raw button, it will give you the raw contents of the file. At this point I can just highlight all of the contents of this file, and create a class in my IDE with the same name and paste the contents in. So back in my IDE we created the basic stack in the default package. I'm going to actually put our testing apps in a new package. I can do that by right-clicking on the default package and click new package. The package that I'm going to call is just simply apps. And now I can create my new class in here, or if you clone the repository you'll copy your new code into here. Okay, the name of the class needs to be CardStackApp. I'll go ahead and click Finish. And now I can simply replace the public class card stack app skeleton holder with the content from the GitHub repository. So I'll paste that in. Okay, so the pasted content has some errors. That's because we haven't imported any of the dependent code or class structures that this class is using. So you'll notice that the first dependency that it can't resolve is the basic stack class that we've created. We've actually created the basic stack in the default package, and here is one weird idiosyncrasy of Java. If you create a class in the default package, you actually can't import that into a sub-package or another package. We've imported the card stack app into an apps package. So we either need to do one of two things. We either need to move the card stack app to the default package, which I don't really want to do because I don't want to mix the data structures in the main classes together. So we can't use that option. But the other option is we can move the basic stack into a non-default package. So I'm going to go ahead and do that just to stick with standardization and help with naming standards. Easiest way to do that is Spring STS is to just move it by refactoring. I can right-click on this and say Refactor, Move, and I can either select to push it into another existing package or I can create a package for this movement. So I'm just going to move it to a package structure called ds for data structures. Click Finish, select that and click OK. My basic stack is now moved into the ds package, and in my card stack app, if we go back to this, we can now import the missing basic stack dependency by doing a Shift-Command-O on the Mac or a Shift-Control-O on Windows and Linux. So go ahead and import my missing dependencies now by doing that, save it, and you'll notice that it added the import ds.BasicStack. So as you start working with Java at a more enterprise level, just watch out for that little gotcha with the default package. Moving forward with this course though we won't run into any more problems because all of our main classes will go in the apps package and all of our data structure classes will go in the ds package. At this point you shouldn't have any compilation errors in the CardStackApp.Java class. So I'm going to briefly cover what this class is going to do. Inside of the main method we create a new card stack app. We're going to go ahead and stack some cards and then unstack them, and we'll do it again, and we'll check the deck size, and we'll do a few other things. The stack card method essentially stacks 52 cards, a deck of cards, onto the stack. It's doing it order so we do the spades first, the diamonds, the clubs, and then the hearts. The on stack card is simply going to do the reverse of that and print out the cards as it pops them off the stack. So, since we ended with the king of hearts as our last stack push, that should be the first card of the stack according to stack rules. I then added a few other helper methods such as if the contains card we have a card in the stack, go to a certain card in the stack, and so on. Yes, I realize now, after I push this code, that my deck size with misnamed but other than that it should still print out the correct size. So what we need to do to run this is we're simply going to just right-click on this and run the main method as a Java app, and I can do that by going to Run As Java Application. Down on the console it will print out what has happened here. So scrolling to the very top, you can see that as we unstack the cards, the king of hearts, the queen of hearts, the jack of hearts, and so on, came off the stack first. If you remember, we started with spades and you'll notice that we ended up with the ace of spades as our last on stack of the card deck. Going back to look at the main method, after we unstack the cards, we restack them so that we want to see how many we have on our stack, we then check the size, and the deck size should be 52. If you look at the bottom of the output, the next line is 52. We then look to see if the stack contains the queen of hearts, and the line after 52 reports that it does, it's true, and we also ask to see if we have a joker. We never stacked a joker and that reports false. And finally we go to the card king of diamonds. So as we access that card, we have to pop cards off the top of the deck to get to the king of diamonds. And when we find it, we can then report the deck size afterwards, how many cards remaining, in this case there's 25 as reported on the console. So that's the basic card app. You can modify this or do some other things, like something that might be interesting to do would be to create a shuffle. Maybe you could pull cards off of the stack and shuffle them around and add them back on. The point is to play around with your stack up. You want to make sure that the stack that you created is actually working and doing what it's supposed to. One other thing to note here is that the basic stack we've generically typed as a string. So all of our cards were simply added as a string, like queen of hearts or ace of spades and so on. We could add a more complex type to put in there. Perhaps we created a card type and on the card we would have an attribute maybe called suit type and another one for the value, whether that's a queen or a two or a four. The nice thing about that generic typing is it allows our stack to work with just about anything. In this case we're using a simple string but we could add a really complex type to add onto our basic stack. So hopefully when you ran this you didn't get any exceptions thrown. If you did, then you probably have a problem in your basic stack implementation. So you want to check that out and make sure that all of your code is working and running before moving on in this course.

Where to Go from Here

Now that we've built a basic stack and we've had a chance to test it out with a little app, we need to talk about some of its limitations. So it's time to think about some improvements that we could make on our basic stack class to make it even more usable. The current stack class we have is a basic Java class with four methods in it. It's built with an underlying object array and it's generically typed. If we ever want to change the underlying implementation to something else, all of our client code would have to change as well. So to prevent direct client code dependencies on our implementation, it would make sense to add a stack interface like this. By creating a stack interface with our four accessor methods, the basic stack can now implement those methods and fulfill the interfaces contract. That means that when our client code uses our basic stack, we can change the code to utilize the interface, rather than the direct class. Then, if for some reason we ever need to utilize a different stack or we create a new stack implementation, the only client code that will change is where the new stack is instantiated. All the other push, pop and other stack code won't have to change. Another glaring problem that you should have noticed when we built our basic stack is that it has a size limit of 1000. There's a couple of ways that you can fix this. You could check the array and reinitialize and resize the array if we run out of room. Or you could switch to a different underlying implementation like an array list. We could add a new implementation called list stack that also implements the stack interface. But this implementation changes out the underlying object array with Java's array list class. This allows for the ability for the stack to automatically grow as items are pushed onto it. The other added advantage of using array list is you can utilize Java generics along type with your current generic stack class that you've built. So at this point for those that really want to understand how data structures and libraries are written, I'm going to give you some homework. When I went through a data structure college class I had to run through a lot of these challenges on my own after learning about the basics of the data structure. And so I learned a lot by doing that and I want to give you that same opportunity. So take some time and see if you can complete the following. So go ahead and make a stack interface and have the basic stack implemented. Then change the card app to reference the stack interface rather than the basic stack. And then you can create a new list stack that also implements the stack class and uses a Java array list as the underlying data structure. And last, change the card app code once again to use the list stack rather than the basic stack. This final step should be quite easy since the only code that will change is where you instantiate the stack class. Instead of instantiating a basic stack you'll instantiate a list stack. Once you're done and you're worked through some of these problems, you are ready to move on.

Core Java Stacks

So we've covered the basic stack concepts and you know a lot more about stacks and hopefully before you started this module, so let's see what the Java API provides out of the box. Java has provided a core stack class for a long, long time. This class extends the vector class which is a synchronized class from early on in the Java 1.0 days. This makes the stack internal threat safe but that safety also comes at a performance cost. Java's been very good about supporting backward compatibility, so the stack classes remain pretty much the same over the years other than the addition of the generic typing. Since the core Java developers try to remain backwards compatible they've added a deque interface as of Java 1.6. That supports all the core stack features and then some. If you want to get a non-threat save stack, you can use the array deque implementation of this interface. If you have a stack with a lot of items, it should perform better than the older stack class. You'll also learn why a deque is a stack coming up in the next module when I discuss queues. You can check out the Java docs for both of these classes and if you really want to get in depth, download the source code and take a look at how the core Java developers designed and implemented these classes.

Summary

At this point we've completed one data structure that we'll be building in this course. I hope that along the way you've thought about some of the problems and issues that we've run into while building the data structures. You have to think about things like performance, the memory pressure, the foot print and other things that maybe you don't quite always think about doing day-to-day coding when you're building a library code such as a basic stack class. All of these concepts are going to help make you a better developer because you're going to start seeing code in a different light after you build a few of these data structures. So let's recap what we've covered about stacks in this module. We first talked about what a stack is conceptually. I used a deck of playing cards for this example but a stack could represent a stack of plates or the Undo feature in your text editor. Next we got our IDE setup and built the basic stack class and created a primitive array as its underlying data structure. Stacks use the push and pop terminology to insert and remove items from the stack. We then implemented these methods onto the underlying stack structure. The big O performance for these are constant time. Constant time is about as good as it gets for performance so these are very performant when you push and pop. Next up we built the contains and access methods. These involve searching through the stack of items giving us linear big O performance. And once we finished this, we put our code to use in a sample app. We then discussed how we could improve on the basic stack implementation, and this ended up with some homework which you hopefully took the time to work through. And finally we covered what Java provides out of the box. You have a couple of stack implementations that you can choose from between the stack and the deque interface classes. I do hope that one of the final takeaways, after completing the stack structure, is that the code really wasn't that complex or hard. It's a fairly minimal amount of code that we added. And by building these data structures, it should have taken some of the mystery and fear away from them, as you take the time to understand them. So up next we'll be diving into queues.

Queues

Introduction

Hey, this is Dan Bunker of Pluralsight and in this module, we're going to be learning about Queues. Queues are pretty much exactly like stacks, but they differ on how data enters and leaves them. Queues are one of the more simplistic data structures out there, it's definitely worth learning, because many of the more complex data structures are built on top of the BasicQueue structure. In this module of the course, we are going to learn what makes a queue a queue, how it performs, what it's useful for and how to build one from scratch. Let's take a look at the queue characteristics and get started.

What Is a Queue?

Queues are one of the easiest data structures to understand, because we deal with the concept of a queue all the time in real life. A queue is simply a line that is ordered by how items are inserted into it. We have lines all around us, we line up to pay for groceries at the store, we wait in lines at Disneyland and often have to wait in lines at restaurants, these lines aren't necessarily queues though, because people can cut in line in front of you and disturb the order of how the line is formed. Here's an example of a line I deal with, living in Utah, this is a ski lift line on a crowded day with the riders trying to get on the ski lift, so is this a queue? No, look at the mess of people jockeying to get in the front of the line, I can never quite understand what kind of line this is, but it's definitely not a queue, however, once you get on the lift, there's no changing order or trying to sneak in front of someone else, each skier or snowboarder is carried to the top of the mountain, where they can exit in the order they loaded, there's no line-cutting, due to the nature of the ski lift. This makes a ski lift a perfect example of a queue and a queue's characteristics.

Queue Characteristics

So let's see what makes a queue a queue by talking about some of its characteristics. We can determine how data is added and removed from a queue by looking at the ski lift. If you happen to catch the first chair in the morning on the lift, you're the first to leave the lift at the top, this is known as a FIFO, First In First Out. The four basic ways you can interact with a queue involves these four operations, you can insert or enQueue data, which in the case of a ski lift, you would get on the lift, you can delete or deQueue data, which is like getting off the lift, you can also search for something in the queue, maybe you're standing underneath the lift looking for your brother on the lift, as it passes by above you and last, maybe you need to access the data at some point in the queue, perhaps once you've found your brother on the lift, you needed to throw him a pair of gloves or something. Each of these operations comes with the performance costs, so let's see how they major up on the Big O scale, inserting and deleting data on a queue is almost identical to a stack, instead of keeping track of the top of the stack, we need to simply keep track of the front and back of the queue, this makes inserting and deleting data constant on the Big O notation scale. Ski lifts are fixed queues, meaning they can't grow in size, they have a limited number of chairs, the loading area and exit area are the pointers for the queue, no matter how many people are on the lift, the loading and exit always takes the same amount of time. Again, searching and accessing data on a queue is linear in performance, just like the stack was, the size of the queue determines how long it takes to find or access data in it, if you have a small ski lift with 10 chairs on it, you won't have to search as long as a ski lift with 100 chairs on it. As the size of the queue grows, the time it takes to search it grows linearly for a worst case scenario. Lastly, queues and stacks really go hand in hand and are very similar overall, the only difference between the two is how data is added or removed from the structure. A queue is a first in, first out, whereas stacks are last in, first out. If you remember from the Stack module, Java uses a deQueue to represent a stack, all the deQueue changes from a queue is how the data is added and removed. The deQueue gives you the ability to pull the last piece of data off of the queue, in addition to the first, giving you a hybrid structure, which makes it a queue and a stack all at once. So that's it for the queue's characteristics, let's go ahead and get started by coding up our queue.

Demo: The Queue Class and Underlying Data Structure

We're going to start off coding a queue data structure, just like we did with the stack structure in the last module. We'll begin by creating the queue class, which will be a plain old Java class, that we'll call BasicQueue, we can then create our underlying queue structure, using a generic primitive array, just like we did for the stack. If you've jumped ahead in this course and bypassed building the stack, you may want to finish that module off, before continuing. Okay, I'm back in Spring Tool Suite and you can see all of the code and work we did for the stack is still here. It's okay to leave it here, because we're simply going to add the queue data structure in the DS package right alongside with the BasicStack. So to create the queue class, I'm going to simply right-click on the DS package and select New Class, I'm going to call the queue BasicQueue and click Finish. You can name your queue whatever you want, but try not to name it just Queue, we want to reserve that name later for some refactoring, just like we did with the BasicStack. To make the BasicQueue as useful as possible, I'm going to add some generic syntax to the class definition, so I can type the queue to whatever kind of data type I want to, so after the class name, I'm simply going to add X and brackets and that'll allow it to be generically typed. If the syntax is confusing, make sure that you check out some tutorials or courses on Java generics. Next, I'm going to add the primitive array data structure, that will represent our underlying queue, I'm going to do that by adding a private class attribute like so. I make sure to scope it private, so that no outside class or code can mess with this underlying data structure, I also set the type to the class's generic type of X, later on when this class gets used, you can think of the X turning into a valid data type, that Java knows, like a string or some other kind of object. I can now initialize the data structure, so it's not null, when the time comes to use it, I'm going to do that by adding two constructors to the basic queue class. The first constructor will look like this, this takes any size argument and initializes the queue to the given size, I still want to have a default constructor though, so I'm going to add one of those, that looks like this. This constructor simply calls the other constructor I created, using the this keyword and passes in 1000 to create a BasicQueue defaulted to a size of 1000 and that's all there is for getting the BasicQueue set up. Next we can add the enQueue and deQueue functionality.

Demo: The Queue Class In/Out Operations

In this demo, we're going to take the queue class structure, that we've just built and add the ability to insert and remove data from the queue. The first method we'll add is the size method. This method will be utilized by our insert and delete operations, so it needs to be added first. Next, we can add the enQueue or insert method, which will allow us to add data to the queue, followed by the deQueue or delete method, which takes data off of the queue. So I'm back in my IDE and I have my basic queue class open. I'm going to start by adding two private attributes to the class to help maintain position of the queue's underlying array structure, these two attributes will look like this. The front and end attributes are array pointer locations to determine where the beginning and the end of the queue lies on the data array, these attributes will work much like the stack pointer did, when we built the stack. Next, I need to initialize these attributes in a constructor, so I'm going to do that in the BasicQueue constructor with the end parameter, I want to initialize the front and end to negative one to start out with, so I can tell if the queue is empty and hasn't had anything added to it yet. If you're confused by the negative one, just think about how arrays are zero based, so negative one is before zero, so once we get to zero, we'll actually have one item in the queue. I can now add my size method and the size method definition will start out like this. By using the data array and the front and end pointers, I can fill in the size method. If you want to give it a shot, before I show my implementation, go ahead and pause the video now. Here's what I came up with. I first checked to see if the queue is empty, by seeing if the front and end pointer is still set to negative one, if so, I just return a size of zero, otherwise all I do is subtract the front from the end and I need to also add a one, because subtracting the front from the end is exclusive in math and I want to get the inclusive count. I'm now ready to start adding the enQueue and deQueue methods, so I'll do that by adding the method signatures like so. I'm going to continue to use my X generic type to represent my data on the queue, again I'd recommend trying to implement these on your own, before continuing on, working on it and thinking about it, before you see my solution will help you build a stronger coding foundation. Here's what I did for the enQueue. The first thing I do is check to make sure there is enough room to enQueue the given data. If I take the overall size and modulus it with the end pointer plus one, which represents the new data coming, it better not equal the front of the queue. The modulus or percent sign may be confusing, but it's simply a division, where the remainder is returned. I'm using modulus here, because the front and end pointers vary up and down on the queue for efficiency and modulus can help me determine where those pointer references are. Next, I have to check to see if this is the first time I'm adding data to the queue, by using the size method we just built, we can determine that quite easily, all I do is simply increment the front and back pointers and assign the item to the end slot on the array. Last, if the previous two conditions don't apply, I simply increment the end point and assign the item to that position on the data array. Now let's take a look at the deQueue, here is what I've come up with. I start by creating a local variable placeholder for the item I'm hoping to return, I then check to see if the queue data size is zero, if so, I can't remove anything else off of the queue, because it's empty, so in this case, I've chosen to throw an exception, you could possibly also return null here, depending on how you want your queue to work. If it's not empty, then I check to see if we are down to our last queue item, if so, then I grab the data and set the front and end pointers back to negative one, we're essentially back to an initialized queue state in this scenario. Last, if we have lots of data on the queue, I grab the next data in line at the front and then increment the front pointer, deQueuing is pretty much just the opposite of enQueuing. Hopefully you notice that we aren't really removing data off the queue, we reference a position in the array and bump the front pointer up. I've chosen to do this for performance reasons, so I'm not reinitializing the array every operation. Later on, if I go to add something to the queue and there's already data in that spot, all I simply do is just overwrite it. For garbage collection purposes though, I may want to also add a null section back into that data spot, I could do that by putting it right after we grab the item off of the array, I will also need to do it right here, that should help to free up some memory. Hopefully you've also been thinking about the Big O performance of the enQueue and deQueue, as you can see, we simply move the pointers around, without having to loop through the array, this means these methods perform in constant time, which is a bonus for this data structure. Next up, we'll add the access and contains methods.

Demo: The Queue Class Contains/Access Operations

The last thing that we can add to the queue class is the ability to find and access items on the queue. For this demo, we'll first create a contains method, which will be followed up by an access method. So back in Spring Tool Suite, in my BasicQueue, I'm going to go down to the bottom and I'm going to add both of my new operations down here, the method signatures will look like this. Again, I recommend that you take some time and try to implement these, before you see my solutions. So I'm going to start with the contains method and here is how I've implemented this. Since the contains method returns a Boolean, I start by initializing a local variable called found and I set it equal to false, if I don't find it, I'll simply just return this and it will contain false. Next, I simply check to see if the size of the queue is empty, if we don't have anything in the queue, then we can't find anything, so I simply return found at that point, otherwise I begin to loop through the primitive data array, I start by looping from the front of the queue to the end of the queue and I inspect each item, in this case, I'm doing an equals on the item, that's passed in versus the item in the data array, if that passed, I set the found equals to true, I break out of the for loop for performance reasons and return found, so as you recall in the Stack module, any time you have a for loop, that creates linear performance and in this case, the queue is no different, so the contains method in the queue is Big O linear. Okay, let's take a look at the access method now and here is what I've come up with this. First off, you should notice that the access method is taking a position and this is a position inside of the queue, we're trying to access something inside of the queue and return its data, since a queue is a array, or essentially a line, we're just saying, hey, get us the second position in this line and return the value to us, so the first thing that I do is I check for some size issues, if the size equals zero or we have a position outside of our size, we want to return that there's no items in the queue or the position that you've given us is greater than what the queue holds. I've chosen to throw an exception here, but you could also do something like return negative one or some other kind of functionality, that makes sense for your queue. Next, I want to loop through the queue, just like we did in the contains method, but I have to set a true index to zero, because the front of the queue is not always going to be at zero, because that pointer moves up and down in the queue for performance reasons, so I do the for loop, where I loop through each item in the queue and I also increment the true index by one, so that I know if I've reached that position, so to tell if you've reached the position, all you need to do is see if the true index equals the position passed in, if that's the case, we just return the data from our for loop. Note that returning out of the for loop will break the loop and return immediately, this is for performance reasons, it's kind of like the break in the contains method, but it just is returning the data, once it's found. If we make it through the for loop, then I also throw an exception saying no item was found at that position. So again, the access method is Big O linear performance, as we loop through the queue, we have to get through possibly all the items in the queue, before we return what we're looking for. So that's it for our BasicQueue, we've implemented most of the functionality, it's now time to test.

Snowbird Lift Queue Test

Since we've been talking about ski lifts for most of this module, I figure we could test our BasicQueue out by coding up a lift at the Snowbird Ski Resort and we can try to load and unload some skiers. To make running the test easier, I've created a small main class, that you can pull from my GitHub examples repo, this is the same repo that we pulled our test class from for the stack class. The URL for the app that we want is located at https colon forward forward slash GitHub dot com forward slash dlbunker forward slash ps dash data dash structure dash helpers forward slash blob forward slash master forward slash source forward slash java forward slash SnowbirdLiftApp dot java, it's quite a long URL, but hopefully you can find it, once you get to the GitHub repo. So at this point, you can either clone the repo and move that file into your IDE or just copy the contents into your IDE with a cut and paste and that's what I'm going to do here, so I'm simply just going to click the Raw button, I'm going to select all the text, copy it and go back to Spring Tool Suite, so in Spring Tool Suite, I'm going to add this over to my apps folder or package, I do that by creating a New Class and the class name was SnowbirdLiftApp, click Finish, I'm going to replace the class definition with the one that I copied from GitHub, but I'm going to leave the package definition in there, since the one from GitHub does not have a package definition. So I've pasted that all in here and we have some errors, but notice that the package apps is still at the top. What we need to do now is add the class references, which is the BasicQueue to our import statements, the easiest way to do that on Spring Tool Suite is to do a Shift + Command + O on Mac or a Shift + Control + O on Windows, so I'm simply just going to make sure my class is selected and do your Shift + Command + O and it should import that BasicQueue, that we just built, automatically for us and all of our compilation errors have gone away. So make sure after you save and import this file, that you have no compilation errors before moving on, so pretty much the two things that you need to adjust is the package name at the top and the import of your BasicQueue. Okay, so let's take a look at what this app does really quickly, first of all, we are building the Gad2Chair lift, Gad2 is one of the two-person chair lifts at Snowbird and that is going to represent our BasicQueue. You'll notice that we generically type it to Gad2Chair, the Gad2Chair is a class defined at the bottom of this file and it represents both seats with the seat name, or essentially the skier name and it has some things like a hash code and an equals, so that we can determine whether this chair is equal to another chair. So back in the main method, this is where the app gets kicked off and we create a new local instance of our Snowbird lift app and then we start the lift up by calling runlift. Looking at the runlift method, we first load the lift, we then print out the size, we want to see if we have Mary and Anna on it, so we're going to check our contains method, we try to see who's on the second position, which is an access method, we unload the lift and then we try to unload the lift to see what would happen again, when no one was left on it. So really quickly, if we look at the load lift, what we do is we instantiate a bunch of Gad2Chairs with some skiers or snowboarders on it and then we queue them up onto the queue, so in this case, we have five chairs, we load them up in order and then we can check them. Later when we go to unload the lift, we essentially deQueue, until we run out of skiers, so since we enQueued five chairs, we're going to look to deQueue five chairs at the top of the lift. Okay, so let's go ahead and run this and see how our queue works, we're going to go ahead and right-click on the class and say Run As Java Application and if we open up our console by double-clicking on that tab, you can see some of the output from the application, so we spit out five, it says that the lift has Mary and Anna on it, the second chair has Samantha and Kelly and then we unload the rest of the lift and spit out the contents and when we unload the lift the last time, we get an exception. All the system dot outs are essentially in this runlift method here and it just tries to flex all of the operations, that we've added to our queue to make sure that it's working correctly. If you want, this is a good time to maybe add another lift or try to load some different people on, keep loading, change the size of the lift, there's all sorts of things you could do to now test out and work with your application. If for some reason, you had an error when you ran this code, you need to double-check to make sure that your lift app is working okay or double-check that you don't have any problems in your basic queue class.

Queue Extra Credit

Okay, it's that time in the module that we can look to see how we can improve our basic queue class. The current queue class that we have is a basic Java class with five methods in it, it's built with an underlying object array and if we ever want to change the underlying implementation to something else, all of our client code would have to change as well. To prevent direct client code dependencies on our implementations, it's going to make sense to add a Queue interface, like this. By creating a Queue interface, with our five access methods, the BasicQueue can now implement those methods and fulfill the interface's contract, that means that when our client codes are BasicQueue, we can change the code to utilize the interface, but if we ever need to utilize a different queue or we create a new queue implementation, the only client code that will change is where the new queue is instantiated, all of the other queue related code won't have to change. Another problem that we have with our current BasicQueue is that it has a size limitation, due to the primitive array, so there's a couple of ways that we can fix this, you could check the object array and reinitialize and resize the array, if we run out of room, or you could switch to a different underlying implementation, like an ArrayList. So if you went through the stack homework, it's quite similar to that, what we can do is essentially add a new implementation called ListQueue, that also implements the Queue interface and this implementation changes out the underlying object array with Java's ArrayList, this allows for the ability for the queue to automatically grow as items are enQueued. The other added advantage of using an ArrayList is you can utilize Java generics, so you can type the stack to match the data, that you are placing in it. So again, here's some extra credit or homework, that you can try to keep improving your data structure skills with the queue. Take some time and try to complete the following, make the Queue interface and have the BasicQueue implemented, then change the Snowbird app to reference the Queue interface, rather than the basic queue, then you can create a new ListQueue, that also implements the Queue interface, but it uses a Java ArrayList as the underlying stack structure and last, change the Snowbird app code once again to use the ListQueue, the final step should be easy, since the only code that will change is where you instantiate the queue, instead of instantiating your BasicQueue, you'll instantiate a ListQueue and once you're done and you've worked through some of these problems, you're ready to move on.

Core Java Queues

Let's take a quick look at what the core Java API provides for queues. The core Java developers have built a Queue interface, much like I talked about in the previous slide, so that they can offer a variety of implementations. Here are a majority of classes, that implement this Queue interface. We've already talked about the ArrayDeque class, when we built the stack, because it can act like a stack and a queue. Some of their queue implementations contain certain underlying data structures, like an array or a list and others that add thread-safe implementations, like the SynchronousQueue. Now that you know exactly what a queue is and how it's supposed to act, take some time and browse the Java docs and look through some of the Java source code to see how the core Java engineers have built their queues.

Summary

Congratulations, you've survived building another data structure, so let's see what we covered with queues. I started off explaining some of the characteristics and attributes of a queue and related it to a ski lift, as an example. Hopefully this concept helped us solidify how a queue is unique and differs slightly from a standard line or list. Next we built out the basic queue class and created a primitive array to hold the data inside the queue, we then built out a size method, followed by the enQueue and deQueue methods, which allowed us to insert or remove data off of the queue, according to the queue's in-out rules, remember that a queue is a FIFO, First In First Out. Next up, we built the contains and access methods, these methods essentially loop through the queue, until the data is found. When you have a loop in your code, you're dealing with Big O linear performance. We then discussed how we could improve on the basic queue implementation by utilizing an interface and hopefully you took some time to do the extra credit homework. Last, we covered what Java provides out of the box. There are many queue implementations you can choose from and each one is dialed in to fit a sweet spot or need. If you need to use a queue in real-life code, look closely at your options and pick one that fits your needs the best. Queues are slightly more involved than stacks were and that has to do with just how we put data on and off of the queue versus the stack. So next up, we're going to be looking at the linked list data structure. This structure takes the concept of a queue to the next level.

Lists

Introduction

Hey, this is Dan Bunker of Pluralsight and this module is all about lists. We've gone over stacks and queues now, and lists are the next logical step when learning data structures. Lists are like queues in the sense that they describe queue like line but how you get data in and out of a list differs and is not quite as strict as a queue. In this module, we'll be building a type of list called a link list. So get ready to learn as we continue on our data structure journey.

What Is a List?

Everything that we've learned about queues apply to lists with the exception of a couple of things. Before we talk about the similarities and differences, let's take a look at a list example. One of the best examples I could come up with is a freight train. Each car of this freight train is in a certain order. There's the front of the train and the back of the train. And the train is constrained to the railroad tracks forcing it into a line. This describes the basics of a list, and so far they aren't too different than a queue. In this module, we'll be building a linked list which is a specific type of list. Each item in a linked list is linked to the next item to it. In this case, this boxcar is a standalone linked list. The size of the list is one. If I couple a tanker car to the boxcar, I've linked the two cars together creating a link list with the size of two. I can keep adding cars to my freight train. So next I'll add a flatbed container car. I now have a list or train with the size of three. Each item or piece of data in a linked list is called a node each node only knows about the other nodes that it's connected to. So the tanker car is aware of the boxcar and the container car, but the container car doesn't necessarily know about the boxcar because it's not directly connected or linked to it. This allows my list to grow however big without any size limitations. This is one of the differences of lists and queues. The queues data structure is handled by an underlying array which needs to be sized out or resized as it grows. A linked list can simply grow as big as it needs to by continuing to link another node onto the list. Let's take a look at some of the other list characteristics.

List Characteristics

Lists are more flexible with how data gets in and out of them. Queues and stacks have really specific ways data enters and leaves them. So how about a list? With a list, anything goes as far as data in and out is concerned. Think about the freight train example. Do new freight cars always have to be added at the end? No, we can add a boxcar in between our two existing cars. We could add it at the front or the back or anywhere inside the list. The same goes for removing cars on the train. You can pull them out in any order or any sequence. Another characteristic of a list is that there aren't any problems with having duplicate data on the list. So we can have a train with a whole bunch of container cars and it's not a problem at all with a list. In the next module where we talk about sets, you aren't allowed duplicate data with those type of data structures. The six basic operations will be building that definer list are add, remove, insert, removeAt, find and get. Let's see how they stack up on the big O scale. Adding data is simply appending a node onto the end of the list, so this remains constant time. And removing data is pulling the first node off of the list which is also constant time. If you only use these two operations on a list, you would essentially have a queue. Because the list also lets you insert and pull off nodes anywhere in the list, those operations are more complex and perform in big O linear time. The find and get are also linear time. What that ultimately means is that these operations will have loops in them as you traverse the list in some fashion. That should give us enough background on lists to get started with coding which is coming up next.

Demo: The LinkedList Class and Underlying Data Structure

If you've been following along with all of our coding demos, you know the drill by now. We're going to start by getting our basic list class created. So we'll start with creating that new linked list class and we'll call it the basic linked list. We're then going to create the node structure which will house the data we'll insert and provide the link pointers to the data in front and behind the current node. Okay, I'm back in my project in spring tool suite and if you remember, we have all of our data structures in the DS package and all of our testing apps in the apps package. So I'm going to begin by creating the linked list class in the DS package. You can do that by right clicking, and going to new class. The name I want to give it is basic linked list. Go ahead and click finish. And we have our basic class structure. Again, I'm going to want to use generics to type this so that we can use it later to be a specific type when we instantiate or linked list. So I can do that by adding the less than sign with an X for my generic type followed by a greater than sign at the end of the class name. I'll go ahead and save that and we have our basic class. I can now create a basic constructor for the class which will look like that. I'm just going to leave it empty for now. When we built our stack or queue, we added an internal private array. We're not going to do that for the list. We need to build a node structure. And the way to do that is to add a node class inside of our basic link list. So at the bottom of the class definition, I'm going to create a private class node. The node's going to contain two attributes. We're going to have a link to the next node. So that's kind of our coupling between for example the boxcar to the tanker car, that's how we couple them by linking each node together. And we're going to have a node item set to our generic type. So this node item would specify what type of node this is. So like a boxcar or a tanker car. Next, I need to create a new constructor which will initialize these attributes. So the constructor will take a generic item, and we'll set the next node to null, and we'll initialize the node item to the past in item. Next I'm going to add three more methods on this private class node, which will help us access and work with the node. These methods are set next node where we can link the next node to the current node. We can get the next node so that allows us to iterate, later on, going from node to node. And we can also get the node item, which allows us to inspect what kind of node this is. So the node represents the basic underlying data structure for the basic linked list. And now we need to add a couple of attributes on the basic linked list that points to the front and back of the list. At the top, I'm going to create two new private attributes called first and last and set their types to node. And last, in our basic linked list constructor, I want to initialize the first and last nodes to null. So I'm going to save that and that's our first coding demo for the linked list. Essentially we've created a basic linked list with a node structure and we've initialized it to an empty list, there's no first or last node at this point. So coming up next, we'll add the operations to get some data in and out of our basic linked list.

Demo: The LinkedList Class Add/Remove Operations

In this coding demo, we're going to go ahead and get some operations on our list that we can add and remove data. So the first thing that we're going to do is define the size method which will help us add and remove data to our linked list. We can then place the add or remove methods on the class and we'll add remove the nodes appropriately. To start off, I'm going to create the size method underneath the constructor. The method signature will look like that. To track the size, I'm simply going to add an attribute called node count, which we can increment and decrement as we add or remove nodes. So inside of the size method, I'll just simply return the node count. And when we initialize the list, I'm going to set the node count to zero. At this point, we can begin by adding some operations to get data in and out of the basic linked list. So we'll start off by putting the add method on here. So the add method simply appins a new item onto the end of the list. So before you implement the add method, take a shot at how you might implement this, give it a try, and then come back and see how I solve this problem. So here's what I've come up with for the add method. So the first thing I do is a check to see if the beginning of the linked list is null, meaning that we don't have any items in the list yet. If it is null, I simply create a new node, pass in the given item, and I set the first and last node equal to each other because we essentially have a list of one, just one node. And the front and back node are the same node. If we already have a front node, what I do is I create a new node, and I call it the new last node, I then set that on the next node of the last item. And then I reinitialize the last node or the pointer to the last node to the new last node item. So these three lines of code is simply just add a new node onto the end of the existing list, link them together and update the last pointer to the last node. And finally, I just update the node count by one to keep track of how many items we have in the list. Next up, we want to add the remove. The remove method signatures looks like this, it returns an item or generic item and it will simply pull the first node off of the list. So again, I recommend pausing the video and take a stab at implement this yourself before you see my solution. Okay here's what I've come up with for the remove method. So again, the first thing that I do is check to see if the linked list is null. So we do that by looking at the first node pointer. If it's null, we can't pull anything off the list. So I've chosen to throw an exception in this case, it just says hey there's no more items on the list. We can't return anything. If I get pass this if check, I then get the first node by looking at the first pointer and I get the node item. This node item is going to be what is returned to the color. But before I return it, I need to update the next node. So I do that by asking the first node for the node that it's connected to, and I update that to become the new first node. Then I can simply decrement my node count to keep my size method in line and I return at the node item for the first item. So you node for both of these methods, we did not do anything by traversing or looping through. We essentially got the pointers and added or removed the item based off of the pointers of our list. So these are constant time. They're quick operations no matter how big the list gets. So for this coding demo, we're going to stop right here because these two methods essentially allow us to add and remove basic data on a list. And if we just left these two methods on the list, we would have a queue because we're pulling data off the front of the list and we're adding it onto the back of the list. But we need to add the insert and removeAt methods which will allow us to put data in and out of the list anywhere on it. And we're going to do that in the next upcoming demo.

Demo: The LinkedList Class Insert/RemoveAt Operations

In this coding demo, we're going to continue by adding some more operations to get data in and out of the list. Those operations will simply be called insert and removeAt and it will take a position that we want to insert or remove data from. Okay, so back on the linked list I'm going to go just below the add method and create a new method signature for insert. This method signature will return void, it will take an item and insert it at a position which is given in the second parameter. So I'm going to warn you, this operation is a little harder than the previous add. But again, take the time to try it on your own before continuing on. Okay, here's what I've come up with. So the first thing I do, is a little bit of code checking. So I make sure that the position that they've passed in is not greater than the current size. Because we can't insert an item on a train that's only five items long at a position of 100. So I threw an illegal state exception in this case, which just says hey the linked list is smaller than the position you're trying to insert at. Otherwise what I do is I get ahold of the first node, and I set that equal to a temporary node variable and then I begin by looping through it. So this foreloop right here is what gets us through looping through our node list. We essentially start by looking at the first node and we try to loop until we reach the position or we run out of the current node. So this is another guard check where it says current node doesn't equal null. We want to make sure that as we get the next node inside of the foreloop, we have a new node to continue looping towards. When the foreloop ends, since we only go to the position that they passed in, the current node is the position that they're asking for. The rest of this code simply severs the linked list, adds the new node, and inserts the node into the new linked list at that position. Again, since we're adding a piece of data, we need to update the node count so that our size method continues to report the correct length of the list. Okay, so that does it for the insert method, let's take a look at the removeAt method. Here is the definition for the removeAt. We need to return the X item, and we need to return it at the position that's given in the parameter. So if you had a hard time implementing the insert, hopefully now that you've tried it and had a stab at it, you can implement the removeAt a little easier. And here's what I've come up with for the removeAt. Okay, so at the beginning of the method, I again check for some code conditions. If the list is empty, which means our first node is null, I go ahead and throw an exception because we can't remove anything from an empty list. Then I want to loop through all of the nodes again. But this time I need to keep track of the current node and the previous node because I'm going to sever the list and take that node out and reconnect the list with the current node and the previous node. So as I start the foreloop, I again start at one and I loop up to where the position is given keeping in mind that I can't have a current node that's not null. So as I loop, I set the previous node to the current node and then I get the next node which is the one beyond where I'm currently at. When I get to the end of the foreloop, my current node is the node that we're trying to remove. So the last four lines of this method essentially takes the node item off of the current node which is what we'll return. We then take the previous node and set it to the current node's next node. This is where we sever the link, pull out the middle and reconnect it. And again I decrement the node count to keep the size method on track and we return the node item. So make sure you save your class. And if you got through the remove at and the insert, those were quite a bit more difficult. I'd recommend taking some time and looking through there and make sure you understand what is going on, essentially our moving nodes in or out of the current node list. And with that, all that's left is to add our find and get. Which is coming up in the next demo.

Demo: The LinkedList Class Contains/Access Operations

Okay, we're almost done with coding and the last thing that we need to take care of is getting some of the finders and getters on our class. So the first two methods that we'll add is the find and the get, and essentially, one just finds the data and the other one gets the data somewhere on the list. And the last piece of code that we'll add is a two string which will just help to print out our list for convenience. So I'm back in spring tool suite, and I'm to go ahead and add the get method underneath the remove method. So the get method signature returns the X or the node item and it passes in a position that we're trying to get the item or node from. So if you've implemented the insert or the removeAt, we had to loop through the nodes for that data. So I'll give you a hint, if you want to try this before seeing my solution, you're going to have to do something similar by looping through the nodes to get the item at the position. Okay, here's what I have come up with. So again, I'm going to check to make sure that our list is not empty. If our list is empty, we can't get any data off it so I throw an illegal state exception. Otherwise, I get the first node and I set it equal to the current node. This time I'm going to loop through all of the nodes from one to the size of the list, making sure that none of my nodes are null and inside the foreloop, I simply check to see if I equals the position. If it does, I return the nodes item immediately and break out the foreloop. Otherwise, I simply get the next node and continue on. If we get to the end of the foreloop and we didn't find it for some reason, I'm going to go ahead and return null. You may want to simply just throw an exception or do something different here, and that's up to you how you decide to implement checks like this. One thing to know, I could've looped through my foreloop up to the position like we did in the insert and remove and just returned that information there. I chose to show you this way which does a comparison inside of the foreloop to use the position and the size differently. Okay, next up let's implement the find method for our list. The find method signature looks like this, it returns the position as you pass in the item. So it's kind of doing the opposite of get. Get you pass in the position, it returns the item, this you pass in the item and it returns to position. And here's what I've come up with for the find method. First off, I again check for an empty list and if so, I go ahead and return that there's no items to find through an illegal state exception. Otherwise, I'm going to go ahead and loop through the list or the nodes just like I did in the get, but in this case, instead of doing a comparison in the foreloop on the position I'm doing a comparison on the passed in item with the current node item's values. If those two things are equal, I return I which is the position of the list. And again, if I got to the end and I couldn't find the item I return a negative one. You could return null or throw an exception here again. That is up to you how you want it to implement your list. Okay the last thing I'm going to do is add a two string method and the reason why I'm doing this is just for a way to print out the list which looks a little prettier than the memory address of the object. And so my two string method will look like this. So two string is a method on the object in Java and I'm simply overwriting that method and doing something with it. So in this case, I simply loop through all the nodes and I get their contents and append it to a string buffer and return the final string buffer content as a string. You don't necessarily need to have this, but I added this because when we add our code demo later when we test our link list, I use it so that we can see what's going on inside of our linked list. A little easier than if we didn't have it. And that wraps up all the coding for the linked list. So as we kind of scroll through this, you'll see that we added a lot of code. The link list was a little bit more complex and verbose than the queue or the stack was. So if anything is confusing to you about what has happened so far, make sure you go back and review and see what is going on. Just follow the code through with the debugger or just line by line to see if you can understand what each part of the link list is doing. Coming up next, we're going to test our new linked list.

Train List Test

Now that our linked list is ready to go, it's time to build a train with it. A heavy train test app on my GitHub rebo that we can use as our test app. If you've cloned this rebo from the other modules, you can find the train code there or you can go here, dlbunker/ps-data-structure-helpers/src/java trainlinkedlistapp on github.com. So I'm going to go ahead and click the raw button and copy the code to paste into my IDE. So I'm going to select everything and copy it and jump over to spring tool suite. Okay, in the project structure we have our apps package and this is where we've been putting our application test code. So I'm going to do the same thing here by right clicking and creating a new class. The class name will be called train the linked list app I'm going to click finish. And this created my new class in the package apps. So I'm simply going to leave the package statement at the top and replace the class definition with the code that I copied from GitHub. Okay, and we have some compilation errors because we're missing some import statements for the basic linked list class and the other items. So I'm going to go ahead and save this, and to import, again I can do shift command O on the Mac or shift control O on other platforms to import the missing statements. So you notice that I imported DS.basic linked list, that's the class I just created for the linked list and that got rid of all the compilation errors. So make sure before you run this, you have no compilation errors. Okay, let's take a quick overview of what we're doing in this test app so you can see what we're trying to flex in the basic linked list app that we created. So the first thing we do is we create an instance of our train which is our basic linked list and we type it to the train car. So let's go take a look at the train car definition. Down the bottom of the app, we have a class called TrainCar, and then here it has a car type and a string. And the car type is simply an enum of different boxcars, tinkers different train cars, and that's down here at the very bottom where it says enum car type. And then the content specify what is inside of the car. So this is our basic data item holder for the train linked list. If you go back up, we're in the main class. This is where the class gets kicked off and started. The first thing we do is instantiate the train linked list app and then we call different operations on the class. So the first thing after we instantiate the app is we build the initial train. If we look at that method, initial train, we create a bunch of train cars and we add it to the linked list. And the way that we're adding it is simply adding via a queue, we're just adding one car after another or linking them in order. After we add all the cars, we do a test to see if the cars are in certain positions and correctly built. So I look for the train car three to get its position. The train car three, if you look at the definition, it's a boxcar full of paper. So after we get the train car three, we want to get its position and then test to see if it was the same train car. And that's what the system.out is providing, I'll go ahead and open this so you can see this better. System.out is saying, hey the train was built correctly and we found the paper car at the position. And lastly, print out the entire train at the end of this method. If you remember in the previous module, we built a two string method that will pretty print the basic linked list. That pretty print is going to be put to use here when we spit out the whole train. After we build our initial train, the train has to make some stops and deliver cars. So the first thing that I do is I added a convenience helper method just to spit out the train size. I can probably just call size on the linked list itself to get the same info, but I for some reason added this train size helper method. And then on the first stop, we're going to do some operations on the list. So let's take a look at the first stop, the second stop, and the last stop to get an idea of what we're performing on the train. The first stop, we're going to pull the first car off the train. We then insert a new car at position one. And then we print out the relevant statements for those operations. The second stop, we're going to remove a car in the middle the train at position five, and we're going to do that several times and then print out the train. And the last stop were to go ahead and remove all of the cars from the train until we have nothing left. And then by the time we get to the end here, we should have nothing left in our train. Okay, let's go ahead and run this and see what happens. So I can right click on my main app, and go to run as Java application and here you can see the printout from the app. So after we built the train, we we're looking for that paper car and we found it, it's a boxcar of paper at position three. And then we spit out the entire linked list train. So you have a boxcar, a flatbed, boxcar and so on. The first stop, we remove the boxcar of Amazon packages and then we spit out the train size again. The train size remains at 10 because we also inserted a car at that stop, which was our farm fence posts and barbed wire boxcar. And in the second stop, we removed cars at position five. So we got rid of all of the tanker cars. And at the last stop, we simply printed out the rest of the train by removing all of the other train cars. So that's it to test the linked list app. That pretty much flexed all of the operations that we added, such as removing, inserting, finding, and getting and so on so make sure that your app was able to run and that your linked list didn't provide any errors. That app should've run correctly, it shouldn't got any exceptions or other items if you ran that. So coming up next, we're going to take a look at core Java classes for lists.

Core Java Lists

Normally we talk about homework or extra credit at this point in the course module, but before I do that I want to talk about the core Java list that the JDK provides. So Java does provide a basic list interface and this interface defines the list contract so all implement classes can adhere to the list operations and implement them accordingly. If you follow the extra credit and homework suggestions for stacks and queues, this concept should hopefully make sense at this point. As far as core Java implementations of this interface, there are several out there, but there are two that are really worth mentioning. First is the vector, this is an extremely old collection class that's been around since the Java JDK 1.O days. The most important thing to know about the vector is that it is synchronized. This makes it thread safe but that thread safety comes at a performance cost. The other extremely popular collection class used all over in Java is the array list. This implementation of the list is not synchronized making it technically not thread safe. You can however wrap it as a synchronized list using the collections class if needed. Chances are, if you've done any programming in Java you have used the array list, it's used quite often. So now that you're familiar with the two core Java lists, let's talk about the extra credit homework coming up next.

List Extra Credit

So you might think they we're going to build out our own list interface similar to what we've done in the past, but I'm going to switch up the extra credit in this module. So here's what you should try on your own. Add some time measurements the main method in the train linked list app to print out how long it takes to build and unload the train with the basic linked list class we made. This code will look something like this. Rerun the train up a few times and jot down the times that it displays. Next, replace the basic linked list declaration in the train app with the core Java array list. This will look something like this at the very top of the train link list app. Import the missing classes and note the compilation errors you get. So fix these errors by using the Java lists method names instead of the ones that we built. Once you get all the errors fixed up, run the train app again and compare the times. And keep in mind that some of the method names are different and throw different errors on certain conditions. For example, we built a method called find but in the array list, the equivalent method would be index of. Utilize the Java docs for the array list to help discover the appropriate methods. Once you've ran the app a few times, which implementation was faster? On my machine, the core Java array list beat out my basic link list by a millisecond or two on average. The real reason for doing this extra credit was to introduce you to the concept of metrics. Sometimes you need to figure out what data structure or implementation of a data structure is needed to squeeze out every last ounce of performance. By adding some basic time calculations, you can start to answer these kind of questions. The real question you should be asking though is how is my time performance on a train that is one million cars long? Bumping up the size of your array list or the basic linked list would provide more interesting metrics and that should be something that you should take some time and play around with.

Summary

We've completed another data structure, so let's see what we learned. Again we began this module with learning about what makes a list a list. And this includes its characteristics. So remember that lists are pretty much exactly like queues with the exception of how data enters and leaves them. Lists also allow for duplicate items. Next, we built our basic link this class with the underlying node structure. We then went through four coding demos that built the six basic operations of our list that allowed us to add, remove and find data in the linked list. We then covered what Java provides for lists and that is a list interface, the basic vector and array list implementations. And finally, we went through the extra credit where you put our list to the test with the Java array list. This involved a little code re-factoring and some time comparisons. This was a very simplistic and easy way to gain some metrics on the code. Next up on our data structure journey, we're going to be tackling sets. Make sure you're caught up at this point because we'll be comparing sets to lists in the next module.

Hashes

Introduction

Hey, this is Dan Bunker with Pluralsight, and the data structure we'll be covering in this module is the hash. Hashes are often quite performant when retrieving, storing, finding and deleting data regardless of the hash's structure size. There's some unique gotchas with hashes though that we'll uncover when we start to code one up. The type of hash we'll be building in this module is a hash table. Before we get to that point though, let's talk about what a hash is and how it works.

What Is a Hash?

A hash has a real life example that we can use to help us understand the concepts, just like the other data structures that we've covered so far in this course. You can think of a hash as a set of items. In this case, we'll compare it to a tool set. A socket set has a variety of sockets and wrenches, and have all been situated in a custom place in the toolbox. The first thing that should be obvious is that there's no certain order with the tools and the sockets. They are all just kind of presented to you. This type of dataset is not like a line or a queue where data follows a certain order inside of the data structure. You're free to choose any of the tools as needed in any particular order. Hashes aren't first in first out or last in first out, but they're maps. They map a key to a value. So for the socket set, I have a key called 20 millimeter, and it's mapped to the 20 millimeter socket. When I need the 20 millimeter socket, I look for the 20 millimeter marking on the toolbox and I then know that the socket there is the 20 millimeter socket. Primitive arrays are also hashes if you stop and think about it. Even though it looks like a primitive array is a list or queue where you have one element after the other, the primitive array maps each item in the array to an ordinal. Arrays begin their hash with the number zero and increase one at a time as the array elements increase. The only real difference between an array and a hash structure is how the key hash is generated for the element. Coming up next, we'll talk about how the hash value is determined.

Determining the Hash Value

To completely master and understand hash tables and hash sets, you need to understand how the hash value for the key is determined. Think about the keys in a hash structure as a set of actual keys. Even though keys in real life contain duplicates, think about your hash key set as a bunch of unique keys that unlock the door to your item or value. The uniqueness of your key is the hash code. So if you look at these two keys, you can see that they are different and their shape is unique, giving them each a unique hash code. So how do you get a unique hash code for data keys in programming? Going back to the tool set example, we might have key values like 8 millimeter, 10 millimeter, 12 millimeter and so on that maps to the actual tool sockets or items. We need to take these strings or whatever other type of data the key might be an generate an int value that is unique hash code for the key. Java provides a method on the base object called hash code. Its method signature looks like this. Since all objects inherit from object, each object can overwrite and implement this method so that you can create a unique hash code for your object's data type. So here's the code for how the string class in Java implements the hash code. It uses each character and the string's length to generate a unique hash code integer for any string representation. We'll rely on this method to successfully build our hash table. Normally on this slide, I talk about the big O notation for the operations it will be building. I'm going to leave that out in this module and discuss it later after we've built each operation to give you the chance to figure it out as we build it. So hopefully by now, every time you create a method, you should quickly think about how it stacks up on the big O scale. Take a look at this string's implementation of the hash code method, can you determine its big O value? If you saw the for loop and guessed linear performance, you were correct. So that's it, let's go ahead and get coding.

Demo: The HashTable Class and Underlying Data Structure

The hash table may seem a little mysterious still because it's not quite in ordered and apparent as a list or a queue. Coding one will definitely take the mystique out of this data structure. So we're going to start by creating the hash table class and give the class a name of BasicHashTable. We can then create the underlying data structure that will hold everything and create a class to hold the hash's mapped key and value. Okay, I'm back in my IDE, and I'm going to go ahead and create our new basic hash table class. I'm going to do that by going into the ds package, right clicking and creating a new class. The class name will be BasicHashTable. I'm going to generically type this data structure like I've done with the other ones, but it's going to be a little different. I'm going to give it two generic types, an X and a Y. The reason why I'm giving it an X and a Y is the X is going to represent the key generic type and the Y is going to represent the item or the value generic type. The underlying data structure for the basic hash table is going to be a hash entry. And I can create a private class hash entry which will represent this basic structure. You'll notice that, on the hash entry, I also pass in an X, Y generic type for my key and value. I can then create two private attributes that represent the key and value for the hash entry. For convenience, I'm going to create some getters and setters to help us set and retrieve these values. So I'm just going to use my IDE to do that by right clicking and going to Source, Generate Getters and Setters. I'm going to go ahead and Select All so that we get one for the key and the value, and then click OK. The last thing I need to do on my hash entry is create a constructor that initializes the hash entry appropriately. And that will simply look like this. We take in the X key and the Y value, initialize it to our private attributes, and we have a hash entry that represents our key value pair. I can now go back to the basic hash table class and create the underlying data structure, which is simply going to be an array of our hash entries. And that'll look like that. Next, I'm going to add a private attribute that represents the capacity of the hash table. That's just going to be an int, and I'm going to call it capacity. Capacity is going to be different from size, because size is going to represent how many items we have inside of the hash table, but the capacity represents how the big the hash table is. So if you think about the toolbox, the toolbox can only hold so many sockets. This is the capacity of the toolbox. And last up in this coding demo, I'm going to create a constructor for the basic hash table, which allows us to set the initial table size. And that constructor will look like this. The incoming int tableSize parameter will determine our capacity and how big we want to initialize our underlying data structure. So to finish off our constructor, we're simply going to add the following code. I'm going to take the incoming table size parameter and initialize this.capacity to that value, and then I'm going to use the capacity to initialize our hash entry data underlying structure. Later on, we'll talk about how the hash size or the capacity of the hash table is important to prevent collisions, which can be quite problematic if your hash size is not big enough to deal with your incoming items. But for now, that ends the coding demo for this exercise, and we'll get ready for the next one.

Demo: The HashTable Class get/put Operations

Now that our basic hash table's in place, we can add a couple of methods that are used quite often with our hash. Just like before, we're going to begin by creating a size method that tells us how many hash keys and values the hash table has in it. We can then create the get and put operations, which is our main way of getting information in and out of the hash table. So I'm back in Spring Tool Suite and I have my basic hash table open. The first thing that I'm going to add for this coding exercise is a private variable which will hold the size. That'll be typed to an int and it will look like that. Next, I can create a method that will return the size value. That'll look like this, and I'm simply just returning the actual size. Since size has not been initialized anywhere, I'm going to go ahead and do that in the constructor that we built in the last exercise. I'll simply initialize it to zero since, when we create a new hash table, we probably won't have any data inside of it yet. Next, I'm going to create the get method which will be slightly different than some of the other operations we've created with other data structures, and it will look like this. The get method takes in a key and it returns the value. Normally I'd tell you at this time to go ahead and try it on your own, but I want you to follow along with the get implementation because it's going to be a little different, and then I'll have you try the put implementation on your own. So before I actually implement the get, what I'm going to do is create a private method on our class called calculateHash. What this private method does is it takes the incoming key and it gets the hash code off of it. Remember that hash code is a default method that is provided by Java, and you can either override or implement it or use the default implementation. Since we don't know what the object coming in is because it's generically typed to X, we need to rely on the provider of that data type to properly implement the hash code. Okay, so the first thing in this method is we take the hash code off of the key and we modulus it or percent with this.capacity. So what we're trying to do is find a slot inside of the hash table using the modulus and the size. That'll return our hash slot. So don't get the hash code confused with the hash slot, the hash code is unique, but the hash slot may not because we're limited to a certain number of entries or the capacity inside of our hash table. Okay, so once we have the hash code, what we can do is check to see if it is the appropriate one to use for the incoming key. Since it's not unique, we have to see if it's null or not for that hash spot on the data array. So this while loop checks that. So if it's not null, that means we already have an item there and that's called a collision. So we have to move on to the next one. If the item actually equals the key, then we're okay because that is the hash for this key. But if we don't have a hash that's appropriate for the key, we have to move up one until we find an empty spot. So this is why you want a larger capacity than the size of your hash table, so that you can prevent collisions. Because we want a lot of options to be able to move up in case we have a collision. So once we found an appropriate hash slot, we'll go ahead and return that hash slot as the int to our color. And this we're going to use to set up our get method. Okay, so the first line in the get method, we're going to call the calculate hash by providing the key, and we're going to store that as a local variable called hash. So first thing I'm going to do is check for some null conditions. So if they give us a key and we look in the data hash spot with that key and it's null, that means that we could not find a value for that key. Otherwise, what we're going to do is return the data value inside of that hash. So when we get the data hash, we're going to get the Y value which contains an actual value on it, there's a key value pair for each hash slot or a hash entry, and we want to get the actual value for the key. So what we do is we go ahead and hash that entry, and then we get the value back and cast it to Y, which is our generic value type. And that's all there is for the get. So now that you've seen the get, think about implementing the put. I'll show you what the put looks like before you can try it out on your own. The put method definition looks like this, it returns void and it takes in two parameters, a key and a value. What you want to do is you want to store the value in that key slot. What you are going to want to do is utilize the calculateHash method, which is a private method on this class that we created. Okay, so here's how I implemented the put. So I start off by getting the hash from the key. I then create a new hash entry, I store its key and value in that hash entry and I set that on the data hash slot. And last, I simply increment the size because we've added something to our hash table and our size has increased. Okay, so last but not least, let's talk about the big O notation for these two methods. If you look at the get and the put, what do you think their big O notation would be? Assuming best case scenario where we don't have any collisions, these are going to be constant time, because we essentially can get the hash almost immediately from the key, and then grab or put that data inside of our underlying data structure. So it's quite performant. If you look at the calculateHash function that we created though, this kind of throws a wrench in the constant time, because if we end up with a lot of collisions, we can possibly turn our gits and puts into linear time because we're looping through all of the capacity or the keys using the while loop. So hopefully, your data structure was initialized to a large enough size that it prevents collisions as we hash our keys. And that's it for this coding demo. Coming up next, we're going to learn how to delete items out of the hash table and find things in the hash table.

Demo: The HashTable Class Delete/Contains Operations

Even though the core methods in the hash table are going to be get and put, this exercise is going to add some helper methods which kind of polish and finish off the data structure. First, we're going to add a delete method, this will remove a key value pair from the hash table altogether. And finally, you will add a couple of finer methods that will tell us whether a key or a value exists in the hash table. Okay, so let's go ahead and start with the delete method. The method signature will look like this. The delete method, I'm not going to lie, is one of the more complex methods inside of the hash table. So if you want to try it on your own, you're more than welcome. But I'm going to go ahead and just show you my solution and we'll talk through it. It's kind of large, so I'm going to go expand this open to see it all. Okay, so inside of the delete method, the first thing that we do is we try to get the value. So we utilize the get method that we've already built to do that. Next, I can check to see if the value is not null. So if the value is not null, then I have actual data to clear out. If the value is null, we don't need to do anything because the data isn't there to begin with. Okay, so back inside of the if block, the first thing that we're going to utilize is our private calculateHash method. That's going to return the hash slot that we're interested in removing. Once we get the hash, then we can go ahead and null out the data structure for that array slot. We then decrement the size by one, and the next line is where this method gets tricky. After we decrement the size, I look for the hash in the next slot up. The reason why I do this is we might've had a collision. And if we had a collision, we should have data in the next slot up. And if we have data in the next slot up, we have to actually go back and rehash all of our items to keep our hash table in line. And that's what's happening here in this while loop. So while the data hash isn't null, meaning that we found another collision or a possible collision, we go ahead and grab that value, which is our hash entry, we set it to null, and then we re-put it back in the data structure. The put method that we're calling here is actually rehashing that structure back in. Since we removed an item, the hash coming back in will either put it back in the normal slot, or it will find it and collision with another item, and go ahead and increment it and put it in. Either way, removing data from the hash structure is difficult because we have to rehash all of our items so that our hashes remain pristine. And even though we put it back, we need to decrement the size because we didn't really add a new one, we just rehashed it. And finally, we up our hash again to see if we have another collision, and we keep going until we run into an empty slot in our array, which means we have no more collisions. So this implementation can also be problematic because if we have a hash table that's completely full, we can run into problems because we're going to put the really expensive rehash, as we remove one item out of it, we'll probably have a collision all the way through the entire capacity of the hash table. Okay, moving on, let's go ahead and talk about the hasKey method and the hasValue method. These two methods simply tell us whether we have the key inside of the hash table or whether we have a value inside of the hash table. So if you want to try these two methods on your own, they're not quite as hard as the delete, and you can try taking a stab at them before we implement them here. Okay, here's how I implemented the hasKey. Again, I start off by utilizing my calculateHash. I then check to see if the hash is null. If it's null, we don't have anything in there, so we can just simply return false. Otherwise, I'm going to get the key out of the hash entry and check to see if it equals the key passed in. If it does, then we'll return true. And we can determine that the key exists in the hash set. If we don't find it that way, we can also just simply return false. Now note, there is another way that you could possibly implement this method. You could loop through all the items in the array, simply checking the key value on the hash entry with the passed in key. I've chosen to utilize the calculateHash with the hash functionality to try and increase performance. Okay, so let's take a look at the hasValue implementation. Since we're passed a value and we're not given a key, the only option that we have is to loop through all of the items in the hash table and look at the hash entry's value. So that's exactly what I do, I start with the for loop and I look through the capacity. 'Cause I want to check every slot, I don't want to use the size because we might miss some slots. Because the size only tells us what's in there. The capacity will let us look at each individual slot in the hash table. So for each entry, I then take that and I check to see if the entry isn't null and if the value of that entry equals the value passed in. If so, I go ahead and return true. Otherwise, if we make it to the end of the loop, we did not find that value and return false. Okay, so let's talk about big O notation for hasKey and hasValue. HasKey is going to be close to a constant time depending on how well our calculateHash function performs. The hasValue, on the other hand, uses a for loop, so it's going to be linear time because any time you loop, it's always linear time inside of big O notation. The delete method is a little harder to comprehend on the big O notation side because we're rehashing. So it really depends on how many collisions we have. In a worst case scenario, we would have to loop through every item and rehash it. The rehash could also screw things up causing another rehash. So this big O notation could go from linear all the way up to possibly quadratic. Okay, go ahead and make sure that your hash table is saved, and that is it for the coding of the hash table. Next up, we'll go ahead and test it out.

Oil Change HashTable Test

Okay, so sticking with the tool set example that we've talked about in this module, the test app for the basic hash table will be to change the oil on a motorcycle using the hash table's tool set. Okay, so again, on my ps-data-structure-helpers repo on GitHub, I have a sample class you can use to pull to test the basic hash table out. You can find it at MotorcycleOilChangeApp.java, which is in the src/java folder. Either clone and move it into your IDE, or you can just copy the raw contents, which I'm going to do by clicking the raw button then doing Command + A on Mac or Ctrl + A on other operating systems, you can select all the text and then copy it. Okay, so back in my editor, I'm going to go ahead and open the apps package, create my new app, a new class called MotorcycleOilChangeApp. Go ahead and click Finish. I can then replace the class definition with the one that I copied from GitHub, making sure to leave the package apps at the top. Okay, so I have some missing class information that I need to import, and that basically is the basic hash table. So go ahead and do Shift + Command + O on Mac or Shift + Ctrl + O on Windows or other operating systems to import that. You can save that, and I should have no compilation errors at this point. If you still have compilation errors, double check your code and make sure that everything is in line and that you've named everything correctly. Okay, so let's take a look at what this app does. So first, we instantiate a tool set. We're setting it to a capacity size of 12. So this is actually very small. I set it to this size to demonstrate how rehashing will work as we use that and how it kind of has to go through and push information in and out of the hash table to keep it pristine. If I wanted this to be more efficient, I would set the basic hash table size to something like 3,000. Okay, so this app has a main method, it's just a Java app, and it goes ahead and instantiates the motorcycle oil change app, and it starts by changing the oil. So the first thing we do when we change the oil is we are going to get our tool set. And building our tool set, we simply just populate our hash table with a bunch of sockets. We then have to remove the skid plate off of the motorcycle, drain and replace the oil, replace the skid plate and then empty the tool set. So empty the tool set just simply pulls all of the items out of the hash table, which will flex the delete method that we created, while building the tool set puts them in there using the put. The removing the skid plate is going to go ahead and use specific tools from the tool set, like the 10 millimeter socket, and then it does some work. And the same thing with replacing the skid plate, just it's the opposite of remove. And then the drain and replace oil, obviously, it needs to use certain tool sets. Now again, this is kind of a silly app because I'm just spitting out text and I'm using strings for things, but you can get an idea of how this would work if you already use it on something else. Okay, so to run the app, I'm simply just going to right click and say Run As Java Application. So if you open up the console, you can see, after we build our tool set size, we have tools that are size of 12. We then do the work, we remove the skid plate. We use the 17 millimeter socket, we drain the oil. We go ahead and replace the oil. Put the skid plate back on. And then finally at the end, we remove all of the tools out of the tool set. So to get an idea how the rehashing works, let's go back to our basic hash table, I'm going to put a system.out in the delete. So I want to do that here where we need to do a rehash. Okay, so the system.out simply just says, "Hey, we're rehashing this key value pair "because we found a collision when "we went to remove the item." So when we go ahead and empty our tool set on the next run, we should see these printed out. So I'm simply just going to run this again. Open up the console tab. Now you can see what happens when we rehash. So right here, we finish replacing this skid plate on the motorcycle, and we're emptying out the tool set. So you notice that the rehashing took a lot more than 12 times. This is where I said because we have a limited number of size of capacity in the hash table, the rehash took a lot longer. If we did not rehash and we simply remove the items, you would get errors because it would think that items have already been removed and the hash keys inside of the hash table are not pristine and appropriate. So for fun, if you wanted to try that, go ahead and comment out this while block or while loop and try the MotorcycleOilChangeApp again and see what happens when you empty out the tool set. And that's it for our app test, let's go ahead and see what Java provides on the hash table and hash world.

Core Java Hashes

Java provides a couple of core hash structures that line up similar to the list structures that we've already talked about. The main interface for a hash in Java is the map. The map's main purpose is to map a key to a value. Java provides a variety of implementations of the map, but there are really two worth noting. First implementation is the hashtable. This structure is a synchronized data structure, just like the vector was in the last module, and the hashtable has also been around from the very beginning, starting with Java 1.0. Most developers these days generally use the HashMap. This structure is not synchronized and is a little more performant than the older hashtable. The HashMap structure also allows for nulls in the value, whereas hashtables do not. As far as functionality is concerned though, the HashMap and hashtable are pretty much the same thing. If you need to use a hash structure in your code, you'll probably be looking to use the HashMap. Now that you're familiar with the core Java hashes, let's go ahead and talk about the extra credit work coming up next.

Hash Extra Credit

Now that we built a few data structures at this point, hopefully you've noticed the hash structure provided some unique challenges. The homework for this module will also be unique. One of the problems with a hash structure is when you run into collisions when putting data into the hash. Collisions slow down the performance of the hash, which kind of defeats one of the main points of using the hash. Instead, if you can think of your hash structure as a parking lot, you want to be able to have enough room for each car and still have some parking spaces left over to accommodate any new vehicles. So for this module's extra credit, think about how you can reduce the collisions inside of your hash structure. I'll give you a couple of hints, one is easy and the other is a little bit harder. First, what would happen if you increased the initial capacity of the hash size? Would this help reduce collisions or increase them? How does this affect your code's memory footprint? This is the easy test that you can perform to see how it affects collisions without having to change any code. Utilize some metrics like we talked about with lists in the last module to help gauge timing and memory use. Second, all hashing in Java is determined by an integer type. That gives you 32 bits of information. Java determines an object's hash value by the hashCode method. If you implement the hashCode method using something simple like an ID or another simple attribute, your hash has a higher chance of producing the same hash value. You could improve the hash code method by utilizing a custom object type that does its best to determine a unique integer for that object. So instead of using a string to represent a socket tool, you could create a socket class and generate a hash code that takes into account the size, color, depth, metric or SAE value to try and improve the hash value's uniqueness. Finally, take a look at some of the core Java classes to see how they implement the hash code method to get an idea on how to write better hash values.

Summary

At this point, we've reached the end with hash data structures. These structures are used often in programming, and you should definitely feel confident with what they are and how they work. So here's what we've covered. We started off by talking about what a hash data structure is. The key point to remember is that inserting and accessing data is very performant because of the concept of the hash. A hash is a way to create a unique integer value from the key. Keys are unique in a hash and point or map to its value. We then built out the basic hash table class, the hash entry class and the underlying data structure for the hash. By using the unique hash value and the modulus function, we're able to determine the hash entry's placement inside of the underlying data array. Next, we added the basic hash operations. These should've been pretty straightforward, but there were two things that were tricky. One was dealing with hash collisions, and the second was handling the rehashing of data on delete. If you're still unclear on these two things, go back to the coding demos and make sure that you understand what the problems were and how we solved them. We then talked about the Java map interface and the hashtable and HashMap implementations. HashMaps are generally what people use now and are slightly more performant due to the fact that they aren't synchronized. Last, we went through some extra credit work where we tried to reduce the collisions you might encounter with a hash to keep its performance top notch over time. And with that, we only have one data structure left in the course, and that's the binary tree which is coming up next.

Trees

Introduction

Hey, this is Dan Bunker with Pluralsight. For the final module in this course, we'll be covering tree data structures. Trees are often used to help sort and find data quickly, but inserting data and keeping the tree balanced tend to be problematic. In this module, we'll be building a binary tree data structure. I'll briefly cover generic tree characteristics, which will help give you a good foundation for all types of trees. If you've made it this far in this course, you should be feeling pretty strong with data structures overall. Finishing off your knowledge with trees will help to solidify your status as a data structure master, so let's get started.

What Is a Tree?

Tree data structures are aptly named since they literally define and look like tree structures. Trees are hierarchical in nature. Here's a representation of the animal kingdom's classification in tree format. Each node in the tree can have one of three states. It can be a parent node. It can be a child node or it can be both a parent and a child at the same time. Trees are very common to us in the real world and here are some other examples of trees. We're born into a family tree with ancestors and descendants branching out. Most organizations are hierarchical in nature with a president, vice-president and so on. The file system on your computer is another commonly used tree structure. Folders can be nested in other folders and documents are leaf nodes of a parent folder. Finally, the document object model or DOM of a browser webpage is a hierarchical tree structure as well. Next up, we'll focus on what a binary tree is and talk about how that works.

Binary Tree Characteristics

Binary trees are a specific type of tree data structure. Binary trees start with a root node and then can contain up to two child nodes, a left and a right. The binary name refers to how the child nodes are limited to two nodes. There's no limit to how deep the tree hierarchical nodes can go. The real advantage of using a binary tree comes down to how data is added and stored into the tree. To understand how to add data, I'll show you how to store some of the presidents of the United States in a binary tree. The first piece of data added to the tree is placed in the root node position. I'll start by adding President Lincoln as the root node. The next piece of data added will be compared to the root node and determined if it is greater than or less than the root node's value. Since we're working with strings, the next president will either be less than or greater than Lincoln alphabetically. If the next president we add is Jefferson, J comes before L, so we place Jefferson as the left node off of the root. The next president added is Washington. That will be placed as the right node since W is greater than L in the alphabet. Any new data added now will get placed on one of the two child nodes. The same rules apply, except we base the comparison on each node as we move down the tree. If I place President Kennedy in next, he'll go here because he's less than Lincoln, but greater than Jefferson, so he ends up as the right node off of Jefferson. Jackson is less than Lincoln and Jefferson, so he's the left node on Jefferson. I can keep adding presidents based off of the rules like so, Madison, Adams, Roosevelt, Buchanan and so on. You'll notice that over time the tree becomes unbalanced. The binary tree doesn't need to always have two nodes per parent. The tree additions are based on comparisons rather than keeping the tree balanced. Because the tree data is dispersed based on comparisons, it makes adding and finding data quite efficient. Even though the total items in this tree are nine, it only takes two steps below the root to find Kennedy and one to find Washington. Compare that to a sorted list, where you would have to traverse all the way to the ninth position to find Washington. Because of the tree like structure, this makes the Big O notation logarithmic, which is one of the best access performance you can get out of a data structure. Good performance comes at a cost though because coding a binary tree is one of the more complex data structures that we'll be building in this course. Get ready because we're going to start on that next.

Demo: The Binary Tree Class and Underlying Data Structure

For this module, this will be the easiest coding demo that we'll do and will be similar to the previous data structures that we've created. We'll start by creating the basic binary tree class and then implement the binary tree node class structure and setup the root node. I switched over to Spring Tool Suite and I'm going to open up the data structure package that we previously been building our data structures in and this is where we'll add the basic binary tree. Go ahead and right-click on the DS package and say new Java class. I'm going to call my tree name BasicBinaryTree. Click Finish. We have our new class. Previous data structures we have been generically typing them and we're going to do the same for the binary tree, but this is going to be slightly different. At the end of the class name, I'm going to add the following, I want my generic type to extend the comparable with a generic type. If you remember when we talked about binary trees, they have to compare each node to each other. This comparable generic type will allow us to do that. Since trees are made of nodes, the next thing that we need to do is define the node class and I'm going to do that as a private class with inside of the basic binary tree and I'm just going to simply call the class node. If you remember, each node in the binary tree can have a left or right node, so I'm going to add that inside of here and this now allows us to have a node with two children, a left and a right node. To make things easier, I'm also going to add another node, which is a reference back up to the parent node. Really quickly, here's what all the nodes represent, the private class node represents the current node that I'm on. It can then have two children, a left and a right node and it can also have a parent. Those nodes represent the tree structure, but we need a place to store the value. I'm going to add a generic type X, called item, which represents our item value for this particular tree node. This represents all the attributes that my node class needs, so I can now add a constructor for the node. The constructor will take in a generic X item and initialize it to the item that we defined above. As far as the nodes, the left, right and the parent, we simply initialize those to null because we do not have that information set yet. The basic binary tree structure will take care of setting those nodes for us. All it's left to do for the private class node now is to implement getters and setters for all of our attributes. I'm simply going to do that by right clicking inside of that class and going down to the source, generate getters and setters. I'm simply going to just select all and put these as the last member of the class and click OK. This will generate getters and setters for all of the attributes in our node class. Go ahead and save that. At this point, our basic binary tree has been defined and we defined the node class. We can go ahead and use that node class to set up an attribute for our root node, which is where our basic binary tree's life will begin as we add data to it. Jumping back up to the basic binary tree class now, I can now create a private node and I'm going to call it the root. This represents the root node of the tree. Finally, I want to create a constructor for our basic binary tree and that will look like this. We won't take in any parameters or arguments and we're simply going to initialize the root node to null. This comprises our basic binary tree data structure and underlying node class value. Coming up in the next coding demo, we'll start to learn how to add data to our binary tree that we've just defined.

Demo: The Binary Tree Class Add Operation

Now that we have the basic binary tree class, we can get started on some of the tree functionality. Again, we'll create a size method to help us implement and use the tree and then we'll implement the add method. Here's the tree class that we just built and I'm going to start by adding a new attribute called size. This will just be an int data type. To go along with that I'm going to add a method called size underneath the constructor that simply just returns our attribute. As we begin to add data and later remove data, we'll need to make sure that we increment and decrement the size attribute appropriately. Next up, we're going to create an add method, which looks like this. We're going to take in a generic X item and we're going to add that to our tree structure. If you want if you want to try to implement this yourself, you can go ahead and try that now, but the trees are a little bit more complex because they're hierarchical in nature, which ends up doing a lot of recursive programming. If you simply want to follow along that should be fine too. Before I fully complete the add method, I want to create a helper method I can use inside this class, which will help me with some of the recursive nature. I'm going to create a new method called insert. The insert method will take in a node, which is the parent and a node, which is the child and its whole purpose is to associate those two things together. If you remember back to the president binary tree that we did, you have to compare nodes as you add them. What we want to do is we want to compare the child node to the parent to determine if it goes on the left side or the right side. I'm going to start the comparison by looking to see if this child is a left child node. Here's what this code is doing, the first thing that we do is we get the item off of the child node and we compare it to the item off of the parent node. The .compareTo is Java's comparable method that determines whether something is less than, equal to or greater than another value. If it's less than, it returns a number less than zero. If it's equal, it returns zero and if it's greater than, it returns a number greater than zero. To determine if the child is less than the parent, I simply need to compare it to the parent's item and see if it's less than zero. If it is less than zero, then I have a left child for that parent node. The first thing I need to do is see if the parent already has a left node. That's happening here on this if check inside. If the parent's left node is null, I then can add this child. I simply set the parent's left child to this child and I set the child's reference back to the parent. I then increment the size and we successfully added a node to the left side of this parent node. If the parent's left node is not null, we have to drop down and do this again. What we're going to do is we're going to call the insert method again and this is where the recursive programming comes because I'm calling insert, which is this private method that I just created. In this case, I'm changing the parent node to the left child of the parent, where I'm trying to add this child to the new node that I'm recursively dropping down into. That case takes care of adding a left child. If we want to add the right child, we're going to do something very similar. This else if condition determines whether we have a right child or not. It's pretty much the same logic as the left child, but the compareTo method is looking to see if our value is greater than zero. Again, this is a comparable object and it's part of the Java core API that tells us whether these two objects are greater than zero. If that comparison passes, we can then check to see if the parent's right node is null, similar to what we did on the left side. If it is null, we can simply add this child to the right position, set its parent reference back and then increment the size doing this.size++. If the parent's right node is not null though that means that it already has a right node and we can't add it there, so we're going to drop down to the parent's new right node and do the comparison all over again, which happens down in this else block. This private insert method can go as deep as needed to add a node way down on a really large binary tree or it can work directly off of the root node. Finally, you might be asking what happens if the comparison comes back as equal to zero? Meaning that both the child and the parent are equal. That's a good question and you can ultimately decide how you want to handle that since we're building these data structures however we want, I'm simply just going to add a comment here. I'm just going to say if the parent and the child are equal, we aren't going to do anything because I'm going to assume that the data in the binary tree is unique and if the value already exists, I'm just not going to do anything. You may not like that functionality. You may want to throw an exception or maybe you do want to add the data again. If you do want to allow for duplicates, you can choose whether you want to add that as the left or the right node or maybe it doesn't matter to you because as you start to compare, you'll move down, but it does get a lot more complex because now when you add a node, you'll have to check to see if the nodes below it also are the same value. Since binary trees are complex enough, I'm going to go ahead and just say, "If it's equal, we're not going to do anything," and just move on. Now that the insert method is defined, I can come back up to the add method and I can attempt to fill this out, so that we can actually add data to our tree structure. The first thing that I want to do when I enter my add method is I want to create a new node with the item value passed in. Once I have my new node, one of two things needs to happen. I need to first check to see if this is an empty tree and if it's an empty tree, I'm going to go ahead and add this new node as the root. This if check here is saying, "Hey, if my root node is "null, I'm going to go ahead and set "the new node to the root." This is what happens the very first time you add data to your binary tree, this condition will run. Since it's kind of a unique thing, I added a System.out here just to say, "Hey, I set the root node, but you don't "necessarily need this, but we do need to increment the size "just so we can see that when we call the size method, we "have an accurate count." If the root is not null, we then have a tree that is already have a populated root. That brings us to this else condition here. In the else condition, if we already have a root node, we're simply going to call the insert method that we just created and that will take care of traversing down the tree and finding the appropriate leaf position for this node that we created. I start off the initial insert by saying, "Here's my root node, here's the new child node, "go ahead and find the appropriate place in the tree." As it moves down into the insert method, this will recursively call over and over, doing our comparisons, determining whether it's left or right and placing it in the tree appropriately. Make sure your code is saved to this point and if you have any questions about it, you may want to just simply look over the code and try to think about what it's doing and refer back to the president example that I gave as we added presidents to the binary tree. This code is doing exactly that but it's doing it generically using the X item. That wraps up this demo. Up next, we will be building the contains method that allows us to search for items in a binary tree.

Demo: The Binary Tree Class Contains Operation

Normally, we'd add the delete functionality next, but I'm going to save that for last because of its complexity. In this demo, we'll create the contains method, which allows us to search and find data in the binary tree. In the last demo, I added the add method and the insert method, so underneath the add method, I'm going to create a method definition for our contains, which will look like this. This method definition takes in an item and then goes to search the tree to see if it exists and returns true if it does and false otherwise. You could also modify this method, so instead of returning true or false, it would return the data or the node, so that if it was null, it didn't find it. Otherwise, if it came back with a valid node data, it found it. For my class, I'm simply going to just return true or false if we found the item in the tree. Now that you've seen the add and the insert method, I'm going to recommend trying to implement the contains method on your own before moving on. You're going to want to do something similar to the insert method, where you can traverse recursively down the tree to find nodes. Here's how I implemented the contains method. I'm going to start off by creating another private method, which will help us obtain our value. The new private method is going to be called getNode. It takes our item, it's going to retrieve all the nodes recursively until it either finds it or not. The first thing that I'm going to do is I'm going to get ahold of the root node and I'm going to set that to the current node. Now, I just mentioned that you could do this recursively, which is fine if you did, but you can also do this in a non-recursive manner. I'm going to actually show you how to do this not recursively, in case you're curious how that might look. Once we have the current node, we can use that as a value to determine if we have other nodes. I can create a while loop that looks at the current node and it does some operations on it as long as it's not null. For example, if we called this method when the root is null, say when we have an empty tree, the while loop here is simply just going to exit immediately and return null. This makes sense right because if we don't have any values in the tree and we go to look for it, it's not in there because it's an empty tree, so we're going to return a null value. If we do have a root node, the first thing that we'll do is we're going to make another call to the compare. Again, this is a part of the comparable interface, where we compare one thing to another and we get the value. Simply taking the item passed in and comparing it to the current node's item and I store that in a local variable called int val. I can then check to see if it's equal, I can check to see if it's less than or greater than. If the value is equal that means that we have found the node. I can simply return the current node and break out of this private method with the current node value. If the value is less than zero that means that we're interested in moving down the left side of the tree. I'm going to take the current node and get its left node and reset that to the current node. Instead of making a recursive call, I'm simply updating the current node based off of the comparison value. Same thing goes here, if we get to this else block, it means we're looking at the right side of our node tree and we can get the current node's right node and set that as the current node value. When we execute the while loop again, we'll then have moved down one section of the tree. This is how you can traverse a binary tree non-recursively, but you could have also done this recursively similar to what we did with the insert method, where we call the insert over and over recursively. Both ways work just fine. I wanted to show you this way as well to see that it is possible to do things on a tree non-recursively. Now that the node helper method is in place, I can back up to the contains method that I've created and I can simply call my node method. I can say, "Get me the current node for this item "and just store it in a local variable called current node." Once I have my current node, this becomes a very simple comparison by just checking to see if it's null or not because if it's null, we didn't find it. If it's not null that means we did, so we can return true and false accordingly. That's it that takes care of our contains method. Coming up next is the delete, which is the most complex piece of a binary tree, so get ready for that.

Demo: The Binary Tree Class Delete Operation

The last coding demo will be providing the delete functionality. This will involve figuring out the delete algorithm for a binary tree. Back in the binary tree we've been working on, I just need to find someplace in here to add my delete definition, so I'm going to place that underneath that contains 1. It's going to look like that. We take in an item, this is the item were interested in deleting and we're returning a Boolean value that specifies whether we deleted that item or not. If you think about deleting a node from a binary tree, couple things need to happen. We first need to find the node, then we need to sever that node and reattach any other nodes around it. That will essentially delete the node. Before I implement the delete functionality, I'm going to create a new private method, which will help with unlinking or severing the nodes from each other. I'll just add this method underneath the delete and I'm going to call it unlink. Here's what this private unlink method does. It takes the current node and the new node and it tries to remove those. Couple things need to happen when we unlink. First, we need to see if the current node is the root node. If it is the root node, we're simply going to go ahead and set the new node as the current node's left and the current node's right. All we're doing here is if it's the root node, we're simply taking the new node and that becomes the new root node. It's like we just kind of rip it off the top of the tree and place the new node right at the top and reset the left and right values. Otherwise in these bottom two else blocks, we do something that looks a little confusing. What we're trying to do here is we're saying, "Look, if the current node's parent "right node equals the current node, "then we're going to go ahead, take that parent and reset "the new node on it." What we're saying is if the current node is on the right side of the parent, we want to go ahead and reset that new parent node to the new node. Just like above where we swapped out the root node, this is saying, "Anywhere on the tree if I'm the right node of a "parent, I'm going to go ahead and swap out the current node "with the new node on the right side." The same thing happens here on the left side. We're simply swapping out the left node position and resetting the new node in that position. Now, we have the unlink method. We have all the private helper methods we need to implement the delete. The delete is quite complex because there's a few cases that we have to deal with. The first thing that I'm going to do in the delete method is declare a local Boolean variable called deleted and initialize it to false. I can then return the deleted value at the end of the method. As we traverse through the tree and figure things out, we'll determine whether we need to flip this deleted flag to true or leave it false. The next code check we need to do is determine if we have a root node or not. If the root of our tree is null, we don't have anything in the tree because it's empty, so we're simply going to return false, meaning that we could not delete anything from an empty tree. If we make it past that check though, we need to get the current node of the item. We already had that when we built the contains method, we had the getNode on there, so I'm simply calling getNode with the current item and that returns whether my current node is null or the valid current node of where this item is. If it found that current node that could be anywhere on the tree because this traverses the whole tree, looking for this particular node. Next, I just do a check to see if the current node is not null because if it is null, we didn't find it, we can just return false, but if we did find a node that means we need to determine how to delete it. Inside of this if block, we're going to have some specific cases we need to deal with. Let's take a look at the first case. The first case here is if the node that we're trying to delete doesn't have any children, this is the easiest one to delete. We can simply just chop it because we don't have to worry about reattaching any of its children. I do that by just looking to see if the current node's left and right nodes are null. If so, I'm going to go ahead and unlink the current node and set it to a null value. Since I did that I can go ahead and set the deleted to true, meaning that we did delete the item from the tree. The next condition that we should check is to see if the node that we're deleting has only a right child. The left node is null, but the right node is not null, we can go ahead and unlink this by reattaching the right node to the current node. To restate that the unlinked method, if you remember, simply swaps out the current node with the node that's the new node. What we're saying is that the current node is the one we're trying to delete, but since we have a right child, we need to reattach it to the parent. We're pulling out the current node and we're taking its right child and replacing it where it currently was. That's what this unlink method does and again we can flag that we deleted that as true. The next condition is similar, but instead of looking for a right only child, we can look for a left only child. This condition is saying, "Hey look, we found a left node, but the current node's right node is null." That means that the node that we're trying to delete only has a left node, so I can unlink or swap it out similarly. I'm just saying, "In my current node, I'm going to get rid of "it and replace it with its left node, which keeps "everything in line." These first three conditions are fairly simple because we're dealing with situation, where we only have no children or one child on each side. Here's where it gets more difficult. What do we do if the node has both children? There's a couple of ways that you can deal with deleting a node with both children. I'm going to probably talk about the easiest way to do this since this delete method is complex enough. If you think about removing a node, the node can traverse all the way down until we can find the left or right most leaf node on this node with two children. You can pick one side or the other. You can pick the left path or the right path. Here's how this will work. What I'm going to do is I'm going to swap out the node that I'm trying to delete with the right-most leaf node on the left side. If you think about this visually, this node has a left tree side and as I traverse down the left hand side, I want to get the initial left node and then, I want to find the right most leaf node on that left side because that node is the next least value below the current node that we're trying to delete. Once I find that right most leaf node, I can then swap it for the current node that I'm trying to delete. Here's how that works programmatically, I first get the left node on the node that I'm trying to delete and then I look for the rightmost child off of that left node. When I find it, I have the two nodes that I need to swap. The child now is the next value that would be appropriate to swap with the current value that I'm replacing. The first thing that we do once we get that child is we just set it to null, meaning that we're pulling it off of the bottom leaf node and now we're going to go back up to the current node, we're going to set its left and right node to that child value. I can then unlink the current node with the child node. That just replaces where the current node is with the new child. Again, I can finally set the deleted equal to true. This is kind of hard to think and visualize, so I recommend going back and looking at the president binary tree and think about removing Jefferson. If you look at the Jefferson's side of the tree, you go to the left hand side, what is the most right hand node on there and does it make sense to swap those two nodes out? You'll look to see that the logic works every time because of the way the binary tree is set out. You could have taken the other side of the tree and done the same thing. You could have said, "Hey, I want to find the right leaf node ..." Excuse me, you want to find the right node and get the most left leaf node off of that to swap it out with. It's just kind of the opposite depending on which side of the tree that you run through. At this point, the only thing we have left to do is to just not forget about that size attribute and if we ended up deleting a node, we just simply decrement the size value by one and that keeps our size of our tree in line and appropriate. That's it, the binary tree is quite a bit more complex than the other data structures just because of the hierarchical nature of it and the weird strange conditions. The delete by far is the most complex, so if you don't understand that write down on paper a few different binary trees and practice removing it on paper to see how it works, what happens if it's the root node when you try to delete it, what happens if it's a node with one left node or what happens if it's a node with both nodes or just one right node? You have to take care of those particular conditions as you remove and unlink that node from the binary tree. The good news is is this completes our coding of the basic binary tree. We can go ahead and test it out by using a demo application coming up next.

Contact Manager Tree Test

Now that the hard part of creating the binary tree structure is in place, we can test it with a simple contact manager app. This app will work kind of like the binary tree example of the presidents I showed in a previous example, except we'll be using people's last names rather than presidents' last names. I've created a simple contacts manager app that lives out on my GitHub repo that we've been using for this course. You can find it here underneath the ps-data-structure-helpers repository. If you browse down into the source Java folder, you'll see that the ContactManagerApp.java file lives there. If you've cloned this repository, you already have it and you can move that into your application. I'm going to go ahead and just grab the raw data and cut and paste it into my project, so simply select all the text and I'll copy it. I'll jump over to Spring Tool Suite. In Spring Tool Suite, we've separated out our applications versus our data structures by package. I'll go ahead and open the apps package and I'm going to create a new class called ContactsManagerApp. I can now take the information I copied from GitHub and replace the class definition. Again, I'm going to leave the packaged apps in place because the one on GitHub doesn't have any package info. When I paste the new class definition into my project, I have some compilation errors and that's because I'm missing the reference to the basic binary tree that we've created. I'm going to go ahead and do a Command Shift O to import that on the Mac or you could do a Shift Control O on other operating systems. Basically, we need to make sure that we have the import statement ds.BasicBinaryTree at the top of our class and that should remove all of the compilation errors. I'll go ahead and save this and we can take a look at what this class is doing. I'm going to double click on the class name just to open up the editor, so it's full screen. We'll take a look at this class. The first thing that we do is we instantiate a basic binary tree. I'm calling the tree contacts because this tree is going to hold a bunch of contact information. I then have created a public static void main method that will start and run this application that we can use to test the basic binary tree. This app pretty much does three things. We load contacts, which will add data to our binary tree. We then search our contacts, which will do the contains or find method that we added and then finally, we clean up the contacts, which will delete or remove nodes or data from the binary tree. Really quickly here, you can see that in our load contacts, we have several contacts here with some fake numbers and some first and last names. You may be wondering where the contact class is coming from that class is defined at the bottom of this class. One thing that you will notice is that it implements the comparable interface, so this is required by our binary search tree that any of the generic types we add allow us to run comparable interface methods on it. Other than the basic attributes like the first name, last name and phone, the interesting part here is the compareTo method. In this case, I've chosen just to use the last name as the compareTo. If we wanted to make this an actual useful contact manager, I probably want to utilize first name and last name in some kind of particular order, but just to keep it simple, I'm simply using the last name, even though we're holding other data in this class. For convenience, I did also add a to string here that just spits out the contact information. After we load a bunch of data into our contacts, we'll go ahead and search for it. In this case, I'm just trying to find a contact that exists in the book and one that does not so we can see if it works. Then, finally we'll go ahead and clean up by deleting some of the contacts from our binary search tree or our contact manager. This app isn't super complex, but it will allow us to test out the functionality of our binary tree. I'll double click the class name again and we can right click here to run this as a Java application. After running, we have some information in the console. It says that we have created a root contact called Abe Lincoln, which if you look at our load contacts, he is the first one into the binary tree. He is set as the root, which is what we want. Next, we display that we've loaded 30 contacts, which is done in this block here and then in our search contacts, we try to look for Fay Maro and that person does live in the binary tree, which is right here. We added them right here. We then look for a Bob Hope, which we did not add up in this load contacts block. We got a false return down here. Finally, after deleting five of the nodes, we look at our size to see that 25 contacts are remaining. That's it, even though it took a lot to build our binary tree, we were able to test it pretty quickly just to see if it was working. If you ended up with any errors, make sure you go back and work through the code demos again, just so that you can get your binary tree working. Coming up next, we're going to talk about how we can improve our binary tree and make sure that what we've really done here really does work.

Binary Tree Extra Credit

So far, we've created and tested a pretty simple binary tree structure. The extra credit for this data structure, we're going to talk about how to enhance the tree to be a little more robust. Let's get started by taking a look at the president binary search tree that we initially talked about. As we add a presidents to this tree in a random order, the tree accepted the president, but we ultimately ended up with an unbalanced tree. If we were to have added the presidents in alphabetical order, we would have ended up with a tree with only right nodes. That's because of how the tree compares data as it's added. This pretty much makes it the same as a linked list if you do this. How can we prevent this type of unbalancing from happening? What if we could add the presidents in random order, but try and balance the tree as best as we could, so it looks more like this tree than the one on the left. Fortunately, there are ways to accomplish this. For the homework for the binary tree, here's what I'd like you to try and work through on your own. Create a rebalance method that you can call whenever you want for your basic binary tree that will take all of the current nodes and rebalance them. This can be tricky and confusing, but you can use the concept of tree rotation to help you set up an algorithm to accomplish this task. Check out this article on Wikipedia for an excellent example of tree rotation, which includes animations to help you see how tree rebalancing can be accomplished. Basically, you need to determine where in your tree needs the rebalancing and then perform a rebalance. Because of the hierarchical nature of a tree, you can utilize recursion to help you with this task. Finally, debugging binary trees programmatically can be hard because they're hard to visualize. To help with this, you can create a print or toString method that will display your binary tree nodes in a tree format. To give you an idea of how to do this, try to determine the level, depth of the binary tree and that can help you indent or add spacing as you print out the values of each node of the tree. As you saw, tree data structures are quite challenging and definitely take the time to work with them and really try to understand them, which will help you become a better programmer. Coming up next, we'll take a look at what Java offers out-of-the-box for tree data structures.

Core Java Trees

Out of all the data structures that Java does provide out-of-the-box, surprisingly Java doesn't have a lot of tree options. As far as a pure binary tree implementation is concerned, there isn't really anything in the Core JDK that provides that functionality. If you have a need for a binary tree or a binary search tree, you will need to either implement your own or find a third-party library. Java does provide one collection that is tree based called the TreeMap. The TreeMap is a type of tree called a red-black tree. A red-black tree is a type of binary tree that is self-balancing. This typically is what you want your tree structure to do anyway and so you can use the TreeMap to get pretty close to a binary search tree as far as functionality is concerned. If you struggled with implementing a balanced binary tree in the previous extra-credit, you may want to take a look at how the red-black tree works. Take a shot at implementing the balanced tree again using those algorithms. You can find more info on red-black trees at Wikipedia at this link.

Summary

That wraps up the binary tree data structure for this module and data structures overall for this course. Let's take a look at what we've covered in this module. I first started by talking about the tree structure and how they are hierarchical in nature. There are many naturally occurring concepts of trees in the computer world, like in HTML'S page structure or the DOM. We then talked about how binary trees consist of nodes that contain at most two child nodes, a left and a right. Data is added to a binary tree by comparing the item value to the root node and then descending the tree, comparing node item values until a proper leaf node position is found. This puts data in the tree in an order, which makes accessing and placing data in this structure very efficient for large data sets. Next, we started coding the binary search tree by creating the basic binary tree class and the tree node class. This was followed by creating the add method, which allowed us to place data into the tree and then we added the contains method, which determines if an item value lives in the tree or not. The last section of code we added to the tree was the delete functionality. The delete was the most complex part of the structure and requires that you implement several different cases to handle deletes properly. After testing the tree structure with a sample app, we talked about how to enhance the binary tree we built with some extra credit like adding a rebalance operation and the ability to print out the tree visually. Finally, we talked about what trees Java support out-of-the-box. Java doesn't have a lot of options, but does provide a tree map class, which is an implementation of a red-black tree, which is a type of a binary tree. That's it for our data structure journey. If you've taken the time to implement and think about the data structures we've covered in this course, you're now going to be a stronger developer with a better foundation than when you began the course. Good luck on your programming journey and I hope to see you at a future course.