Data Structures & Algorithms in Python: Fundamental Data Structures

Explore Python data structures and delve into the details of some of the basic structures, such as linked lists, stacks, and queues. Key concepts covered in this 12-video course include the metrics on which algorithms and operations on data are evaluated; learning how the performance of operations and algorithms is expressed in terms of size of input; and learning about linked lists and their contents and structure. Next, study different ways in which nodes can be added to a linked list and how search operations work on this data structure; learn methods to remove nodes from a linked list and the process of reversing the order of nodes in this data structure; and learn techniques used to keep track of numbers of elements in linked lists. You will examine workings of a stack data structure, including the addition and removal of elements; learn some of the operations on stacks, such as ISEMPTY and ISFULL, and the complexities of different stack operations; and learn the queue data structure and how to compare it to stacks.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview.* [Video description ends]

Hi, and welcome to this course on fundamental data structures. My name is Kishan Iyer, and I will be your instructor for this course.

[Video description begins] *Your host for this session is Kishan Iyer. He is a software engineer and a big data expert*. [Video description ends]

A little about myself first. I have a master's degree in computer science from Columbia University, and have previously worked in companies such as Deutsche Bank and WebMD in New York. I presently work for Loonycorn, a studio for high-quality video content. Data structures and algorithms are the bread and butter of coding. A programmer will use these every single day.

And a good programmer will have an in-depth understanding of the various data structures and algorithms which can be applied in different situations. Data structures and algorithms go hand in hand. And which ones you pick determine not only the kinds of operations which you can perform but also the speed with which they are executed and the compute resources which they consume.

This course is purely theoretical. And we begin by taking a look at the concept of complexity, which is a measure of how the number of operations of an algorithm varies with the change in its input and how this complexity can be represented with a Big O notation.

We then move on to the most fundamental data structures in computer science – linked lists, stacks, and queues. We take a look at what their properties are, how data is stored within them, and the types of operations which can be performed with them. We also cover the complexities of these different operations as well as the use cases for each of these structures.

By the end of this course, you will have a clear understanding of complexity, the Big O notation, and will have in-depth theoretical knowledge of linked lists, stacks, and queues. This will prepare you to move on to a course where you can implement what you have just learned here in code.

An Overview of Data Structures

[Video description begins] *Topic title: An Overview of Data Structures. Your host for this session is Kishan Iyer.* [Video description ends]

Hi, and welcome to this learning path where we will be discussing data structures and algorithms. So let us begin by taking a look at what exactly is meant by data structures and algorithms. A rather simple definition for data structures is that these are a way to organize information in such a manner that something useful can be done with it. That is, data structures are a way for you to structure your data in a useful manner.

Now once you have your data in some form or structure, then you can perform some computing operations on that data. [Video description begins] *On the slide, this point is written as follows: "Computing is all about performing operations on information."* [Video description ends] The specific methods which you apply in order to perform these computing operations will be referred to as your algorithm. So we now have data, which is organized into some form or structure, and an algorithm with which to perform computing operations.

Now why exactly do we study data structures and algorithms? Well, this is because the specific data structure which we use and also the algorithm which is implemented go a long way towards determining how efficient our operations on that data are. The right choice of data structures and algorithms can significantly reduce the time taken to perform operations on your data.

So the aim of data structures is to make common operations on your data very quick. For a real life and less abstract analogy, consider a shipping container into which you can pack several products which needs to be shipped across the world. You can consider the shipping container to be analogous to a data structure. And one of the goals of a shipping container is to make common operations, such as loading and unloading of goods, very quick.

While this is often the most important consideration when using data structures, another crucial aspect is that the correct data structures can make difficult operations possible in the first place. Sticking with the shipping container analogy, consider that difficult operations such as loading containers onto vessels occur less frequently than loading and unloading tasks. However, this is very much a difficult operation which is made possible by the shape, size, and other properties of the container.

And moving on to one more property of data structures which is to be as space efficient as possible. It should do this without losing any of the information which it is meant to store. The space efficiency will enable more data to be stored on a system at any given point in time and potentially more operations to be performed. [Video description begins] *On the slide, this point is written as follows: "Occupy less space and still represent the complexity of information and its interrelationships in an intuitive way."* [Video description ends]

The same thing also applies to shipping containers where space efficiency can lead to more goods being loaded into a single container, more containers being loaded into a vessel, and all of these will combine to lower transportation costs.

So how does the choice of data structures affect the algorithms which are used in order to operate on the data? Well, the two are very strongly linked. And this is because the choice of data structure can determine the efficiency of the algorithms which are used on that data. In fact, there are several standard algorithms which are built upon very specific types of data structures.

We will explore exactly what those are a little later in this course. So in conclusion, you can say that data structures and algorithms in fact go hand in hand. And it is difficult to think of each of them just in isolation. So what exactly are some of the applications of data structures? Well, depending on the specific application, there are several data structures which might be the most efficient for the task.

For example, if your most common operation is to check whether a particular entity belongs to a group, then you can implement the set data structure. These are ideally suited for both membership as well as containment operations. In the case of compilers in programming languages, they will often need to perform very quick lookups in order to bind methods during runtime.

They will also need to bind variables to their corresponding values, and for these types of operations, hash tables are the most efficient data structure. Another very commonly used data structure is the stack. And this is very useful in order to access the most recently performed operations. For example, the back functionality in many browsers or the undo functionality in text editors make use of the stack data structures for its implementation.

The most common operations in search engines are to quickly look up web pages which match a particular search query. And for this, index data structures are very useful. [Video description begins] *Examples of index data structures given on this slide are: a suffix tree and an inverted INDEX*. [Video description ends] And in applications such as social networking, where you wish to represent relationships between different groups of people, you can make use of the graph data structure in order to represent those relationships.

All in all, depending on the specific operations which you wish to perform, there will be a specific data structure which is likely to be the most efficient. As a result, an understanding of the different types of data structures as well as their specific strengths and weaknesses are crucial in order for you to build the most efficient applications.

At this point, you may ask what distinguishes data structures from abstract data types. Now the main point of distinction between the two is that data structures are concrete representations of data whereas abstract data types are a little more abstract. That is, these are mathematical models of data types with some high-level definitions of what the data type should contain and how it should be used, along with the types of operations which can be performed on the data. [Video description begins] *An example of how the data type should be used, as shown on this slide, is "from the point of view of the user."* [Video description ends]

If you're familiar with abstract types in any programming language, you will know that it is meant to provide some form of guideline for what should be implemented. And the actual implementation can be performed within the code of data structures. One way to think of it is that abstract data types define a high-level behavior whereas data structures implement that behavior.

When defining abstract data types, you will define a set of values and a set of operations at a high level, whereas the actual implementation is left to any instance which extends that abstract data type. [Video description begins] *According to the slide, an abstract data type can be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations*." [Video description ends]

In the case of a language such as Python, you can consider that abstract data types are represented by abstract classes, which includes a standard definition of fields and methods without an implementation. Whereas data structures are nonabstract classes which include an implementation. [Video description begins] *On the slide, this point is written as follows: "It does not specify how the type will actually exhibit that behavior."* [Video description ends]

Measuring the Performance of Operations

[Video description begins] *Topic title: Measuring the Performance of Operations. Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we discussed the fact that the goal of data structures as well as the algorithms which are used to operate on them was to make a lot of computation operations as time efficient as possible. An additional goal was to make the best use of space. And we will now take a look at how we can measure this efficiency for the different algorithms.

So there are three different metrics on which we can measure the performance of an algorithm. One of these, of course, is time. And this is often the default performance measurement. This time metric, however, is not measured in terms of seconds but in terms of the number of operations which need to be performed. This is something we will take a look at in just a little bit.

This time measurement, however, does affect the actual time taken in order to execute an algorithm. Because the more operations you'll perform, the more time it'll take to run. The next metric which is used to gauge the performance of a particular algorithm is the amount of space which it takes up.

The goal of any algorithm will be to make the best possible use of the available space. And this is particularly relevant for machines where the available memory is limited and also in cases where there is a lot of data to work with, which cannot all fit into memory at the same time. In such cases, an inefficient utilization of space can lead to an increase in execution time for the algorithm.

So while the space and time metrics for an algorithm can be measured separately, these two metrics are not entirely independent of each other. Another factor to consider when executing an algorithm on a network of machines is the overall network bandwidth which is consumed by the algorithm.

So just to reiterate exactly what is being measured by each of these metrics, when it comes to time, what we are counting is the amount of processing or the number of operations which a particular algorithm needs to perform in order to accomplish its task. The amount of actual time taken in order to execute each of these operations is not being calculated here. However, the number of operations will be directly proportional to the total time taken, which is why we refer to this as a measurement of time.

When it comes to measuring the space used up by an algorithm, this is usually measured in terms of the memory requirements in order to run the code but also in terms of the disk space required for all the data which is processed by the algorithm to be stored. [Video description begins] *On the slide, this point is written as follows: "This is both memory needed by code to store information at run-time as well as disk space needed by code for persistent storage.*" [Video description ends]

And the network measurement considers the amount of bandwidth which is required by the code in order to pass information between the different machines which make up the network. Now all three of these measurements will contribute to the overall time taken in order to execute the algorithm.

The actual time taken for the algorithm though will be determined by the available hardware resources, in terms of CPUs as well as memory, disk space, and the network bandwidth. What these performance indicators do help us with, though, is to measure how much of each of these resources will be used by the algorithm.

An algorithm can be considered efficient if it uses far fewer of each of these resources, CPU, disk space, memory, and network, compared to some of the other algorithms. However, this is not the only factor to consider. In fact, code can also be considered more performant by our definition if it happens to use more of some specific resource which we have in plenty and fewer of other resources which we lack.

For example, if your system and network has limited CPU power available but memory and hard disk space is plentiful, then an algorithm which is not very efficient in terms of space utilization is not a big concern for you.

And each of the metrics which we just discussed is measured in terms of something known as complexity. And what exactly do we mean when we say complexity?

Here, we don't measure the raw amount of CPU required or the amount of memory or disk required or the network bandwidth which is needed for your algorithm. Instead, complexity is a measure of how the requirements for the available resources change as the size of the problem gets larger. For example, if you double the size of the input to your algorithm, does the requirement for the resources also double or does the resource requirement remain exactly the same?

It is this relationship between the input and the resource requirements which is reflected in the complexity of an algorithm. And the reason for measuring an algorithm in terms of complexity is because these complexity measures have a direct bearing on how well an algorithm performs on any hardware.

[Video description begins] *Screen title: Complexity Affects Performance.* [Video description ends] The higher the complexity of a problem, the lower is the overall performance. For example, if you have one algorithm which requires ten times as many resources when you double the size of the input, this can be said to be a worse performer than an equivalent algorithm which takes up three times as many resources when doubling the input. The exact relationship between the input size and the overall resource requirements though will depend on the specific algorithm.

[Video description begins] *Screen title: Building Blocks.* [Video description ends] As I have stressed previously though, each of these measures of performance are not a direct calculation of the time required for the code to run. That, in fact, very much depends on the basic operations within an algorithm. And all of the measures for complexity are expressed in terms of these basic operations.

Each set of basic operations will have their own specific resource requirements. And what our complexity measures will calculate is how many of these basic operations will need to be performed when the size of the input changes. Now what exactly are these basic operations which make up the building block for any algorithm? Well, as you can imagine, they can be a vast array of such operations, including operations to read from either memory or disk and also write operations.

Operations can also include assignments and a sequence of tests. So a fundamental operation for any algorithm can involve any sequence of all of these or some other common programming tasks. Now in order for you to get a clear understanding of what exactly complexity represents, you don't need to worry about the exact number of operations which are involved.

What you are concerned with is how the number of operations changes based on the input size. So if a doubling of the input leads to a tenfold increase in the number of operations, that is all you need to know. You don't need to worry about the specific number of operations involved or the time it takes to perform each of them.

Now when you are measuring the complexity of any algorithm, it is possible that you may end up with a range of values rather than a very specific one. [Video description begins] *On the slide, this point is written as follows: "That is how performance changes based on input size."* [Video description ends] When that is the case, it is important for you to take into consideration the worst case performance even if that is not a likely scenario.

And once again, what complexity seeks to measure is the maximum number of basic operations of your algorithm which will need to be performed and how this varies with input size. And with that, we have come to the end of this video on complexity, and we can quickly summarize what we have covered so far.

We have explored the fact that executing code does require resources in terms of time, space, as well as the network bandwidth and it is the amount of these resources which are used, which determine how well a particular algorithm performs. The utilization of these resources is measured in terms of something known as complexity. [Video description begins] *On the slide, this point is written as follows: "Complexity is a measure of performance."* [Video description ends]

And when we make these complexity calculations, we are not specifically concerned with the exact operations which are involved in our code and instead focus on how many times the basic set of operations will vary depending on the size of the input. [Video description begins] *On the slide, this point is written as follows: "Complexity ignores actual operations in code – the focus is on how performance scales with input size."* [Video description ends] And finally, all of the three measures of complexity, time, space, as well as network, will vary with the specific algorithm which is in use.

The Big O Notation

[Video description begins] *Topic title: The Big O Notation. Your host for this session is Kishan Iyer.* [Video description ends]

In this video, we will take a look at how exactly that complexity will be represented. Specifically, we will take a look at the Big-O notation to represent the complexity of an algorithm or an operation. So when we talk about the Big-O notation, we can take a look at a specific example. So when expressing the complexity of an algorithm whose complexity does not change with the size of the input, we say that the complexity is O(1). So this is expressed with a capital O and then followed by that, within parentheses, the number 1.

This means that the specific algorithm is set to have a constant time complexity. That is, no matter what the size of the input, the number of operations which the algorithm will perform will be constant. You will recall that the number of operations here will represent a basic building block for the algorithm. And an algorithm which has a complexity expressed as O(1) will take the same amount of time even if the input size is doubled, tripled, or pretty much increased or decreased to any level.

An operation which is considered to take a constant time for execution is a hash lookup. And this is because any input which is provided for a hash lookup will be subject to a hash function. And this hash function will execute the exact same tasks no matter what the input. Since the same set of tasks will be executed each time, the number of operations will be constant and the algorithm or the operation is set to execute in constant time.

[Video description begins] *Screen title: What Is the Complexity of Common Operations?* [Video description ends] Moving along now to some of the other common values for complexity, we express the complexity of an algorithm as O(N) if the time taken by the algorithm increases linearly with the size of the input. [Video description begins] *On the slide, this point is written as follows: "The complexity of an algorithm is O(N) if the time taken by the algorithm increases linearly when N increases."* [Video description ends]

As an example, consider that your task is to search for a particular book, which is on a shelf, and that there is no ordering of those books. You will end up going through each of those books one by one until you find the one which you're looking for. And the amount of time taken for you to conduct the search is directly proportional to the number of books which you have in front of you.

If the size of the pile of books were to double, then the time taken for you to conduct your search will also double, both in the average case as well as the worst case. Since the time taken for your search varies linearly with the size of the pile of books, this operation is set to have a O(N) complexity. There will also be occasions when the number of operations will essentially be proportional to the square of the size of the input.

That is, the number of operations executed by a algorithm will increase quadratically with the size of the input, which is measured as N, in which case, the complexity is expressed as O(N^2). Now both of these are relatively simple representations of the relationship between the input and the number of operations, where they can be expressed solely in terms of N or N^2.

However, the relationship is often a little more complicated and the total number of operations can be expressed in terms of both N as well as N^2. In this case though, we assume that the value of N is very large, in which case, all of the lower-order terms can be eliminated from our calculation. [Video description begins] *On the slide, this* *point is written as follows: "Lower order terms and constants do not matter while expressing complexity, the assumption is that N is very large.*" [Video description ends]

For example, instead of expressing the complexity as O(N^2 + 1000), we will simply eliminate the number 1000, which is a constant value, and express this as O(N^2). Because 1000 will be insignificant if the value of N is very large. [Video description begins] *On the slide, this point is written as follows: "O(N^2 + 1000) is equivalent to O(N^2).*" [Video description ends]

Similarly, even O(N^2 + N) is simply expressed as O(N^2). Of course, the complexity may in fact be represented by a higher power of N, in which case, it is only the highest power which will be used in order to represent the complexity of your algorithm.

[Video description begins] *Screen title: "Which Algorithms Are Faster?*" [Video description ends] So now that we have seen how exactly complexity can be represented, how do we make use of this complexity measurement? How can we take a look at the complexity value and decide whether the algorithm is quick or slow? So if you were to construct a scale from the fastest to the slowest, then the quickest algorithms are usually the ones which execute in constant time.

[Video description begins] *A diagram displays. It shows a rectangular box comprising four speedometer icons placed adjacent to each other. Below the box and at its left-hand corner, the word Fastest displays. And at its right-hand corner, the word Slowest displays. Between the two words, the following expression displays: O(1) < O(N) < O(N^2) < O(N^3).* [Video description ends]

This may not be the case if the size of the input, expressed as N, is small, but we always assume that N is extremely large. As a result, an algorithm of the complexity of O(1) is set to be quicker than one which has a complexity of O(N), which is quicker than one with the complexity of O(N^2), and so on. There are algorithms whose complexity is effectively the logarithm of the input size to the base of two. And their complexity lies somewhere between O(1) and O(N).

An Introduction to Linked Lists

[Video description begins] *Topic title: An Introduction to Linked Lists. Your host for this session is Kishan Iyer.* [Video description ends]

It is now time for us to examine our first data structure which is a linked list. Here we will take a look at exactly what comprises a linked list, how exactly it is structured, and then also what the complexity of some of the common operations on a linked list are. So let us go ahead and take a look at what exactly a linked list will look like. As with any data structure, a linked list is meant to structure your information in a manner which makes it useful for certain operations.

In the case of a linked list, it will arrange all of the elements which make up your data into a sequence or a list. It will then go ahead and link each of those elements together in order to form a linked list. So what exactly might this look like? Well, consider that one of your elements of the data is represented right here. This is often referred to as a node of a linked list, and it has two different fields.

The first field contains the data for the node, and the second field represents a pointer to the next node in the linked list. In the case of this particular node, we call it the head of the linked list since it happens to be the first element. Now the next pointer from this node will point to the second element in the linked list or the second node. This in turn will have a field for the data and another field for the next pointer.

And the next pointer in turn will point to the third node in the linked list. And the next pointer now will point to element number four. And this will go on and on until all of the elements which make up your data have been represented in this linked list. Finally, once the last element has been inserted into the linked list, the value of the next pointer will be null.

[Video description begins] *A diagram displays. It shows four boxes arranged in a downward stepwise order. Each of the boxes, which are of the same dimensions, are horizontally split into two same-sized fields. The first field of each box is labeled data and the second field is labeled next. An arrow points from the second field of each of the first three boxes to the first field of the subsequent box. The diagram indicates that the first box is the Head and that the value of the second field of the fourth box is Null.* [Video description ends]

So based on what we have seen, what can we say about linked lists? Well, very clearly, linked lists are essentially list data structures. That is, they are meant to store an ordering of elements which make up your data. [Video description begins] *On the slide, this point is written as follows: "They can store multiple elements as a list."* [Video description ends] The linked list places no limit on the amount of data which you can store within the list.

And even the type of data which is stored within each node is entirely up to you. It could very simply be just an integer, or you could also store more complex information such as all of the details representing an individual, for example. Whatever data you do store, however, each element or node in the linked list will reference the next node of that list. And in this manner, all of the elements which represent your data will be chained together.

So all in all, a linked list is a data structure which you can consider if all the information which you have can be divided into elements or nodes, which can be arranged in some form of sequence. Now the implementation of a linked list is supported in pretty much all of the high-level programming languages. And some languages also support data structures which are similar to linked lists. For example, ArrayLists in Java fall into this category.

So now that we have some idea of linked lists, let us take a look at some of the terms which are used to represent specific elements which make up the list. The first node of a link list is referred to as the head, and this is often the entry point into the linked list. So the way to get to any of the elements in the list is through the head. And the last element of the linked list, whose pointer to the next element is a null value, is referred to as the tail of the linked list.

And we can now quickly summarize what exactly a node of a linked list represents. We have already covered the fact that it includes one field for the data and another field representing the pointer to the next node. In languages such as C++, for example, the next node will represent a memory location where that node will be stored. Within the data field of a linked list, we can have any kind of data. But in this particular example, we assume that it is an integer value.

When we implement linked lists in the labs of this learning path, you will be using them in order to store integer values. The second field, of course, is a reference to the next node in the linked list. And in languages which support the notion of pointers, this field could in fact be a pointer. And of course, if the node happens to be the last one in the linked list, the value for this will be null. It is this combination of the data and the next field which combine to form a single node of the linked list.

Adding and Searching for Data in a Linked List

[Video description begins] *Topic title: Adding and Searching for Data in a Linked List. Your host for this session is Kishan Iyer.* [Video description ends]

It is time for us to take a look at some of the common operations on this data structure. The first operation we will examine is adding a new node to a linked list. There are different ways in which this can be done. And we'll start off with the most common way to add a new node, which is to append one to the end of a linked list.

Any node which we add to the linked list will contain some value for the data field. However, when adding this node to the end of the linked list, the value of the next field will be set to null. [Video description begins] *On the slide, this point is written as follows: "Remember the newly added node should have the next set to null."* [Video description ends]

In order to add a new node to the end of a linked list though, we will need to traverse the entire list. This is because the entry point to any linked list is via the head. And then in order to get to the end of the linked list, we will have to iterate through each of the existing nodes. And the amount of time taken in order to perform this operation is directly proportional to the size of the linked list. That is, it is a O(N) operation. In this case, N represents the number of elements which make up the linked list.

Now the specific case which we have talked about so far is adding an element to the end of a linked list. However, this is not the only way in which we can append a node to that list. For example, we can also insert a new node to the beginning of the list. We have already talked about the insertion at the end of the list. And if implemented in such a manner, we can also allow the insertion of elements in the middle of a list, potentially after a specific node.

So what are the different steps involved when inserting a new node to an existing linked list? For this, consider a linked list where the head element is this node which contains the data B. And then its next pointer points to the second and last node of the linked list, whose data contains the string C. Now since this is the last node in the list, the value of its next field will be null. Do keep in mind that head is essentially a variable, which is a pointer to the first node of the linked list.

So what if we were to insert a new element A into this linked list, specifically through the head of this linked list? To do that, we will need to readjust the head pointer to point to this new node A. And the next field of this new node will point to the node B of the linked list. In this manner, we now have the elements A, B, and C represented in our linked list.

One thing to note here is that inserting an element at the beginning or the head of the list is always an O(1) operation. That is, it takes constant time because no matter what the size of the linked list, this insert operation will be exactly the same. On the other hand, let us take a look at what needs to be done if we were to insert a new element containing the data E to the end of this linked list.

We need to start off from the head and then use the next pointer of E to the nodes A, B, and C and then iterate through to the end of the linked list. And then we adjust the next pointer of the node C to point to this new node E. And finally, we ensure that the next pointer of this new node is null. Appending an element to the end of the list does involve iterating through each of the existing nodes. And because of that, the complexity of such an insert operation is O(N), where N represents the number of nodes in the linked list.

Now that we have examined the insertion of nodes to the beginning of a linked list and also to the end, we consider the case where we wish to insert this element D in the middle of our list, specifically after the node C and then before the node E. To do that, we once again start off at the head of the list and then iterate through each of the nodes until we find the one containing the value C. And then after that, the next pointer of the node C will be made to point to this node D. And then the next pointer of D in turn will be pointing to the node E.

[Video description begins] *A diagram displays. It shows four boxes of the same size placed next to each other. Each of the boxes is vertically divided into two fields of the same size. The first field of each box is labeled A, B, C, and E, respectively. For each of the first two boxes, an arrow points from its second field to the first field of the succeeding box. A dotted arrow points from the second field of the third box to the first field of the fourth box. A fifth same-sized box, which is also vertically divided into two fields of the same size, is placed below the third and the fourth boxes. The corresponding first field is labeled D. An arrow points from the second field of the third box to the first field of the fifth box. And another arrow points from the second field of the fifth box to the first field of the fourth box. The diagram indicates that the first box is the Head and that the value of the second field of the fourth box is Null.* [Video description ends]

So in this case, we need to override the existing value of the next pointer for C in order to point to the node D. And with that done, we now have this linked list containing five elements in sequence – A, B, C, D, and E.

[Video description begins] *A diagram displays. It shows five boxes of the same size placed next to each other. Each of the boxes is vertically divided into two fields of the same size. The first field of each box is labeled A, B,* *C, D, and E, respectively. For each of the first four boxes, an arrow points from its second field to the first field of the succeeding box. The diagram indicates that the first box is the Head and that the value of the second field of the fifth box is Null.* [Video description ends]

And what about the complexity of this operation? Well, in the average case, there will be N^2 iterations. And in the worst case, we end up inserting this element to the end of the list, in which case, there will be N iterations. In either case though, the number of iterations will be proportional to the length of the list. So the complexity of this insert operation is also O(N).

And with that, we have covered the first of the basic operations which can be performed on a linked list, which is to insert elements into it. We now move along to one more standard operation, which is to search for a particular element; that is, to search for a node containing specific data within the linked list.

[Video description begins] *Screen title: Lookup.* [Video description ends] So for that, we can bring back our five-element linked list now. And assume that the specific search which we need to perform is to find the node containing the data D. Once again, the entry point into our linked list is the element at the head. So we start off with the first node in the linked list. The data for this node does not match the data which we are searching for. So we proceed to the next element.

This node does not contain the data we are searching for either. So we iterate over to the third element. Once again, the comparison of the data does not give us a match. So we move on to node number four. And on this occasion, there is a match. So on this occasion, our search was a success. And we can return either the position or the entire contents of the node, which match the search value.

Now let us consider a second search where we look for an element which is not present within our linked list. So we search for a node which contains the data G. Just as we did previously, we begin with the head element, which does not give us a match. We iterate over to node number two, and once again, there is no match here. And we continue with this process where we do a comparison with node number three and then with node number four.

And finally, we get to the last element in our linked list. At this point, we find that there is no more data for us to iterate over, in which case, we need to stop the search and then deem it a failure. So what exactly is the complexity of the search operation? As we have seen, this is once again proportional to the number of elements which are present in the linked list.

So a search for an element will have a complexity of O(N). The way in which a linked list is structured ensures that we will need to iterate over each of the elements for many standard operations. For example, there is no way for us to jump directly into the middle of the list. As a result, many of these operations will have a O(N) complexity.

Deleting Nodes from a Linked List

[Video description begins] *Topic title: Deleting Nodes from a Linked List. Your host for this session is Kishan Iyer.* [Video description ends]

We now examine one more linked list operation, which is to remove an element from a linked list. So we start off where we left off in the previous video, where we have a linked list containing five elements, A, B, C, D, and E in that order.

[Video description begins] *A diagram displays. It shows five boxes of the same size placed next to each other. Each of the boxes is vertically divided into two fields of the same size. The first field of each of the boxes is labeled A, B, C, D, and E, respectively. For each of the first four boxes, an arrow points from its second field to the first field of the succeeding box. The diagram indicates that the first box is the Head and that the value of the second field of the fifth box is Null.* [Video description ends]

Now in order to delete an element, we'll first need to search for it in the linked list. And so we first conduct a search operation, but assume that the element which we delete happens to be the one at the head of the linked list. There will be a couple of changes which need to be performed in order to implement this deletion. So a second element at the linked list will then become the one at the head.

So we need to make sure that the head pointer points to what was then the second element. In our linked list, this is the one containing the data B. And in fact, this is pretty much all that needs to be done. Once the head is pointing to this node in the linked list, we have effectively removed the node A and then all of the other links in the linked list will remain as they are.

All right, so we have now seen what it takes to remove the head element from the linked list. But we go back to the original linked list containing five elements now in order to see what needs to be done in order to remove the element at the tail of the list. So let's just say that we're asked to remove the node containing the data E. So we start off with a search operation where we compare the data in the first node with the data E. Of course, there is no match here.

So we iterate over the linked list and perform a comparison with the data in node number two. There is a mismatch. So we move on to node number three and then to node number four. And then in our linked list, node number five will give us a match. In order to remove this particular element, we will need to make sure that the next pointer of the previous node in the list will end up pointing to Null.

So here, we use the pointer of the node which we are deleting and use that value in the next field of the previous node in the linked list. So in this manner, node D now has a next pointer, which is Null. And this becomes the new end element of the linked list. We have now taken a look at the two cases where we remove elements from either end of the linked list.

But what if the element which we want to delete happens to be somewhere in the middle? For that, we go back to our original linked list containing five elements. And then let's just say that the element which we need to remove is the one containing the data C. We start off once more by performing a search. So here we enter the linked list from the head. There is no match.

So we move on to node number two; no match again. So we head to node number three. And this is the element which we need to remove from the linked list. The way to do this now is similar to what we did previously. When we remove the node from the end of the list, the value of the next pointer of the node being deleted will become the new value of the next pointer of the previous node. That is, the value of next in node C will become the value of next in node B.

So instead of B pointing to C and C pointing to D, what we end up with is B is pointing to D. By virtue of making this single change for the next pointer in the second node, we have effectively removed what was previously the third node in our linked list containing the data C. Even though the next field of the node C may still be pointing to node D, it is effectively out of the linked list because there is no element in this linked list which is pointing to node C.

So in all of these delete cases where we're removing elements either from the beginning or the end or from the middle of a linked list, all we need to do is to update the value of the next pointer of the previous element. In the case of the removal of the head node, this of course pertains to the head variable. Now since removing an element from the list first involves a search and then a removal, the complexity of this operation is also O(N).

It is time to move along to one more operation which can be performed on linked lists, though this particular step is not carried out nearly as often as either inserting nodes or searching for nodes or removing them. And this is reversing a linked list. The implementation of a linked list reversal is a frequently used programming interview question.

So let us now take a look at the logic of how this can be implemented. So we begin once again with our original linked list containing the five nodes, A, B, C, D, and E, in that order. And what we wish to accomplish now is to reverse this ordering, where we have a linked list which goes E, D, C, B, and A.

How exactly can this be accomplished? Well, let us think about what this new linked list would look like. The head would point to node E, and the next pointer for node E will point to D. The next for node D will point to C and so on. Whereas the next pointer for node A will be Null. The way to accomplish this iteratively is to first move the head to point to node B, and the next field for node A, instead of pointing to B, will be changed to null.

We then move the head over to node C, and now the next for node B will point to node A. We continue with this process where we move the head to node D and the value of the next field for node C will point to node B. We then move the head to node E, and then update the value of the next field for node D. And now the next field for node E will need to point to D rather than Null. And this is how we accomplish a reversal of a linked list.

[Video description begins] *A diagram displays. It shows five boxes of the same size placed next to each other. Each of the boxes is vertically divided into two fields of the same size. The first field of each of the boxes is labeled A, B, C, D, and E, respectively. For each of the last four boxes, an arrow points from its second field to the second field of the preceding box. The diagram indicates that the fifth box is the Head and that the value of the second field of the first box is Null.* [Video description ends]

One thing to keep in mind for the deletion of nodes as well as the reversal of a linked list is that we will be traversing over the linked list in each case and the complexity of this traversal is always a O(N) operation. In addition, we will often need to make use of two different pointers during our traversal, one which points to the current node and one which points to the previous one.

We need to do this because it is possible for us to move onto the next node in a linked list. However, we cannot go back to the previous node, which is why we have a pointer to that previous node. And we will be using that pointer to the previous node in order to make whatever updates which we require to the linked list when performing the reversal or the deletion.

Counting the Nodes in a Linked List

[Video description begins] *Topic title: Counting the Nodes in a Linked List. Your host for this session is Kishan Iyer.* [Video description ends]

We continue our examination of the linked list data structure, and the specific operation we will now focus on is to count the number of nodes in a linked list. So what exactly is involved when you wish to count the number of elements stored in a linked list? Well, very simply, we just need to iterate through each of the nodes in the linked list and then keep a count of how many elements we have encountered.

The complexity of such a count operation will, of course, be O(N). Now each time we wish to perform a count of the number of nodes, we can perform this iteration. However, this does seem a little wasteful. A more efficient way is to have a variable which will store this count of nodes in the linked list. Each time we either add or remove an element from the linked list, we will simply update this counter.

In this manner, when we wish to know how many elements are present in the linked list, all we need to do is to look up this counter variable. This operation will always run in constant time, which is why it is a O(1) operation in terms of complexity. Now if such a counter variable has not been created at the time the linked list was provisioned, you will need to iterate through the entire linked list at least once.

[Video description begins] *A diagram displays. It shows five boxes of the same size placed next to each other. Each of the boxes is vertically divided into two fields of the same size. The first field of each of the boxes is labeled A, B, C, D, and E, respectively. For each of the first four boxes, an arrow points from its second field to the first field of the succeeding box. The diagram indicates that the first box is the Head and that the value of the second field of the fifth box is Null.* [Video description ends]

So here we go back to our linked list containing five different nodes. And we will perform this count by iterating through each of the elements. We initialize a variable called Count and set it to a value of 0. And then when we begin iterating through the elements, we will start off at the head and then the value of Count will be set to 1. We then iterate over to the next node in the linked list, and if you do encounter a node, we increment the Count to 2.

And this process continues where we increment the value of Count to 3 and then to 4 and then to 5 in the case of this linked list. At this point, we have hit the end of this linked list, which is why the value of our Count variable will be set to 5. The most efficient way to maintain a count of the number of nodes as discussed previously is to update this value of the Count variable each time there is either an insertion of a new node or the removal of one of the existing nodes.

And with that, we have now covered all of the common operations for a linked list for this course. It is now time to summarize the complexity of all the linked list operations we have taken a look at so far. As far as appending a new element to the end of a linked list is concerned, we will need to iterate through the entire list. So for a list containing N elements, the complexity of such an operation will be O(N).

On the other hand, if you wish to add a new element to the beginning of the linked list, this is going to run in constant time. Because it doesn't matter what the size of the linked list is. All we need to do is to update the value of the head variable for the linked list and get the newly added node to point to what was the previous head.

Moving along now to finding an element in a linked list, once again we saw that we will need to iterate through the entire list in the worst case. And in the average case, we will need to go through half of the linked list. In each of those cases, however, the total number of operations to perform is directly proportional to the length of the linked list, which is why this particular search also becomes a O(N) operation.

Moving on now to the Delete operation which can be performed on the linked list, if your task is to delete the first element in the linked list – that is, the element at the head – irrespective of what the value of data is stored on it, then the complexity of such a Delete operation is O(1). Because we will be performing the same number of steps each time irrespective of the size of the list.

If the Delete involves searching for a particular node bearing some data and then deleting that, then the number of steps in the execution will become similar to performing a search and then doing a delete, which is why the complexity of such an operation is O(N). We also saw that reversing a linked list also involves traversing the entire list which is a O(N) operation.

And as far as getting a count of the number of nodes in the linked list is concerned, this can either be a O(N) operation if you need to iterate over the list each time or if you maintain a dynamic counter, which is updated each time there is an insertion or a deletion, fetching the number of nodes will be a O(1) operation.

An Introduction to Stacks

[Video description begins] *Topic title: An Introduction to Stacks. Your host for this session is Kishan Iyer.* [Video description ends]

Having already studied linked lists, we will now move on to another very commonly used data structure, which is the stack. Now the stack in computer science has a very specific meaning in that it is a data structure which holds elements in such a manner that the last element which gets added to the stack happens to be the first one which gets accessed from it.

So you can say that the stack follows the principle of Last In First Out. [Video description begins] *On the slide, this point is written as follows: "Principle of a stack – Last In First Out (LIFO)."* [Video description ends] What exactly does that mean? We can take a look with this visualization. [Video description begins] *A diagram displays. It shows four same-sized cubes placed on top of one another in a cylinder. The cubes, from bottom to top, are labeled 1, 2, 3, and 4, respectively.* [Video description ends]

Assume that you have this stack, represented by this cylinder here, and we have four different elements which are placed within it. We started off by placing element number 1 into the stack. On top of that, we placed element 2, then number 3, and then number 4. When accessing elements from the stack, whether to just have a look at it or to remove it from the stack, the one which is accessible to us at first is the element at the top.

So if you were to access the stack, then the element we would read would be number 4. So you can think of a stack data structure as we would a stack of books, where you pile one book on top of another. The most readily accessible book is the one which was added to the pile most recently, which is the book at the top of the stack.

And the one which is the most difficult to access and the one which will be accessed at the very end will be the one which was the first to be placed in the stack. So now that we have an understanding of what a stack is, we can examine some of the common stack operations. These, of course, are to add and remove elements from the stack.

So the action of adding an element to the stack is referred to as a PUSH. To be precise, pushing an element onto the stack means that we add a new element to the top of the existing stack. The complementary action to this is referred to as popping an element from the stack. So a PUSH adds a new element to the top of the stack, and a POP removes the element in that same position.

We move on now to another stack operation which is called Peek. And this is the one which we use to take a peek at the element which is on top of the stack. One significant way in which this operation differs from the PUSH and POP actions which we took a look at is that peek does not modify the stack data structure in any way. All it does is perform a read-only operation on the element at the top of the stack.

All right, so let us now take a look at some examples of data getting added and removed from a stack and also read from it. So if we consider that on the left we have a stack which has a capacity for six different elements. There are six different elements here on the right, represented by these colored circles, and these is the ones which we wish to add to our stack.

In order to add this purple element to our stack, we will perform a PUSH operation on it. And this gets pushed onto the stack. And there is a variable called Top, which references the element which is currently at the top of the stack. So now that we have pushed our first element into our stack, you will go ahead and push another one, specifically the gray element.

So the gray element gets added on top of the purple one, and the Top variable now references this new element which has been pushed onto the stack. We'll go ahead and place this yellowish-green element into the stack as well. It just gets added to the top, and the Top variable references that. We perform another PUSH operation, and the red element get added to the top.

You will observe that the purple element, which was the first to be pushed onto the stack, has always been at the bottom, which is now the new top element. [Video description begins] *A diagram displays. It shows a stack comprising six rows. The first five rows, starting from the bottom, are filled with purple, gray, yellowish green, red, and orange colors, respectively. Placed against the stack is a circle filled with blue color. The top variable now references the row filled with the orange color.* [Video description ends]

And then finally, we will fill our stack to capacity by pushing the blue element onto it as well. All right, so we now have a full stack to work with, and the Top variable points to the most recently added element onto the stack. So what will happen if you were to perform a POP operation? Well, we will follow the principle of Last In First Out. The last element to go into the stack was the blue one.

And when we perform a POP operation, this will be the first element to come out of it. Also take note that the element now at the top of the stack is the orange one. This is relevant because each time we perform a POP operation, it is the element which is at the top of the stack which gets removed. So when we perform this POP, it is the orange element which gets removed.

Now the Top points to the red element. So a third POP operation will remove that from the stack. And then our next POP operation will remove this green element from it. So we continue popping elements from the stack. So the gray element now comes out. And with just one element remaining, we perform one more POP operation. And the element which was the first to be added to the stack becomes the last one to be removed from it.

So we definitely follow a policy of Last In First Out, which is also referred to as First In Last Out in some cases. So now we have a fairly good understanding of what a stack is and what some of its fundamental operations are. In the next video, we will examine the complexity of these operations. And we'll also take a look at some examples of where stack data structures can be applied.

Additional Stack Operations

[Video description begins] *Topic title: Additional Stack Operations. Your host for this session is Kishan Iyer.* [Video description ends]

Given that we have a fundamental understanding of stacks at this point, we can quickly summarize the operations of a stack which we have just covered. We have seen that the PUSH operation adds a new element to the top of the stack. And then a POP operation will remove an element from the top of the stack. So the most recently added element becomes the first element to be removed from the stack.

As far as the PEEK operation is concerned, this will read the element which is, at that point in time, at the top of the stack. These three operations are the most commonly used one when it comes to working with a stack data structure. But there are a few additional operations which you may find associated with a stack. One of these is referred to as the ISEMPTY operation. And as the name implies, this will a perform check to see whether the stack currently has any elements.

There is also a complementary ISFULL operation. And you will want to check whether a stack is full before you try to push any elements onto it. Both the ISEMPTY and ISFULL operations represent some of the extreme conditions of a stack, where it has either zero elements or is packed to capacity.

In most cases though, your stack will be very large and will have at least some elements and not be full to capacity, in which case, you'll want to check the size of the stack; that is, the number of elements which are already placed within it. For this there is the SIZE operation. And you could also have additional operations, for example, to check the capacity of the stack.

All right, so now that we have taken a look at some of the common operations related to the stack data structure, let us consider some special conditions. So what if we try to pop an element from a stack which happens to be empty? Conversely, what if we try to push an element into a stack which is already full? Again these are two extreme conditions, but in each of these cases, the operations will result in an exception being thrown. [Video description begins] *On the slide, this point is written as follows: "It's an error, throw an exception!"* [Video description ends]

It is to prevent the throwing of such an exception that we have the ISEMPTY and ISFULL operations where you can check whether a stack is empty before you try to pop an element from it and also perform a check to see if it is full before you try to perform a PUSH operation.

[Video description begins] *Screen title: The Stack - Underlying Data Structure*. [Video description ends] All right, so let us now quickly summarize some things about implementing a stack. So the most common operations when using a stack data structure are the PUSH and POP operations. And we have seen that both of these tasks involve working with just one end of the stack data structure, specifically the top of the stack.

As a result, in order to implement a stack, we can pretty much use a linked list data structure. So even though we consider both stacks and linked lists as data structures in their own right, a linked list is a more general data structure whereas a stack is a more specialized one. As a result, we can use a linked list as pretty much a building block in order to implement a stack. [Video description begins] *On the slide, this point is written as follows: "A linked list lends itself perfectly to build a stack*." [Video description ends]

The top of the stack will be the head of the linked list, and any PUSH and POP operations will occur at the head of that linked list.

[Video description begins] *Screen title: The Stack - Performance and Complexity*. [Video description ends] With that in mind, we can now take a look at the complexity of the different operations which can be performed on a stack. The PUSH and POP operations are analogous to adding an element or removing one from the head of the linked list.

And in either of these cases, the size of the linked list is entirely irrelevant because of which these operations have a time complexity of O(1). Also in order to get the number of elements in the stack, we can use the SIZE function and this also has a O(1) time complexity. This assumes though that we implement a variable which dynamically keeps track of the number of elements in the stack.

As far as the ISEMPTY and ISFULL operations are concerned, these also have a constant time complexity. [Video description begins] *On the slide, this point is written as follows: ISEMPTY and ISFULL have O(1) complexity*." [Video description ends] Once again, this is when we implement variables which will keep track of the capacity of the stack as well as the number of elements currently within it. So these cover the time complexity of the different operations.

The space complexity for using a stack will be O(N) since we will need exactly N spots in memory in order to implement the stack data structure. So now that we have a fairly good understanding of what a stack data structure is, we can take a look at some of the applications of such a data structure.

[Video description begins] *Screen title: Where Can Stacks Be Used?* [Video description ends] One very common use case is to implement an undo operation in any application. You'll be aware that when you hit the undo button anytime, you will effectively be undoing the most recent action. And if you hit the undo button again, then the action just before that will be the one which is undone.

To put it in another way, the most recently performed task should be the first one which is undone and the least recently performed one should be the last one which is undone. And this is best implemented by using a stack data structure in order to store the most recently performed operations.

Another common use case for a stack data structure is to implement the back button on the web browser. Once again this is similar to the undo operation, where the most recently visited page should be the first one which is accessed when we hit the back button. And the stack data structure is ideal for this implementation.

Stacks are also commonly used by programming languages. And this is the best way to implement recursive function calls, where once again the last invocation of a recursive function will be the first one which is executed to completion. [Video description begins] *On the slide, this point is written as follows: Holding the memory for recursive calls in a programming language*. [Video description ends]

Programming languages will also make use of stacks in order to evaluate expressions, specifically to translate infix notation into postfix. So all in all, we can say that the Last In First Out nature of a stack data structure lends itself as the ideal option in order to implement very specific kinds of features.

An Introduction to Queues

[Video description begins] *Topic title: An Introduction to Queues. Your host for this session is Kishan Iyer.* [Video description ends]

We will now take a look at another specialized data structure which is the queue. So a queue is something all of us should be familiar with if you have ever waited in line for something. And just as a queue in real life, a queue in the context of computer science is a data structure where you can add elements to the end of the queue and then remove elements from the beginning.

So in some ways this is similar to a stack, but there is a notable difference. Whereas in a stack all of the operations or the main operations such as PUSH and POP are performed at one end of the stack, in the case of a queue, we make use of both ends of this data structure. So we add elements to the end and remove elements from the beginning.

And while with a stack data structure we follow a policy of Last In First Out, the principle which is followed for a queue is First In First Out. So just as with a queue in real life, the first person to join the queue will also be the first one to be processed and leave the queue. And the converse of that is that the last one to join the queue will also be the last one to get out of it. So if you ever hear people discussing FIFO and LILO, then you know that it is a queue data structure which they're talking about.

Now what are the common operations when it comes to working with a queue? Well, in order to add an element to the queue, you perform an Enqueue operation. This is where a new element gets added to the end of the queue. And then the converse operation is to remove an element from the queue, and this is the Dequeue operation. The Dequeue operates at the opposite end of the queue, where an element is removed from the beginning.

Some of the other common operations when working with queues include the Peek, and this is where we take a look at the first element in the queue. This is the same element which will be removed when performing a Dequeue operation. Another task which can be performed on a queue is known as an Offer. And this is where an element is added to the queue, provided that there is space available within it.

All right, so we now have some kind of high-level understanding of the queue data structure. Now we perform an Enqueue operation on this orange person, who will then get added to the first empty spot available within the queue. We still have empty spaces within the queue, so we perform one more Enqueue operation.

And now there is this red person within the queue as well. A third Enqueue is performed, and the blue person gets added to the queue. And we can continue filling up the queue as long as there is capacity. So we do one more Enqueue, and the green person gets added on. And then finally, one more Enqueue so that our queue is packed to capacity with five different people.

[Video description begins] *An image displays. It shows a group of five outlines, each of which represents a person. The outlines are filled with orange, red, purple, blue, and green colors, respectively. There is also an empty queue containing five spots. When the first Enqueue operation is performed, the orange-colored person occupies the first spot in the queue. The second, third, fourth, and fifth Enqueue operations result in the red-colored person, the blue-colored person, the green-colored person, and the purple-colored person occupying the second, third, fourth, and fifth spots, respectively.* [Video description ends]

It is now time to begin processing everyone who is part of the queue, for which we perform a Dequeue operation. So this is where the queue differs from the stack, where the Dequeue removes the element which was the first to be added among those which are present in the queue. So the orange person gets dequeued.

And then when we perform a second Dequeue operation, the second person to be added to our queue gets processed and is taken out of the queue. A third Dequeue will remove the third person who got added to the queue. And this will go on, where the next Dequeue will remove the green element. And we can perform one more Dequeue in order to process the sole remaining element within our queue data structure.

[Video description begins] *On the same queue comprising five persons, when a Dequeue operation is performed, the orange-colored person exits the first spot. And the next four persons occupy the first, second, third, and fourth spots, respectively. When the second Dequeue operation is performed, the red-colored person exits the first spot. And the next three persons occupy the first, second, and third spots, respectively. The pattern continues* *with the performance of the third and fourth Dequeue operations. And the fifth Dequeue operation results in an empty queue.* [Video description ends]

So now that we have an understanding of how the Enqueue and Dequeue operations work in a queue and how they differ from the PUSH and POP operations in a stack, we can quickly summarize the different types of operations which can be performed on the queue. We of course, saw that the Enqueue operation adds an element to the back of the queue and then the Dequeue operation will remove an element from the front.

The Peek operation, which we have not visualized, will effectively take a peek at the first element within the queue. In real life, you can compare this to taking a look at who is the next person in line who needs to be processed. Just like with a stack data structure, we can also implement an Isempty operation for a queue, where we can check whether the queue currently has any elements within it.

And conversely, we can also check whether the queue is full to capacity using an Isfull function. Moving on now to the boundary conditions, so what will happen if we try to dequeue from a queue which currently does not have any elements? Conversely, what if we try to perform an Enqueue operation on a queue which is full?

As you would expect, each of these operations will result in an exception being thrown. [Video description begins] *On the slide, this point is written as follows: "It's an error, throw an exception!"* [Video description ends] And this behavior is analogous to performing a PUSH operation on a full stack or a POP operation on an empty stack. And just as we discussed in the stack video, the way to avoid these errors is to make use of the Isempty and Isfull operations before we execute Dequeue or Enqueue.

It is now time for us to summarize some of what we have covered about the queue data structure. So the most common operations when working with queues are the Enqueue and Dequeue operations. When comparing queues to the stack data structure, you can see that these correspond to the PUSH and POP operations.

However, there is a significant difference in that Enqueue and Dequeue operate at different ends of the queue whereas both PUSH and POP in a stack will work at the same end, which is the top of the stack. And just as a stack can be implemented using a linked list, we can do the same to implement a queue as well. The way to do this is to have a pointer at the head of the linked list and another one at the tail.

In this manner, when we perform an Enqueue operation, we can add an element to the tail of the linked list and a Dequeue operation will remove an element from the head. Having a pointer to the tail of the linked list will ensure that when we perform an Enqueue operation, we don't need to iterate over all of the linked list in order to add an element. [Video description begins] *On the slide, this point is written as follows: "A linked list with a pointer to head and the tail works well."* [Video description ends]

There is also a special kind of queue known as a circular queue where in the underlying linked list, the last element in the list will not have null in the next field but will instead point to the first element in the list. [Video description begins] *On the slide, this point is written as follows: "A common data structure used is a circular queue with pointers to the head and to the tail."* [Video description ends] The specific details of circular queues though are beyond the scope for this course.

[Video description begins] *Screen title: The Queue - Performance And Complexity.* [Video description ends] So what about the complexity of the different operations on a queue data structure? Well, when enqueuing or dequeuing from a queue, the time complexity for these operations will be O(1), especially if we have two different pointers, one at the head and one at the tail of the linked list when implementing the queue.

In this manner, an Enqueue operation can directly reference the tail of the queue rather than having to iterate through each of the elements within. The ISEMPTY and ISFULL operations also have a constant time complexity. [Video description begins] *On the slide, this point is written as follows: "ISEMPTY and ISFULL are also O(1)."* [Video description ends]

And once again, this is when we make use of variables in order to keep track of the number of elements within the queue and its capacity. The space complexity for a queue data structure, though, is O(N) because we need as many spots in memory as there are elements within the queue.

So what are some of the common applications of a queue data structure? If you ever dial a customer service hotline, you know that you get placed into a queue since – as many of them like to remind us – each call gets processed in the order in which it was received. The same thing also applies to e-commerce websites and banking transaction systems where the first transaction to be initiated is the first one to be processed. And queues are also used by printers in order to process any printing jobs which are sent to them.

Exercise: Fundamental Data Structures

[Video description begins] *Topic title: Exercise: Fundamental Data Structures. Your host for this session is Kishan Iyer.* [Video description ends]

In this exercise, you will start off by listing down some of the common operations which are performed on linked lists and also summarize the time complexities for each of those operations. And following that, you will compare two of the other data structures which we took a close look at during the course, namely stacks and queues.

Both of these data structures do have many things in common, but they're also different in very important ways. And you'll need to highlight both the similarities as well as what distinguishes them. Each of these topics were covered during this course. So please pause this video and then take some time to do this exercise on your own.

[Video description begins] *A solution banner appears at the bottom.* [Video description ends]

For the first task in the exercise, you needed to recall some of the common operations on linked lists and also identify their complexities. Once you start off with an empty linked list, the first thing you would probably do is to add an element to it. As we saw in this course, there are different ways in which we can add an element to a linked list.

But the default Insert operation for a node involves appending it to the end of the linked list. Since the only entry point to a linked list happens to be the head, we will need to iterate through each and every element which already exists in the linked list until we get to the end, at which point, we can append this element. Clearly, the amount of time taken for this operation will be proportional to the length of the linked list, which is why we say that it has a time complexity of O(n).

Now the other way in which we can insert a new node into a linked list is to add one at the head. In such a case, no matter what the size of the linked list, we will be performing exactly the same operations. So we say that this particular insertion takes place in constant time and the complexity can be represented as O(1).

The other Insert case which we covered during this course involved adding a new node after one of the existing nodes in the linked list, based on its value. The complexity of that operation will also be O(n) since in the worst case, we will need to go to the very end of the list. Another common linked list operation involves searching for a particular node which contains a specific value.

In the best case, this node can be found at the head of the linked list. But in the worst case, we will iterate over the entire list until we find the element we're looking for. In that case, the complexity is O(n). The next operation which we can perform is to delete an element based on its value. In this case, we will need to effectively perform a search in our linked list where we iterate over the nodes until we find one which contains the specific value which we're looking for. And then we can perform the Delete operation.

However, since this once again involves an iteration which could span the entire linked list, its complexity is O(n). However, if you were to delete the first element in the list, we don't really care what the length of the linked list is and we will perform the same operations regardless of the size. Because of that, this operation has a constant time complexity, which is denoted as a O(1) operation.

For the next task in the exercise, you needed to compare the stack and queue data structures. We will start off with the similarities of these two structures. First of all, for many of the common operations such as inserting new elements, removing elements, or taking a peek at the next element to be processed, the time complexity is O(1).

Also given both of these are linear data structures, they can be implemented using a linked list, which is also a linear structure. A linked list is a generic data structure whereas stacks and queues are very specific implementations with a few notable differences. The biggest one of these is, of course, that stacks follow a principle of Last In First Out whereas queues follow one of First In First Out.

This does have a bearing on how these data structures are implemented using linked lists, for example. So in the case of the Insert and Delete operations, in the case of stacks, both of these occur at the same end. You add elements to the top of the stack and you also remove elements from the top of the stack. However, in the case of queue data structures, the addition and deletion of nodes take place at opposite ends. You enqueue elements to the end of the queue and then you dequeue elements or remove them from the front of the queue.