Data Structures & Algorithms in Python: Trees & Graphs

This 13-video course explores the theory of graph and tree data structures in Python. Learners will examine a specific type of tree: the binary search tree, its structures and properties. You will then observe how to execute common tasks in binary tree; examine the binary search algorithm; and review data structures of linked lists, stacks, and queues. Next, learners will examine how a binary tree structure offers several applications that cannot be done by using stacks or queues. The course demonstrates different depth first traversals, including pre-order, in-order traversals, and post-order traversals. Explore graphs, which are data structures used to model relationships, and different representations of a graph, and learn to model a vertex. Learners continue by observing how to represent an adjacency list as a graph, and examining the adjacency matrix, the adjacency list, and the adjacency set. Then you will explore graph traversal algorithms, including the topological sort. Finally, learn how to traverse through each of the vertices in a graph.

Table of Contents

Course Overview

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

Hi, and welcome to this course on trees and graphs. My name is Kishan Iyer, and I will be your instructor for this course. A little about myself first. I have a Master's degree in computer science from Columbia university, and have previously worked in company 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 focuses mainly on the theory of two specific types of data structures, trees, and graphs. While trees have a rather broad definition, we will focus on a very specific type of tree which is the binary search tree. We will examine the structure and properties of such a tree. And explore the different ways they can be traversed using both breadth first, as well as depth first techniques. We then move on to graphs, which are data structures used to model relationships between entities. We cover ways in which graph can be represented.

Such as the adjacency matrix, the adjacency list, and the adjacency set, and we'll explore graph traversal algorithms, including the topological sort. By the end of this course, you will have a firm grasp of both trees and graphs, how they are structured, how they can be represented, and very importantly, how these data structures can be traversed. You can then move on to a hands-on course, where you implement each of these structures, and the algorithms for their traversal.

The Binary Search

[Video description begins] *Topic title: The Binary Search. The presenter is Kishan Iyer.* [Video description ends]

When it comes to data structures and algorithms, the algorithms we tend to study include sorting algorithms. And another class of algorithm which you should be familiar with involve searching for an element in a list. One of the more popular searching algorithms is known as the binary search, and this is something which we will take a look at in this video. So given a list of elements, how exactly can we search for a specific element within that? The most naive approach is to iterate through each of the elements one by one, until we find the element which we are looking for. Now this is something which will definitely work, and it'll work whether the list happens to be sorted or unsorted. And the complexity of this particular search is O(N).

Since in the worst case we will need to iterate through each and every element until we find the one we're looking for. Or we may end up not finding the element, in which case we would have iterated through the entire list. Now, this particular search is known as a linear search. However, if the data which we are searching through happens to be sorted already, then we can do much better than a O(n) complexity. And this is where we can perform a binary search. When we do a binary search, we do not need to examine each and every element in the list. Given that the list already happens to be sorted, when we are given an element to search for, we will begin the search at the midpoint of the sorted list.

When we get this midpoint value, we will check whether it happens to be smaller or greater than the element which we're searching for. Now, consider the case that the element at the midpoint happens to be larger than the one which we are searching for. We know that this is a sorted list, which means all of the elements to the right of this midpoint, will be larger than the search term as well. So we can now restrict our search to the left of the midpoint. If the midpoint element happens to be less than the search term, then we can restrict our search to the right half of the array. Now the binary search is a recursive algorithm, which means that if we restrict a search to one half of the array, we further divide that sub-array into two halves, and compare our search term with the midpoint value.

We keep repeating this process until we either find the element which we're looking for, which will happen when the midpoint of one of our sublist happens to be equal to the search term. Or we end the search at the point we encounter a sublist of size one, and still don't find the element which we're searching for. It is now time to visualize exactly how a binary search works. For this, we will be using this array of nine integers, and you'll observe that at this point, this array is not sorted.

[Video description begins] *The following array displays: 15, 32, 26, 11, 36, 19, 42, 44, 14.* [Video description ends]

An important thing to keep in mind about a binary search is that it strictly works only with a sorted array. For a list which is unsorted, you will need to perform a linear search. So for a use case, we will work with a sorted list, and our goal is to search for the value 42 within this list. Taking a quick look at the elements, we know that this element does exist in the second to last position in the list.

[Video description begins] *The following array displays: 11, 14, 15, 19, 26, 32, 36, 42, 44.* [Video description ends]

But how exactly will a binary search find this element? Well, there are a few index values which we will need to keep track of for each of the subarrays we would be conducting our search in. This includes the lower bound of

the subarray which is the min-index, the upper bound which is the max-index, as well as the midpoint of the subarray. In the first iteration, our subarray happens to be this entire list. So the min-index points to 11, the max-index points to 44, and the middle points to the element 26. Our search term is 42, which happens to be greater than our middle element of 26.

[Video description begins] *The following text displays: 26 (middle value) < 42 (lookup value)*. [Video description ends]

So we now know that this search term can only be found in the right half of our array. So we can safely discard the left half, and we set the value for the min, max, and mid-index to the right half of our array.

[Video description begins] *These elements are 32, 36, 42, 44 from index 0 to index 3*. [Video description ends]

Given this right subarray contains four elements, with the index going from zero to three, the midpoint will be at index one. So how does a search term compare to the value at the middle index?

[Video description begins] *The following text displays: 36 (middle value) < 42 (lookup value)*. [Video description ends]

Well, once again, our search term happens to be greater, so we now need to restrict our search to the right half of this subarray. We again reset all of the index values.

[Video description begins] *The elements in the right half of sub-array are 42 and 44*. [Video description ends]

And on this occasion, the middle index and the min-index are exactly the same.

[Video description begins] *The min-index and mid-index is element 42*. [Video description ends]

And when we compare our search term with the value at the middle index, we get a match.

[Video description begins] *The following text displays: It's a match! The element at the midpoint is the one we're looking for and our search has ended*. [Video description ends]

So at each iteration, we check whether the value at the middle of the array happens to match our search term. And if not, we divide this array into two subarrays, and repeat the binary search in either the left half or the right half. If the length of our subarray happens to be one and we still don't find a match, then we can deem our search a failure. So with that, let us quickly summarize the properties of the binary search.

Working with a sorted list, allows us to effectively half the search area at every iteration. And the result is that a binary search works much faster than a linear search. Exactly how much faster though? So while a linear search has a time complexity of O(n), in the case of a binary search, the complexity is O(log N). And once again, the logarithm has the base 2.

The Binary Search Tree

[Video description begins] *Topic title: The Binary Search Tree. The presenter is Kishan Iyer.* [Video description ends]

Previously in this learning path, we took a look at the data structures of linked lists, stacks and queues. All of those are linear data structures, where the ordering of the elements within them is very clear and there is no hierarchy present. We will now explore the binary tree data structure, which is non-linear and offers several applications which cannot really be done using stacks or queues, for example. So what exactly is a binary tree? Well, to get there we can start off with what a tree data structure is. This is one which is made up of a number of nodes, very much like a linked list. However, the difference is that in a tree data structure, each node can point to a number of other nodes and not just one.

And unlike a stack or a queue, the ordering of elements within a tree data structure is not quite as important. In stacks and queues, there is a clear ordering of the nodes. However, in the case of a tree data structure, since you have single nodes pointing to many others, there is no such clear ordering present. Which is why you can say that a tree is a non-linear data structure. So what exactly is the purpose of having such a data structure? Well, a tree is very good at conveying hierarchical information. If you have ever visualized the org chart for any company, you will know exactly what such a tree data structure looks like.

However, the focus in this course is not going to be on a general tree data structure, but on a very specific variation of it, which is the binary tree. Why exactly is this the case? Well, in a general tree data structure, each node can have any number of children. However, such data structures are less useful overall. And while they are very good at conveying hierarchical information, the lack of a clear ordering of the data does limit its use cases.

[Video description begins] *Trees are not very commonly used as a data structure*. [Video description ends]

This however, is not the case with a binary tree, where each node can have at most 2 children. So while a binary tree is a special version of a tree data structure, there is a special case of a binary tree itself, which is a binary search tree. And this is one in which you can store data in an ordered manner.

[Video description begins] *In a binary tree, each node can have 0, 1, or 2 children*. [Video description ends]

Due to the wider range of applications, this course will focus on binary trees and the different variations which include binary search trees. So let us now take a look at a binary tree data structure and then get used to some of the terms which are associated with it. So here you have it, this is a binary tree containing 8 different nodes. Each node, which has arrows going out of it, are considered parents.

[Video description begins] *The binary tree displayed has eight elements. Element 2 is the parent node, which has two child nodes, 1 and 8. Element 8 further has two child nodes, 7 and 9. Element 7 has one child node 3. Element 9 has two child nodes, 5 and 14*. [Video description ends]

And those which have arrows coming into them are children. And to comply with the definition of a binary tree, a parent can have, at most, two children. So what are the special terms associated with binary trees? Well, there will always be one parent node which has no parents of its own. In our example, this is a node containing the data 2, and this is known as the root of the binary tree. Now all of these arrows which convey the parent-child relationships are known as the edges of the binary tree. And again, keep in mind that the edges do have a direction associated with them. And the nodes which don't have any children of their own are known as the leaf nodes.

[Video description begins] *Elements 3, 5, and 14 are highlighted*. [Video description ends]

Now these are the most commonly used terms which you'll come across when discussing binary trees, but there are a few others. You can see here that these three sets of nodes each have the same parent or grandparent.

[Video description begins] *Three sets of nodes are 1 and 8, 7 and 9 and 3, 5, and 14*. [Video description ends]

That is, they are of the same generation. And these are referred to as sibling nodes. Continuing now with this terminology, the root node is an ancestor of every node within this binary tree data structure. In any binary tree, every single node, aside from the root itself, is a descendant of the root node.

[Video description begins] *In this case, the root is element 2*. [Video description ends]

So the terms ancestor and descendant are not just in the context of the root node. You can say that in this example, the node 8 is an ancestor of 3, and also that node 5 is a descendant of 8. Also, nodes 3 and 5 have a common ancestor in 8. Also, the node 8, along with all of its decedents, themselves form what is called a subtree. And because they belong to the right side of node 2, you can say that it is the right subtree of the node 2. In fact, the left subtree of the node 2 contains just a single node, which is 1.

And with that, you now have a clear picture of what a binary tree data structure is. We can now take a look at one of its special variations, and in fact, one of its most widely used ones, which is the binary search tree. So what exactly is a binary search tree? So this is one which we call an ordered binary tree. And at its heart, it's just a binary tree, but with some special characteristics. Just like with a regular binary tree, any node in a binary search tree will have a left subtree as well as a right subtree. But these have special characteristics of their own. When it comes to the left subtree of any node, all of the nodes in that subtree will have a value which is less than or equal to the value contained within that node.

And correspondingly, all of the nodes in the right subtree of a node will have a value which is greater than the value of the node. So this is what brings about an ordering of the nodes in a binary search tree. Lower values can be found along the left and higher values along the right. So what is an example of a binary search tree? Well, this is one which we have right here. The root of this binary search tree contains the value 8, and you can see that this contains two subtrees.

[Video description begins] *The binary search tree displayed has nine elements. Element 8 is the parent node which has two child nodes, 3 on the left and 10 on the right. Element 3 further has two child nodes, 1 on the left and 6 on the right . Element 6 has two child nodes, 4 on the left and 7 on the right. Element 10 has one child node 14 on the left, which further has a child node, 13 on the right* . [Video description ends]

Within the left subtree, you have values which are all less than 8. And in the right subtree, all of the nodes have values which are greater than 8. Now we have already taken a look at all of the nodes in the context of the root node. Where every element in the right subtree is greater than the root, and everything on the left is less than or equal to it. However, this is not just with regards to the root node, the same rule applies to every single node within the binary search tree. So if you were to apply this to the node 3, for example, its left subtree contains the value 1, which is less than 3.

And all of the three nodes in the right subtree contain values which are greater than 3. And upon closer examination, you'll observe that this is true for every single node which is a parent. Even the nodes 6, 10 and 14, which have children, there is a clear placement of the child nodes depending upon the data within them. So what are some of the implications for this constraint on the elements in a binary search tree? Well, with regards to insertion of new nodes within the tree, there will be exactly one location within that tree where an element can be placed. And in fact, it is not just the content of the nodes within the tree, but also the order in which they are added, which affects the structure of the tree.

Exactly why this is a case is something we will cover in just a little bit when we further explore the insertion of nodes within a tree. And while this may seem a little too complicated, where we need to be very careful about inserting elements into the tree, the benefit of this approach is that lookup operations become much easier. When we are searching for a node in the tree, if we know that there is just one place where a node can be inserted, it also means that we know exactly where it can be found. And in order to find our particular node within the tree, we simply have to follow path along the left and right subtrees based on the value we are searching for. This is another operation we will explore a little later in this course.

BST: Insert and Lookup

[Video description begins] *Topic title: BST: Insert and Lookup. The presenter is Kishan Iyer. Insertion into a Binary Search Tree*. [Video description ends]

In the previous video, we discussed that when inserting any new nodes into a binary search tree, there is just one place it can go. I had also discussed the fact that the order in which nodes are inserted into a binary search tree have an effect on the structure of the tree. We will now see exactly why this is the case by taking a look at an example of how data is inserted into a tree.

[Video description begins] *The binary search tree displayed has nine elements. Element 8 is the parent node which has two child nodes, 3 and 10. Element 3 further has two child nodes, 1 and 6. Element 6 has two child*

nodes, 4 and 7. Element 10 has one child node 14, which further has a child node, 13. [Video description ends]

So here, we have an existing binary search tree which contains 10 different nodes, with the node 8 at the root. We now need to insert the node 2 into this binary search tree. How exactly will that happen? Well, remember the constraint of the binary search tree. For every node in the tree, its left subtree must only contain nodes whose data happens to be less than the data within the node itself. And all of the elements in the right subtree must contain data which is greater than the node value.

It is with this constraint in mind where we decide exactly where the node 2 fits into this binary search tree. To begin, we compare the value in this node with the value in the root. Since 2 happens to be less than 8, we know that this node belongs to the left subtree of the root. So we will head over to the left child of the root node. And we next perform a comparison with this new node with this left child.

[Video description begins] *The left child is element 3.* [Video description ends]

Now, once again, in order to comply with the constraint of the binary search tree, we need to make sure that all the elements less than 3 are moved to its left subtree. And since 3 already has a left child, we move further down this left subtree. And next, we compare the element 2 with the existing node 1. Now here, 2 happens to be greater than 1, so we know that it belongs to the right subtree of this node. Now, the node 1 does not have a right child, which means that we can now make this node 2 the right child of the node 1.

And when we do that, then the node 2 becomes a leaf node in this binary search tree. And the node 1, which was previously a leaf node, now becomes a parent. So any new node which is inserted into a binary search tree will become a leaf node. And its parent will be one of the existing nodes which has either no child of its own or has just a single child. So we have now inserted a node into this binary search tree which was less than the value at the root. The second node which we insert will have a value which is greater than the root, which means that it needs to go down somewhere in the right subtree of the root.

[Video description begins] *The value to be inserted is 12.* [Video description ends]

So since 12 is greater than 8, we iterate down to the right child. And now we perform a comparison with this node 10. Since 12 happens to be greater than 10, we iterate down to its own right subtree and then perform a comparison with its right child. You can see that this is a recursive operation. So we now perform this comparison and determine that this new node belongs to the left subtree of 14. And since it already has a left child, we need to, once again, perform a comparison with this child.

Now, this node 13 does not have any children of its own, so the node 12 can become one of its children. And the only question now is whether it becomes a left child or a right child of 13. Because 12 happens to be less than 13, it will become a left child. So now 12 is a new leaf node within this binary search tree. And the node 13 is no longer a leaf. So a big takeaway from these two insert operations we have just seen is that given the existing state of the binary search tree, there was exactly one location into which each of those elements could be inserted. And there is precisely what we have done. This predictability of the placement of elements in a binary search tree also has a bearing on lookup operations, where we search for a particular element within the tree.

[Video description begins] *Lookup in a Binary Search Tree.* [Video description ends]

So here, we will now explore how exactly we can perform such a search operation. For which, we go back to the same binary search tree which we had at the beginning of the video. So this is without the nodes 2 and 12. So let us just say that the specific node which we want to look up is one which contains the data 4. Just like with the insert operations, even lookup operations will begin at the root node of the binary search tree. So we perform a comparison of the root value with this value to be searched.

Now given 4 is less than 8, if there is a node within this binary search tree which contains the data 4, we know that it can exist only within the left subtree of the root. So we can completely disregard half of the binary search tree at this point. So we'll now iterate down towards the left child of 8, and then we recursively perform the

comparison with each of its children. So the comparison continues with this node 3. And since 4 is greater than 3, we know that if this value does exist within the tree, it can only be found within the right subtree of this node. So we go towards its right child, and then once again, perform this comparison. Since 4 is less than 6, we know that we need to perform this comparison with its left child next.

And at this point, we have found a match. At this point, we can end our search. And you will observe that at each step, we have effectively eliminated half of the binary tree from contention when it comes to searching for our node. This is what makes lookup operations in a binary search tree extremely efficient. So now that we have performed a successful search operation, let us visualize what an unsuccessful search will look like. So on this occasion, we will look for a node containing the value 15, which we know does not exist within the tree. But how exactly will such a lookup take place? As usual, we will begin our search at the root node.

And since 15 is greater than 8, we will need to iterate down towards its right subtree. Since 8 has a right child, we compare the value 15 with the node 10. Now since 15 is once again greater than 10, we need to iterate further down its own right subtree. This is where we perform the comparison with 14, and we know that 15 is greater than 14, and we will need to move down its right subtree. But this right subtree does not exist, this means that the value 15 does not exist within this binary search tree. So when performing any lookup operation within a binary search tree, we keep iterating down either the left or the right subtree, depending upon the value.

And then we either find the element we are looking for, in which case, our search is a success, or if we encounter a null value, then it means that our search was a failure. You'll now observe that this binary tree we have been working with has some asymmetry in it. In the context of the root node, there are more elements in its left subtree, specifically 5, as compared to its right, which has only three nodes.

[Video description begins] *A Balanced Tree*. [Video description ends]

So let us consider another case where we have what is called a balanced tree. This is where there are as many elements within the left subtree of the root as there are in its right subtree. In this specific example, there are a total of seven nodes within the binary search tree containing the data 1 through 7. One reason for the structure is that the first element to be inserted into this tree would be the node containing 4. And this is because the first node to be inserted into any tree becomes its root. For example, if we went in the sequence 1 through 7, so the first element to be inserted becomes the root. And then when we add 2, it will become the right child of that root.

[Video description begins] *An Imbalanced Tree*. [Video description ends]

When we insert the note 3, it becomes the right child of 2. And this will go on until we end up with this kind of structure. So this is a highly skewed tree. And in fact, this is not very different from a linked list. Why exactly is this relevant?

[Video description begins] *The Binary Search Tree*. [Video description ends]

Well, when it comes to insert operations, we can say that the average case complexity will be O(logN) to the base 2, but this is only in the case of a somewhat balanced tree. In the case of a highly skewed tree, like the one we just saw, the complexity will in fact be O(N), just like with a linked list. This also applies with lookup operations, where the complexity will be O(logN) in the average case, but will be O(N) in the worst case.

[Video description begins] *The following text is displayed: For both insertion and lookup, we halve the tree we have to traverse at every step (this gives us the logN complexity)*. [Video description ends]

BST: Extreme Values, Max Depth, and Sum Path

[Video description begins] *Topic title: BST: Extreme Values, Max Depth, and Sum Path. The presenter is Kishan Iyer. Minimum Value in a Binary Search Tree*. [Video description ends]

In this video, we will take a look at some of the common tasks which will need to be performed when using a binary search tree. One of these is to identify the minimum or the maximum value which is stored in the tree. These two operations are quite similar, but they just happen to be symmetric. So let us now explore what exactly needs to be done to find the minimum value. And the one-sentence answer to that is that you keep traversing the left subtree of every node until you hit a leaf node.

And this, of course, is because every left child will have a value which is smaller than that of the parent. One common way to implement such a search for the minimum value in a binary search tree is to use a recursive function call. And these recursive calls will stop once we happen to hit a node which has no left child. So this is the left-most value in the tree, and this is the node which contains the minimum value. Now in order to find the maximum value in a binary search tree, you would effectively perform the same type of search, except instead of traversing down the left subtree starting from the root, you will traverse down the right subtrees.

[Video description begins] *Maximum Depth of a Binary Tree*. [Video description ends]

We move on now to another common operation in a binary search tree, which is to find its maximum depth. So what exactly is meant by maximum depth? For that, consider this example binary search tree.

[Video description begins] *Find the Maximum Depth of a Binary Tree. The binary tree displayed has nine elements. Element 8 is the parent node which has two child nodes, 3 and 10. Element 3 further has two child nodes, 1 and 6. Element 6 has two child nodes, 4 and 7. Element 10 has one child node 14, which further has a child node, 13*. [Video description ends]

So we start off at the root element, which is 8, and if we traverse down to one of its children, we say that this element, in this case, 10, is at the depth of 1. If we traverse further down to the child of 10, we say that we are at a depth of 2. And then another traversal to the child of 14 will take us to a depth of 3. So when we talk about the depth of a particular node in a binary search tree, what we mean is its distance from the root. If we stick to the ancestor-descendant terminology, the depth refers to the number of generations between a particular node and the root of the tree.

So in the case of the maximum depth of a binary search tree, this refers to the maximum distance between the root node and any of its leaf nodes. In this particular example, you will observe that the maximum depth of the tree is 3. However, if we were to add an additional node, which becomes a child of one of the existing leaf nodes at depth 3, then this binary search tree will have a new maximum depth value of 4. Now how would you find the maximum depth of a tree using code? Well, this is where you will once again make use of recursive function calls and calculate the maximum depth from every single node within the tree. And out of those, pick the one with the maximum value.

[Video description begins] *Sum Paths in a Binary Search Tree*. [Video description ends]

We move on now to another common operation with a binary search tree, and this is to find sum paths within the binary tree. So what exactly is a sum path? Well, given a particular number, you will check if there is any path from the root to a leaf node within the binary search tree where the sum of all the values in the nodes along that path sum up to the given number.

[Video description begins] *Sum Path*. [Video description ends]

I can imagine that it seems a little abstract, so let us visualize this using a real example. So we construct this binary search tree here, and then we can calculate the sum paths to the various leaf nodes.

[Video description begins] *The binary search tree displayed has eight elements. Element 2 is the parent node which has two child nodes, 1 and 8. Element 8 further has two child nodes, 7 and 9. Element 7 has one child node 3. Element 9 has two child nodes, 5 and 14*. [Video description ends]

So here, we calculate the sum path to this leaf node with the value 1, and this includes the value 2 at the root, plus the value 1 at the leaf, which gives us a sum of 3. If we traverse down this path, right up to the leaf of 3, then the value that we need to sum up are 2, 8, 7 and 3, which gives us a sum of 20. Going down this other path, which includes 2, 8, 9 and 5, the sum of this path is 24. And then the final path within our binary search tree includes these four elements, which sum up to 33. Whenever we calculate a sum path, it is usually when a number is given to us and we need to calculate whether there is any path to a leaf which sums up to that given value.

So the way to do this is, is for every leaf node within the tree, we calculate the sum of values in the nodes to that leaf and then we check whether it matches with the given number. And one way to do this is that you start off with the value which has been given. And then for each node which you hit when traversing a path, you subtract the value of that node from the value which has been specified. Once you hit a leaf node and the cumulative subtraction results in a value of zero, then you have found a valid sum path.

BST: Breadth First Traversal

[Video description begins] *Topic title: BST: Breadth First Traversal. The presenter is Kishan Iyer. The Binary Tree - Traversal*. [Video description ends]

In this video, we will explore different ways in which a binary tree can be traversed. Earlier in the course, we had discussed that there is a clear ordering of elements which are stored in linear data structures. Such as linked list, stacks, and queues, but that there is no strict ordering of elements in tree data structures. However, there is an order in which we can traverse the nodes in trees. And here we will take a look at how this can be done in binary trees. So when it comes to traversal in binary trees, the interesting fact is that there is not just one way to go through it. But in fact, there are several ways in which you can visit every single node within a tree. This means that even for the same tree, there are many different orders in which the nodes can be accessed. Depending on the method which you have chosen for your traversal.

Now, just to keep in mind that it is the visiting of nodes in a tree which is called traversing a tree. And we will now explore two different ways in which this traversal can be done. The first of these is a breadth-first traversal, and the other one is a depth-first traversal. Now this is something which applies generally to any tree data structure and not just a binary tree. But we will be looking into these in the context of a binary tree. We will start off though with a breadth-first traversal. And this is one which involves traversing the nodes of a binary tree in such a way that you visit all of the nodes at a particular depth or level in a tree. Before you move on to the next level. As with any tree traversal, a breadth-first traversal also begins with the root node, and this is set to be the node at level 0 in the tree.

We will then visit all of its immediate children which are at level 1. Then all of the grandchildren which are at level 2. And then all of the nodes which are at level 3 and so on until the entire tree has been traversed. So each time we hit a node in our traversal, we say that we have visited that node. And generally speaking, when we have performed a visit on a specific node. We say that we would need to perform a check to see whether there are other node at the same level which have not been visited. And then visit those first before we move on to any children of nodes at that same level. So it is only when a level has been completely exhausted that we move on to the next level. And this is the main factor in a breadth-first traversal.

And of course, we continue with this particular method until every single node within the tree has been visited. And we end up with a specific ordering of the elements based on this breadth-first traversal. While this gives you some idea of how a breadth-first traversal works, it will be far easier to understand if we just visualized it. So we use this particular tree in order to demonstrate how a breadth-first traversal will work. So it is the element 2 which is at the root of this tree.

[Video description begins] *A binary tree is displayed that has eight elements. Element 2 is the parent node which has two child nodes, 1 and 8. Element 8 further has two child nodes, 7 and 10. Element 7 has one child node 3. Element 10 has two child nodes, 9 and 14*. [Video description ends]

Or if this represents a subtree in a larger tree. We can say that this represents level N. Its immediate children are at level N+1, and their children, that is the grandchildren of the root, are at level N+2. In our example, this includes the nodes 7 and 10, and their children are at level N+3. Which happens to be the highest level in this particular tree or subtree. Now, how exactly will a breadth-first traversal work? Well, broadly speaking, we start off at the root, visit each of its children, and then from there we visit each of its grandchildren. So note here that we are going from top to bottom and then also within each level in this particular case from left to right. So what exactly will the output of this breadth-first traversal be?

Well, we always visit the root node at the very beginning. So we will now print out the value of the root node, which is 2. So this is the first element in our breadth-first traversal. And now to move on to the next level. So this contains the elements 1 and 8, and within the same level we move from left to right. So the first element to visit at this level will be 1. So we now have the second element in our breadth-first traversal. And the next element which we visit, is 8. So at this point all of the nodes at level N and level N+1 have been visited. Now it's time to move on to level 2. Just as we did with the previous level, the first node to be visited at any given level happens to be the leftmost node. So here, we print out the value 7.

And then the next node we move on to is 10. Now there are no more nodes at this level, which means that we can now move on to level 3 in our binary tree. And the leftmost node here happens to be 3, so we print that out. The next node at this level is 9, so that gets printed out as well. And the final value to print out at this level is 14. With that done, there are no more children and no more levels for us to go through, so we can end the breadth-first traversal. And the order of nodes which we end up with is 2, 1, 8, 7, 10, 3, 9, 14. So we now have an understanding of the theory of a breadth-first search. But how exactly would we implement it in code? Well, we will start off with the root node and then we can make use of a queue data structure for this traversal. Given that we have already covered queues previously in this learning path,

I won't get into the details of that data structure. However, once we have added the root element to the queue, we will find that there are no more elements at the same level. At which point, we can invoke a dequeue operation and process the node. So what exactly is involved in processing a node? Well, in addition to printing out the contents of the node, we will add the left child and then the right child of that node to the queue. So note this order of adding the left child first and then the right. Since we're just one level down from the root at this point, all of the nodes at that particular level are now on the queue. In which case we can begin triggering the dequeue operations and process each of the nodes in turn.

So in this way it is the left child of the root which gets processed first and then the right child. In fact, we continue with this process as long as the queue is not empty. So at each level we keep adding nodes to queue from left to right. And once all of the elements have been added to a queue at a particular level, then we dequeue each of them. And since a queue follows a first in, first out policy. It will be the left nodes which will be processed first and the rightmost node will be the last to be processed at any given level. So, that covers breadth-first traversals, and we now move on to the depth-first traversal of a binary tree. In contrast to a breadth-first traversal, when we go depth-first.

We go right down to the leaf of a binary tree before moving up the tree and visiting the higher nodes. In other words, we go deep first before we move up. Now even when we do go depth-first, there are different methods which we can apply for this traversal. And these are pre-order traversal, in-order traversal, and post-order traversals. We will get into the details of each of these traversals in just a little bit. But just to give a general overview, all of the depth-first traversals are most efficiently implemented using recursion. This is because when we are considering a particular node during our traversal, we will assume that it happens to be the root of some subtree. Since we can effectively treat every single node as some form of root, we can make use of a recursive algorithm.

So the same techniques we used to process a node will also be used on each of its children. And then the same technique will be applied on each of their children and so on. The base case for any of these depth-first traversals is when the root node happens to be null. This will occur when we have traversed all the way down to a leaf node, and then we try to process each of its children which will be null. So we can now take a look at one sentence descriptions of each of the different depth-first traversals of a tree. If we visit a node before we visit

each of its children, then this is known as a pre-order traversal. If we visit a node between visiting each of its children, then this is an in-order traversal. So this is where we first visit its left child, then the node itself, and then the right child. And if we visit the node only after both of its children have been visited, then it is a post-order traversal.

BST: Depth First Traversal - Pre-Order and In-Order

[Video description begins] *Topic title: BST: Depth First Traversal - Pre-Order and In-Order. The presenter is Kishan Iyer. Pre-Order Traversal*. [Video description ends]

We will now take a look at each of the different depth first traversals, which are possible for a binary tree. The first of these is a pre-order traversal. So we will now visualize exactly how a pre-order traversal of this binary tree will work.

[Video description begins] *A binary tree is displayed that has eight elements. Element 2 is the parent node which has two child nodes, 1 and 8. Element 8 further has two child nodes, 7 and 10. Element 7 has one child node, 3. Element 10 has two child nodes, 9 and 14*. [Video description ends]

And this, of course, has its root element as 2. All of the depth first traversals can be applied recursively on any of the nodes within a tree. So we could also consider this to be a subtree within a larger binary tree. In any case, when we are processing a particular node, we will consider that this is the root of a particular subtree, and that it has a left subtree, as well as a right subtree. The difference between each of the different depth first traversals is the order in which the nodes are visited, the node itself, its left tree, and its right tree. In the case of a pre-order traversal, it is the node being processed which is visited first, which is why the term pre-order.

[Video description begins] *Each node is processed first (PRE) before its right and left subtrees*. [Video description ends]

And once the node has been visited, we move along to the left subtree, and then over to the right subtree. So in this particular example, it is the node 2 which is visited first. And then following that we move ahead to the left subtree, which is rooted at node 1, and then over to the right subtree, which is rooted at node 8. So just to clarify, when you have a node, its left subtree and right subtree in a pre-order traversal, it is the node itself which will be visited first. And then you move ahead to its left subtree, and then finally you visit the right subtree. We can now apply this on our binary tree example.

So after the node 2 has been visited, which is the root, it is time to process the left subtree. Once we get here, we visit the node before we process any of its children. So the first element in our traversal is already 2, and the next one will be 1. Given that node 1 does not have any children of its own, we can exit from the left subtree at this point, and then move over to the right subtree of the root, which is rooted at node 8. How do we process this particular node? Again, in a pre-order traversal, you first visit the node. So in addition to the nodes 2 and 1, which we already have visited, we will also visit the node 8. So now that this node have been visited, the next node to be processed is its left subtree, which takes us to the node 7. 7 itself has a left child but no right child, but in a pre-order traversal, we will visit this node first. So the order of the nodes we have processed so far is 2,1,8,7.

At this point, you will notice that it does seem very similar to a breadth-first traversal. However, this is simply a coincidence. Because the next node we will visit at this point is the one which is the left child of the node we have just processed, and this is the node 3. Once again, in a pre-order traversal, we visit this node, and then process each of its children. Given three is a leaf, there are no children to process. In which case, we head back up the tree, which will take us to the node 7, and then we will process its right child. Given there is no such right child, we move back up one level to node 8, and then process that node's right child, which happens to be 10. Applying a pre-order traversal again, we visit this node 10 and then process its left child. We visit that node so that 9 gets added to our traversal.

And then, given 9 doesn't have any children, we go back up to 10, and then process its right child. At this point we have visited all of the leaf nodes in our tree, and we end up with the final ordering of our pre-order traversal. So the result of this pre-order traversal is 2, 1, 8, 7, 3, 10, 9, and 14. And we can now take a look at how an in-order traversal of that same tree will work. So now we start off with the root node, and it of course contains a left subtree, as well as a right subtree. In the case of an in-order traversal, however, we will be processing the left subtree first, which is rooted at node 1. And then we will process the node itself, which is the node 2. And then the right subtree, which has its root at 8. So just to clarify the order, when processing a node during an in-order traversal, we will first visit its left subtree.

And then following that, we will visit the node itself, and then we will go ahead and visit the right subtree. And this is something, once again, which will be applied recursively. So going back to our binary tree, when we go from the root and then process its left subtree, we end up at the node 1. And then from there, we need to process that node's left subtree. Given it does not have a left subtree, we will visit the node. And 1 becomes the first element which we print out in our in-order traversal. Since all of the left subtrees of the node 2 have been processed at this stage, this is when we visit that node and 2 becomes the second node in our in-order traversal. And this is when we will go ahead and process node 2's right subtree.

It is now the turn of node 8 to be processed, but we won't be visiting it just yet. Instead we will process node 8's left subtree, which takes us on to node 7. This in turn has a left subtree, so we will go ahead and process that. And given 3 does not have any left children of its own, we will visit 3. And this becomes the third element which we visit. It is at this point where we go further up the tree. And given that all of the elements in the left subtree of 7 have been processed, it is time for us to visit 7, and this becomes the fourth element in our traversal. Given 7 does not have any right children, we go back up a level, and this is the time where we visit node 8. The next node to be processed will be the right child of node 8, and this is node 10. But because 10 has left children of its own, we need to move down its left subtree.

Once we process node 9, we see that it does not have any left children of its own. And this is the time where we can visit node 9 and then go back up the tree and visit node 10. Now we can process the right child of node 10. And given that this is a leaf node, we can visit this. All of the nodes in our binary tree have been visited. So one thing you will note about this particular ordering of 1,2, 3, 7, 8, 9,10 and 14, is that the elements are in ascending order. In fact, this is exactly why this particular type of traversal is called in-order, because all of the elements will be visited in ascending order. This is, of course, if you're working with a binary search tree.

BST: Depth First Traversal - Post-Order

[Video description begins] *Topic title: BST: Depth First Traversal - Post-Order. The presenter is Kishan Iyer.* [Video description ends]

We have already looked at pre-order and in-order traversals of a binary tree. And it is now time for us to take a look at what a post-order traversal will look like. So once again, we consider this particular binary tree, which has a root, and then this root has a left subtree and a right subtree. In the case of a post-order traversal, it is the left subtree which will be processed first, then the right subtree, and only then will the node itself be visited. So just to summarize the order, we first process the left subtree, then the right subtree, and the last node to be visited will be the node itself.

[Video description begins] *A binary tree is displayed that has eight elements. Element 2 is the parent node which has two child nodes, 1 and 8. Element 8 further has two child nodes, 7 and 10. Element 7 has one child node, 3. Element 10 has two child nodes, 9 and 14.* [Video description ends]

So, what would a post-order traversal of this binary tree look like? So we start off with the root node, and then we will process the left subtree. This will take us to the node 1, in which case we will need to process its own left subtree. But given 1 does not have a left subtree, we will need to process its right subtree next. Given 1 is a leaf, there is no right subtree either, so we will now visit node 1. We then go back up the tree and we're back at node 2. But now we need to process its right subtree, which takes us to node 8. We then move on to its left subtree,

which takes us to node 7. And given that 7 has a left child of its own, that is the direction in which we head. So in short, we keep moving further and further to the left until we hit a leaf.

This brings us to the node 3, and given that this is a leaf, this is the one which will be visited next. This is when we move up the tree back to 7. And given all of its children have been visited, we can visit this node, and 7 becomes the third element in our traversal. We then move back up to 8, and all of its children have not yet been processed. So we now move ahead to its right child, which is 10. 10 has children of its own which have not been visited, so we will first take a look at its left child, which takes us to 9. 9 does not have any children of its own, so we can visit that next, and then we move up the tree. And given 10 has some children of its own which have yet to be processed, specifically its right child, we move ahead to 14.

And given this is a leaf, this is the one which is visited next. Now all of 10's children have been visited, so we can finally visit 10 itself. At this point, we move further up the tree, and all of 8's children have been visited, so now it is time to visit 8. And at the very end, it is the root node which will be visited, given we have processed and visited all of its children already. So this is the final order of a post-order traversal. For this tree in our example, it is 1, 3, 7, 9, 14, 10, 8, and 2. When it comes to a post-order traversal, the root node is always the last one to be visited.

An Introduction to Graphs

[Video description begins] *Topic title: An Introduction to Graphs. The presenter is Kishan Iyer.* [Video description ends]

It is time to take a look at the last of the data structures we will be studying in this learning path and that is the graph. So what exactly is a graph? This is a data structure which is used to represent relationships between entities. The entities in question here can be pretty much anything. It is up to a graph to merely show exactly how all of those entities are related to each other. Note that these relationships can be rather complicated and we will be taking a look at a few such examples in this course. So what exactly makes up a graph? Well, broadly speaking, the constituent elements of a graph are a vertex, which represents an entity and an edge, which represents relationships.

To get a clearer picture of this, let us consider a concrete example, where we have a number of different vertices which represent people. Now, if you have a number of edges which connect these entities, what you may end up with is a professional graph. This will represent all of the people who work together say in an organization, and how they relate to each other. Unlike tree data structures though, these relationships need not be hierarchical in nature. Now, you could also have a collection of vertices representing people and edges showing their relationships in order to represent a social graph, say in a social network.

Such a graph could help you visualize the number of direct connections for a particular individuals and also a number of their indirect connections. So we can now take a look at another example, where the vertices this time represent locations. Let's just say geographical locations. In this case, you could have edges which connect these together in order to portray a collection of locations which are linked to one another which could form a transportation map. Within the graph data structure, you could store, for example, the nature of the connection between the different locations, whether it is by road, by rail, or by air.

[Video description begins] *By road, it can be bus, car, or taxi. By rail, it can be by individual trains. By air, it can be by airlines.* [Video description ends]

If you have a collection of vertices, which represent computers scattered across the world, these along with the collection of edges, which show which of these are connected to each other, could combine to represent the Internet. The graph data structure in this instance will be able to convey how one can connect from one computer to another whether it is a direct connection or an indirect one. You could in fact also store information about the type of connection between the computers in order to convey whether the connection represents a wired one or a wireless one. So just to give a brief overview of graph data structures, these are used to represent information for

many many real world applications, as we have already seen. Questions related to graph are in fact commonly asked in many programming interviews.

And these questions often start from simple graph related concepts and then build upon them to the point where they can get very, very complex. There is also a field known as graph theory, and this is very much a field in and of itself. Some other problems which are studied in graph theory include algorithms, which are used to optimize many problems related to graphs. For example, if you have a graph which represents connections between different geographical locations, it is in graph theory where you would write algorithms in order to solve for the shortest path between different locations. Given the rather broad scope of graph related topics, in this course, we will very much just be scratching the surface. However, what we do cover should lay a strong foundation for you and these will be enough for you to solve graph related problems from first principles.

So let us now zoom out a little bit and then ask the question, what exactly is a graph? I have already alluded to the fact that a graph is a collection of vertices and edges. In its simplest form, a graph can just be two vertices and a single edge which connects them together. To visualize this, consider that we have an entity A which happens to be related to an entity B. As represented by this edge E. Now in this case, both of these entities, A and B, will be the vertex and then the connection between them will be the edge of a graph. So this itself is a basic graph but we can also convey additional information within this with a slight modification. For example, by including an arrowhead, we can convey that this relationship is one way. As an example, consider that A represents a source and B a destination.

So this is also a valid graph. Now when you have two vertices which are connected by an edge containing this arrowhead, then we say that this edge is a directed edge. And that the relationship between the two vertices has a specific direction. And this contrasts with edges without an arrow, which are undirected edges. So this is an example of an undirected edge which is what we saw earlier on. And this is a directed edge which conveys a 1-way relationship. So when the edge is undirected, it is used to convey a 2-way relationship. Examples of this include 2-way roads but also for example, 2-way friendships.

In contrast, the directed edges represent one way relationships such as 1-way roads, but also instances such as the relationship between managers and those who report to them. So the organization chart for any company can be represented as a graph containing directed edges. You will now get used to some of the terminology associated with graphs by taking a look at a sample graph. So here we have six different vertices and seven different edges.

[Video description begins] *A graph is displayed. It has six vertices: A, B, C, D, E, and F. Vertex B is connected to A, which is connected to C. Vertex C is connected to vertices D and E, which are connected to vertex F. Vertex E is also connected to vertex D.* [Video description ends]

Now, we say that the nodes A and B are adjacent nodes because they are directly connected to each other with an edge. Similarly, nodes A and C are also adjacent nodes but B and C are not. Also, this particular node D, is connected to three different edges. In such a case, we can say that the degree of D is 3. Similarly, the degree of the node A is equal to 2, and this is because there are two different edges which are incident on the vertex A. Now, it is possible for us to get from the node B over to the node C via A.

And in such a case, we say that there is a path between B and C. Also notice the series of edges which connect the nodes D, E and F, and back to D again. In this case, we say that these three nodes together form a cycle. In fact, we can also say that there is a cycle which exists between the nodes D, C, and E. So you can already see that a lot of these relationships can get a little complex. In many graphs though, such cycles are not permitted. So for example, if you were to get rid of these two edges, what we end up with is an acyclic graph. And given that the edges are undirected, it is in fact an undirected acyclic graph. One more feature of this graph is that there exists a path from each of these nodes to every other node.

[Video description begins] *Every node is connected to every other node via a series of edges.* [Video description ends]

In such a case, we say that the graph is a connected graph. And if you really look at this, you will see that a connected graph is nothing but a tree. And this becomes evident if you were to just restructure this particular graph in this manner. Going back to our original layout, if you were to take a subset of the nodes and edges, we can see that even these three nodes and the two edges together form a graph in and of itself.

[Video description begins] *A tree is displayed in which node C is the parent node, which has two child nodes A and E. Node A further has a child node, B. Node E has two child nodes, D and F.* [Video description ends]

In fact, the nodes D, E, and F also form a graph. And in the current form, both of these graphs are connected together, thanks to the edge between C and E. But what if we were to remove it? Well, what we have here is still very much a graph, but it is just no longer a connected graph. So since both of these individual graphs have no cycles, they are in fact, trees. And if your graph contains more than a single tree, which are not connected to each other, then what you have can be termed a forest

Graphs as an Adjacency Matrix

[Video description begins] *Topic title: Graphs as an Adjacency Matrix. The presenter is Kishan Iyer. The Graph Representation.* [Video description ends]

We saw in the previous video that graphs can represent all kinds of information and have a wide array of use cases. But how exactly do we represent graphs? This is what we will cover in this video. Now, what exactly do we need in order to represent a graph? Well, we will need some means of modeling a vertex. Since vertices represent entities, each entity will have some information connected with it. And we need some structure to store such data. The other component which makes up a graph is the set of edges. And we need some way to model these for both the directed, as well as the undirected edges. So given these requirements, what are the options which we have in order to represent graphs?

Broadly speaking, there are three standard graph representations. So these are an adjacency matrix, an adjacency list, and an adjacency set. We will now zoom in on each of these, starting with an adjacency matrix. As the name implies, this is represented using a matrix data structure, which, of course, contains rows and columns. A matrix data structure very much resembles a table. But a question you may ask is, what exactly do the rows and columns represent? Well, one thing to note is that this matrix is a square matrix, where there are as many rows as there are columns. And both of them in fact represent the vertices in the graph. So if you have n different vertices in the graph, the size of your matrix will be n by n.

So each vertex has its own row and column represented in the matrix. As for the data within the matrix, well, each cell represents the relationship between the corresponding vertices. So the matrix cell is a representation of an edge in the graph. Now, this may be a little abstract right now, so let us make use of a real graph and then represent it using an adjacency matrix.

[Video description begins] *A graph is displayed. It has five vertices: A, B, C, D, and E. Vertex A is connected to B, which is connected to E and D. Vertex E is further connected to vertex C. Vertex D is also connected to vertices C and E.* [Video description ends]

So here we have this graph containing five different vertices, these are labeled A through E. And there are also six different edges. Do keep in mind that all of these edges are directed. So I had mentioned that if there are n edges in any graph, we represent it with an n by n matrix. So we now have a five by five matrix to represent this graph. Each row and column corresponds to a specific vertex within our graph, as you can see. And the data within the matrix will represent the relationship between the corresponding vertices, that is, they represent the edges. Now, within the matrix, we can populate the diagonals with zeros. This means that within our graph, there is no edge which connects a node with itself.

With that done, we move over to the first row in this matrix, which corresponds to the vertex A, but more specifically to all of the outgoing edges from A. In our specific graph, there is just one such edge, which moves

from A to B. So in our adjacency matrix, the cell corresponding to the row A and the column B gets populated with one. And all of the other values in this row will be zero. So each of the rows within this adjacency matrix represent edges, where the corresponding vertex is a source. And for each of the columns, the corresponding vertex is a destination. So when we move along to the second row in this adjacency matrix, there are two outgoing edges from B, which terminate at C and D. Which is why the columns corresponding to C and D in this row will be populated with one and the others will remain zero.

To represent the edge from C to E, we populate the last cell in this third row with one. And then to represent the two edges from B, which terminate at C and E, these are the values which we fill up for the fourth row. And given that there is no outgoing edge from the vertex E, all of the values in the final row of this adjacency matrix will be zero. So the graph on the left of our screen can be represented using an adjacency matrix, which we see on the screen's right. So what if this particular graph happened to be undirected? Well, if you consider that the edge connecting A and B will no longer be unidirectional but bidirectional, then we will have to change the value in the row representing B and the column corresponding to A.

So the relationship between these two vertices becomes bidirectional, which is reflected by the symmetry in the matrix. Moving on now to the edge between B and D, we once again make sure that the corresponding cells are both one. And we do the same for the cells corresponding to B and C. You will notice that making all of these edges undirected creates a certain symmetry in the matrix along the diagonal. So we now make this particular edge between C and E undirected, so we now make this particular edge between C and E undirected.

And then we move on to the edge between C and E, and finally, the edge between D and E as well. So towards the right, we have an adjacency matrix which represents the graph on the left as though the edges were undirected, the edge between D and E as well. So towards the right, we have an adjacency matrix, which represents the graph on the left as though the edges were undirected. So now that you know what an adjacency matrix looks like, in the next video, we will take a look at the other graph representations, which are adjacency lists and adjacency sets.

Graphs as an Adjacency List and Set

[Video description begins] *Topic title: Graphs as an Adjacency List and Set. The presenter is Kishan Iyer. Adjacency Matrix.* [Video description ends]

In the previous video, we took a look at representing a graph in the form of an adjacency matrix. But now we can take a look at an adjacency list and how it is represented as a graph.

[Video description begins] *The Graph Representation vs. Adjacency List.* [Video description ends]

So in an adjacency list, each vertex is represented as a node. Now, each vertex also has a pointer to a linked list. A linked list, as you know, itself contains a number of nodes. And the specific nodes in this linked list represent all of the vertices in the graph to which a particular vertex has a connection. So in a graphical representation of an adjacency list, if there is a vertex which has an edge leading to another vertex U. Then we will have an associated linked list of which one of the nodes will represent U. We can now visualize exactly what this will look like for our sample graph.

[Video description begins] *Adjacency List. A graph is displayed. It has five vertices: A, B, C, D, and E. Vertex A is connected to B, which is connected to vertices C and D. Vertex C is further connected to vertex E. Vertex D is also connected to vertices C and E.* [Video description ends]

Since this graph has a total of five vertices, these five vertices are represented by these five circles on the right.

[Video description begins] *These are A, B, C, D, and E.* [Video description ends]

Now there is a single edge which originates at A, and then terminates at B. The top circle on the right represents the vertex A, and it points to a linked list. This linked list will contain exactly one element, because there is just one vertex in the graph to which A is connected. So this linked list has a node which corresponds to the vertex D. Similarly, the vertex B in the graph has outgoing connections to C as well as D. So the linked list corresponding to B will have nodes which represent C and D. As for the vertex C, there is just one outgoing edge and this is to the node E.

And there will be just a one element linked list corresponding to C in the adjacency list. Moving on to the vertex D, there are outgoing edges to C and E, which is why both of those vertices are represented as nodes in the linked list. The node E, though, does not have any outgoing edges, so within the adjacency list, there is no linked list associated with E. So now we have an adjacency list for our sample graph. So, what if this particular graph were undirected? Well, we simply assume that all of the edges are bi-directional, in which case the adjacency list changes to what you now see on the right.

[Video description begins] *Five linked lists are displayed. In first, A is connected to B. In second, B is connected to A, C, and D. In third, C is connected to B, D, and E. In fourth, D is connected to B, C, and E. In fifth, E is connected to C and D.* [Video description ends]

So with this in mind, let us now summarize some of the properties of adjacency lists. For one, these data structures are much more space efficient than adjacency matrices. This is because for a very large graph with a very large number of vertices and edges, an adjacency matrix will have plenty of cells which have a value of zero. On the other hand when it comes to an adjacency list, there will be a node in a linked list only for the existing edges within the graph. Since an adjacency matrix has a cell even for non-existent edges, it tends to be rather wasteful. However, while adjacency lists are indeed quite space efficient, they may not be the best representations of graphs. The reason for this is that the order of the vertices in the linked list in the adjacency lists does not really matter.

For example, if a vertex A has edges going out to vertices B and C, then the linked list corresponding to A will contain the nodes representing B and C, but those elements can be in any order. A side effect of this is that the same graph can have several different adjacency lists representing it. Where the only difference between the variations of adjacency lists are the ordering of elements within the linked lists. There are also other drawbacks when using an adjacency list to represent a graph. For example, consider a case where you delete one node or one vertex from a graph. The node which you have deleted may have several outgoing edges. And the removal of this node means that its corresponding linked list in the adjacency list will need to go. Now that is simple enough, however, the trickier part is for all of the incoming edges to the vertex which you have removed from the graph.

In order to change the adjacency list to reflect this removal, you will need to iterate over each and every linked list in the adjacency list and then delete the node corresponding to the removed vertex. One way to speed up such operations is to replace the linked list which is used to denote all of the nodes to which a vertex is connected with a binary search tree. And this is accomplished when using an adjacency set. So what exactly is an adjacency set? Well, this is very much like an adjacency list, except that instead of a linked list, you use a set to represent all of the outgoing edges from a particular vertex. In most programming languages, sets are implemented internally as binary search trees. So searching for an element within a set, becomes a O(log n) operation. While searching for that same element in a linked list is a O(n) task.

So now that we have an idea of the three different graph representations, let us take a look at the situations where we might use each of them. An adjacency matrix works really well when the graph is well connected. This is when each of the vertices in the graph have outgoing edges to many of the other vertices, and not just a few of them. When this happens, all of the overhead in terms of space, which is required for an adjacency matrix, is worth it. This is because the complexity of many operations related to graphs are very low when it comes to an adjacency matrix. Especially when compared to an adjacency list and even with an adjacency set. You will see exactly what those operations are in just a little bit.

Now an adjacency list or an adjacency set would be the preferred graph representation if the graph you're working with happens to be sparsely connected. In this case, we will say that the added time complexity is worth the more efficient use of space which these graph representations have to offer. So let us now quickly summarize the complexity for different operations when using these different graph representations. So consider that the number of vertices in your graph is V and the number of edges is E. Then the space complexity for an adjacency matrix will be O(V*V). Whereas it'll be proportional to the sum of the edges and vertices in the case of an adjacency list and set.

The big benefit of an adjacency matrix though is in determining whether there is an edge present between two vertices in the graph. Given we already have the values of the two vertices, there is just a single cell which we need to look up in the adjacency matrix. So this is a O(1) operation. In the case of an adjacency list, we will have to iterate over all of the nodes in the linked list corresponding to the vertex from which the edge originates. And this operation is a little quicker in the case of an adjacency set.

[Video description begins] *For adjacency list, it is a Degree of V. For adjacency set, it is log(Degree of V).* [Video description ends]

Because we're not going through a linked list but rather through a binary search tree.

[Video description begins] *For iterate over edges on a vertex, for adjacency matrix, adjacency list, and adjacency set, the values are V, Degree of V, and Degree of V, respectively.* [Video description ends]

The Topological Sort

[Video description begins] *Topic title: The Topological Sort. The presenter is Kishan Iyer.* [Video description ends]

So far in this course, we have covered what graphs are and how they can be represented. It is now time to take a look at how we can traverse through each of the vertices in a graph. You could say that graph traversal is not very much unlike traversing through a tree. However, graph traversal includes a couple of complications. First of all, in a graph, there can be multiple paths between any two nodes, which is not really the case with a tree data structure where there is always exactly one path between the root node and a particular node within the tree. Another important factor is that a graph can have cycles.

So if you're not careful, you could end up visiting the same node multiple times during the traversal. The are specific traversal algorithms which will ensure that for a given graph, it will always print all of the vertices in a given order. So that even if there are multiple paths between the nodes, the order of the traversal will be constant. And in order to avoid getting stuck in infinite loops due to the cycles present in the graph, we need to keep track of the nodes which we have already visited previously in the traversal and make sure we don't visit them once again. So what are the different types of traversal which are available for graph?

Again, broadly speaking, as we saw with tree traversals, we have a depth-first traversal, where we start at one of the vertices and then go down one path, right up to its very end, and then visit each of the vertices along the way. Once one path has been exhausted, we go down another path until we hit a vertex which has no outgoing edges from it or if we happen to encounter an already visited node again. This process continues until all the paths within our graph have been covered. And then there is the breadth-first traversal.

So just like with a breadth-first traversal in a tree data structure, we will start off with one of the vertices in the graph and then visit all of the vertices which are adjacent to that. Then we pick one of those adjacent vertices and visit all of its neighbors, and so on. Given the broad similarities to the depth-first and breadth-first tree traversals, we won't go into too much details of these graph traversals in this course. But we can now take a high-level overview of such traversals on this particular graph.

[Video description begins] *A graph is displayed. It has five nodes: 1, 2, 3, 4, and 5. Node 1 is connected to 2, which is connected to 3 and 4. Node 3 is further connected to nodes 4 and 5. Nodes 4 and 5 are also connected.* [Video description ends]

As we have seen, there are multiple paths which are present within this graph, but let's just say we start off at node 1 and we need to make sure that this particular node is not visited again. More than node 1, though, the cycles are present in this graph, covering nodes 2, 3, 4 and also 5. And we need to make sure that we don't keep visiting those nodes and get stuck in an infinite loop. The way we can ensure this is to implement something like a Boolean array. The number of elements within this array will correspond to the number of vertices in our graph. And then there will be a corresponding Boolean value attached to those vertices.

A value of true means that the node has been seen before during the traversal and we don't need to visit it again. An alternative to the Boolean array implementation is to maintain a list of the nodes already visited. As I had mentioned previously in this course, there is a separate field known as graph theory which covers various aspects of graph traversals. Given the vast complexity of graphs and graph traversals, it is beyond the scope of this course to get into the details of its various implementations. But there is one traversal which we will explore. And this is the topological sort. So what exactly is this sort? Well, this applies specifically to directed and acyclic graph.

And when doing a graph traversal using this technique, we ensure that the ordering of the vertices in the traversal is such that a particular node will appear before any other nodes in the traversal to which it has an outgoing edge. So if there is an edge in the graph from A to B, then A will definitely precede A in the topological sort. To get a better understanding of how this works, we will use this particular graph once more as an example. So the first question which we need to ask before we begin the topological sort is, what is the indegree for each of the vertices in this graph?

[Video description begins] *A graph is displayed. It has five vertices: A, B, C, D, and E. Vertex A is connected to B, which is connected to verticesC and D. Vertex C is further connected to vertex E. Vertex D is also connected to vertices C and E.* [Video description ends]

So the indegree is a term which represents the number of incoming edges to that vertex. So the indegree for the vertex A will be 0 because there are no edges coming into it. On the other hand, for the vertex B, the indegree is 1, because we have an edge from A to B. As for the node C, there are edges coming in from both B as well as D, which is why its indegree is 2. As for the vertex D, there is an edge coming in from B, so its indegree is 1. And the indegree of E is 2, thanks to the two edges coming in from both C and D. So given this graph and the indegrees of the vertices, where exactly do we start a topological sort? Unlike a tree data structure, there is no root node over here, so we need to pick one of these vertices as our starting point.

And in the case of a topological sort, this would be the vertex which has an indegree of 0. So in our example, this is the node A. So this becomes the first element in our topological sort. What about the next element? Well, for that we assume that the vertex A is removed from this graph, and also any outgoing edges from that vertex. Given A has already been processed, that is the one which needs to be removed from this graph, where we now make B the next element in our topological sort. And what we are left with is this four-vertex graph. So let us now recalculate the indegrees for each of them. And what we are left with is that the node B has an indegree of 0, and the indegrees of the other vertices are 2, 1 and 2.

So we follow the same logic which we did previously. So with A and B processed, we also remove B from this graph, and then all of the outgoing edges from B as well, and we'll recalculate the indegree for the remaining vertices. What we are left with is this three-vertex graph with indegrees of 1, 0, and 2 for C, D and E. So the third element in our topological sort will be the vertex D. Once we remove D and its outgoing edges, we're left with this two-vortex and single-edge graph. So it is now C's turn to get added to the visited list.

And finally, once we remove C from the graph as well, we are only left with the vertex E, which has an indegree of 0. We are only left with the vertex E, which has an indegree of 0. So the topological sort of this five-element graph produces a traversal of A, B, D, C and E. Once again, keep in mind that the topological sort only applies to

directed and acyclic graphs. These are in fact the simplest graphs to traverse. And during the labs of this course, we will be implementing such a topological sort.

Exercise: Trees and Graphs

[Video description begins] *Topic title: Exercise: Trees and Graphs. The presenter is Kishan Iyer.* [Video description ends]

In this exercise, you will begin by summarizing the properties of a binary search tree. We started off this course by taking a look at what tree data structures are. And we also covered both binary trees and then binary search trees. Similarly, you will come up with a high-level description of what trees are, what makes a binary tree, and what exactly a binary search tree is. Next, you will list the three different ways in which a graph can be represented. During the course, we saw that a graph is a collection of vertices and edges which connect those vertices. Each of the vertices can be considered to be entities, whereas the edges represent relationships. In order to model such information, though, we had taken a look at three different representations of graphs. And you will now recall exactly what those are and their individual properties. Each of these topics were covered during this course.

So please, pause this video and then spend some time to do each of the tasks on your own. For the first task in the exercise, you needed to summarize the properties of a binary search tree. Now, a binary search tree is a very specific form of tree data structure. So we will start off by summarizing what a tree is. So a tree contains a number of nodes, and each of these nodes in turn can be connected to a number of other nodes. Such a data structure is very useful in order to represent information such as hierarchical relationships. The nodes within a tree data structure can be considered either children or parents or even ancestors and descendants, as we saw. Now, any tree data structure typically has a root, which is considered to be the top node in a tree. And there are also nodes in a tree which won't have any children, and these are referred to as leaves.

So now that we know what a tree is, we get to a specific form of tree, which is the binary tree. Now this is one where each of the nodes can have at most two children. So a node can be either childless, have one child or two, but not more than that. And the children in this case are referred to as the left child and the right child. And then we move along to a special variation of a binary tree, which is the binary search tree. So for any node in a binary search tree, if it has a left child, then the data stored within that left child must be either less than or equal to the value in the node itself. And the data in the right child of a node must be greater than the value within the node.

We move ahead then to the last task in the exercise, which was to summarize the three different ways in which graphs can be represented. Now a graph, as we have discussed, contains a number of vertices and edges which represent relationships between those vertices. To represent a graph, we can make use of an adjacency matrix. And this is a matrix whose size is NxN, where N represents the number of vertices in the graph. As for each of the cells within the matrix, this is one which represents an edge. So the cell corresponding to row A and column B represents an edge from node A to node B. If there is such an edge within the graph, this cell will contain a value representing the weight of the edge.

And if there is no such edge, then the cell will contain 0. An alternative to an adjacency matrix when it comes to representing graphs is an adjacency list. And this is one where we will maintain a list containing all of the vertices for the graph. This is not all, however, since each of those vertices has an associated linked list. And the linked list for a vertex contains a number of nodes which correspond to all of the vertices in the graph which are adjacent to that vertex. So each vertex has a list of adjacent vertices which are represented as a linked list.

And this is what makes up the adjacency list. Which moves us along now to an adjacency set, which is in fact very similar to an adjacency list, in that you have a list of vertices for the graph. However, the collection of adjacent nodes for each vertex is stored, not as a linked list, but as a set. In many programming languages, sets are implemented internally in the form of binary search trees, which is why many operations on adjacency sets are more efficient than those on adjacency lists.