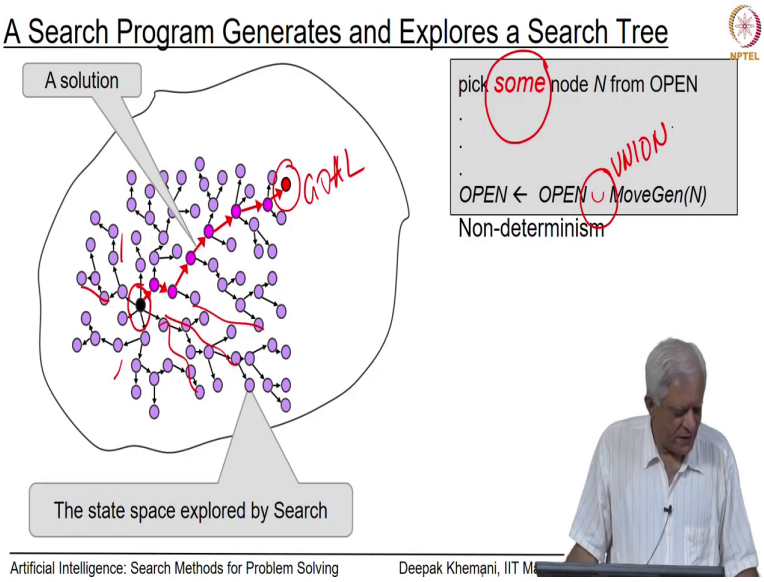
Artificial Intelligence: Search Methods for Problem Solving
Prof. Deepak Khemani
Department of Computer Science & Engineering
Indian Institute of Technology, Madras

Lecture – 19
Deterministic Search

(Refer Slide Time: 00:14)



So, welcome back. So, far we have seen that we have built a mechanism for doing search in some state space. And the mechanism is essentially to generate the neighbors, and pick one of them and test whether it is a goal or not. And if it is not the goal, you add its own neighbors to the set of candidates. And in this process the search algorithm explores the space essentially.

Now, this figure depicts what the search algorithm has done at the point when it has reach the goal. So, if this is a start node, it has explored different parts of the search space which are

showed in these nodes in purple all over the place. And at some point it has found a path to the goal node, and it has been able to reconstruct the path as we can see here.

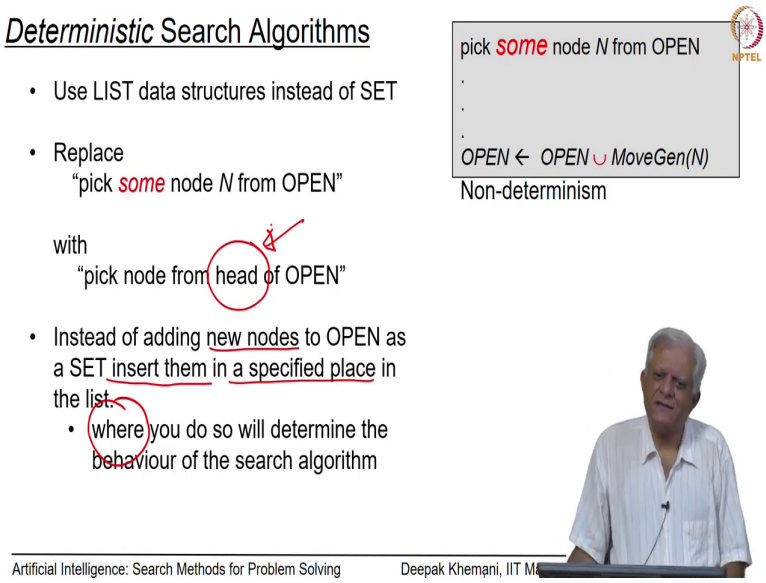
Now, this mechanism for exploring the space is a function called movegen function or a neighborhood function which we expect that the user will provide to us. And the mechanism for termination is also a function called goal test function which again we expect the user to tell us when we have reach the goal. And our algorithm essentially picks some node from the set of candidates and test whether it is a goal; and if not, it adds new nodes essentially.

So, the key features of our mechanism were that we had not specified which node to pick from the set of candidates, we had simply said pick some node from open. And later on when we wanted to add new nodes to open, we also did not specify where the new node should be open, add it to the open, and we had simply use the union operator which is available on sets essentially.

So, this was a basically a non-deterministic non-deterministic approach to solving the problem is somehow left it to some other mechanism may be an oracle or something to decide which node to pick from open and then where to add new nodes into open.

So, let us now move forward since we want to actually implement computer programs and it is not so straight forward to implement non-determinism. And as you probably know non-determinism is implemented by resorting to search. So, you may make a guess, but if the guess turns out to be wrong, then you have to go back and do something else essentially.

(Refer Slide Time: 02:59)



So, let us implement a deterministic algorithm to do search. And in this deterministic algorithm or a sequence of algorithm that we will see, we will replace this non-deterministic choices with deterministic choices ok. So, the first thing we will do is that instead of a SET, we will use a LIST, because a list is easier to implement. And in fact, very often you might implement sets also as list though there may be other ways of doing that. And it is easier to operate on LIST. So, we will assume that we are working with LIST.

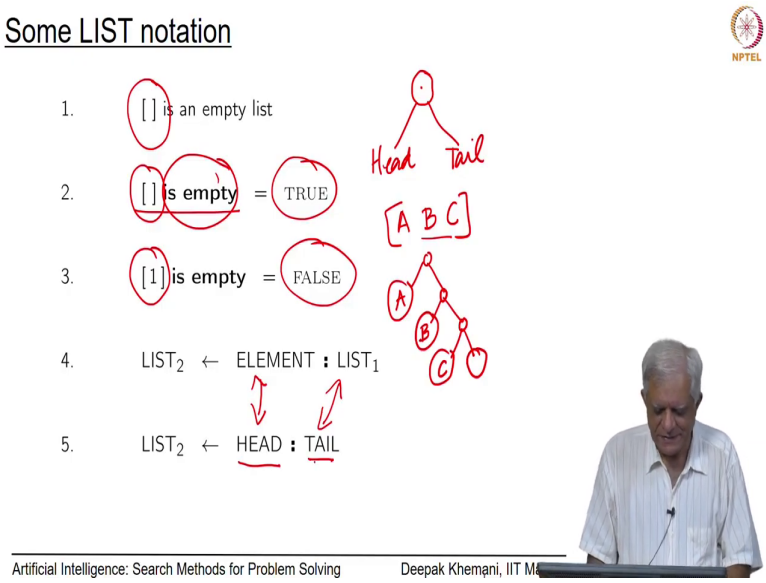
Now, since we are working with LIST, we will replace the statement, pick some node from N, some node N from OPEN. With pick the node which is at the head of OPEN. Now, we know that in a LIST structure a head element is very easy to access, and this will make our life a little bit simpler essentially.

The second part which was said that you simply treat OPEN as the SET and add the new nodes to open, we will now since we are treating it as a LIST, we will specify where to add the new nodes essentially. Insert the new nodes into a well-defined location. And based on where we specify the behavior of the search algorithm will be influenced by that.

In fact, in the next few classes we will experiment with this where here, and see as to what is the impact of putting those new nodes in different places in the OPEN LIST, because we have decided that we will always pick the next node to be inspected from the head of OPEN. So, the only control we have left is how do we maintain open, and we will see various ways of doing this. Initially we will look at some blind approaches which are oblivious of what the goal is that you are trying to achieve.

Now, obviously, an intelligent agent must try to work towards a goal, but since we are simply building a underlying mechanism which is a search mechanism, we can at least smaller problems resort to some brute force like search algorithms and that is what we will start with essentially. Before we do that let us look at some notation that we will use in writing these algorithms.

(Refer Slide Time: 05:26)



And this notation is to do with handling list, because now we have decided that open list will be a list essentially. So, a list will be denoted by two square brackets. And this particular first example here is it denotes an empty list, it is a list containing nothing.

If you want to test whether a list is empty or not, we can write a statement that a list is empty; and if it is indeed empty, then that is answer would be true and the program will return TRUE to us essentially. Remember that we are looking at a somewhat functional kind of approached programming where everything is a function and it return something to us.

And in this case, it is a predicate. So, a predicate is just a function which returns true or false. And if we give an empty list as an argument to this phrase is empty, then answer would be true. But on the other hand if we return if we input non-empty list, for example, this list

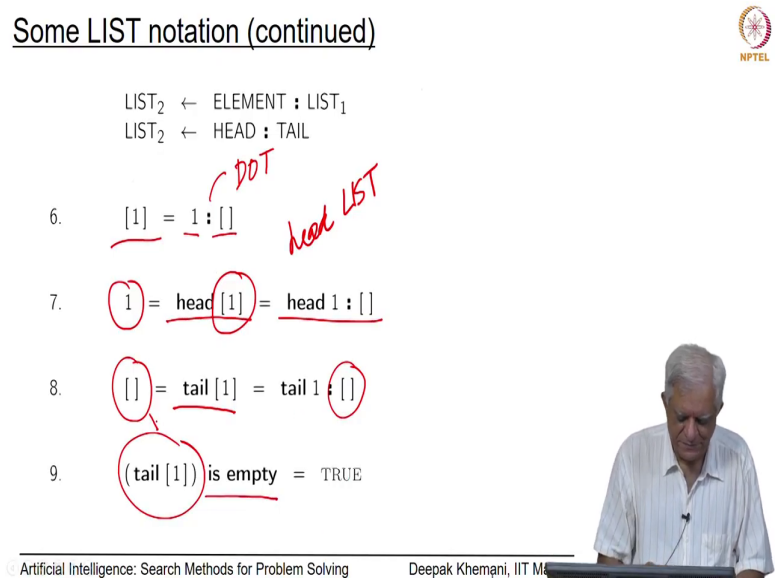
containing the element 1, and ask whether it is empty, then the answer would be FALSE essentially.

Now, how do we build list and manipulate list? We can define list as made up of an element which is the head of the list as you can see in both these statements, and the rest of the list which is a tail of the list. So, you must have studied in a data structures course or somewhere that this lists are often represent represented at dot products or cons structures and so on.

So, that what you have is an element here which is called the dot, and the left child of that is the HEAD, and the right child of that is the TAIL. So, a list is internally represented as the binary tree where the first child of the left child is a head, and the right child is a tail.

So, if I had a small list like for example, A B C, I would represent it as a dot product followed by an element A here, followed by another dot product because what I have is a tail which is a non-empty list this would have the element B, followed by another dot product where the first element is C and in the end I have an empty list or, nil list. So, the list ABC is internally represented as a binary tree. And in our notation we will denoted by saying that it is made up of a head and a TAIL, where the tail is a remaining part of the list essentially.

(Refer Slide Time: 08:21)



So, we have seen this that a list is made up of a head and a tail. Let us look at some examples here. So, here is a list containing only one element 1, and we can also represent it by saying that it is made up of the element 1, and the tail is a empty list here essentially. So, this represents a dot operator that we had talked about, which separates the head from the tail essentially.

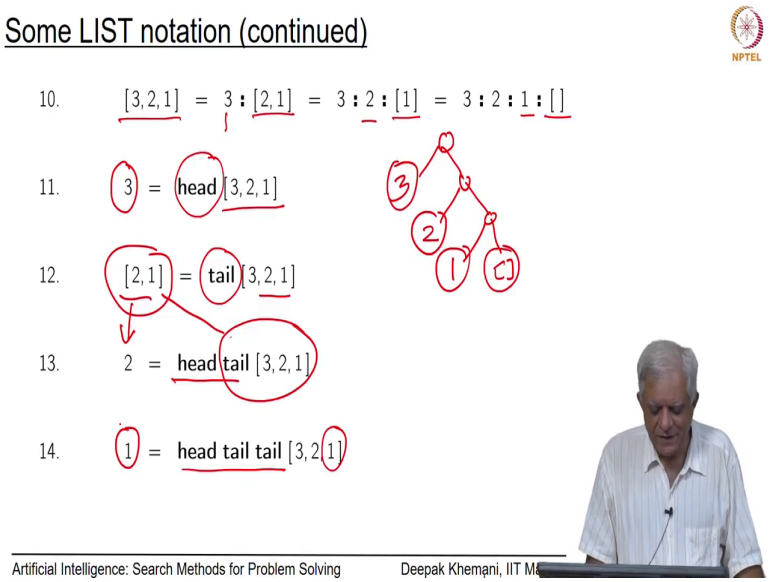
If we were to use the expression head of 1, then what it would return is 1 which is the head of the list. And this is equivalent to writing head of 1 and followed by an empty list essentially. So, a head of a list may be obtained by using head followed by name of the list.

In this case, the list is not named we have said it explicitly as a within square brackets as the element containing 1, but it can be a name of a list which is what typically you would do in a program. We can likewise get the tail of a list by using the keyword tail. So, we give tail as an

example we have already seen that this list containing only 1 has an empty tail. So, it is not a surprise that this tail of 1 returns and element which is the empty list essentially.

Now, we have seen that we had this test called is empty. And we can apply this test is empty to tail of 1, where 1 is this list containing only 1. And we will see that it indeed it is an empty list essentially that we have seen already here essentially.

(Refer Slide Time: 09:58)



So, let us look at a list containing three elements – the elements are 3, 2, 1 here. We can visualize this as 3 being the head, and 2, 1 being the tail. Now, since 2, 1 is a tail, it can itself we visualize this 2 being the head, and a list containing 1 being the tail. And since the list containing 1 is also a list, we can visualize this as containing an element 1 followed by an empty list.

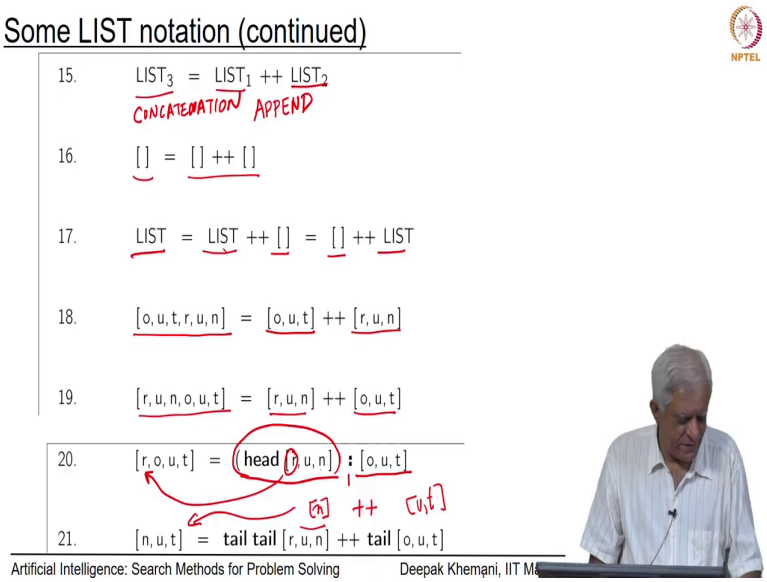
This is consistent with the diagram that I had drawn earlier instead of A B C, now we have 3. So, we have 3 at the head of the list we have another list here which contains 2, at its head then we have a third list which contains 1 as an element and an empty list as a tail essentially.

So, internally as we have seen lists are binary tree essentially given a list like 3, 2, 1, if we give the keyword head what we get is the element at the head of the list. If we use the element tail, if we use a keyword tail what we get is the remaining list which is the tail of the list. So, we can access different parts of the list, using these head tail keywords, we can even use something like head of tail essentially.

So, head of tail applies sequentially. So, first we look at tail of, so the argument to head is this element called tail of 3, 2, 1. What is tail of 3, 2, 1? Tail of 3, 2, 1 we have already seen is this element 2, 1. And when we take the head of this element we get the first element which is 2 essentially.

So, we can use combinations of these operators, we can say head of tail of tail of tail which will take us to the last element of the list and return the last element of the list. So, we have powerful ways of handling list, and we shall keep this in mind when we look at algorithms in the rest of the course we will often use list for representing things.

(Refer Slide Time: 11:55)



Now, some of the operators which are commonly used for list is the concatenation or the append operator. So, you can think of this as append if you might have you might have written at some point written a program called append to list. So, what this does is that it takes two lists list 1 and list 2, and appends them to give you a list 3, and we can also think of this as a concatenation operator both mean the same thing essentially.

So, let us look at some examples here. If you append an empty list to an empty list, you get an empty list which is fine. If you take any list and append it with a empty list, this is equivalent to taking an empty list and appending into that list. And it remains the same as the list, because appending an empty list does not add anything to the list.

If you take a list called out and append it to a list called run you will get a new list which is called out run. Well, I am using it as a English word, but essentially these are 6 alphabets of the character o, u, t in one list, and r, u, n in another list. And the concatenated list or the

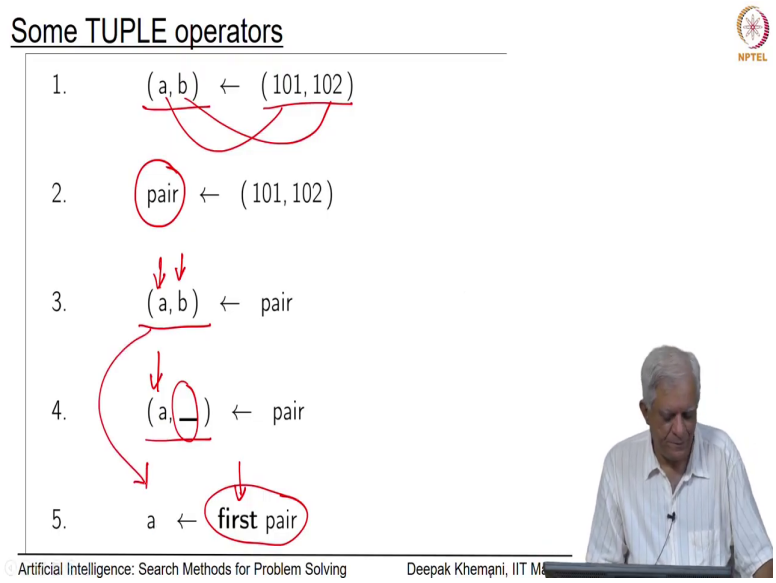
appended list has o, u, t, followed by r, u, n essentially. Likewise if you have r, u, n, and o, u, t, you will get appended a list containing all the 6 elements essentially.

Now, let us look at this slightly more involved example. You are talking of a list in which the head is the complex expression and the tail is a list containing o, u, t. Remember that this is a cons operator that we had talked about or the dot operator. The head is at the this whole head is the head of a, this whole, whole expression is the head of this list which is o, u, t.

So, whatever this returns or whatever this evaluates to will be inserted at the head of the list that will be composed by o, u, t. What does this expression evaluate to it evaluates head of this expression which is r, and then you can see that r is what is inserted at the head of the list.

So, the outcome of this expression is r, o, u, t, which is taken by taking the first element of this list r, u, n, and adding it to the head of o, u, t, essentially. Another example where you have tail of tail of run. So, tail of run is u, n; tail of tail of run is the list containing n; tail of out is a list containing u, t. And when you append these two together, you can get the list which contains n, u, t essentially.

(Refer Slide Time: 15:08)



So, we have interesting ways of operating upon list essentially. We also have a notion of a tuple. A tuple is a it is like a list except that when you define tuples, they all of fixed length list can be of arbitrary length. You can make a list of 20 elements, 30 elements, 4 50 elements, it does not matter. It is a data structure which allows you to use an arbitrary number of elements, but the tuple is like a list except where it has a fixed number of elements essentially.

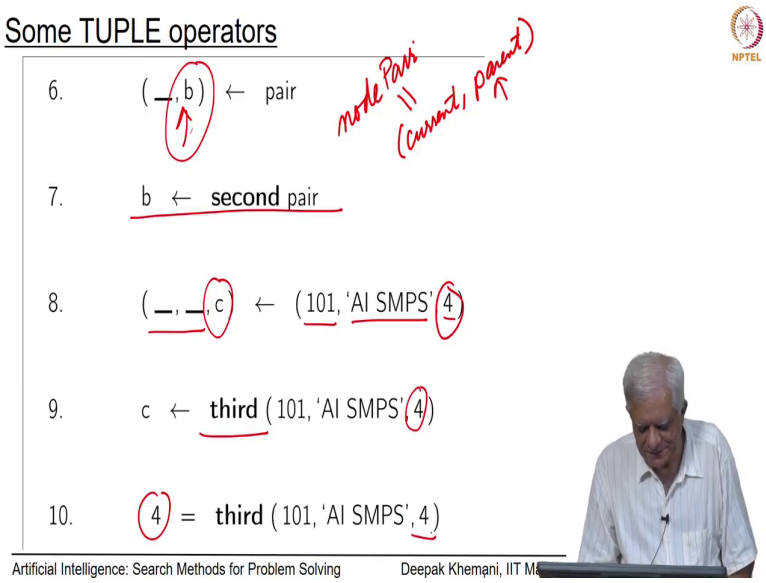
So, here we see that it is a pair made up of two elements and we can identify it with a list containing with a pair containing two elements, where a identifies with the first element, and b identifies with the second element. We can also give it a name called pair, and then we will know that it is a pair. I mean the name does not mean anything, but we can assign a variable to this list in this case we have call it a pair essentially.

So, we can if somebody if some program returns a pair to us, we can access the two elements by using variable names a and b here. And we will know that the first element we will call a, and the second element we will call b.

And in fact, we do not even have to give a name to the second element we can just use an underscore which many programming languages allow is that if I only interested in the first element of a which we will often be, because we are often interested in the first element of remember we talked about the node pair in the previous class where a node pair was a pair made up of a current node and a parent node.

And if you are only interested in the current node you can simply referred to as by an expression where you do not give a name to the second element at all essentially. We could also extract this first element by using an operator called first. So, when we say first of something, it just gives us the first of that element essentially. So, the if the pair is a, b, first of the pair is a essentially. So, as you can see this is the first element here.

(Refer Slide Time: 17:19)



Some more tuple operators. We can access the likewise the second element of a pair. So, remember that we had said that a node pair in our search algorithm would be made up of the current node followed by the parent node.

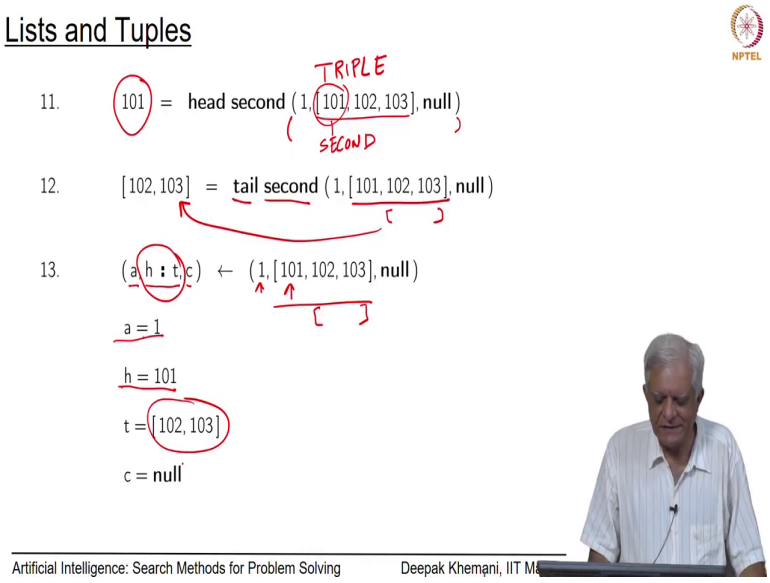
And remember that we had said that if you want to reconstruct the path because these node pairs are stored in the closed list, we need to access the parent node and then the parent of the parent node and the parent of the parent node and so on, and that can be done simply by as you can see here accessing the second element of the pair essentially. So, you can refer to it directly or if you please we can write this expression second pair which will give us the second pair essentially.

So, tuples do not have to be pairs they can be triples as well. So, for example, if you have a triple which is let say a course number and a course name called AI, SMPS search method. So,

problem solving that we are doing here, and this may be the set of credits which is the credit set this course has in IIT Madras we can access any of these elements.

So, if we just want the third element here, we just refer to the third element and we do not name the other two elements essentially. We could have got as before the third element by an operator called third which is again gives us the same value essentially. We can even write this equality and say that four is a third element of this list here.

(Refer Slide Time: 19:03)

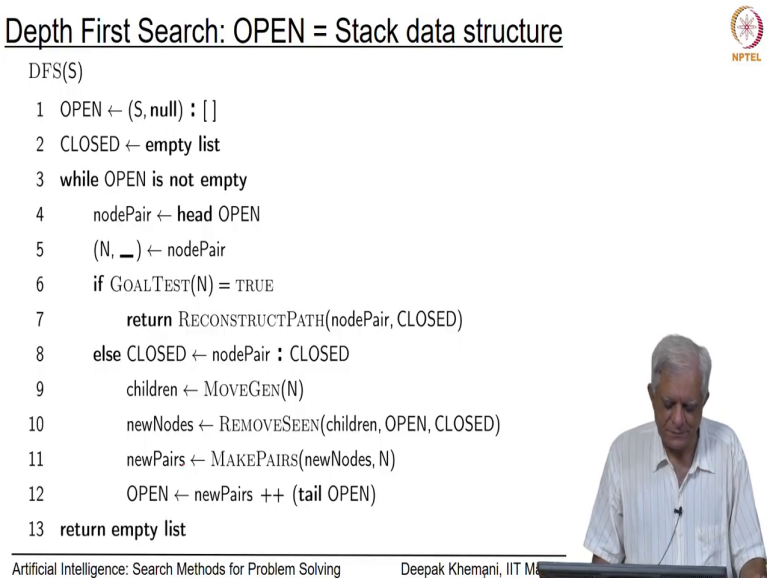


So, let us look at one more example here. So, what do we have here we have a triple, because it has three elements inside this in this pair of brackets which represents the triple. The first element is just an element. The second element is a list containing three elements, and the third element is the constant called null.

And we can see that this element 1 0 1 is the head of the second of the triple. The second of the triple is this whole thing, and the head of this is the element 1 0 1. As you can see essentially if you write tail of second, then again second of this thing is one thing and the tail is this much and which is what we get here essentially. We can access the similar thing by an expression a followed by a list followed by a constant.

So, the same thing, here a will be one which is the first element of the triple, h is 1 o 1 which is the head of this list which is this list that we are talking about; t is 1 0 2, 1 0 3 which is the tail of this list as we can see; and c corresponds to null ok. So, now that we are equipped with this mechanism for dealing with list and we have decided to write deterministic search algorithms. Let us move on and write our first algorithm, and then we will start looking at the details a little bit later essentially.

(Refer Slide Time: 20:52)



So, this is a algorithm that we are going to study first it is called depth first search, and the key property of this is that it treats open as a stack data structure ok. So, we will do that in the next video which is going to focus on a couple of algorithms where we look at the variations and it will give us two algorithms to start with one is called depth first search and the other is called breadth first search.