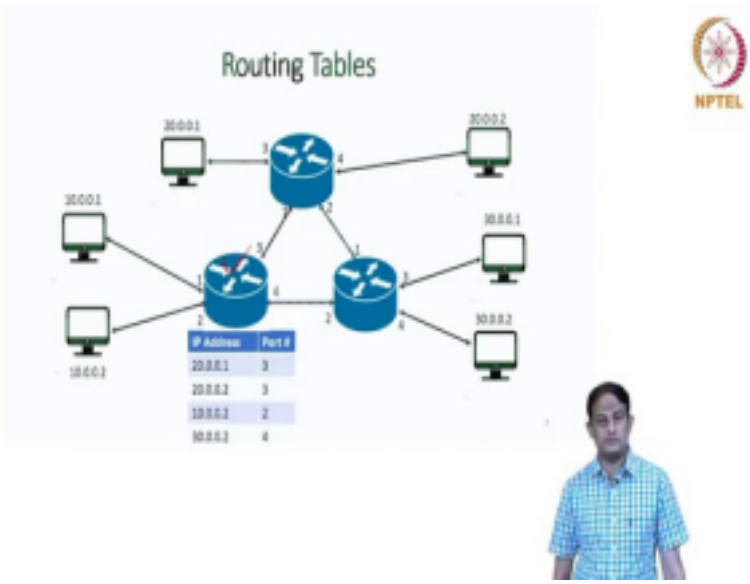
Advanced Computer Network
Professor Dr. Neminath Hubballi
Department of Computer Science and Engineering
Indian Institute of Technology, Indore

Lecture 5
IP Table lookup: Trie-based data structures, Part-I

(Refer Slide Time: 00:18)

Welcome back! In the last class, we started our discussion on the IP address lookup and what all we can do to do the fast IP address lookup, given that the network routers have to function very fastly and need to handle the packets with several gigabits of speed.

(Refer Slide Time: 00:42)



So, this is the diagram that we saw in the last class, there are three routers and some computers connected to each other, and every router, in fact, has got a routing table, here router number 2 has got this routing table. So, which has got the destination IP address and the corresponding port number on which the packets are to be forwarded.

So, if this router receives a packet, let us say this router receives a packet with the destination IP address 20.0.0.1, it means that it needs to forward it to a packet to port number 3. Now, we saw one data structure in the last class which is the binary trie, one example we saw how to construct the binary tree, given the IP address prefixes.

(Refer Slide Time: 01:33)

So, let us now continue the discussion, the agenda for today is we will try to look at the variants of the Binary Trie data structures. First, we will again take an example of the Binary Trie and then look at what are the performance issues? what we can do for probably to improve the performance further? and then look at the different variants of those data structures.

So, here is a set of prefixes. There are nine of them on the screen, P_1 to P_9, and the prefix binary digits are actually shown. So, let us try to construct a Binary Trie for this set of prefixes. So, as usual, P_1 is a * then I need to start with the root node. And then that is my prefix P_1 and P_2 is saying 1*, I go to the right and have a node here this is my P_2, and P3 saying 00*. So, I need to come to the left and have two nodes. This is my P_3, and P_4 is 101. So 1, I already have this and 00, and this P_4 is 101, 1 is already there, and then I go to the left to have a node on 0 and then 1*.

So, I go to the right this is my P_4 and P_5 is 111; 1 is already taken care of after that we need 1 once and then followed by a third 1, this is my P_5, and P_6 is 1000, 1 followed by one 0 is already there and a second 0 there is a node here, and then on the third 0, there is a node here.

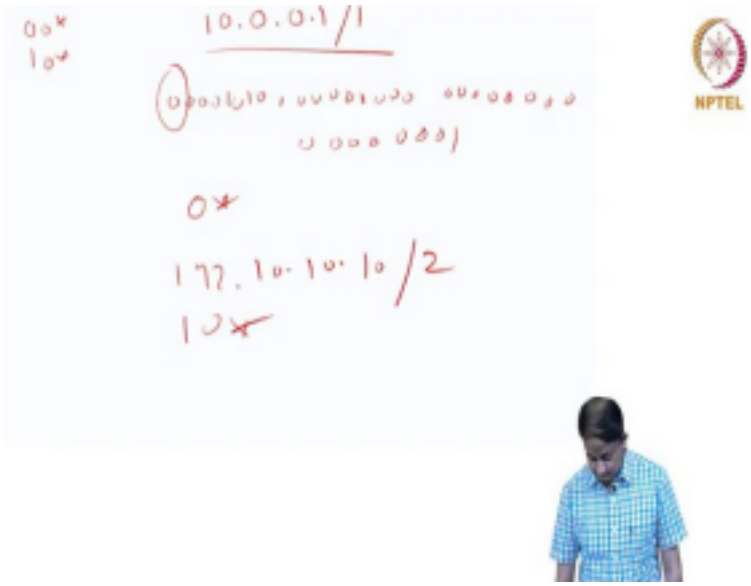
So, this is on 0, and this is my P_6 and P_7 is 11101, triple 1 is already there followed by 0, a node here, and then 1. So, I come here, this is my P_6 and P_8 is triple 1001, this 10 is already there, one node and then followed by 1, you are here this is my P_8, P_9 is 1000011*.

So, 1 followed by three 0 is already there one more time 0, this is a node, and then a 1 and then a

second 1, this is my P_9. And I need to mark the nodes which are corresponding to prefixes. That I will do that by using the blue color here. And this is my P_4 similarly P_5, P_7 and then P_8, P_9 and then P_6. So, that is the Binary Trie for this set of prefixes, given these nine prefixes, this is how you construct the Binary Trie.

So, a little bit about the ones. So, how many times it depends on the height of the Trie can be depending on the length of the prefix. So, here the length is: P_9 has got this, 1000011. So, that will correspond to the longest length prefix in this Trie structure. So, you require those many numbers of pointer access in this Trie to find out what is the next hop router when you make the routing decisions.

(Refer Slide Time: 05:50)



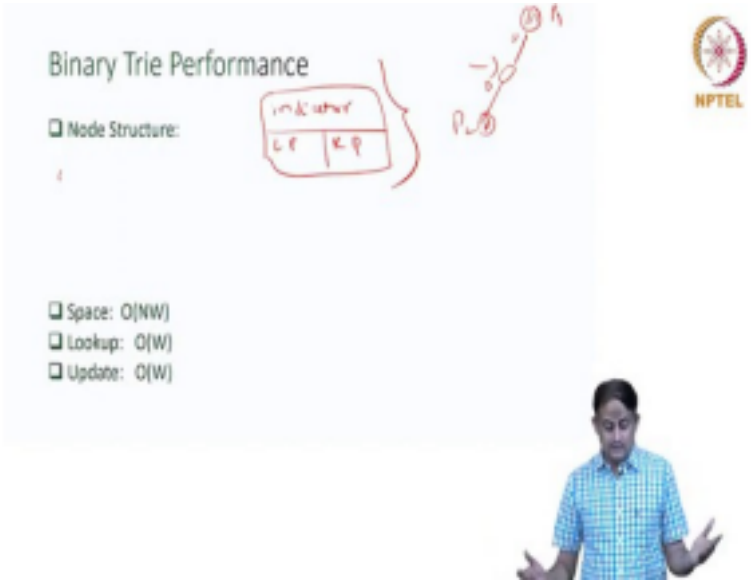
So, the prefix examples that we saw in the previous case, whatever the examples we talk are all in this format, maybe 00*, 10*, something like this, or * like that. But in reality, the prefixes might be expressed in a different way. So, maybe, for example, if you take the CIDR case, the prefix are in this format, let us say I have the 10.0.0.1/1, what it means is, you convert this thing into binary 10.0.0.1. So, ten if you convert it to binary, you will have 00001010. And then second 00000000 and followed 00000000 and followed by 00000001.

So, you take the first bit, only the network portion is actually only the one bit because /1 is there, So, the prefix corresponding to this expression is 0 followed by irrespective of whatever is there, I do not care. So, 0* is the prefix corresponding to this CIDR notation. So, similarly, I might have something else as well. So, let us say I have 192.10.10.10/2, what it means is again, you convert 192 into binary ten into binary, concatenate all the binary numbers and pick the initial two bits of this binary number which will be the prefix. So, let us say for 192, it turns out to be 10. So, 10* would be the prefix corresponding to this number.

So, this is not the exact 10, just the example that I am giving. So, given the CIDR notation, that is how you can convert that into binary prefixes of this format, and then subsequently, you can construct the Binary Trie for that to prefix.

So, there might be some examples corresponding to this in the tutorial and later exercise session, you can take a look at that.

(Refer Slide Time: 08:16)



So, now let us take a deep look at the performance of the Binary Trie, how does actually it, what is the space complexity? What is the number of the lookups I need to do if I have to do the update operation? how much effort do I need to put in ? So, in asymptotic format, we will take a look at that. And before that, given this, how the Binary Trie is constructed, what is the structure of the node what is the data structure, the actual physical data structure that I am going to build in terms of the software implementation.

So, this is a simple Binary Trie is a simple tree, where you can have two nodes and two children and then the node value. I can have a linked list kind of the data structure used for this one. The node constituents can be the following: one is the indication of whether this particular node is a kind of representing a prefix or it is not representing a prefix.

For example, if I have this one, this is my *, this is the prefix P_1 and then I might have 0 followed by 0 this is 00 and this node is actually representing prefix P_2, but the middle node is not representing any of the prefixes.

I can have an indicator variable to indicate whether this is representing a prefix or not. Each node can have two children. We will have two pointers, one is the left pointer and the second one is the right pointer. Every node can have this kind of structure and you need to create those many numbers of structures as there are the number of nodes in the Binary Trie.

So, if there are n number of the nodes in the Binary Trie, so, n times whatever the space this is occupying, that would be the space required for the entire Binary Trie and the space requirement is n times one of the given W, W is the width referring to the prefix which is of the longest length.

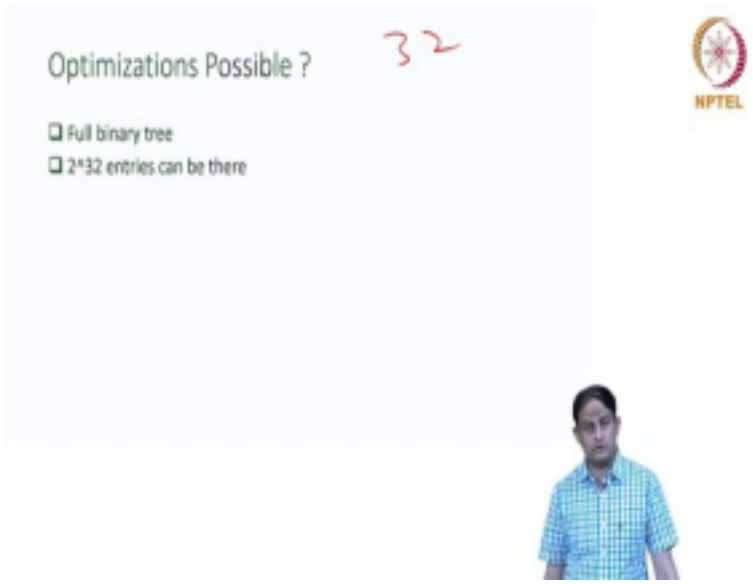
So, remember the prefixes are ordered in increasing format, so, here the P_9 has got the length of 7 and n here is the 9, 9 times the whatever the width, here it is 7. So, 7 times the 9 is the total space required for the entire Binary Trie. Since this is the asymptotic notation, the maximum that we have is the one that we take. So, that is bound and that is the space requirement if I have to do the lookup operation.

Given that if we are given a binary trie that data structure, and then in real-time when the router receives a packet, you pick up the destination IP address from the header of the packet and then against the data structure that you have got whatever Binary Trie you have built, you will go into that, evaluate that, what is the best match for this particular destination IP address.

So, that would take those many numbers of the lookup operations what is the width of the or the longest prefix that I have in the prefix table and if I had to do the update, so, let us say today I have got these nine prefixes, but as the more number of the computers get connected to the network, the number of the prefixes in the table would increase and if you have to do an update operation, then what is the cost of that, that would again take the order of the W, W is the width of the longest prefix.

We discuss routers have to do many numbers of the packet forwarding decisions, So, millions of packets per second and for forwarding one packet, you need to do these many numbers of the memory references, how many, what is the width of the longest length prefix in my table. So, what could be the longest length that I can have in the binary trie.

(Refer Slide Time: 13:01)

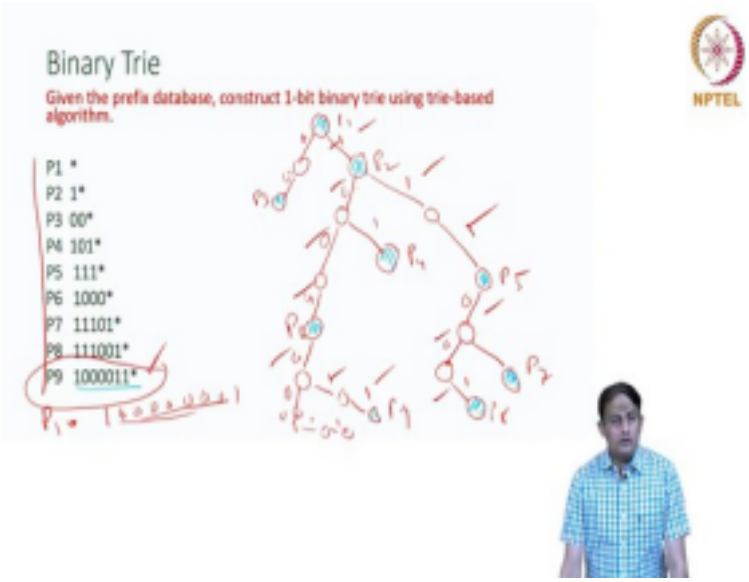


If you take the case of the, IP version 4, there is possibility you can have the longest width as 32, the height of the tree would be 32 and then in the worst case, for forwarding one packet, I need to do 32 number of memory references those pointer references from one node to another node, either you take the left child or the right child and then keep traversing that, those many numbers of the memory references.

So, in the worst case 32 references and the second thing is what could be the size, total size of this Binary Trie. In the worst case, it can be a full binary tree. So, it can have the 2^{32} all possible IP addresses might be listed on the Binary Trie, those many number of prefixes particularly, you do not have any room for every possible IP address there in your network. So, although that is theoretical, when you talk about the data structure, that is the possibility that we have got.

Now the question that we ask is instead of doing 32 number of the pointer references or traversing the Trie, that is the worst case, can we do a little better in terms of the memory references? So, remember, the memory references traversing the pointers, going there, would take time, can I do some optimization?

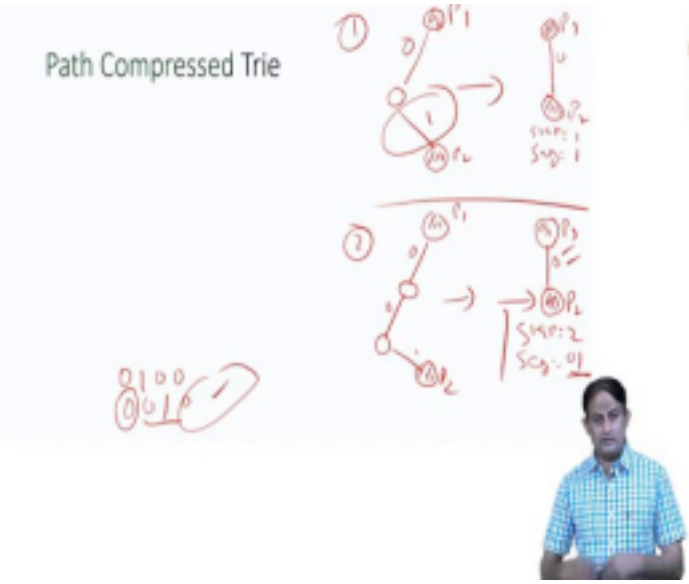
(Refer Slide Time: 15:05)



So, let us go back to the previous example here that we have, what we notice is some of the internal nodes. So, for example, if I take the case of this left subtree, which is here, from P_1 to P_3 from the root node to this node, there are two bits one 0 followed by 0, the intermediate node that is here, it is not representing any of the prefixes.

So, can I get rid of this intermediate node so, that I can bring reduce the height of the tree here, in this case, if I do only this compression, the height is not reduced. But if I apply the same rule across all paths that exist in this binary tree, then you possibly reduce the height of the tree.

(Refer Slide Time: 15:58)



Let us try to understand one variant of such automation that is possible, called the Path compressed trie. So, what I am trying to do here is the path that exists from the root node to any of the leaf nodes, if any of the intermediate nodes have got a single child, and it is not representing any of the prefixes then I am going to get rid of that.

Here is a simple example. So, I start the node here, 01 and this is my P_1 and this is my P_2, what I want to do is convert this into the trie that has got the height of 2. So, I want to convert this into this format.

This is my P_1, and then 0 followed by 1, then make this note as the prefix P_2 and then I need to remember this entire thing is actually skipped. How many numbers of bits I have skipped? And the second thing is, what are the bits that are skipped?

That is going to be indicated with the two variables inside this, one is called the skip length. So, I am going to write it as one because I skipped one bit. And what is the segment that is skipped in this case. So, it might be the left child, or the right child here in this case is the right child. So, I am going to write to remember that the segment that I skipped is 1. So, it is not necessary that only one bit is skipped, it might be more than that.

So, for example, if I have a Trie, which is in this format, 01 and then a 0, and then a 1. So, this is my P_2, this is example number 1, this is the example number 2, and when I do this conversion,

what I want to do is to get this, so, this is my prefix P_1, and then I retain this node and make this node as representing P_2. And here is this case again with the variable value the skip length here is the two I skipped two bits, and then the segment that is skipped is, 0 followed by 1, so, I am going to remember this.

So, how does this actually help? When I am trying to mask the prefixes given the packet arriving at a router, I pick up the destination IP address from that header and then try to match with this slide. So, what is my prefix if let us say this destination IP address is four bits, it is 0100?

So, what I am going to do now is pick up this trie, so, 0 is matched I come here and then 1 is there, but the segment is saying 0 followed by 1 that does not match. The best previous prefix that is matched is the P_1 that is the default route where you forwarded it. On the other hand, if you have a 0010, this is the destination IP address.

So, I pick up the first bit here and then match here, you come to node P_2 and it says that the skip length is two and the segment that is skipped is 01, matched 0 followed by 1 at the node P_2, and that matches. And so, this is the node which is actually indicating the prefix because we have highlighted so, irrespective of what is there in the third, fourth or these fields so, I am going to say that so, far, whatever the best match that I got is the P_2.

That is how actually it is evaluated. In effect, what I am trying to do is that the expense of storing some more content in the node, I am trying to minimize the number of memory references the pointer references that I have got in the table.

So, this is possible if you have a processor inside the router which is capable of doing a multiple number of bit comparisons in one go. So, let us say my processor has got an instruction. When I say the processor, it is the processor sitting on the router, because you need to do some computation, you need to install this routing table on that router. And then I am building a data structure, picking up the destination IP address from the packet, and then doing the comparison the hardware that exists on the router is capable of doing this. And if that hardware is able to do multiple bit comparisons, then I can squeeze this and build a compressed data structure that has got a reduced height comparable to the binary trie that we discussed.

So, let us take some of the examples and then see what is the advantage that we get by doing this how many number of the memory references we can reduce whether the asymptotically is it going to offer any benefits or not, we will try to find out in a minute.

(Refer Slide Time: 21:34)

Let us say I got these prefixes, let us say P_1 is the prefix which is a * and P_2 is the prefix which is getting 00* and P_3 is a prefix which is representing 11* and P_4 is a prefix which is representing 11110* and P_5 is the prefix which is referring to 11100*, I have got five prefixes, and then let me try to construct the path compressed trie for the set of the prefixes. So, before we do that, let us construct the traditional binary trie and then look at the height of the binary trie and compare it to the height of the path compressed tree then we will be able to proceed to what advantage that it is bringing.

As usual, I start with the root node and then that is my prefix P_1 and then it says I got the 2 zeros, I come to the left this is my P_2 and P_3 is 11* then I go to the right which is 11 followed by the second 1, and then P_4 is 11110, two times one is already there, and the third one and the fourth one and then 0.

This is my P_4 and P_5 with 11100, 111 is here, and then I take a branch here go to this node 100,

this is my P_5, so, let me change the color of this or highlight the prefixes.

So, this is the prefix P_1, prefix P_2, prefix P_3, prefix P_4, and prefix P_5. So, let me again construct the binary trie and the Path compressed trie for this, this set of the prefixes looks something like this. So, I take this as the input. As usual, I start with the root node, which is my P_1 the default prefix, and then any node which is intermediate and has a single child that would be eliminated.

If you have multiple of them in sequence, then you eliminate all of them in one go. And remember what is the skip length and the segment that you have skipped. In this case, if I go to the left side, I am going to retain this 0, and then make this as my P_2, and I am going to say that the skip length here is one and the segment that is skipped is a 0 and come to the right, on this particular path this node which is intermediate between P_1 and P_3 can be eliminated.

The same logic I retain in this one and then say mark this as my P_3 and what I skipped here is of length one, and the segment that is skipped is 1. And similarly, the next node has got two children this node nothing we can do about it; just keep it as it is. But on the path from this node to the P_5, I have got an intermediate node this can be removed. Similarly, this can be removed. So, I am going to retain the edge and then make this as P_5, and the skip length is one and the segment that I skipped is 0.

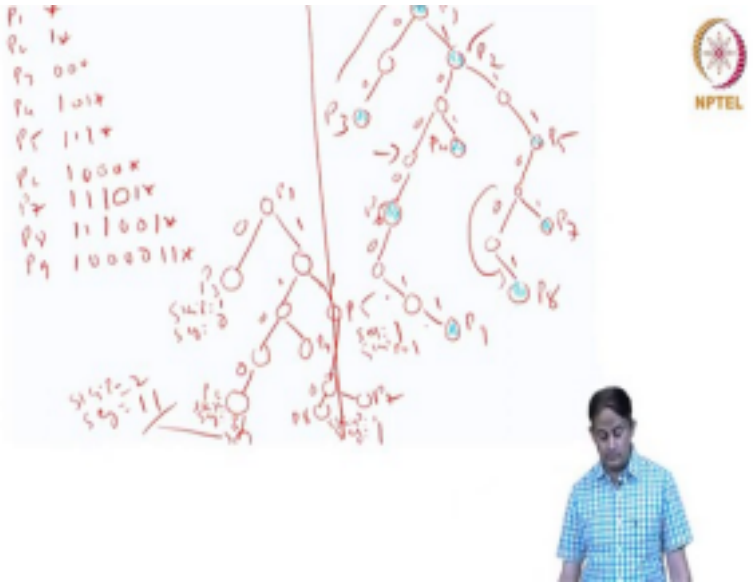
And similarly, I am going to retain this level, and make this my P_4, and the skip length is one and the segment that I skipped is a 0 segment. And then again, I need to highlight these nodes representing whether they are representing the prefixes, this is representing a prefix, this is representing a prefix, this one and this one, and then this one.

So, you can now compare, five is the height of this binary trie and path compressed trie has got height of the three. So, two pointer references we got rid of and then that reduced the height of the tree and then you are able to do the comparison with the minimal cost, minimal in the sense that is the best that we can do, at least to using this structure, you will be able to do the decision where to forward that packet on which port number.

So, that is how you construct you optimize by having something in the nodes extra things by

remembering what I skipped and how many numbers of the bits I have skipped these two quantities, then I am able to reduce the height of the trie and thereby the number of the memory references, remember you need to concatenate what is the label on this path between the node and the next child, and then what is this segment that it is indicating then you do the comparison in one go.

(Refer Slide Time: 27:37)



So, let us try to take one more example and then construct the path-compressed trie before we actually look at the asymptotic performance of this optimized tree structure. So, here is so, my P_1 is, as usual, the * P_2 is 1,* and P_3 is 00*, P_4 is 101*, P_5 is 111* P_6 is 1000*, P_7 is 11101* and P_8 is 111001,* and P_9 is 1000011*.

So, as usual, construct the compressed binary trie for this example. So, as usual, we will first try to do the binary trie operation. So, this is my P_1 and 1* this is my P_2 and 00 *, that is 1 on the left, and on the left, this is my P_3, and P_4 is 101. So, 1 already we have and the 0 you come here, and then in 1 you go to the right that is your P_4, P_5 is 111. So, on the right and 1 on the right. This is my P_5 and P_6 is 1000010 is there, particular child. And then the one this is my P_6. P_7 is 11101. So, 0 and then 1.

This is my P_7, P8 is 111001 and P_9 is 1000011.

So, seven is the height of the trie and I need to as usual mark the nodes which are representing the prefix with the blue color, let me do that and this node, this node this one, and again let us try to construct the path compressed trie for this one, this is one let me do it here. And usually, if I take this path, then I can get rid of one unit node this is my P_3, this is my P_1 and the skip length is 1 and the segment that is skipped is a 0 and on the right-hand side, we have a 1.

So, this node has got the two children, I cannot do anything about it, just to retain them as it is 1 on 0 and then 1 and take this; this has also got the two children 1 on 0 and another 1 on 1. So, retain that as it is, this is P_4. So, P_6 can be this node that can be got rid of, and then the P_6 can move upwards. So, this is 0, and then this node is the P_6.

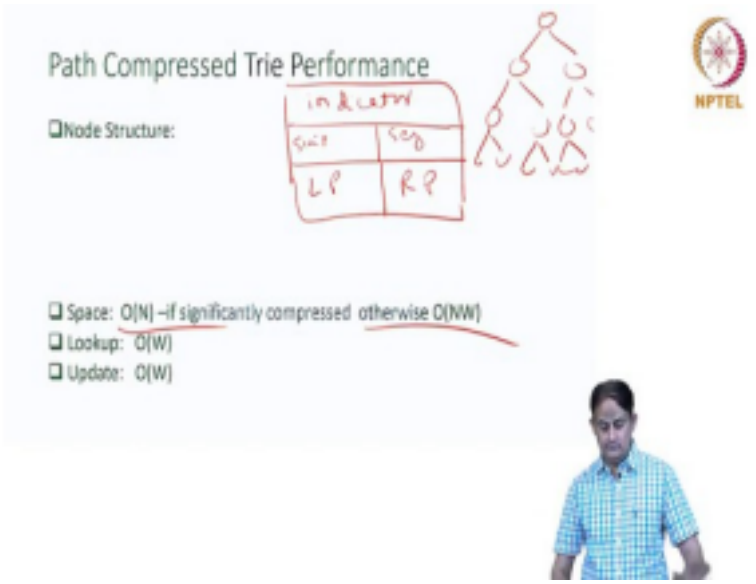
So, what I skipped 10 that I am going to remember skip length is equal to one and the segment that is skipped is 0 and then, followed by P_9; we then have a 0 here and a node here; this is my P_9. And I am going to write the skip length two and what I skipped which is 11.

So, 1000011, which is there that is your P_9, and similarly, this is done. So, on the right-hand side, the P_5 can move what this can become P_5, and then the segment that is skipped is 1, and the skip value is also 1, and this is your P_5, and then on the left-hand side, you know this node it has got 2 children you cannot do anything about it, it retains them.

This is going to be your P_7, and this one has got one intermediate node which has only a single child that can be removed. So, we got to 0, and this will become your P_8. And you remember what you skipped, skip length is equal to 1, and the segment that skipped is also 1.

So, that is how the trie corresponding to the binary trie that is in here. So, from the height of seven, we came to the height of five. So, we could reduce the height of the tree by a length of two, using five memory reference you are able to actually do the traversal of this trie and then making the decision on where to forward the packet to.

(Refer Slide Time: 34:30)



So, that is how you optimize any trie. And what if given this thing, what is the performance benefit that you get out of this? And before we actually take a look at that, let us try to see this, because now we have introduced a couple of more variables inside the node; earlier it was only indicating whether the node is actually representing a prefix or not, and then the left and the right child in the binary trie, but now we have a couple of more variables the node structure has got changed.

So, one is the indicator which was there as it is there, so, whether this node is actually representing a prefix or not. And the second thing that it has got is what is the skip length that we introduced that is one part of the trie and then we are also remembering what is the segment that we skipped so, the sequence of binary digits so, whether it is 0101 whatever it is, so, and then it has got the two pointers as it is: one is the left pointer and second one is the right pointer.

Each node has got more space now, you need to tell whether it can skip, length can be 0 as well, by default every node has got a skip length if it is 0, whatever the skip length is, segment portion should have those many number of the bits and then the left and the right child.

So, at the expense of some space at each node, we are able to compress the height of the binary trie and by reducing the number of the pointer traversal, we are able to make the decision.

So, if I had to how much space it requires is the order of n, where n is the number of the prefixes you have got, only those many nodes probably I can have provided you are able to reduce the

height of the tries significantly.

This means you will have many intermediate nodes which are having a single child which apparently, if your routing table is very sparse with very less number of entries, it is possible to have such a structure, although this is not exact thing you will have an order of n (O(n)), where n is the number of the elements in the trie, it requires that space and otherwise, you will fall back to the in the worst case you will fall back to the order of N cross W (O(NW)) so that is the binary trie.

This happens if the binary trie is full tree so for example, if this is the structure, every node has got two children then you cannot do any elimination. So, remember, elimination will happen only when the intermediate nodes have got a single child that can be removed. So, if they are two children then elimination is not possible.

So, if your trie is having this kind of structure it is full, then you cannot do any elimination, in that case, in fact, it takes more space because you introduced a couple of more variables inside the node structure, and it will take more space no optimization is actually possible.

But practically, how the routing tables looked like, how the prefixes looked like, reduction is indeed possible. Basically researchers found to use it looking at the practical routing table how they look like and how how much space is actually binary trie taking, can we get some optimization using the practically found routing tables, they came up with this kind of the optimization.

And they look up in the worst case still takes the order of W, where W is the width of the longest prefix that you have in the routing table. And the update time still takes the order of W time when the new prefix is just the extension of the longest prefix that we have. So, currently, let us say in the previous case, I have a height of like seven which is the prefix length, and then the new prefix that you get is just an extension of these seven bit then you will have to traverse those seven references, first and then you add the new node into the trie structure.

Again this is in the worst case when the height of the binary trie is equal to path compressed trie constructed, no optimization is possible and then the height of the trie is W and then the extension is going to take the first as you traverse W bits, and then the new addition can be done.

So, that is the one optimization that you can do on the binary trie, and remember what is optimization that we did here instead of matching one bit at a time, I am trying to match multiple bits at a time.

So, whatever the segment that we indicated here, those bits concatenated with the label that exists from the previous node to the child node, you can concatenate these ,two and then you will try to match in one go.