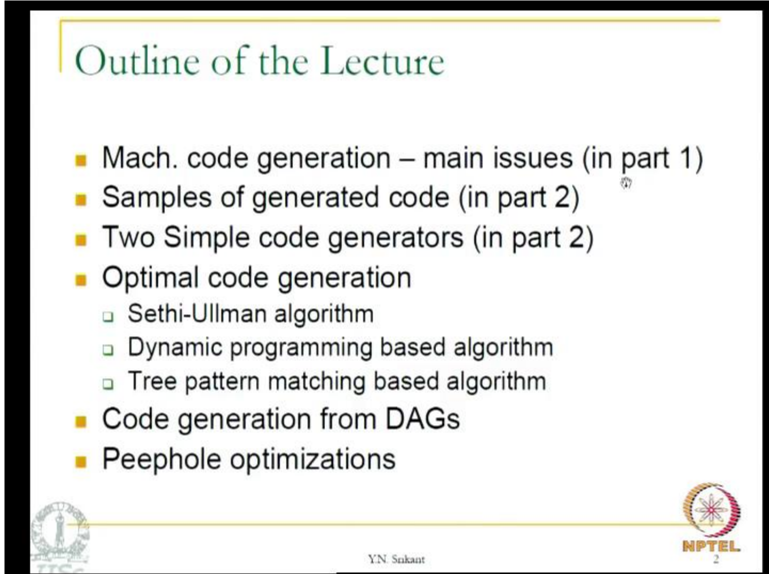
Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

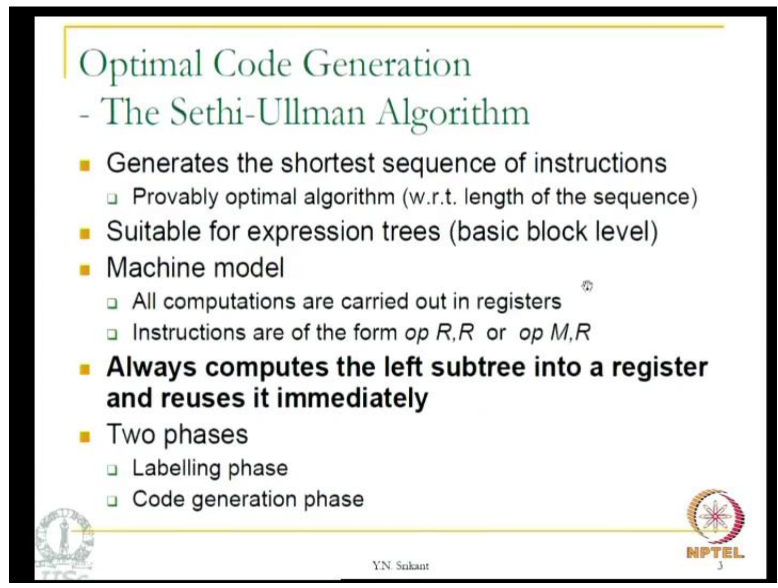
Lecture - 26
Machine code generation Part – 3

(Refer Slide Time: 00:24)



Welcome to part three of the lecture on machine code generation. Today we will continue with our discussion on Sethi-Ulman algorithm for optimal code generation and also look at the dynamic programming based and tree pattern based algorithms.

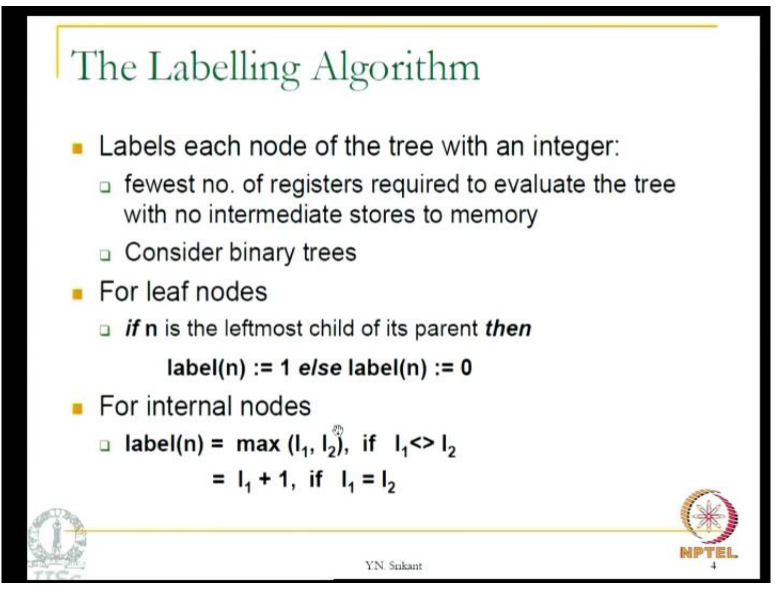
(Refer Slide Time: 00:34)



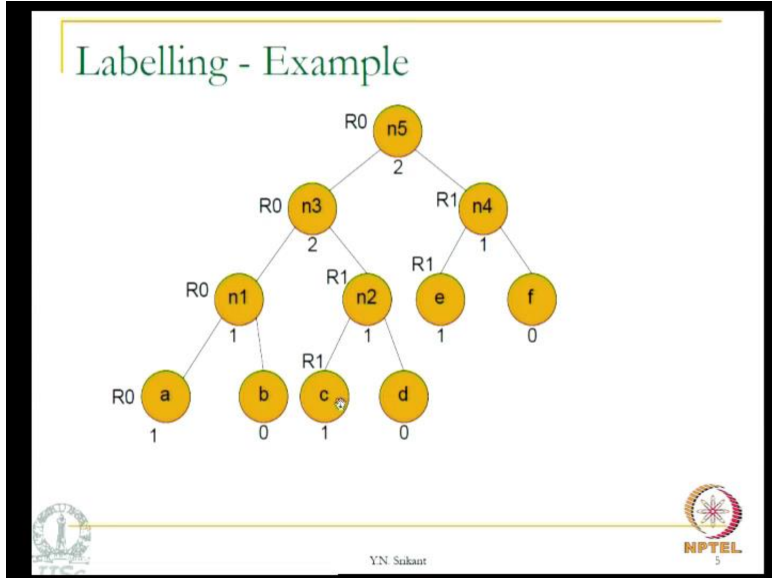
So, to do a bit of recap, the Sethi-Ulman algorithm is one of the provably optimal algorithm. So, in other words, it generates the shortest sequence of instructions possible for a given basic block. Of course, the limitation of this algorithm is that it works only for expression trees and it does not work for DAGs. So, if there is a DAG given to us, you know then we will have to break it into trees and apply the Sethi-Ulam algorithm on individual trees.

The machine model is, that all competitions are carried out only in registers. So, and instructions are of the form of R, R or op M, R. So, the point here is the left, you know, operand should always be in a register. So, it computes the left subtree into a register and reuses it immediately. So, in this type of instruction this register is the left operand and the memory could be the right operand. It cannot be the other way; it must always have the left operand in a register. So, it has two phases one is the labelling phase and the other is the code generation phase.

(Refer Slide Time: 02:03)



(Refer Slide Time: 02:07)



The labeling phase is quite simple. So, let me do the recap with the example itself. So, assuming, that this is the expression tree given, the operators in the nodes are not very important to us, so we have just named them as n1, n2, n3, n4, n5, the labeling algorithm labels the left most leaf of its parent as 1 and all other leafs of the parent as 0. So, this is a 1 and this is 0. The same thing is true for n2 and n3 as well. When we go to the internal nodes n1, n2 and n3, since the left subtree has a label of 1 and the right subtree has a label of 0, which implies, that the two labels are not equal, we take the max and assign it to the parent. So, this becomes 1. Similarly, this becomes 1 and this also becomes 1.

Now, we have a situation where n1, n2 have labels of 1 and n3 is the parent, with equal, with children, with children which have equal labels. So, we take, you know, increment the label value and take it as a label value for the parent. So, this becomes 2. So, now we have 2 and 1 here. So, n5 automatically becomes the max of these two, which is 2.

How do we evaluate such a tree with two registers? So, there are two possibilities, let us examine both of them. One of them, you know, gives us the correct evaluation; the other one does not. So, let me look at the incorrect evaluation first. So, let us assume, that we evaluate n4 into, you know, n4 first and then n3, ok. So, the subtree at n4 first and then, substitute tree at n3. So, if you have to evaluate this we must, obviously, evaluate both its operands. So, the left operand must be in registers. So, we must load it into a register R 1, let us say, and then we evaluate the parent n4, which, an operate, and operator n4. So,

which permits the right operand to be a memory operand. So, this gets evaluated and the result will always be in a register. So, that let us say, that is R 1. So, if we have finished this now, with this evaluation has been finished now and it says, yes, it can be done with one register, which we have satisfied.

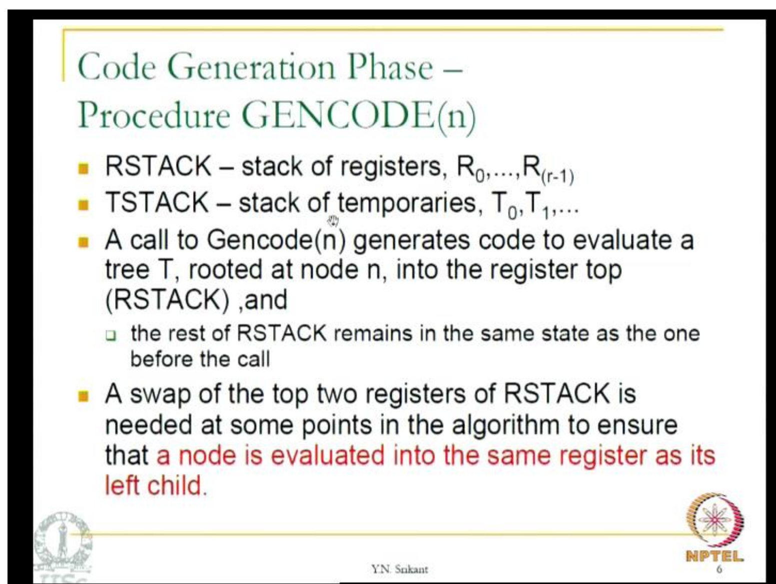
Now, remember, that the result is held in the register R 1. So, when we go to n3, the min reg value says it is 2. So, without two registers it will not be possible to evaluate this tree without storing the intermediate values into memory location. Let us see why.

Of course, this n1 is very similar to n4, so it can be evaluated into R 0. But then, we do not have R 1 free, it holds the value of n4. So, now unless we store R 0 into a memory location, which is not permitted by the algorithm because there exist a better order with which we can do the evaluation, so this code, you know, evaluation fails.

So, suppose we start with the evaluation of n3 and then go to n4. Now, we have a happy situation, we evaluate n1 into R 0 as before. Now, R 1 is still free. So, we can evaluate into, into R 1. So, this is a second register that we are using. And now n3 can be evaluated into R 0 with its two operands being in registers. After that value of n3 is held in R 0 and R 1 is free. This R 1 can be used, evaluate n4 and then we can evaluate n5.

So, the basic principle in the algorithm is take two, take the, node, substitute tree at a particular node, try to, you know, evaluate the subtree, which requires more number of registers and then the subtree, which requires less number of registers. But there are details, so we will come to those details very soon.

(Refer Slide Time: 06:37)



Here is the procedure Gencode, which is the main procedure to generate code for the subtree rooted at the node n. The procedure uses two stacks, one is the RSTACK, which is the stack of registers starting with R 0 to or R minus 1. So, there are ((Refer Time: 07:01)) R number of registers. The stack contains R 0 at the top and R of R minus 1 at the bottom.

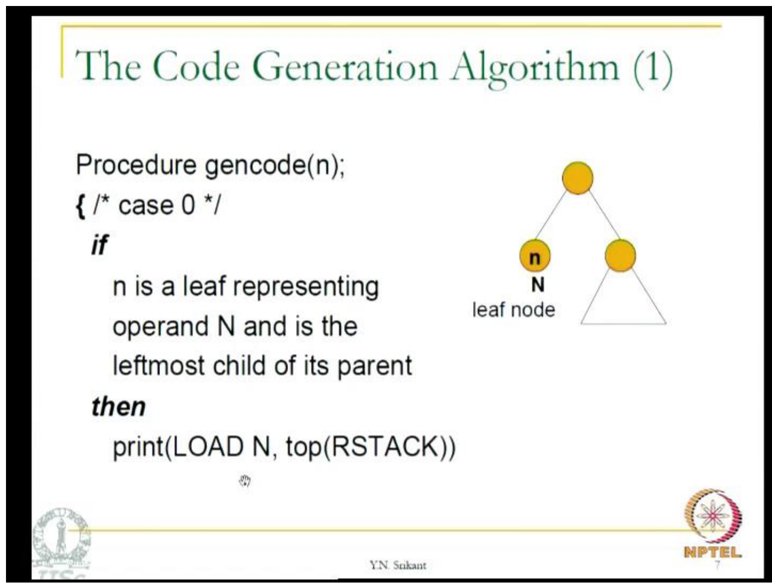
TSTACK is a stack of temporaries. So, we generally assume, that there are enough number of temporaries and that there is no scarcity of supply. That is the reason why the right bottom of the stack is not mentioned to be a fairly large number. The top of the stack is T 0. So, these are the various temporaries that are generated.

So, whenever we want a register, you know, we can use the registers from the stack of registers and then, whenever we want a temporary we can use it from this stack of temporaries. But we must remember, that we should use the top of the stacks, both register stack and temporary and then, go into the deeper some parts of the stack.

So, a call to Gencode n generates code to evaluate a tree T rooted at the node n into the register, which is at the top of the RSTACK; so top of RSTACK. And the rest of the RSTACK remains in the same state as the one before the calls. So, this is a very important consideration to prove the correctness and optimality of the algorithm. So, we must make sure, that the result is always at the top of the stack in the register located at the top of the RSTACK and that the rest of the RSTACK is in exactly the same state as

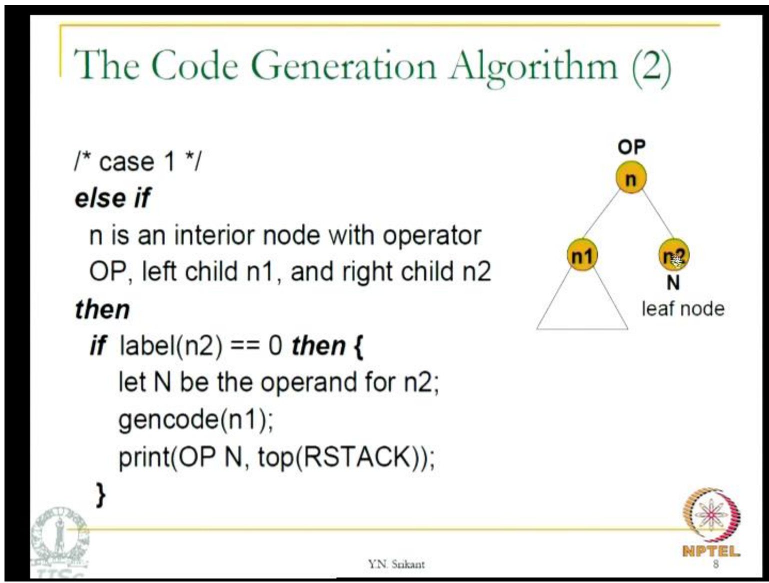
the one before the call to Gencode was made. So, these are, these are two very important things. So, incidentally, somewhere in the middle of the algorithm we will also have a situation to swap the top two registers of RSTACK. So, this becomes necessary to ensure, that a node is evaluated into the same register as its left child. So, we will see this as we go on.

(Refer Slide Time: 09:10)



So, let us look at the details of the algorithm. There are many cases, 0 through 4. Case 0, we are looking at a leaf node n. So, n is a leaf representing the operand n and is the left most child of its parents. So, this means it has a label of 1. So, if it were to be not the left most child, then it would have had a label of 0. So, this has a label of 1 that means, you know, you can evaluate it with, with one register, which is very trivial. We just generate the load instructions (load N, top of RSTACK). So, automatically, this implies the value of N will be put into the register, which is at the top of RSTACK at the time of evaluation. That is the execution. So, this is a trivial case.

(Refer Slide Time: 10:09)

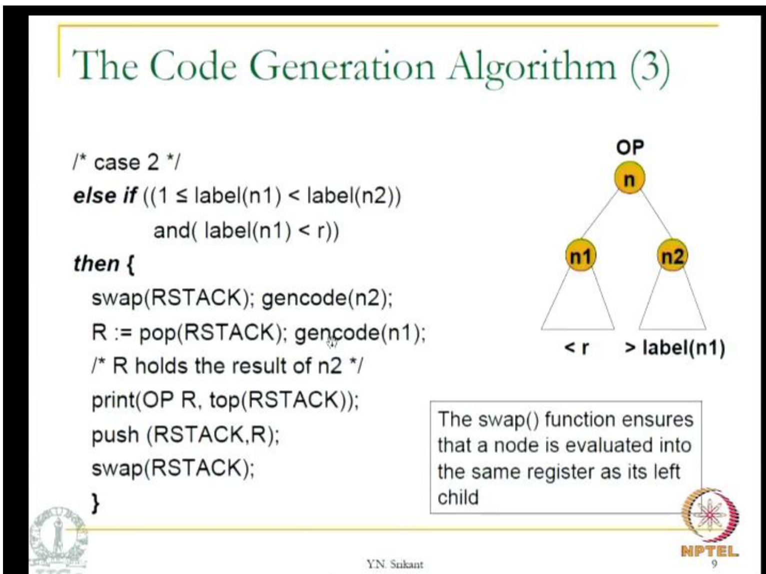


Now, case one, similar, but not the left child, but the right child is the leaf node. So, the left child is the root of a subtree, that is, n 1 and the right child of n is a leaf node. So, if because this is a right child it will have its value as label as 0. So, n is an interior node with operator OP left child n1 and right child n2 and the label of n2 is 0, that means, it is a right child. So, then let n be the operand for n2. So, this is the memory location corresponding to n. So, we first generate the code for n 1, you know, we can use the memory operand as it is. So, let us finish the code generation for the root, for the trees rooted at n1. So, that requires a certain number of registers label of n1. So, once we have finished, code generation Gencode prints out the code for that tree rooted at n1. Now, it is time to generate the code for n, alright.

So, that is what we print out. So, code print (OP n, top of RSTACK). So, why top of RSTACK? Top of RSTACK contains the result of the tree n1, right. So, that is the leftmost child and we are now reusing the leftmost child value immediately in the parent. So, and the result will also be now left in the same register as that of the leftmost child. So, that is top of RSTACK. So, remember Gencode n1 will, you know, make sure, that the top of RSTACK, the register will contain the value of the tree at execution time. So, this is correct, print (OP n, top of RSTACK) will, that is an instruction, when execute, which when executed will leave the result in the top of RSTACK again.

So, for this entire tree we have satisfied the rules of the game, that is, the rules of the algorithm, the value of the entire tree is available in top of RSTACK and rest of the RSTACK below the top is in the same condition as it was when we called the procedure.

(Refer Slide Time: 12:45)



The next case. So, we have both n1 and n2, you know, as roots of subtrees. But the case is, the tree n1 requires less than R number of registers. R is the number of available registers and the subtree rooted at n2 requires more than what n1 requires. So, it not more than R, not necessarily, but it definitely requires more than what n1 requires. So, the thing is, both of them, you know, are sub trees. So, we will have to generate code for them.

It is not, that we can use an operand as it is. So, one less than or equal to label of n1 less than label n2. So, that is what we have written here and label of n is less than R. So, because n2 requires more than the, more registers than n1. That is what the label means and n1 requires less than R number of registers. So, when we evaluate n1, we do not require any stores into memory. We really cannot say much about n2 because the condition says, label of n2 greater than label of n1, does not say it is greater than or equal to R. So, since we are evaluating the right subtree and suppose we evaluate it as it is, then the top of RSTACK will contain the result of n2, right.

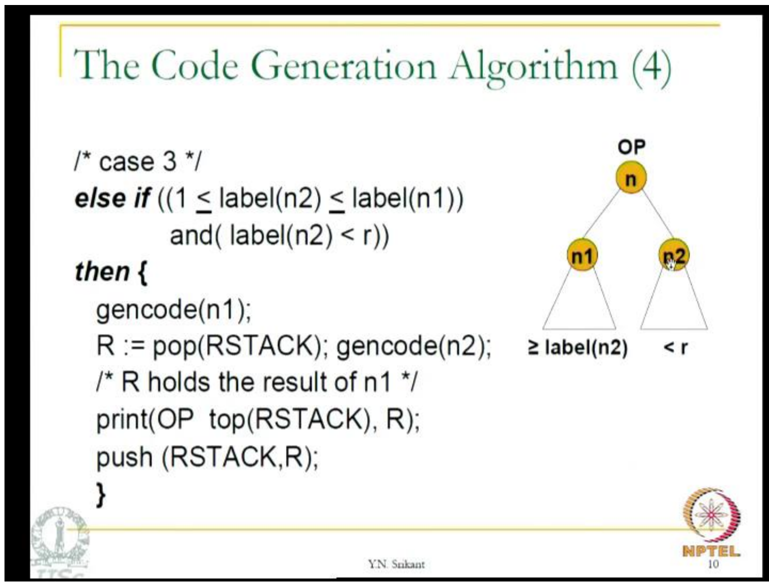
So, this would get us into some trouble because the left subtree must leave its result in the top of the RSTACK and that will be reused immediately by the root. So, to take care

of this we do a swap on the RSTACK. So, that swap functions, swaps the top two registers of stack. It ensures, that a node is evaluated into the same register as its left side. So, once it is swapped, the second register from the top will become the topmost register. Now, we call the code on n2. So, the second register, which is at the top now will contain the value of the entire tree. In other words, it generates instructions, which evaluate this tree and place its value in the top of RSTACK, which is the second register from the top after the swap operation.

Now, pop the RSTACK and remove that register. Remember, which register it is, that is R. Now, we have, you know, the original top of RSTACK at the top. So, we call Gencode n1. Now, it leaves the value in the top of RSTACK. So, this code sequence when evaluated will leave the value in the top of RSTACK, the register at the top of RSTACK. So, and R now holds the result of n2. So, remember n1 is in the top of RSTACK register and n2 is in R. Now, we are in a position to generate the code for the root n. So, we say, (OP R, top of RSTACK). So, this is correct.

This top of RSTACK right now is the original top of RSTACK before the swap. So, now R is of no use to us because the top of RSTACK will contain the result of the n3 after evaluation. So, we push the register into RSTACK. Now, the top of the RSTACK originally, you know, now becomes a second one. So, we swap it, brings it to the top. So, the invariant of the algorithm remains true. The top of RSTACK now contains the value of the entire register and rest of the RSTACK is in the same state as it was before the call. So, this is the reason we require a swap in this part in this particular case. So, we are evaluating the right node, right sub tree and to make sure, that the left sub tree leaves its result at the top of RSTACK we do a swap.

(Refer Slide Time: 17:08)



Then, the next case. So, again, you know, it is the case similar to the previous one. The right subtree requires less R number of registers and the left subtree requires greater than what the right subtree requires. And of course, we also say, label of n2 is less than r. So, this can be evaluated with no intermediate source, but there is no guarantee about n1. So, now, we have a very simple case because n1 requires more registers than n2 according to the label, we simply call Gencode n1, which evaluates after the, when the code is executed, the result will be left in the top of RSTACK register. So, remove that register pop by pop operation.

Now, Gencodes for n2. So, the code for n 2 will now be emitted. R holds the result of n1 and the top of RSTACK, after this pop operation, will hold the result of n2. Now, generate the code for the root. So, OP of, print OP top (RSTACK, R). So, now the top of RSTACK, you know, is the register currently at the top, but R was at the top of RSTACK before the algorithm, ((Refer Time: 18:40)) procedure as called. So, now, R will hold the value of the entire subtree. So, push (RSTACK, R). So, now R becomes the top of RSTACK and thereby, the invariant of the algorithm is maintained. R is the top of RSTACK and it contains the result of the entire tree. So, this is the last case in this algorithm.

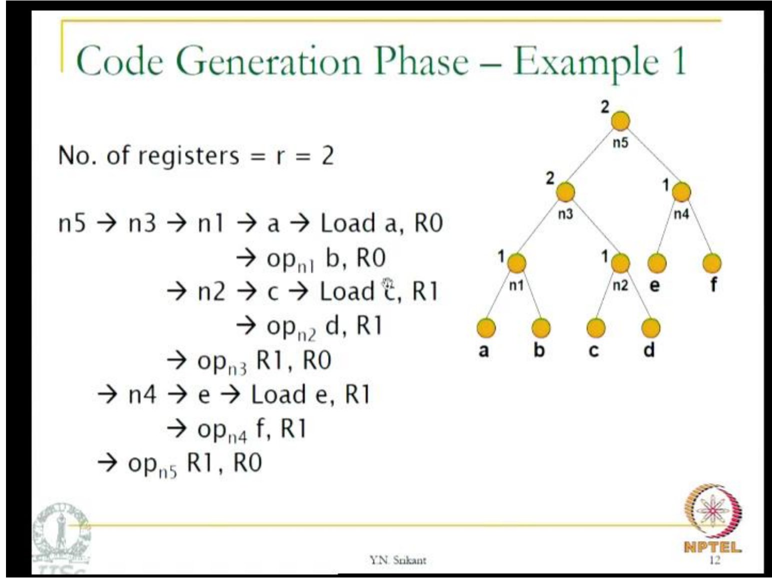
So, both the subtrees, n1 and n2, require more than R, more than or equal to R, number of registers. So, in such a case it is not possible to generate code without storing some of

the intermediate results into memory. So, in such a case, since both of them require ((Refer Time: 18:31)) into memory, it really does not matter, which one we take up. But since we need to leave the result of n1 in the top of RSTACK and then reuse it immediately for the parent, obviously we must generate the code for n2. So, we generate the code for n2.

So, if we had generate a code for n1, we would have had to put it into a temporary and load it back from the temporary at the time of, you know, evaluating n. So, this would be inefficiency. So, we, whereas, if the result of n2 is kept in a temporary, it can be reused as a memory operand in the instruction well while evaluating n. So, this is the reason we generate the code for n2. So, that is the Gencode n 2. Then, pop the, pop a temporary from the stack and generate an instruction to place the top of RSTACK value into the temporary. So, now all the registers are free again.

So, generate code for n1, otherwise we would have held back on registers, which is not correct. So, generate code for n1, then print OP (t, top of RSTACK). So, we now have, you know, the value of the entire tree in the top of RSTACK register. Of course, the temporary is not required anymore, so push it back into the TSTACK. So, again the invariant of the algorithm is satisfied. So, this is the detailed, you know, description of the algorithm

(Refer Slide Time: 21:11)



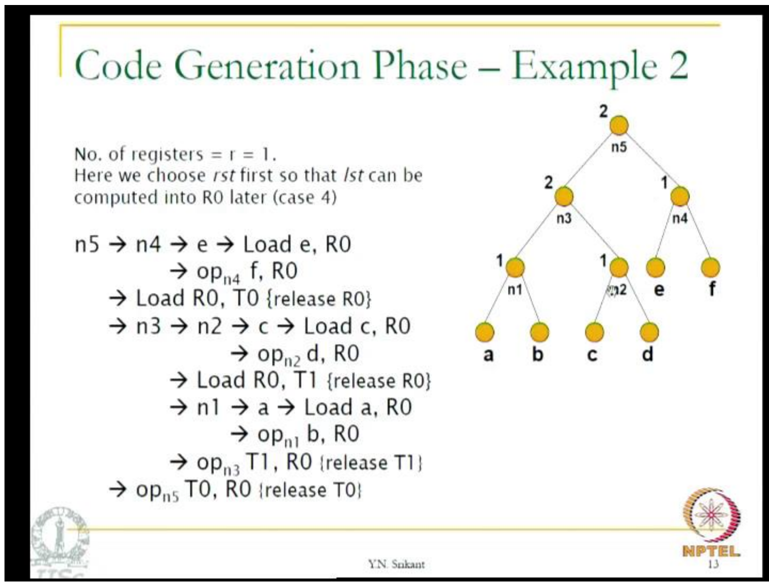
So, now let us look at two examples, one in which the number of registers is two and we do not require more than two, two registers. And the other would be, we have one register and we require more than one register.

So, obviously, the algorithm is very simply applied starting from n5. So, so is the min reg values are all mentioned here. So, this is 1 and this is 2 and this is 2, this is 1, this is 1. So, now we start with n5 and we have two registers available. This requires one register and this requires two registers. So, we call Gencode on this, right. So, once we call Gencode on this, the task is to evaluate these two, right. So, both have one register requirement and it is less than the number of available registers.

So, we can simply go to left subtree and that requires loading from memory location. So, load (a, R0), then op (b, R0). So, will be the evaluation of this would be completed by this stage. Then, we go to n2, generate the code for loading C into R1. So, R0 holds the value of this. Now, R1 is still 3, so we load this into R 1 and then op of (d, R1). So, that completes the n2. We go to n3 and op of (R1, R0). So, that completes n3 as well.

Now, R1 is free, so we can generate the code for n4, that requires loading e into R1. So, that is, load (a, R1). Then, op (f, R1) evaluates n4 and keeps it in R1 and when, then we generate the code for n5, which is op of (R1, R0). So, without storing any result into any temporary location we are able to generate the code for this. So, the code sequence would be load (a, R0) op n1 (b, R0) load (C, R1) op n2 (d, R1), op n, op n3 R (1, R0) load (e, R1) op n4 (f, R1) op n5 (R1, R0). So, this is the sequence of instructions for this particular tree.

(Refer Slide Time: 23:45)



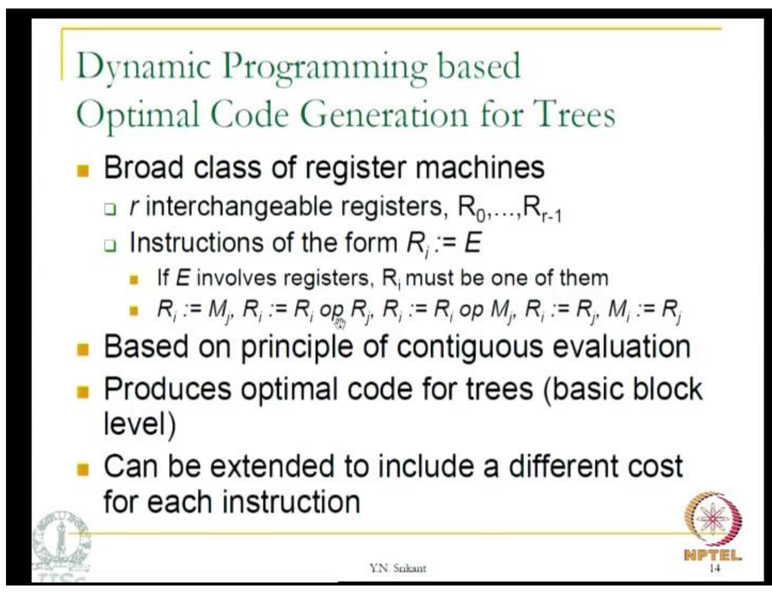
So, let us look at another example. The number of registers is just one, whereas we require two registers here and also here. So, when we start code generation at this point, this requires two and this requires one, but both of them require greater than or equal to number of registers available which is one. So, as I said, we, in this case, that is, here you know, so we would always go to n2, right. Both, both require more than, more than or equal to R number of registers. So, we go to this, then this is as usual load e and then op. So, those two are done.

Now, R0 contains the results of n4, but we must free it otherwise we cannot evaluate n3 at all. So, load, there are no instructions to operate with two memory operands, that is the problem here and so we free this register by generating an instruction to load it into a temporary. Now, we go to n3 and then, obviously, there is n1 and n2, both require greater than or equal to R number of registers. So, again it is a right by us that we have. So, we go to n2, then load C and then op into. So, these two require two instructions to generate two code and which evaluates n2.

Again, we must free the register, so that this can be evaluated. So, load (R0, T1). So, we generate, we take another temporary T1 and move the result into T1. Now, we are ready to generate code for n1, which is very similar load and then op. So, load (a, R0) op n1 (b, R0). So, that completes the evaluation of this n1. Now, left subtree is in a register, right subtree is in a temporary, we can generate a code for n3. So, that would be op n3 (T1,

R0). So, now the result will be in a register R 0 and this evaluation is completed. T1, of course, can be released at this point. Now, we are ready to generate a code for n5, which would be op n5 (T0, R0). So, that place is the result again in R0 and releases the temporary. So, so we have generated code for the entire subtree and we also have the result in a register. So, remember this requires storing result, intermediate results, into temporary locations.

(Refer Slide Time: 26:27)



So, now we move on to another optimal code generation algorithm. So, this is a dynamic programming based optimal code generation. Again, it caters to only trees and it is not possible to handle directed acyclic graphs. Unfortunately, DAGs, you know, code generation for DAGs is ((Refer Time: 26:52)) complete. So, there is no optimal algorithm that is possible. So, so at least there is no optimal algorithm, which works in polynomial time that is possible. We will see that later.

This dynamic programming based algorithm caters to a slightly border class of machines. The first requirement is, that there are R interchangeable registers, R 0 through R r minus 1. Remember, this is not a stack any more, ok. So, it is just a set of registers, R 0 through R r minus 1. And the requirement is, that there are, there is nothing like a special purpose register here. Any register can be used in any instruction. Then, the instructions are of the form R i equal to E. Not much restriction is placed on the form of E, like in the case of the Sethi-Ulman algorithm. The only restriction is, if E involves registers, R i must be

one of them. So, the left operand, left operand of the assignment is a register and the right, right part, RHS is an expression. So, if E uses expressions, then R i must be one of these. Sorry, if E uses registers, then R i must be one of those registers. If all the operands in E are memory locations, then R i can be any other register. So, this is the, so again the competition is performed in a register. Of course, we definitely require instructions of the form M i equal to R j to store values, which are in registers into memory, but all other instructions are of this form.

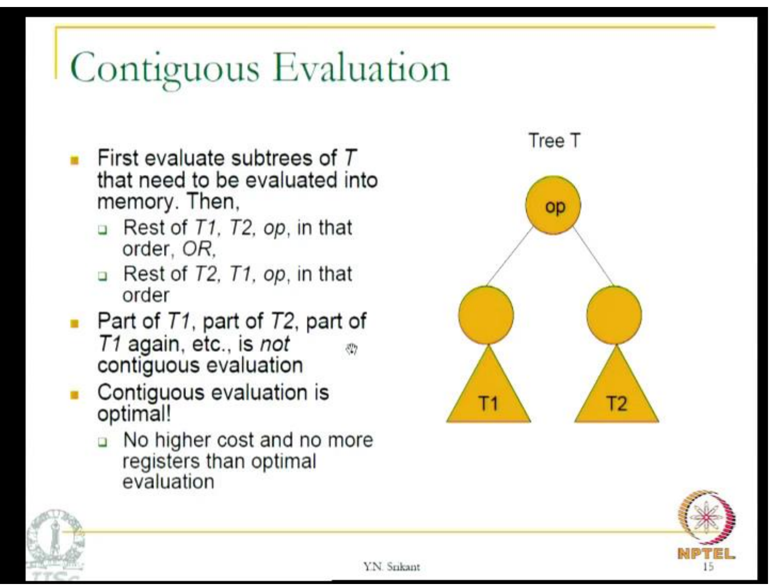
So, here is a sequence of possible rather set of possible instructions. So, assuming, that there are many registers available, for each one of those registers there is an instruction. So, R i equal to M j. of course, M j is any memory address. So, if we have two registers, then we would have instructions possible such as R 0 equal to M j and R 1 equal to M j. So, similarly for all the others. Another possible instruction is R i equal to R i op R j. So, see, that R i is used in the expression here. So, R i must be on the LHS also. Similarly, R i equal to R i op M j. So, one of the operands can be memory.

Then, we have R i equal to R j. So, of course, we would have, we may also have R i equal to M i op M j. But in this example, we do not have it that is all. So, R i equal to R j, which is a copy instruction. So, again this j must be i. You know, if this is i, then it is of no use. So, R i equal to R i is a useless instruction, whereas if R i and j are different, then it is a copy operation and that is a useful instruction. So, this is, this and this are the two variations of this. So, this E involves registers, R i must be one of them, is not really satisfied here, alright.

So, this is the format of instructions that we are going to assume in our examples. But any general instruction with more than two operators is also possible. So, more than one operator rather or one operator with three operands. So, this has two operands. We can have, you know, two operands now, but we can also have three operands. In other words, this is a very general purpose machine. So, we may even have complicated instructions such as multiply and accumulate, so that would be something like two operations in the same instruction, but with a special operator for it, MAC, for example, multiply accumulate. Based on the, this is based, what is known as, the principle of contiguous evaluation, which I am going to explain very soon and it is, it produces optimal code for trees.

So, again basic block level trees, no DAGs are permitted and it can be extended to include a different cost for each instruction. In the Sethi-Ulman algorithm, there was no question of a cost for each instruction. The length of the sequence of instruction generated was the cost of the entire code, whereas here we can attach an, you know, a cost to each instruction. So, this could have a higher cost compared to R i equal to R i op R j simply because this involves a load from memory, right. The same is true for M i equal to R j. This involves a store into memory, whereas this is just a copy operation. So, each instruction could have a different type of cost.

(Refer Slide Time: 32:18)



Then, we still have to explain what is contiguous evaluation? So, consider a tree in which there is an operator op at the root, then we have two subtrees T1 and T2. How does one evaluates such a tree? Well, there are many possibilities, right. So, before evaluating op it is certain, that we must evaluate both T1 and T2. But we can evaluate T1 first, then T2 and then op or we could evaluate T2 first, then T1 and then op. These are two possible orders. But then while evaluating T1 we could evaluate a part of T1, then jump to T2, evaluate a part of T2, continue with evaluation of T1, then again hop to T2, etcetera, etcetera. So, and finally, once the evaluations of T1 and T2, once they are completed, we can evaluate the tree T.

But the principle of contiguous evaluation allows only two possible orders, one is evaluate T1 completely or evaluate T2 completely. So, one of these must be done first.

Then, we can once assume, that we have evaluated T 1 first, then we must evaluate T2. And then, we can evaluate op, otherwise we must do T2 completely, then T1 and then op.

Now, the final details. So, when we are evaluating T1 and T2, there is a rule, which says, evaluate subtrees of T that need to be evaluated into memory first. So, so there may be parts of T1, which require too many registers. So, what we do is evaluate that small subtree of T1 into, we are using all the registers stored into a memory location, we do that for all parts of T1, which must be evaluated into, you know, memory. The same is true for T2. Now, after all these evaluations are done we would be, you know, we would actually be left with parts of T1 and parts of T2, which can be evaluated in using only registers in such a case. After the evaluations into memory are over, we must evaluate the entire T1 first and then T2 and then op, or entire T2 first and then T1 and then op. So, this is called as contiguous evaluation order.

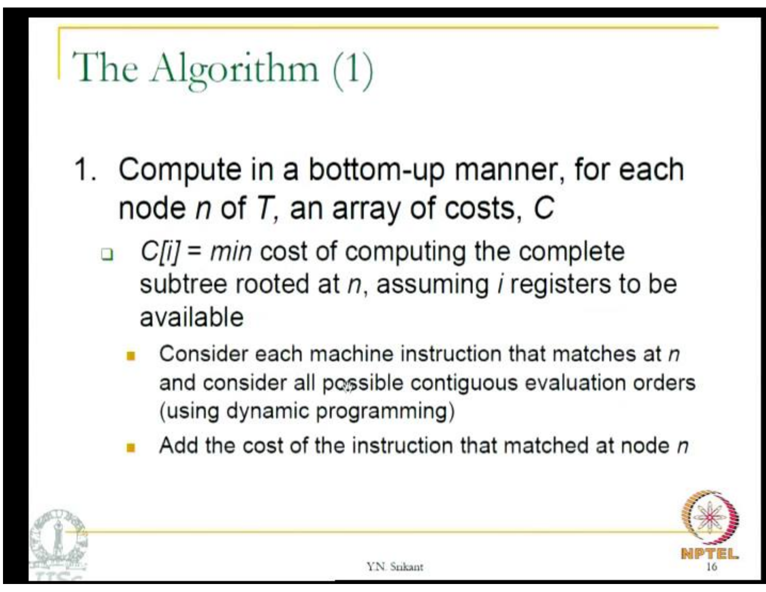
Why should we say evaluate subtrees of T, that require, that can need to be evaluated into memory first, then the rest. The problem is, when we have something to evaluated, something to be evaluated into memory, we might as well use all the registers available, load that result into memory and then again go to some other part, which requires evaluation into memory, use all the registers and then store it into memory and so on and so forth. If we do not do this, if we try evaluating other parts of the tree, which require registers alone, we would be actually left with lesser number of registers to evaluate the small subtrees, which have to be evaluated into memory. How does one decide whether to evaluate something into memory or using registers alone? That we will learn very soon because there is going to be a cost attached to every type of evaluation.

So, what is not permitted in contiguous evaluation? Well, we cannot assume, that everything can be evaluated into in using registers alone, right. So, we cannot evaluate parts of T1 and then parts of T2, again parts of T1, again parts of T2, etcetera, etcetera. We cannot do that. That is permitted only when they need to be evaluated into memory, otherwise when we are using only registers we must finish the evaluation of T1, if we start it, and then T2 and then op or the other way, first T2, then T1, then op. These decisions are actually driven by the cost considerations and we will see how to do it very soon.

Why this contiguous evaluation? Well, the most important property of contiguous evaluation is that it is optimal. So, the code, which is going to do contiguous evaluation is optimal code in what sense? The code does not require any higher cost and it does not require more registers than any other optimal evaluation. So, there may be many optimal evaluations possible using the same number of registers and having the same cost. What this contiguous evaluation theorem tells us is, generate code for contiguous evaluation, then it actually cannot be worse than any other optimal evaluation. It is as good.

So, the contiguous evaluation, you know, principle is very simple and it allows us to generate code in a very simple fashion that is, having decided that we generate code for T1, we do not have to jump to T2 at all. We can finish the code generation for T1 and then go to T2, generate the code for T2 and then generate the code for op. If the cost consideration had told us, that we must do the other way, first T2, then T1, then op, we finish the code generation for T2, then T1 and then op. So, we do not have to consider the operation, you know, evaluation orders, which, which require parts of T1, you know, optimal evaluation order, which requires part of T1 to be evaluated first and then parts of T2, etcetera, etcetera. We can, we can stick to contiguous evaluation orders and still be certain, that we will generate optimal code.

(Refer Slide Time: 38:39)

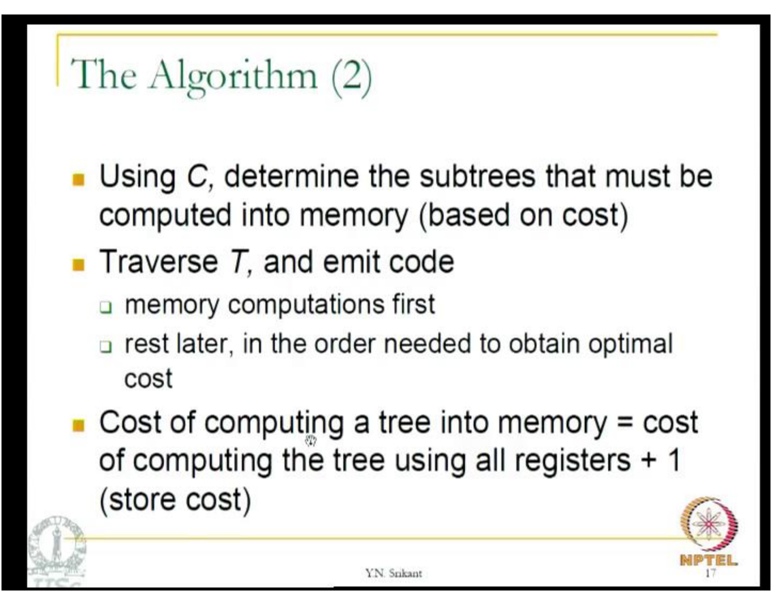


So, let us look at the algorithm now. There are three steps in the algorithm. The first step says, compute in a bottom-up manner for each node of n, an array of cost, C. This is a

very important, probably the most important part of the algorithm. What does the array C contain? C of i is the minimum cost of computing, the complete subtree rooted at n, assuming i registers to be available. So, if there are 3 registers available, then this C will have four locations, C of 0, cost of, minimum cost of computing the entire subtree assuming 0 registers. Similarly, C1 would be with 1 register, C2 contains the cost with 2 registers and C 3 contains a cost with 3 registers. So, 2 and 3 registers, right.

Then, consider each machine instruction that matches at n. So, this is the way we compute Ci and consider all possible contiguous evaluation orders using dynamic programming at the node n. So, and then add the cost of the instruction that matched at node n. So, this is the way to compute Ci. So, we consider all possible orders, then take the minimum and assign it to Ci. So, we assume one register, two register, three registers, etcetera, etcetera, say R registers, which are available.

(Refer Slide Time: 40:24)

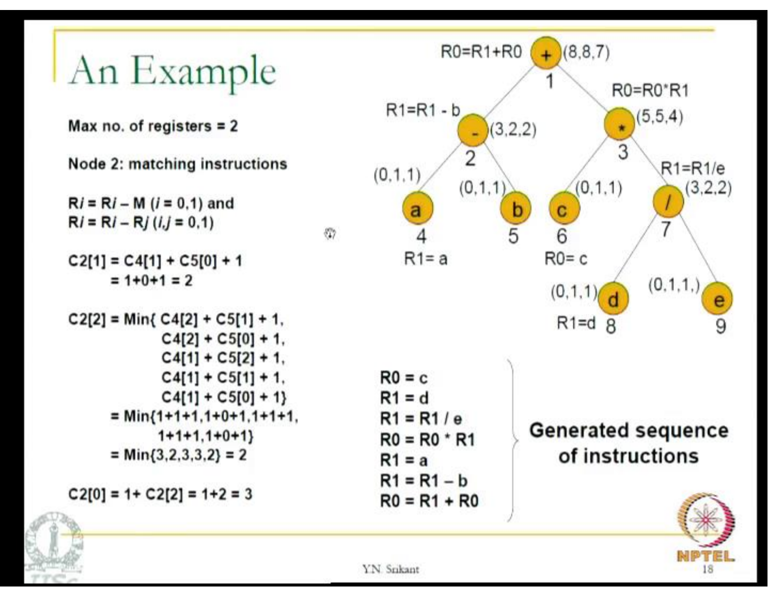


And then, to compute the cost using, you know, into memory, the cost of computing a tree into memory would cost of computing the tree using all registers plus 1, which is the store cost. So, we do not have to do anything special. Compute the cost using one register, two register, etcetera, up to R registers. Then, fill up the cost of computing into memory by saying cost of computing the tree, we are using all registers plus 1. So, as I said, this is the way we do it.

Then, once we do this, once the story is over, all the registers can be free. Then, using C determine the subtrees that must be computed into memory. So, that I mentioned. Then, you know, traverse T and emit code. So, memory computations first and then the rest, later in the order needed to obtain optimal cost. So, this is, you know, fairly straightforward as I am going to explain.

Now, the most important step is to compute this C. So, the subtrees with that must be computed into memory, must be at the lowest cost otherwise we will compute it into registers rather than into memory. So, one trivial possibility here is the leaves, which are already in memory, the operands corresponding to the leaves will be in memory. So, computing something into memory, in that case, would be of cost 0, whereas computing it into a register means, loading it into a register, which is of higher cost.

(Refer Slide Time: 42:08)



Here is an example. So, this is our tree, right. So, there is a plus, there is a minus, there is start, there is a slash and these are the operands a, b, c, d, e, which are in memory. So, as I already said, the cost of computing a in to memory will be 0 because it is already a memory operand. The same is true for all the leaves b, c, d and e.

So, remember the computation of C proceeds in a bottom-up fashion. So, we do this, then this, then this, then we do this, we do this, we do this, then we do this, then this and finally, this. So, that is the post order traversal of the tree. There is the way we would do it.

So, let us assume, that there are 2 registers and it is very elaborate to present the whole example. So, let us take node 2 and beat it to death, find out all the details regarding the evaluation. So, node number 2 is here and in our instruction set, that instructions, which match at node number 2 would be Ri equal to Ri minus M with i equal to 0 and i equal to 1. So, that means, this right operand can be in memory, but the left operand must be in a register. We do not have instructions with both operands as in memory, whereas we definitely have instruction in which both operands are in registers. So, Ri equal to Ri minus Rj; i, j, both can be either 0 or 1. So, both of them can be in registers or left can be in register and right can be in memory. So, these are the only two types of instructions, which match at node number 2. Obviously, this is the same for this as well.

Now, so we are going to compute C. Let us say, let us call it as C2, so that we know, that it is the cost at node number 1 with 1 register, had 2 registers and then, compute the cost of, I know, cost of computing it into memory by saying it is 1 plus C 2 of 2, that is, cost of 1 plus cost of computing it with 2 registers. Let us see how to compute node number 2 with 1 register. So, if we want to do that, since the left operand is always in a register, so we must look at the possibility of C4, that is, node number 4 with 1 register, C5 with 0 number of registers plus the cost of computing the root, that is, 1. This is the only possibility.

We cannot say C4 of 0 because there is no pattern, which matches here and once we have done C4 of 1 we are left with 0 number of registers. So, saying C5 of 1 has no meaning, so it will be C5 of 0 always. Of course, 1 is for the root itself. So, C4 of 1 means, it is a leaf node, it is in memory, so loading it into a register will require, cost 1 and that is trivial. So, that is, so if we have a cost of 1 for an instruction of type Ri equal to Mj, then it would be 1. If we have a higher cost, then that is the number, which we are going to put here.

So, this is what I meant when I said different instruction could have different costs as well. So, C4 1 in this case is 1. It could have been 2 or 3 based on the machine. C5 was 0, computing something into memory when it is already in memory is obviously, 0. And we are assuming, that minus requires, one cost, has cost one, so that would be plus 1 again. So, the whole cost is 2.

It is more interesting to look at C2 of 2 with 2 registers. There are many possibilities. So, we have finished C2 with 1 register. Now, we are looking at C2 with 2 registers. So, we could evaluate this with 2 registers. Of course, evaluating this with 2 registers has no meaning, you know, 1 register is enough. So, the other would be just left as it is, right. So, whether we have 1 register or 2 registers, the value will be moved into only one register. So, the cost will always be 1 with 1 register or 2 registers for this particular node 4.

So, after C4 of 2 we will be left with 1 register, one register being used for node 4. So, C5 can be evaluated with one register and then, we have 1. So, C5 of 1 means, move it into a register and then, one for the root. This is common always, alright. Other possibility is, C4 with 2 registers, C5 with 0 number of registers. Just because just because we have a free register, it does mean we must use it. So, C5 is 0 registers plus 1. So, these are the two possibilities with 2 registers. Now, we look at the possibilities with C4 using one register. So, C4 1, then obviously, we would have C5 2 plus 1, then C4 1 plus C5 1 plus 1, C4 1 plus C5 0 plus 1. So, these are all the possible ways of evaluating the tree with two registers, right.

So, the costs are, this is cost 1, very obvious; this would also be cost 1 because we want to move it into a register plus 1. So, 1 plus 1 plus 1. This is cost 1, this is cost 0, this is cost 1. So, 1 plus 0 plus 1. Again, this would be cost 1, this would be 1, this would be 1, so 3. This would be 1 and this would be 1 and this would be 1, again 3. And the last one, this would be 1, this would be 0, this would be 1. So, this is 1 plus 0 plus 1. So, the minimum of this would be really 2.

So, cost of evaluating node 2 with 2 registers is also 2. It was 2 with one register and it is 2 with two registers, also minimum cost. Well, that is very intuitive because whether we use one register or two registers, this requires only one register. So, and this is best in memory. There is no point in loading it into the register and then using it. So, this cost is 2 even with two registers. If you want to store it back into memory, then we need another extra cost, so that would become 3. So, the triple for this node C0, C1, C2 would be (3, 2, 2) and for the leaves it was (0, 1, 1) simply because it is already in memory. So, cost 0 where we use one register or two registers, it is always 1.

So, all the leaves have the same triples. This is very similar to 2. So, this also has the same triple. Then, let us look at node number 3. This has (5, 5, 4). So, remember the cost here is (0, 1, 1), whereas the cost here is (3, 2, 2).

(Refer Slide Time: 50:24)

Example – continued
Cost of computing node 3 with 2 registers

#regs for node 6	#regs for node 7	cost for node 3
2	0	1+3+1 = 5
2	1	1+2+1 = 4
1	0	1+3+1 = 5
1	1	1+2+1 = 4
1	2	1+2+1 = 4
	min value	4

Cost of computing with 1 register = 5 (row 4, red)
Cost of computing into memory = 4 + 1 = 5

Triple = (5,5,4)

Y.N. Srikant

NPTEL

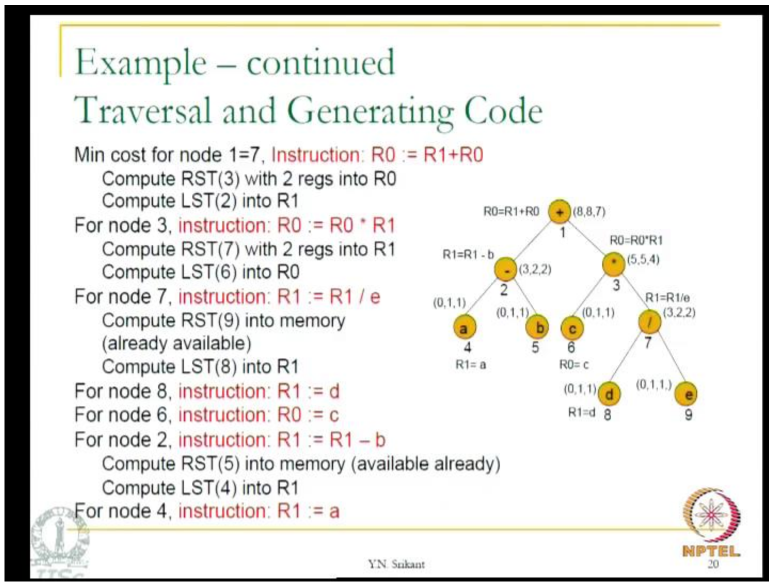
19

So, we could, computing the node 3 with 2 registers, let us look at that, that is the most complicated case. So, we could do with, left subtree with 2, right with 0, left subtree with 2, right with 1, left subtree with 1, right, with 0. Similarly, left subtree with 1, right with 1, left subtree with 1 and right with 2.

So, in all these cases we can, you know, compute the cost. So, with 2 registers this would be 1; with 0 registers for node 7 it was 3, right. So, that is 3 and the root would be 1, so this is 5. This would be again just a load. So, this would be 1 and this would be with 1 register, that would be 2 here, so this is 2 and then we have 1, so this would be 4. So, similarly, 1 and 0 would mean, 1 plus 3 plus 1. This is, next one would be 1 plus 2 plus 1, etcetera, etcetera. The minimum value is 4.

Cost of computing with one register is very simple. So, this is 1 and 0. So, I have avoided writing it again. So, that would be 5. Cost of computing into memory would be cost of computing with 2 registers, that is, 4 plus 1, that is, 5. So, the triple will now turn out to be (5, 5, 4). So, it is more advantageous to evaluate subtree with node 3, into 2, with 2 registers rather than with either 1 or 0 registers.

(Refer Slide Time: 52:10)



So, this is entire tree. So, annotated with the triples, right. Finally, when we finish this root we would be left with the triple (8, 8, 7), and all others have already been evaluated. Now, the way we generate code, we pick the component of the triple, which requires minimum cost. So, in this case it happens to be the cost with 2 registers, right. So, while computing the triple I must mention, that we must keep track of the instruction pattern, which actually gives us that cost as well, otherwise it is not possible to generate code. So, with this particular cost 7, the instruction pattern would have been Ri equal to Rj plus Ri, something like that. So, we can instantiate that with any number registers, assuming that there are 2 registers, we have instantiated it as R0 equal to R1 plus R0. So, the result this implies.

So, this is the pattern, which matches here and that is the least cost pattern. It implies, that the RST would be computed into R0 and LST would be computed into R1. The cost information should also be borne in mind. So, this would be with, LST would be with, you know, the least cost would be again, with one register and here the least cost would be with 2 registers. So, that is what is mentioned here.

Then, we go down. So, for node number 3 this is the one, which requires higher number of registers, so 2 registers. So, that is the one, which should be dealt with first. So, again we compute RST with 2 registers into R1. So, this is the least cost and LST would be into R0.

Then, we go down further. For 7 we compute RST into memory already available and LST into R1. So, the code generated would be in the other order, so right. So, this would be R1 equal to d first, then R1 equal to R1, you know, slash e. Then, we have R0 equal to, then we have R 0 equal to c, then we have R0 equal to R0 star R1 and so on and so forth.

So, once we have annotated the tree with all these instructions, you know, code generation would be very easy. So, annotation requires a top down traversal of the tree starting from the top, pick the minimum cost, then it will also tell you the pattern, it will also tell you, which particular register is to be used for the LST and the RST. So, there we again pick the number of available registers with minimum cost, sorry, the cost of evaluating the subtree with minimum cost and so on. Pick that, and go on doing that further rest of the tree.

So, the basic idea is when we start at the root. We always follow the direction, which you know, the subtree, which requires more number of registers will be dealt with first. So, this information of which pattern matches there and what is the number of registers that is required for this pattern, etcetera, will have to be kept track of. So, remember, that we have that information here, you know, C4 with 2 registers, C5 with 1 register, etcetera, etcetera. So, we have one pattern for each of these, that is possible and that pattern and the number of registers required for LST and RST will have to be maintained in separate data structures, so that we can generate code, you know. We can cover the tree with the appropriate code during this pass and once that is over, we could simply do a post order traversal and generate the code itself.

So, the code that is generated is listed here. So, R 0 equal to… So, remember this is not the order in which we do it; it is post order, alright. But we will have to do the right subtree first and then the left subtree and so on and so forth. So, R0 equal to c. So, we have to follow the order in which we have generated code and that is the way in which we finally, emit the sequence. So, we will stop here and continue in the next lecture.

Thank you.