Parallel Algorithms

Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 16
Applications

Welcome to the 16th lecture of the MOOC on Parallel Algorithms. In this lecture, we continue with the discussion on tree algorithms that we started in the previous lecture. We started looking at the tree algorithms as applications of optimal list ranking. In the lecture in the 14th lecture we discussed an optimal algorithm for this ranking that runs in order of log n time using n by log n processors on an EREW PRAM. And, then by way of applications of this list ranking algorithm, in the previous lecture we looked at the tree algorithms.

So, in particular we considered the case where we are given a tree that is in adjacency list representation. So, let us say the size of the tree is n that is the number of vertices is n, then we found that an Euler circuit of the tree can be found in order of 1 time using n processors on EREW PRAM. Once, we have the Euler circuit to root the tree at a particular node v, we can disconnect 1 incoming edge into v, then the Euler circuit that we found becomes an Euler tour that is starting with v, and terminating with a neighbor of v.

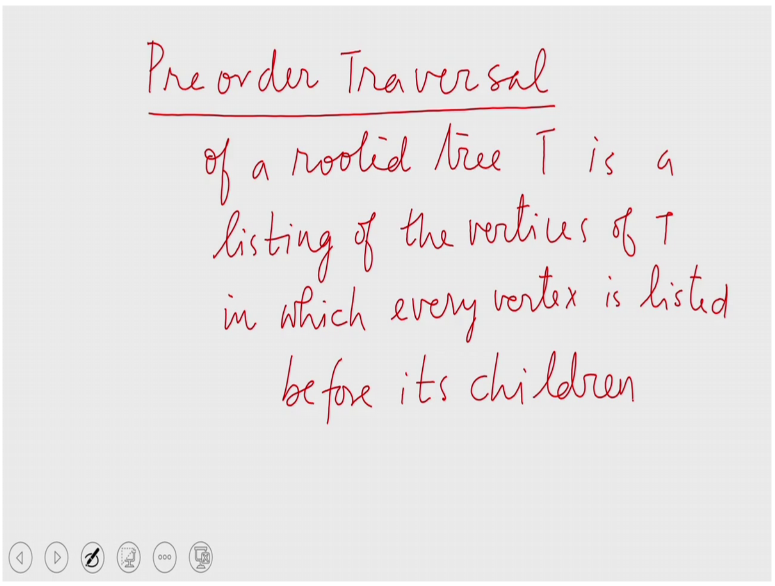
When we rank the Euler's Euler tour we can convert the Euler tour on to an array that is a linked list that is ranked can be copied into an array, every element in the list can copy itself to it is a position. For example, a node which is logically the i'th in the list can copy itself to the i'th physical location, then we get a representation of the linked list in an array. In which the physical representation is exactly the same as the logical representation.

So, once we have the linked list converted into an array in this fashion, we can find the parent for every single node in the tree that is we root the tree this is done by looking at every edge and it is twin for an edge ij.

If, ij is ranked before it is twin ji then i is the parent of j otherwise j is the parent of i. In this fashion, we can define the parent of every single node in the tree. So, that is what we mean by rooting the tree. And, then we saw that once the tree is rooted it is possible to find the level number for every single node, to find the level number for every single node what we do is this for every parent child parent to child edge we assign a weight of 1, and for every child to parent node we are the child to pair an edge we assign a weight of minus 1.

So, with these initial weights when we perform a prefix sum on the array version of the Euler tool, we find that with every single vertex we can associate the level number of it. Today, we shall see some more applications of the list ranking problem for tree problems. The first 1 we are going to look at is the pre order traversal.

(Refer Slide Time: 03:32)



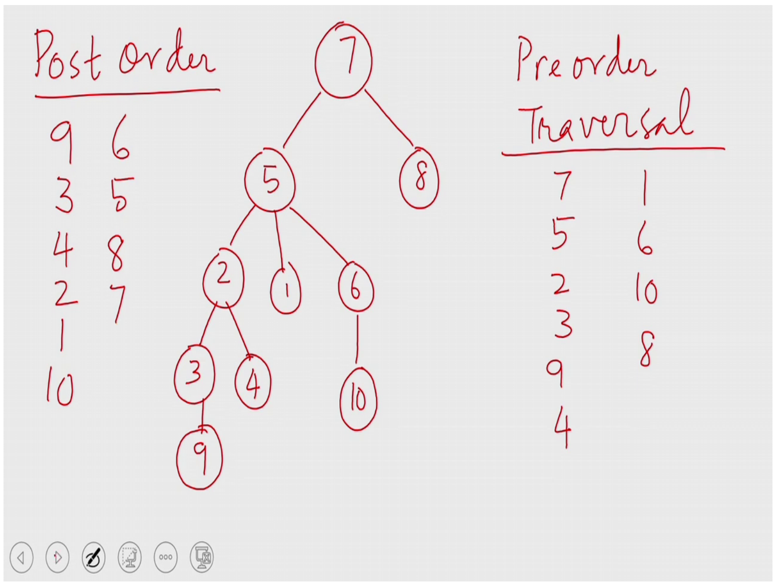
Preorder Traversal

of a rooted tree T is a listing of the vertices of T in which every vertex is listed before its children

The preorder traversal of a rooted tree T is a listing of the vertices of T in which every vertex is listed before it is children.

So, let us see how to find the preorder traversal of a given tree a given root a tree.

(Refer Slide Time: 04:43)

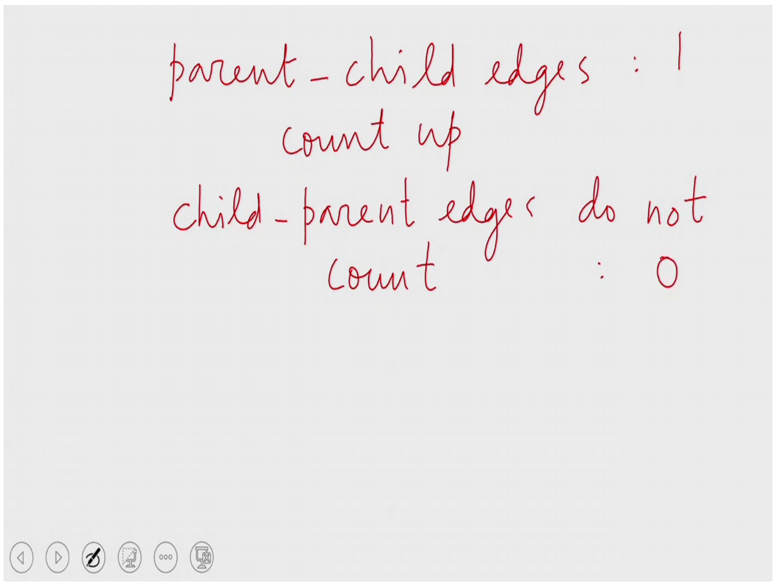


So, we will continue with the same example that we had in the last class, we took a tree and rooted the tree at 7; after rooting the tree at node 7 it looked like this. So, this was the rooted tree that we had a preorder traversal of this tree, begins with the root a node has to be visited before all it is children.

And, then let us say we go to the left child which is 5 and then 5 has 3 children 2 1 and 6. Let us say we go to 2 and then we visit 3, then 9, then 4, and then we come back to 2, 2 has already been visited we come back to 5 and then 1, we are back to 5, and then 6 and then 10; we backtrack to 6 and then 5 and then 7 and then 8.

So, this is a pre order traversal of this tree T. So, the question is how do you calculate the compute the pre order traversal in parallel.

(Refer Slide Time: 06:16)



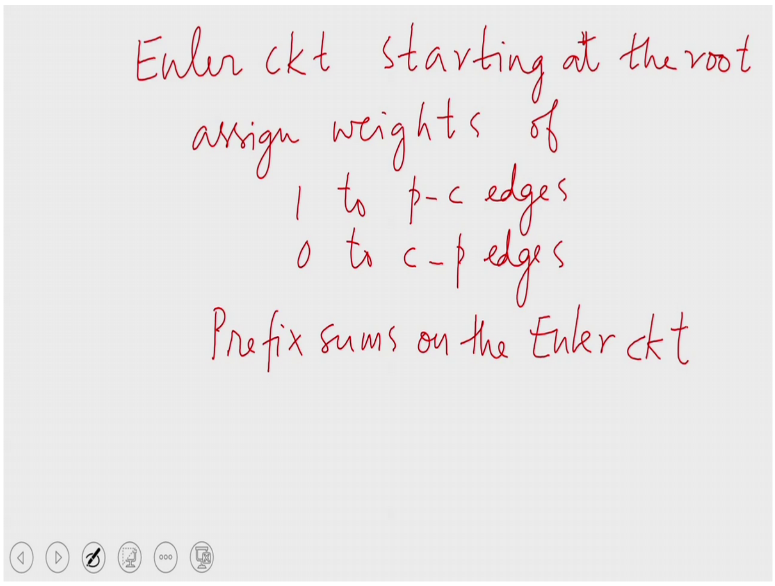
parent - child edges : 1
count up
child - parent edges do not
count : 0

So, in the pre order traversal we find that the parent to child edges count up. So, in this example we found that starting from 7, we know that the root should come first in the preorder traversal. So, we have 7.

Then we choose an outgoing edge of the root which is 7 5, and then 5 is the next node to be visited, and then from 5 we choose to go to 2 and 2 is the next node visited. So, a node is written down as soon as it is visited, but then on the way back we do not have to write anything. For example, when we return from 9 we go back from 9 to 3 3 to 2. And, there is nothing to write in this backward traversal and then when we move from a forward from 2 to 4 we have to write 4, and then from 4 we backtrack again to 2 and 5 and there is nothing to write here.

We what it means is that when we are moving backwards there is nothing to write, but when we are moving forward every new vertex that is encountered should be written in the traversal. So, the parent child edges count up whereas, the child parent edges do not count, this immediately suggests an algorithm they suggest that for parent child edges, we have to assign a weight of one, because they count and for child parent edges we have to assign a weight of 0, because they do not count.

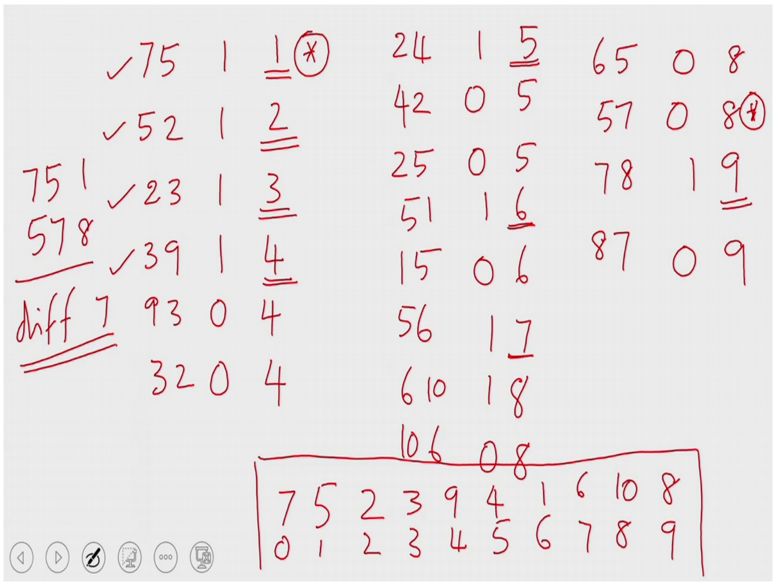
(Refer Slide Time: 08:04)



Euler ckt starting at the root
assign weights of
1 to p-c edges
0 to c-p edges
Prefix sums on the Euler ckt

So, what we do is this? We take the Euler circuit starting in the root and assign weights of 1 and 0 to parent child edges and child parent edges respectively. And, then perform a prefix sums on the Euler circuit. So, let us take an example and do this.

(Refer Slide Time: 09:05)



75 1
57 8
diff 7

✓75	1	1 (*)	24	1	5	65	0	8
✓52	1	2	42	0	5	57	0	8 (*)
✓23	1	3	25	0	5	78	1	9
✓39	1	4	51	1	6	87	0	9
93	0	4	15	0	6			
32	0	4	56	1	7			
			610	1	8			
			106	0	8			

7	5	2	3	9	4	1	6	10	8
0	1	2	3	4	5	6	7	8	9

In our example we had the Euler circuit in this fashion starting from 7 we had 75 52 23 39 93 32 24 42 25 51 15, 56, 610, 106, 65, 57, 78 and then finally, we had the closing edge 87.

So, what we have agreed here is that for every parent child edge we have to assign a weight of 1 75 is a parent child edge. So, is 785 has 3 children 21 and 6, 2 has 2 children, 2 3 and 2 4, and 3 has 1 child 3 9, 6 has 1 child 6 10 and then every remaining edge is a child parentage. So, they will get a weight of 0. Then, we find the prefix sum over this array. So, this is the Euler circuit that we have in the form of an array.

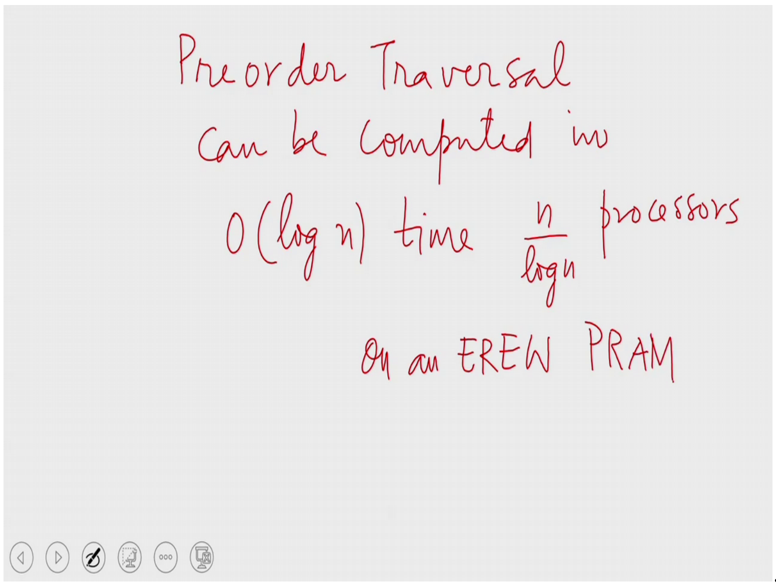
Once we have R and the Euler circuit as we did in the last class, we can copy it into an array on this array we have now these weights. Let us say we find the prefix sum of these weights. So, the prefix sums would be like this. So, once we have the prefix sums, we observe that in the preorder traversal 7 should come first in any case the root should come first. Now, 7 5 has a rank of 1 therefore, 5 occupies the first position 7 occupies the zeroth position 5 occupies the first position, then 5 2 is has a rank of 2.

Therefore 2 occupies the second position and then 3 occupies the third position, 9 occupies the 4th position, and then what would be at the 5th position 24. So, 4 should come at the 5th position and 1 will come at the 6th position 7 and 8 positions will be occupied by 6 and 10 respectively and 9th position is occupied by 8.

So, what we have done is to. Look at the parent child edges all of these are counting up and for all these edges, the rank that we obtain will be the ordinal position for the destination of the parent child edge that is a child in the preorder traversal. In particular for the edge 2 3 the destination is 3 and this has a rank of 3, which means the edge 2 3 should occupy the third position in the preorder traversal.

So, once we have computed the ranked of the prefix sums in this fashion. Once we know which are the parent child edges, then all that we have to do is to fill the prefix the fill the pre order traversal array, which is shown on the bottom here. So, this filling can be done in order of one time once you have one processor per element of the Euler circuit array.

(Refer Slide Time: 13:29)



Preorder Traversal

Can be computed in

$O(\log n)$ time $\frac{n}{\log n}$ processors

On an EREW PRAM

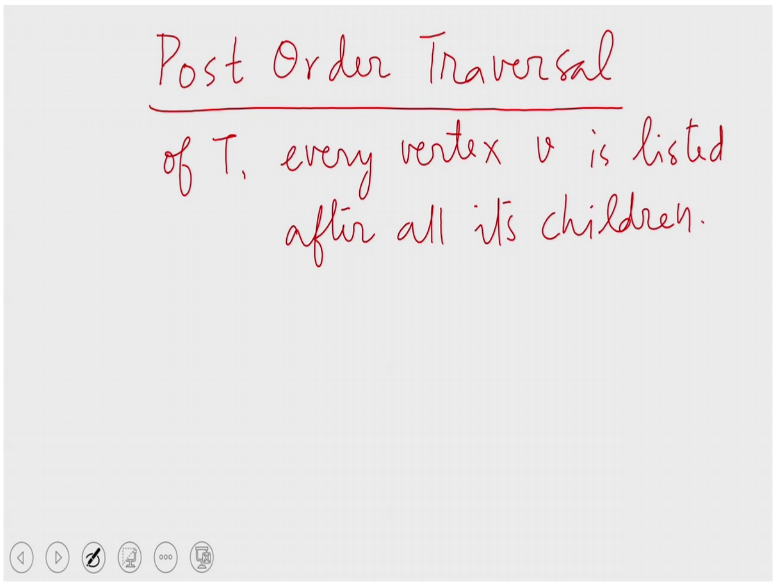
.

So, we find that we can find the preorder traversal can be computed in order of log n time, using n by log n processors on an EREW PRAM.

Most of these operations we discussed in the previous class. The only additional operations after computing the Euler circuit is copying the Euler circuit on to an array which can be done in order one time. And, then assigning the weights to the parent by parent child edges in the child parent edges in this manner parent, child edges will get a weight of 1, and the child parent edges will get a weight of 0, this can be done in order of one time as well if you have n processors. Once this is done commute the prefix sums using an optimal algorithm which can be run in order of log n time using n by log in processors.

And, then all we have to do is to look at the parent child edges and for each parent child edge take the rank and use that as the position number for the child in the pre order traversal. So, then we can fill in the array that represents the preorder traversal in order of 1 time using n processors. So, putting together if you have n by log n processors, then every single step can be executed in order of log in time and therefore, we have that result. So, that is about the pre order traversal.

(Refer Slide Time: 15:14)



Post Order Traversal

of T, every vertex v is listed after all its children.

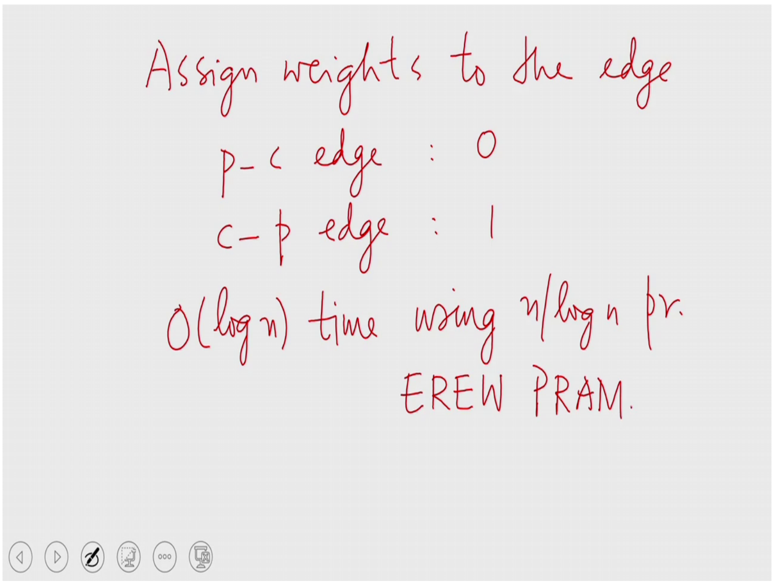
Similarly, we can also compute the post order traversal. In a post order traversal of a tree T, every vertex v is listed after all it is children are listed. So, in particular for our example the post order traversal would be going back to our diagram. So, the post order traversal will be where we list the vertex after all it is children have been listed. So, starting from 7 we go down the tree we go to let us say we go the same way as before, from 7 we go down to 5 from 5 we go down to 2 then 2 3 then 2 9. So, 9 does not have any children. So, we are now backtracking from 9.

So, once we are backtracking we list 9, then we go back to 3 3 does not have any more children we are backtracking from 3. So, we list 3 then we go back to 2 2 is not ready for backtracking yet because 3 has 2 has an unexplored child which is 4. So, we go to 4 and then immediately back tracked from 4, then 4 is listed. And, then 2 is now ready for backtracking we go from 2 to 5 and 2 is listed, and then once we are in 5 we go to 1 and then backtrack immediately so, 1 is listed.

Again we go back to 5 we come down to 6 and then to 10 and then backtrack 10 is listed, when 6 does not have any more children. So, 6 is now backtracking. So, we list 6 at 5 we have exhausted all it is children. So, 6 is of 5 is now backtracking. So, they can now list 5. At 7 we have a waiting child which is 8 so, 7 is not ready to finish it. So, we go to 8 8 backtracks 8 is listed and then finally, 7 is listed. So, the post order traversal of this tree is 9 3 4 2 1 10 6 5 8 and 7 I suppose to the pre order traversal which is shown on the right side.

So, now the question is how do we calculate the post order traversal?

(Refer Slide Time: 17:49)



Assign weights to the edge

p-c edge : 0

c-p edge : 1

$O(\log n)$ time using $n/\log n$ pr.

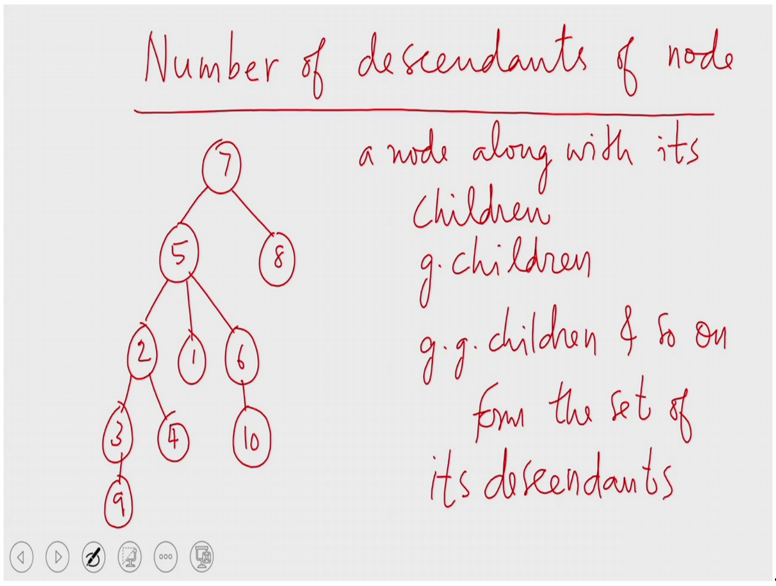
EREW PRAM.

This is easy to show we assign weights to the edges in the following fashion, we found that while going down the tree we are not listing the vertices. So, for every parent child edge, we have to assign the weight of 0, but then when we are backtracking through a child parent edge, we are now going to list the vertex. So, we are going to list the vertex while we are backtracking therefore, we have to assign a weight of 1 to every child parent edge.

After assigning weights like this, we do exactly as we did in the case of the preorder traversal we find the prefix sums of the weights. And, then the ranks that we get we have to use for every single child parent edge. And, then for every child parent edge, we have to take the rank of the edge as the ordinal value of the child. For example, when we are backtracking from 9 to 3 at this point the prefix sum will appear as one therefore, the rank of 1 will be assigned to vertex number 9.

So, I am leaving the working out of the example to you the correctness should now be clear it is the exact inverse of the pre order traversal case. So, in this case also we can argue that in order of log n time, using n by log n processors on an EREW PRAM, we can find the pre or post order traversal of a tree in so, that is the result we have. So, pre order traversal as well as the post order traversal can be found in order of log n time using n by log n processors.

(Refer Slide Time: 19:44)



Number of descendants of node

a node along with its children

g. children

g. g. children & so on

form the set of its descendants

.

He next thing is finding the number of descendants of the node. In our example the number of descendants the descendant of a node is the set of it is children grandchildren great grandchildren and so on. A node along with it is children, grandchildren, great grandchildren; and so on form the set of it is descendants. Or descendent of vertex v which is not the same as vertex v is called a proper descendant of vertex v. So, that set of the proper set of proper descendants will always be 1 smaller than the set of descendants and node itself is an improper descendant.

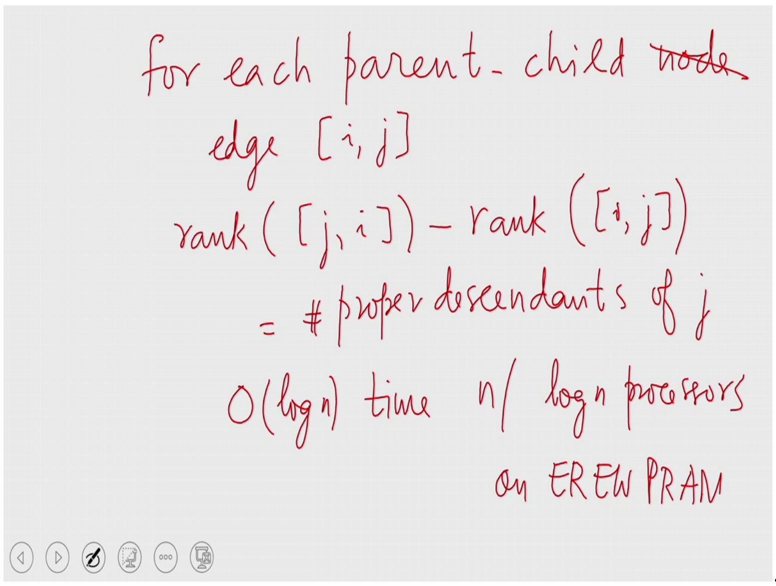
So, to find the set of all descendants we use the same prefix value that we had the prefix sums we had before, in the case of the preorder traversal that is we assign a weight of 1 to every parent child edge, and a weight of 0 to every child parent edge, and then compute the prefix sums. Now, in particular let us consider edges 75 and 57 here. These are the parent child edge the child parent edge respectively involving vertices 7 and 5.

So, we find that 7 5 has a rank of 1 and 5 7 has a rank of 8, the difference in their ranks is 7. This is the number of proper descendants that 1 has that 5 has. So, in this case the child 5 has 7 proper descendants. The number of proper descendents of 5 are 2 1 6 3 4 10 and 9 which is 7 elements. So, 5 has 7 proper descendants.

So, we consider the 2 edges 7 and 5. And, then the prefix sum values when we take the difference between the prefix sum values what we find are the total number of proper descendents of 5, because in the prefix sum values we are actually counting the order in which the pre order traversal visits them. Then, we get the total number of nodes that are visited before the traversal returns from 5 after visiting 5. So, 7 5 gives the rank of 5 and 5 7th ranked tells us how many nodes have been visited after 5, but before returning from 5 to 7.

So, that that is exactly the number of proper descendants that known 5 has. So, that immediately suggests an algorithm for finding the number of descendants of a node.

(Refer Slide Time: 23:54)



for each parent-child ~~node~~ edge $[i, j]$

$\text{rank}([j, i]) - \text{rank}([i, j])$

$= \#$ proper descendants of j

$O(\log n)$ time $n/\log n$ processors

on EREW PRAM

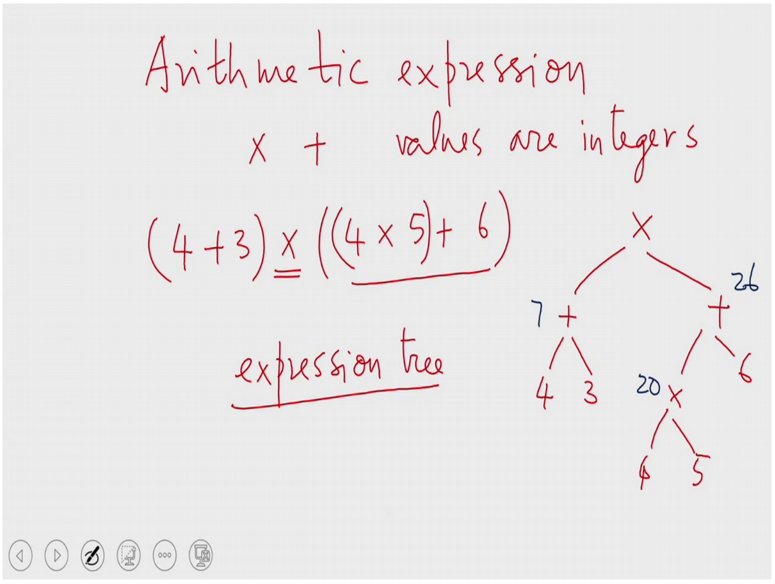
.

For each parent child node of each parent child edge i, j take the difference between the rank of j i, which is the twin of i, j and the rank of i, j this will give the number of proper descendants of g.

So, the number of descendants of j would be 1 more than this because j is also to be added to the set of proper descendants to form the set of all descendants. So, this can be done in order one time if you have n processors for every single parent child edge. So, this would assign the number of proper descendants for every single node other than the root and the number of proper descendants of the root of course, would be 1 less than the total number of nodes that are available.

So, this is an algorithm for finding the total number of proper descendants for every single node in the tree. So, as is clear this would run in order of log n time using n by log n processors on an EREW PRAM. So, all these problems on trees can be solved in order of log n time using n by log n processors.

(Refer Slide Time: 26:07)


Arithmetic expression

$\times \quad +$ values are integers

$(4+3) \underline{\underline{\times}} (\underline{(4 \times 5) + 6})$

<u>expression tree</u>

Now, let us consider a problem of a different queue. Let us say we are given an arithmetic expression for simplicity.

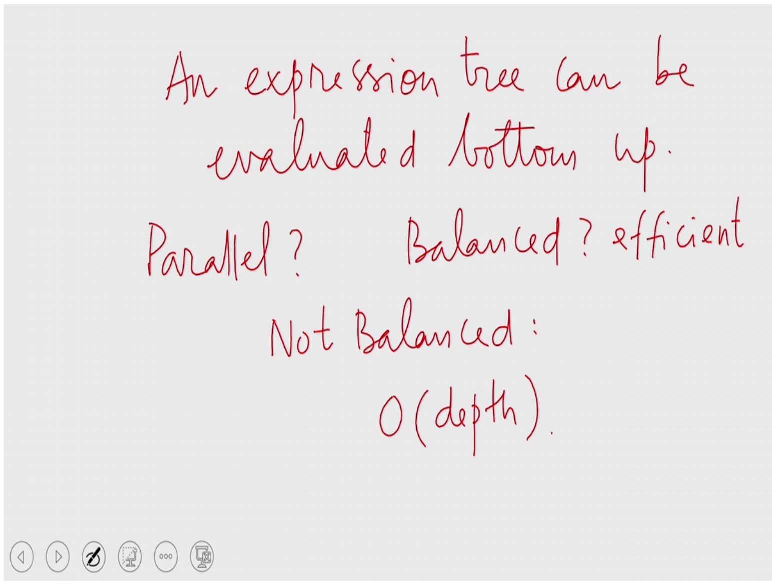
Let me assume that there are only 2 kinds of operations multiplication and addition. And, let us say the values are all integers of course, our discussion can be easily extended to arithmetic expressions of more complex form, but for now let me assume that these are the constraints available. And, then the arithmetic expression that is given to us which is of this form can be translated into a tree.

So, in this case the root has a multiplication and the root has 2 sub trees on the left side we have 4 plus 3 which can be written as a binary tree of this form plus at the root with 4 n tree on the leaves. And, here the natural precedence says that multiplication has to be performed first therefore, here we have a plus the plus is the last operation to be performed on the right side of plus we have 6, on the right on the left side of the plus we have 4 into 5.

So, given an arithmetic expression with our precedence rules we can convert the arithmetic expression in 2 or 3 of this form this is called an expression tree. So, given an arithmetic expression, we know how to convert that into an expression tree. Let us say we are given the expression tree and we need to evaluate this, the usual sequential algorithm to evaluate the expression tree is to reduce the tree bottom up 4 plus 3 is 7. So, this node evaluates to 7 4 into 5 is 20.

So, this node evaluates to 20 and then here we have 20 plus 6 26 and then at the root we have the product. So, this is how we evaluate a tree the tree can be evaluated bottom up.

(Refer Slide Time: 28:48)



An expression tree can be evaluated bottom up.

Parallel ? Balanced ? efficient

Not Balanced :

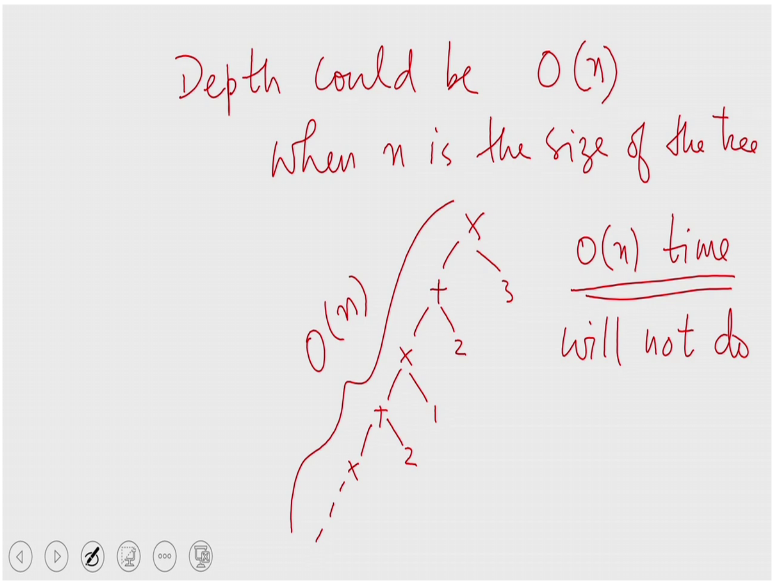
O(depth).

But, how would we do this in parallel? If, the tree were balanced then we could do exactly the same thing, if the tree were balanced we could start at the bottom of the tree and then every operation which is at the bottom of the tree can be evaluated simultaneously.

So, in this case 4 plus 3 equals 7 and 4 into 5 equal to 20. These 2 operations are independent of each other. So, both of these can be executed simultaneously. So, we will have a reduced tree with the 7 here and 20 here, then the next operation to be performed would be 20 plus 6 equals 26. And, then finally, that reduced tree is 7 into 26 which can be evaluated at 1 go. So, in 3 steps this tree can be evaluated.

So, if the tree is balanced then we have a way of efficient parallel execution. So, this bottom of bottom up approach works fine if the tree is balanced, but if the tree is not balanced, then the time taken by those parallel execution is the order of the depth of the tree which is problematic.

(Refer Slide Time: 30:38)

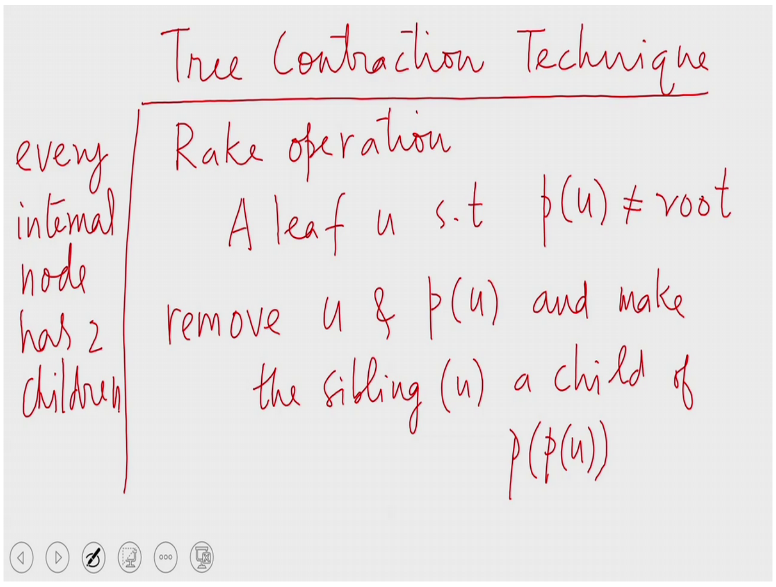


That is because the depth of a tree could be order of n where n is the size of the tree, this happens when the tree is very skewed.

For example, let us say we have a tree of this form a skewed tree of this form will have a part of length order of n where n is the total size of the tree. The total size of the tree is the total number of nodes in the tree. Now, this path itself has about half the number of nodes in the tree. Therefore, the time taken to evaluate this tree would be order of n, the number of process does not matter you might be able to optimize on the number of process, but still the algorithm runs in order of n time that certainly will not do. Because, the tree itself can be evaluated in evaluated sequentially in order of n time.

So, this is not an efficient parallelization. So, we have to device a technique for efficiently paralyze in this algorithm.

(Refer Slide Time: 32:28)



Tree Contraction Technique

every internal node has 2 children

Rake operation

A leaf u s.t $p(u) \neq root$

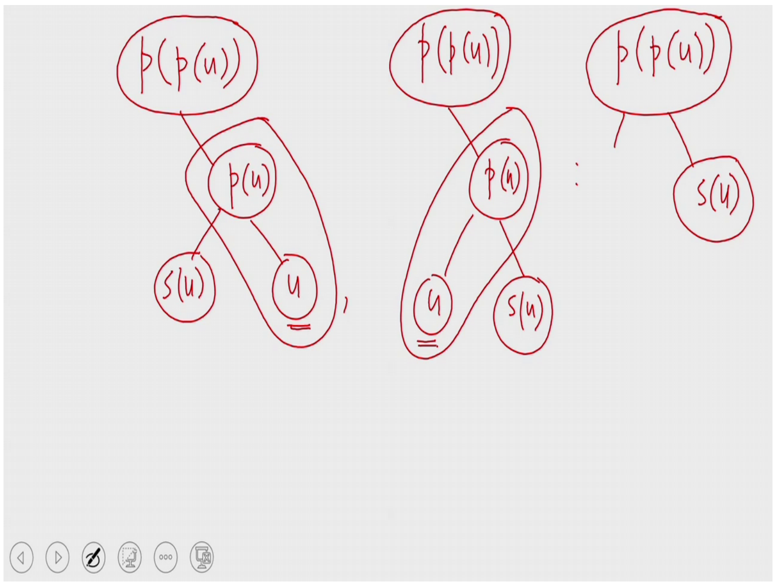
remove u & $p(u)$ and make the sibling (u) a child of $p(p(u))$

This we can do using the tree contraction technique. The tree contraction technique relies on what is called a rake operation. A rake operation is performed on a leaf u such that the parent of u is not the root of the tree.

So, when u is a node. So, that it is parent is not the root what we do is this, remove node u and it is parent and make the sibling of u, a child of the grandparent of u. So, here we have assumed that every node has a sibling. In other words every internal node in the tree has 2 children exactly.

So, we define the rake operation for trees of this form where every internal node has 2 children of course, the expression trees that we consider where every internal node is either a multiplication or a or an addition this condition is satisfied. So, the rake operation proceeds like this.

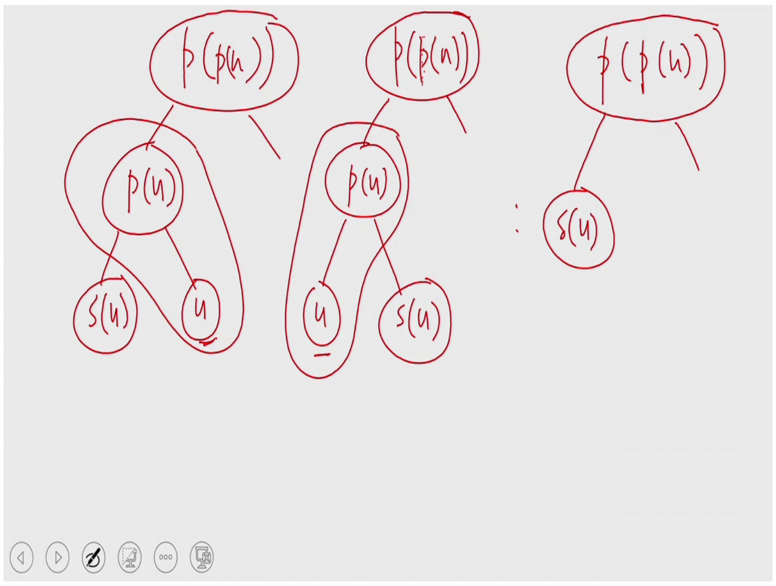
(Refer Slide Time: 34:31)



The various cases are these this is the grandparent of u and this is the parent of u. Suppose the parent of u is a right child of the grandparent S of u is let us say the sibling of u, and u happens to be the right child of p of u. The alternate case is where u is the left child of the parent of u. In both of these cases what we do is to remove u and p of u and make the sibling of u a child of the parent of grandparent of u from the same side that p of u held.

So, p of u was a right child of the grandparent of u before now p of u is replaced with s of u p of u and u are removed. So, here use a leaf. So, this is the case where the grandparent of u has p of u as a right child.

(Refer Slide Time: 36:07)

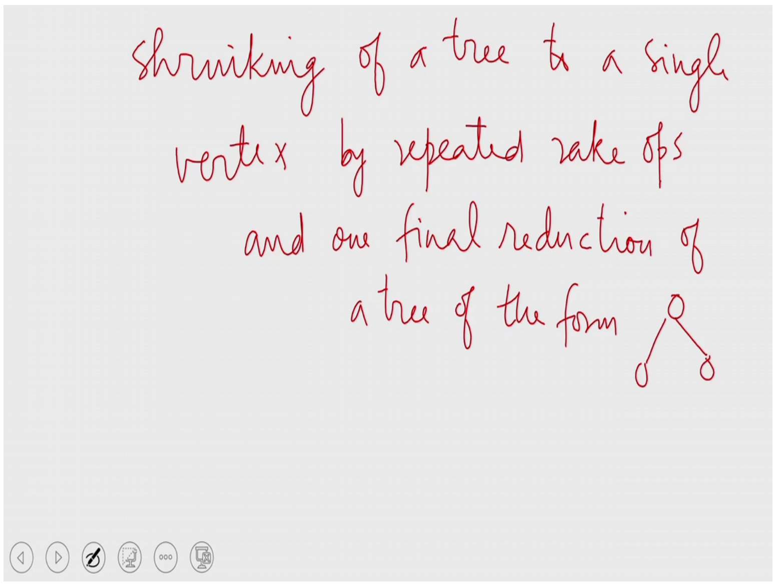


The other case is symmetric when the grandparent of u has p of u as a left child. Again there are 2 cases u could be either the right child or the left child of p of u. In both of these cases we remove p of u and u and make s of u a child of the grandparent of u from the same side in which p of u was it is child. So, in this case that is the left side.

So, the principle involved in all cases is the same when u is a leaf that has been chosen to be raked. And p of u is not the root, which means u has a grandparent what we do is that the node u and p of u are both removed from the tree. And the sibling of u s of u is made a child of the grandparent of u from the same side in which p of u was it is child.

So, in this case we consider that the case where p of u is a left child of the grandparent of u in the previous diagram we considered the case where p of u was a right child of the grandparent of u. So, this operation is what we call a rake operation.

(Refer Slide Time: 37:55)

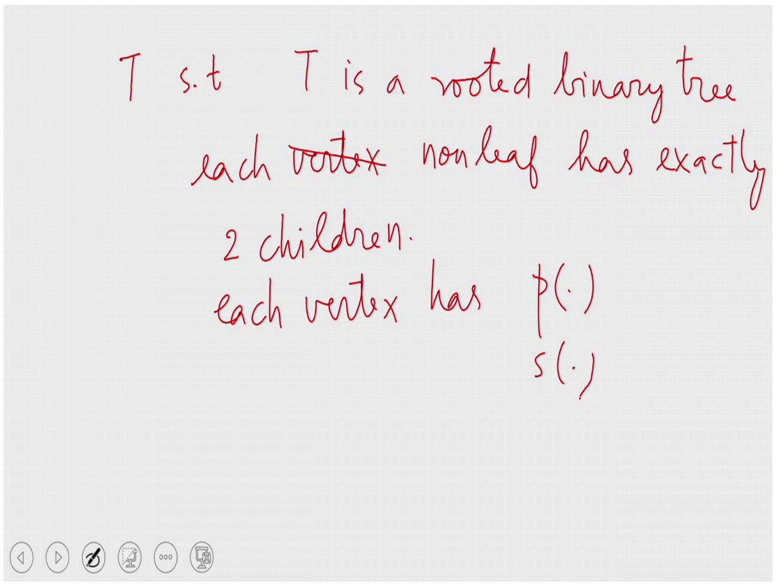


Shrinking of a tree to a single vertex by repeated rake ops and one final reduction of a tree of the form

By tree contraction we mean the shrinking of a tree to a single vertex, by repeated rake operations and one final reduction of a tree of the form, where the root has exactly 2 children which are both leaves.

So, the tree contraction algorithm that we have is this given the tree satisfying these conditions.

(Refer Slide Time: 38:59)



T s.t T is a rooted binary tree

each ~~vertex~~ non leaf has exactly 2 children.

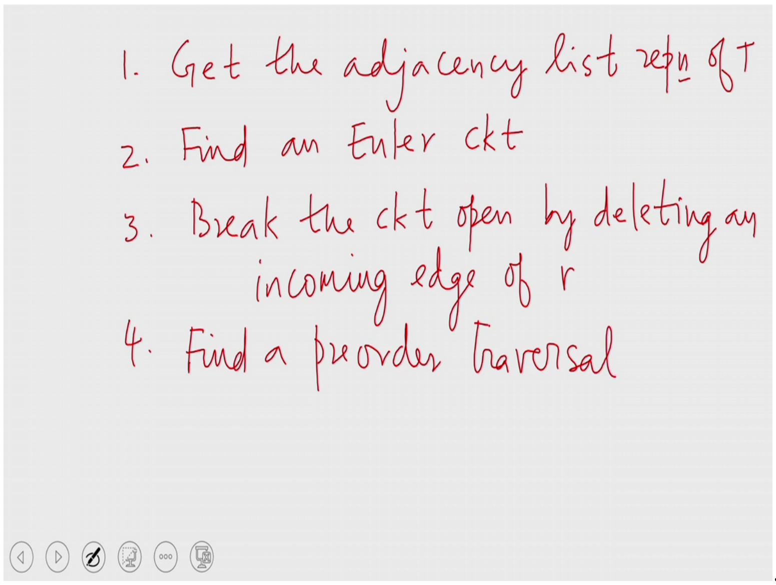
each vertex has $p(\cdot)$

$s(\cdot)$

So, we are given a tree T such that T is a rooted binary tree as in expression trees each vertex, each non leaf, has exactly 2 children. And each vertex has 2 pointers the parent pointer and sibling point. So, such a tree is what is given to us and we want to contract the tree through repeated break operations. So, this is the challenge that we have.

So, the algorithm for T contraction proceeds in this fashion.

(Refer Slide Time: 40:03)

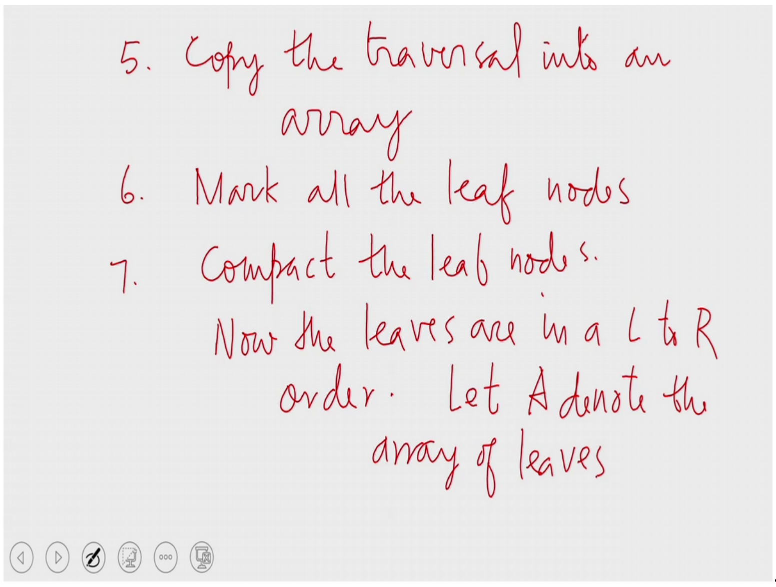


1. Get the adjacency list repn of T
2. Find an Euler ckt
3. Break the ckt open by deleting any incoming edge of r
4. Find a preorder traversal

.

First we get the adjacency list representation of tree and then we find an Euler circuit, we break the Euler circuit open by deleting an incoming edge to the root. And, then we find a preorder traversal all these are familiar operations all these we have done in the other algorithms.

(Refer Slide Time: 41:18)



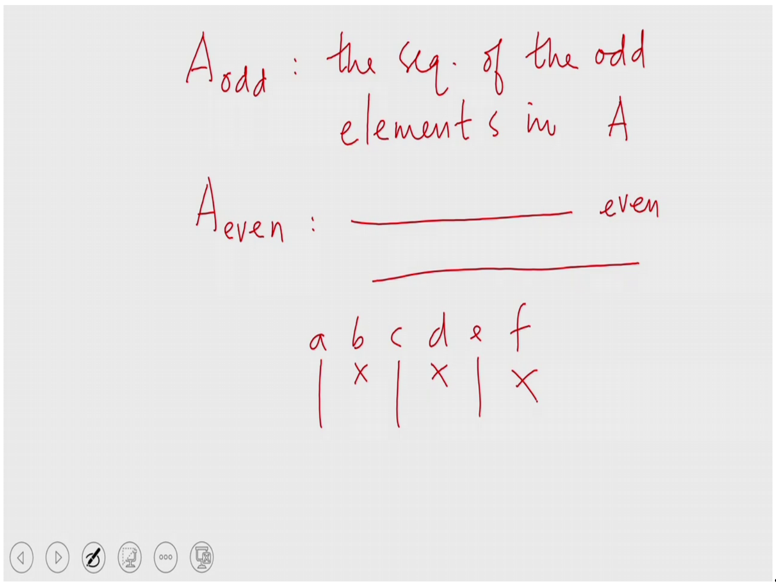
5. Copy the traversal into an array
6. Mark all the leaf nodes
7. Compact the leaf nodes.
Now the leaves are in a L to R order. Let A denote the array of leaves

.

And, then this traversal is copied into an array which can be done in order one time, if you have one pro super element. So, in this array the logical order of the traversal is identical to it is physical order and then we mark all the leaf nodes. And, then essentially we delete all non leaves, leaving only the leaf nodes, but the leaf nodes need not be consecutive in the traversal, but then we make them consecutive by compacting the leaf nodes.

Now, the leaf nodes are in a left to right order. Let a denote the array of leaves. So, we have managed to get all the leaves of the tree in the left to right order in an array, we call this array A.

(Refer Slide Time: 43:04)



A_{odd} : the seq. of the odd elements in A

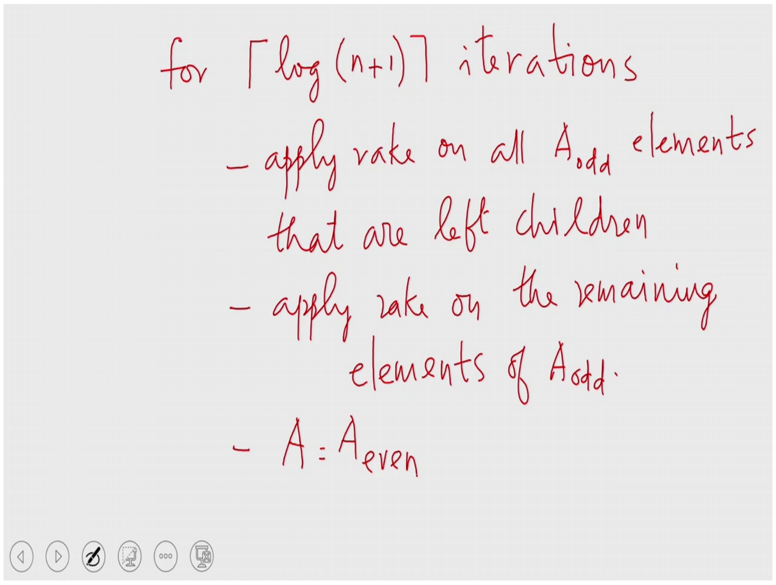
A_{even} : ________ even

$$\begin{array}{cccccc} a & b & c & d & e & f \\ | & \times & | & \times & | & \times \end{array}$$

Then from this A we define 2 sequences, A odd is the sequence of the odd elements in A. Similarly, A even this defined us the sequence of the even elements in A. We define A odd and A even in this manner the sequence of the order elements in A and the even elements in A.

Fact for example; in a sequence of this form a b c d e f, a c and e from the odd elements, b d and f from the even elements. So, this way we can convert an array into the odd sequence in the even sequence.

(Refer Slide Time: 44:00)



for $\lceil \log(n+1) \rceil$ iterations

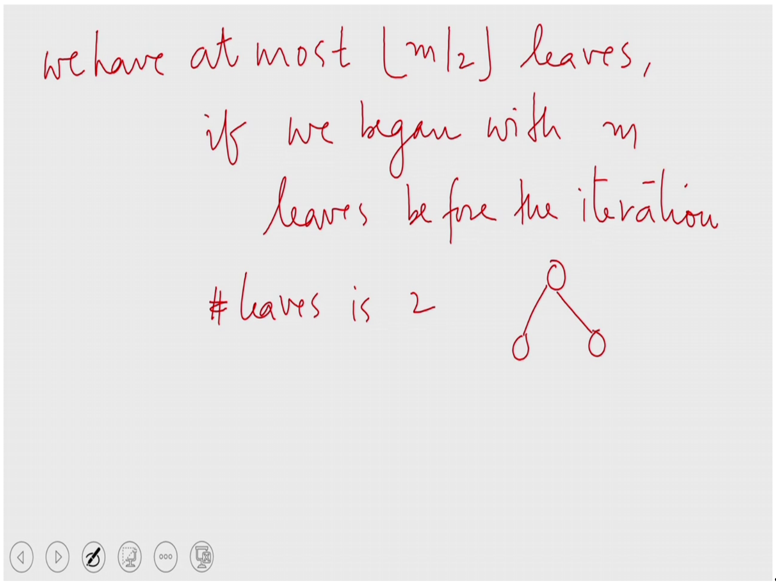
- apply rake on all A_{odd} elements that are left children
- apply rake on the remaining elements of A_{odd}.
- $A := A_{even}$

And, then the algorithm proceeds to the second phase. In the second phase of the algorithm what we do is this for ceiling of log of n plus 1 iterations, where n is the total number of nodes in the tree or the total number of leaves. We apply in rake on all A odd elements that is all odd numbered leaves, that are left children, that are left children of their respective parents.

After this we apply rake on all the remaining elements of A odd. So, after performing rake on all A odd elements in this fashion they have all vanished from the array. Now all that remain in the array are the even elements. So, we redefine A as A evens and then continue with the algorithm. So, here what we have to observe is that rake is performed on a set of vertices. So, that their parents are non-adjacent, which means 2 rake operations being performed simultaneously will not interfere with each other.

Therefore all these nodes along with their parents vanish from the tree. So, if we are beginning with the n plus 1 know beginning with n leaf nodes, then after ceiling of n plus 1 iterations we will be left with at most 2 leaf nodes. Once, we have annihilated all A odd elements from the tree.

(Refer Slide Time: 46:05)



we have at most $\lfloor m/2 \rfloor$ leaves,
if we began with m
leaves before the iteration

leaves is 2

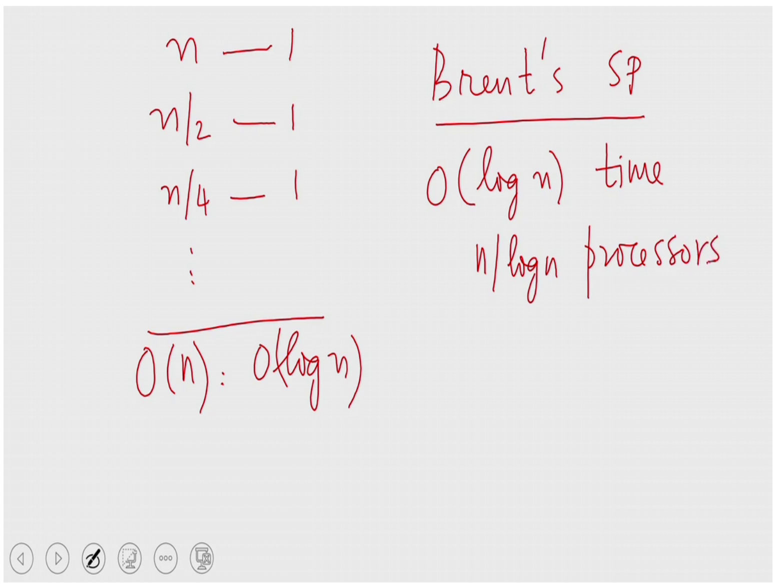
.

We have at most floor of m by 2 leaves left in the tree after the iteration, if we began with m leaves before the iteration.

What it means is that every iteration that reduces the number of leaf nodes by a factor of 2. So, we have log n plus 1 iterations by which the total number of leaves has reduced to 2. So, the number of leaves in the tree now is at most 2, that is the tree now looks like this after log n plus 1 iteration. And, the cost of the iterations as you can see in the first item in the first iteration, we have to get rid of all A odd elements there are n by 2 A odd elements.

So, there are so, many rake operations to be performed we know that if we have one processor per week operation the operation can be performed in order one time. So, assuming that we have n processors, the first step can be executed in order 1 time. In the second iteration the number of rake operation house therefore, we require half as many processors.

(Refer Slide Time: 47:42)



So, with n processors the first step can be executed in 1 time with n by 2 processors the second step can be executed in 1 time, with n by 4 processors the third step can be executed in unit time and so on.

And, there are log n such steps or the log n such steps and the total cost of all the steps put together as order of n again. Therefore, we can use Brent's scheduling principle using Brent's scheduling principle we will be able to execute the algorithm in order of log n time, using n by log n processors. This is about the second phase of the algorithm, but the first phase of the algorithm is already familiar to you all these steps we have already executed in the previous algorithms.

Therefore, we know that all these steps in the first phase of the phase of the algorithm these 7 steps in the first phase of the algorithm can be executed in order of log n time with n by log n processors. Therefore, putting everything together the tree contraction algorithm runs in order of log n time using n by log n processors. Now, the question is how we will we use the tree contraction algorithm for evaluating an expression which we shall see in the next lecture. So, that is it from this lecture hope to see you in the next lecture.

Thank you.