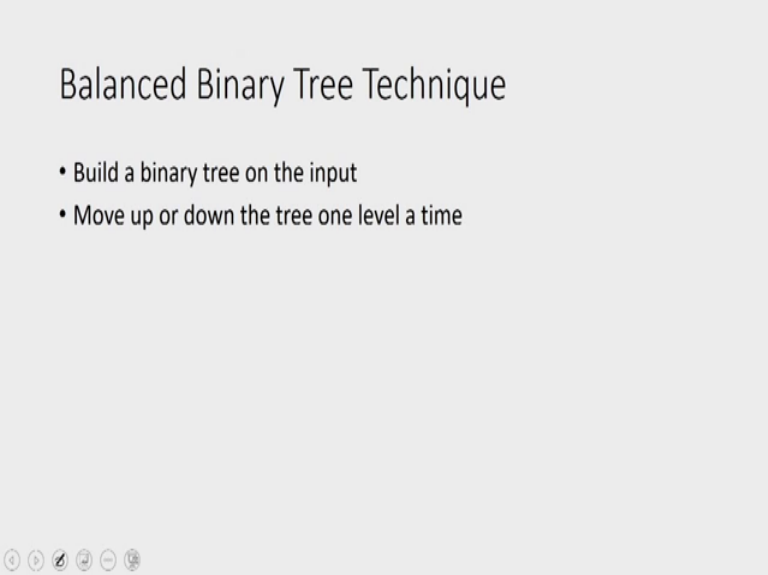
Parallel Algorithms
Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 06
Basic Techniques 2

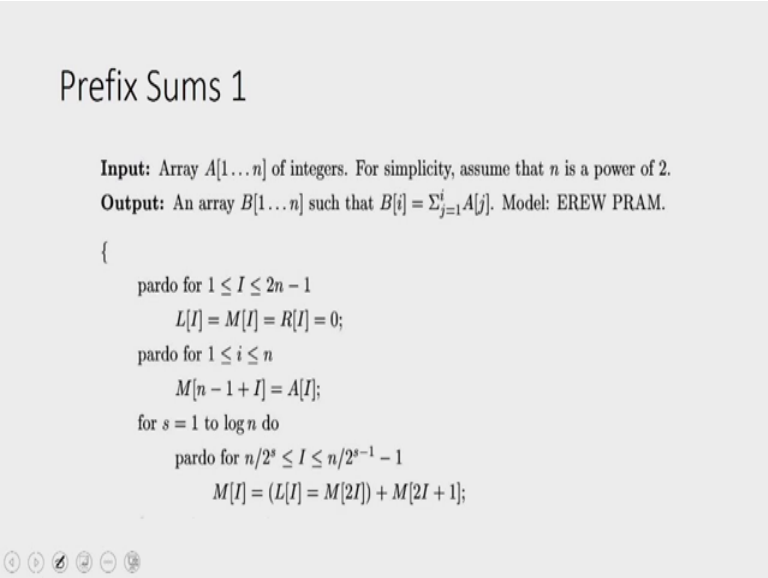
Welcome to the sixth lecture of the NPTEL MOOC on parallel algorithms. In the previous lecture, we had started discussing algorithm design techniques in particular, we talked about balanced binary tree techniques.

(Refer Slide Time: 00:46).



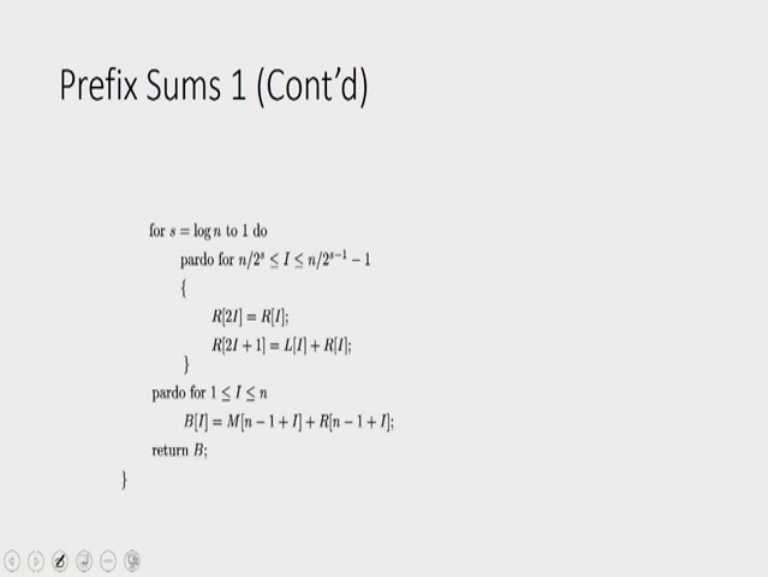
In balanced binary tree technique, we build a binary tree on the given input and then solved the problem by moving up and down the tree.

(Refer Slide Time: 00:54).



In the last class we had seen a couple of algorithms of this sort; one of them was for finding the OR of n bits and the other one was for finding the prefix sums of an array of n integers. And then we had started discussing the applications of the prefix algorithms.

(Refer Slide Time: 01:06).

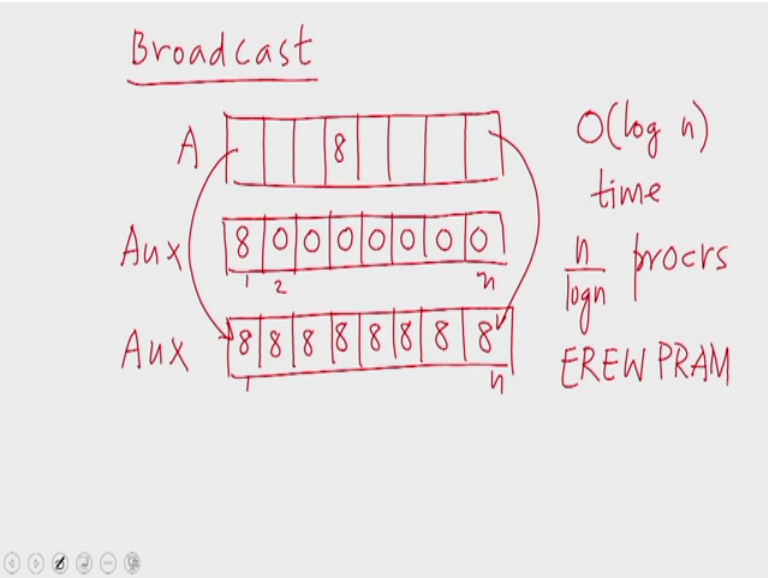


(Refer Slide Time: 01:11).

In particular, we had seen the compaction problem, where you are, you have an array, in which only some of the elements are useful elements and then you want to compact the useful elements into consecutive locations.

Starting from the left end we saw how to do this using prefix sums, at for the same time complexity, that is order of log n time using n by log n processors on an EREW PRAM. Now, let us see another application of the prefix sums problem. This is the broadcast problem.

(Refer Slide Time: 01:41).



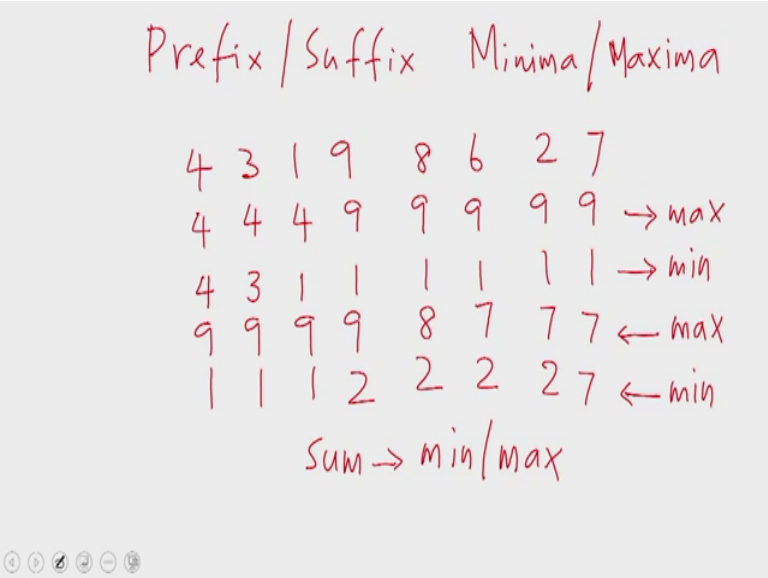
Here, we have an array, in which one particular element holds a value, let us say the fourth element in this array, holds a value 8. This value has to be broadcast to the other elements in the array. So, to do the broadcast, what we could do is this, take an auxiliary array of the same size and the element that wants to do the broadcast will write the value to be broadcast in the first location. In every other location the corresponding processors will enter a value of 0, that is processors, 2 through n will end the values 0 in locations 2 through n.

The processor of which wants to do the broadcast, separately will write the broadcast value in the first location. Now, if we invoke the prefix sums, algorithm on this array, that is on the, in the auxiliary array. The array fills up in this fashion. The prefix sums of the present contents of the auxiliary array, will be to distribute this value over the entire array.

Now, the value 8 has been broadcast from all locations 1 to n. So, the original processors which was seated in the given array A, can copy the value from the corresponding location of the auxiliary array without any concurrent read, that is the first element here, can see that the broadcast value is 8 from the first location, the nth processor here can see that the broadcast value is 8 from the nth location and so on. So, there does not seem to be any concurrent read.

So, the time complexity is identical to that of prefix sums. This runs in order of log n time, using n by log n processors on any EREW PRAM. Another application of the prefix sum problem or essentially the prefix sums algorithm is to solve a related problem, which is that of solving the Prefix or Suffix Minima or Maxima.

(Refer Slide Time: 04:51).



In the, in the case of the prefix maximum problem, let us say, we are given an array that contains 4 3 1 9 8 6 2 7 in that order. Let us say, we want to find the prefix maxima of this array, to find the prefix maxima, we start from the left end and find the largest element seen (Refer Time: 05:29) at each position.

So, the prefix maxima for this array would be 4 in the first location, again 4 in the second location, because 4 is a larger of 4 and 3, again 4 in the third location, because 4 is the largest of 4 3 and 1, but in the fourth location, it will be 9 and thereafter at every single location it will be 9. 9 is the largest of this array. Similarly, if you compute prefix minima of the same array, you have to find the smallest element from the left end till that point.

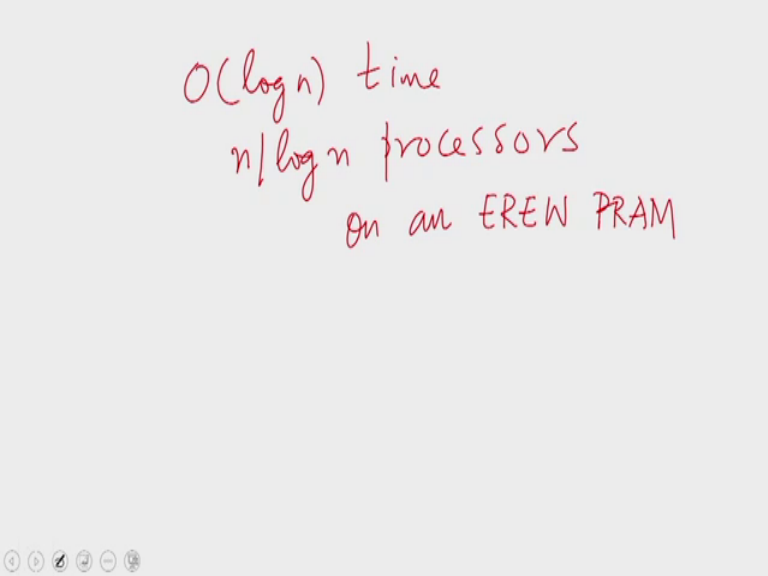
So, if the first location we again have 4 at the second location, we have to write the smaller of 4 and 3, which is 3 at the third location. We have 1 at the fourth location, we have 1 again and then 1 at every single location, because 1 is the smallest in this array. So, these are the prefix maxima and prefix minima of the given array, if you were to compute the prefix maxima and minima from the right side, we get what are called the suffix minima and maxima of the given array.

So, the suffix maxima would be computing prefix maxima from the right end. So, when you start from the right end, the rightmost prefix maximum will be 7, at the next location, you have to enter the maximum of 2 n 7, which is 7, then the largest of 6 2 and 7, which is 7 again then the largest of 8 6 2 and 7, which is 8 then 9 and then 9 in all the

remaining locations. Similarly, if you do a suffix minimum starting from the right end of the prefix minimum starting from the right end, we do get suffix minima of the given array.

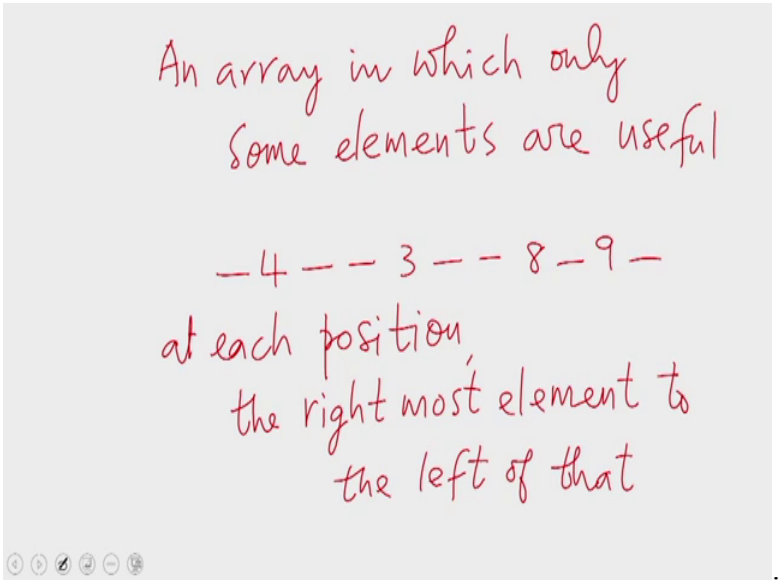
So, the suffix minima of the array would be 7 here, smaller of 2 and 7, which is 2 smaller of 6 2 and 7, which is 2 again, 2 here again, 2 here again, but since 1 is smaller than 2 here, it will be 1 and then it will be 1 at the remaining locations. So, the suffix minima of the array would be these. In the array, when you replace some with minimum or maximum as the case maybe, you would get an algorithm for prefix minimum or maximum or suffix minimum and suffix maximum.

(Refer Slide Time: 08:10).



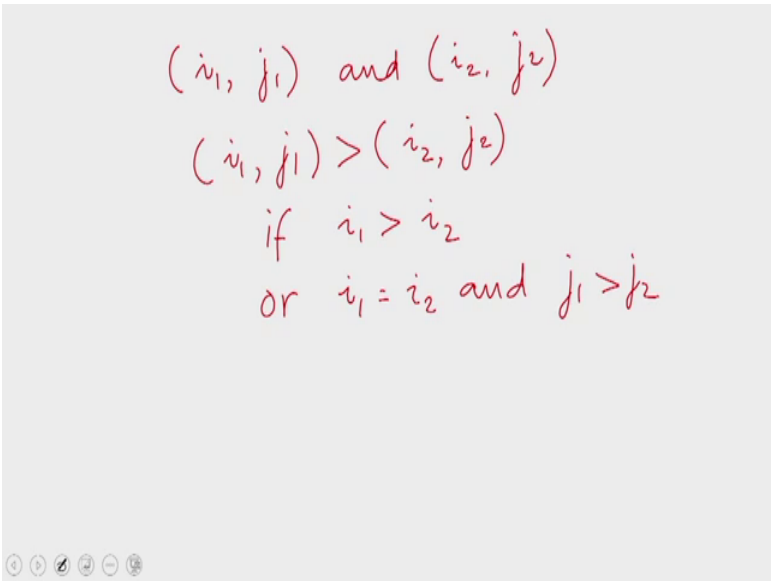
So, all these algorithms run in order of log n time, using n by log n processors on an EREW PRAM. Now, let us see an a problem, which can be solved using prefix minima. Let us say, we are given an array in which only some elements are useful like here.

(Refer Slide Time: 08:49)



So, this array has only four useful elements, the remaining locations are all empty. Let us say at each position in the array, we want to find the rightmost element to the left of that position. So, this is the problem that we want to solve. Given an array of size n, in which only some of the elements are useful. At each position in the array we want to find the rightmost element to the left of it. So, let us see how we can solve this.

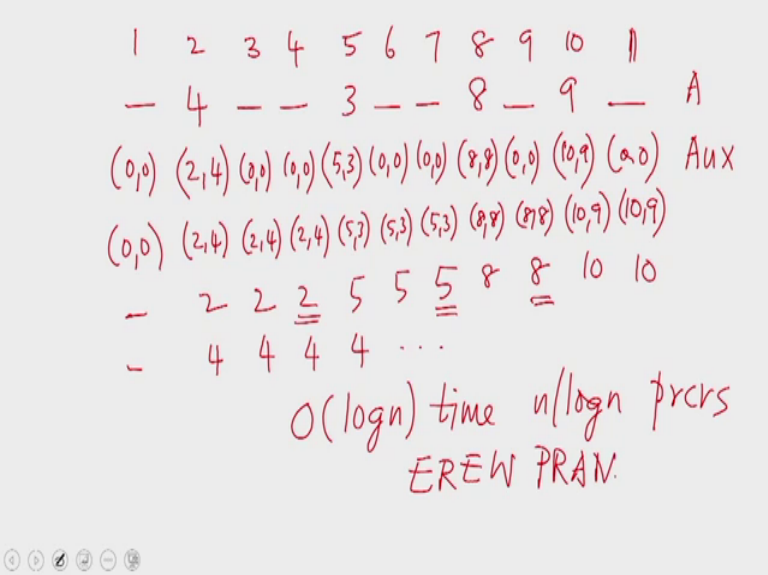
(Refer Slide Time: 10:17).



Let us say we have given we are given two ordered pairs i 1 j 1 and i 2 j 2. We say that ordered pair i 1 j 1 is greater than the ordered pair i 2 j 2, if i 1 greater than i 2 or i 1 is

equal to i 2, but j 1 is greater than j 2. So, using this definition we can compare two ordered pairs. So, let us say for the given array we form ordered pairs in this fashion.

(Refer Slide Time: 11:10).



Let us say, we are given this array, the same one that we saw before. This is the instance of the problem that we want to solve, let me also write the indices; this is the first location, this is the second location and so on. This is an array of size 11. Let me replace each element in the array with an ordered pair. Here, I will replace 4 with 2 4, because 4 is occurring at the second position. Here, I will replace 3 with 5 3, here I will replace 8 with 8 8 and here I will replace this with 10 9. At every other location, where an element does not appear.

Let me introduce the ordered pair 0 0. So, this is the given array A and I have formed an auxiliary array in this fashion. Now, if I find the prefix maximum of this array from the left end using the maximum operation on ordered pairs, what I get is this. We find that at each position we now know, the index of the rightmost element to the left, that is at the first position we know that there is no element to the left of it, at the second position we know that there is nothing to the left of it and this is indeed the rightmost element to the left.

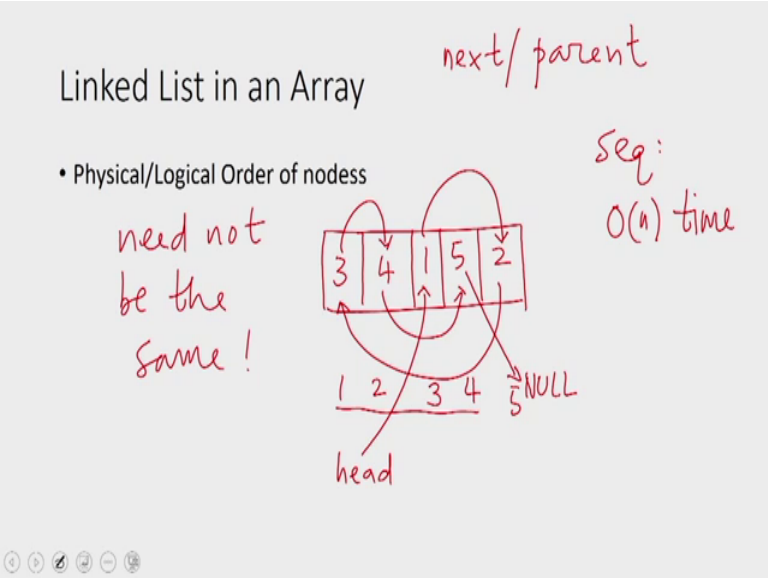
So, the second element knows that there is nothing to the left of it and this is the first element in the array, the fifth element knows that the rightmost element to the left of it is the value, which is in the neighbor, which is 2. Similarly, the eighth element knows that

the rightmost element to the left of it is the value, which is in the neighbor of it, which has 5 and the 9, the tenth element knows that the rightmost element to the left of it is, is in position 8. So, copying these values, all these positions will now know that there is nothing to the left of 1 and the rightmost element to the left of 2 is 4, 4 here, 4 here and here it is 4 again and so on.

So, in this fashion filling the auxiliary array, we find that at each position now, we know the rightmost element to the left of that position. All we did was to form the ordered pairs and invoke the prefix sums algorithm that we already know. So, this also runs in order of log n time, using n by log n processors on any EREW PRAM. So, those were some algorithms using balanced binary trees. The next algorithm design technique is called Pointer Jumping. Pointer Jumping is a technique that is used primarily on linked lists.

(Refer Slide Time: 15:25).

(Refer Slide Time: 15:35).

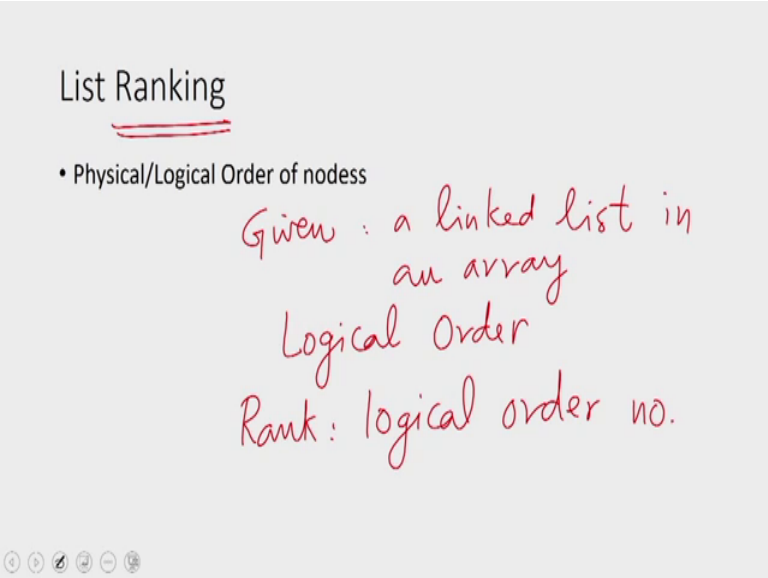


So, let us say we are given a linked list in an array, what I mean is this. Let us say, we have an array in which every node has a next pointer or a parent pointer. A linked list has been stored in this array; the first node in the linked list can be accessed through a pointer called head. So, head points to the first node in the linked list. Let us say, this is the parent pointer from the head node, this node in turn points to this node, which in turn points to this and let us say it finally, points to this and the fourth node in the list, in the, in the array has no parent pointer or in other words, it has a null pointer for it is parent.

So, this is the given linked list, in this linked list, the order in which the elements are arranged in the array, forms the physical ordering. So, this is the first way node in the physical order, this is the second node in the physical order, this is the third in the physical order, this is the fourth and this is the fifth. So, starting from the left 1 2 3 4 5 are the physical orders, but the logical order is given by the pointers. The head pointer takes you to the first node in the list, from there the parent pointer will take you to the second node in the list, from there the parent pointer will take you to the third node in the list and so on.

So, what is now written inside the array are the logical order values and what is written outside are the physical order values. So, as you can see the physical order need not be the same as the logical order, the physical order of a linked list need not be the same as the logical order.

(Refer Slide Time: 18:11).

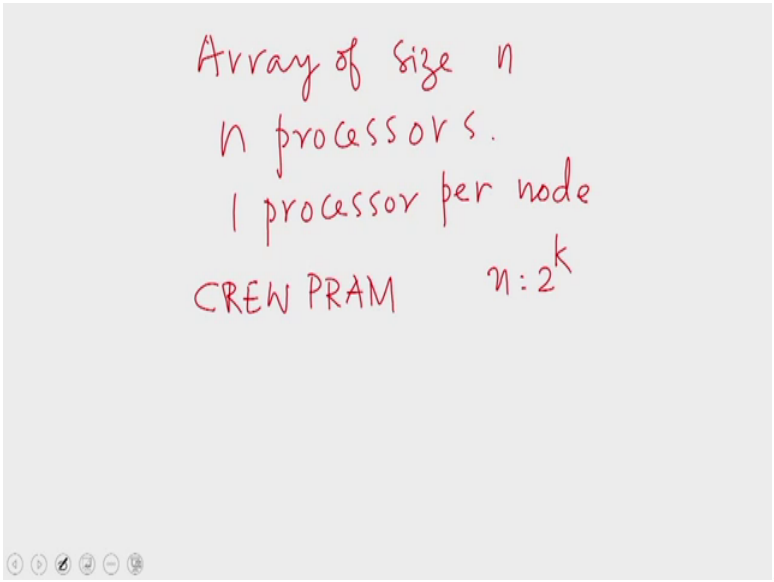


We consider the problem of list ranking, that is we are given a linked list in an array, what is needed to find is the logical order that is we have to say, which of the nodes is the first one, which of the nodes of the second one and so on, which is as good as ranking the nodes. Rank of a node is the logical order number.

So, in our example the third node in the physical order, happens to be the first node in the logical order, the fifth node in the physical order, happens to be the second node in the logical order. So, we say the ranks of the nodes are 3 4 1 5 2 from the left end. So, what is needed is to find the ranks. In the sequential setting, this problem would seem to be very easy, all you have to do is to trace the head pointer goes to go to the first node. This node will be ranked one and then trace the parent pointer go to the second node and give it a rank of 2 and so on.

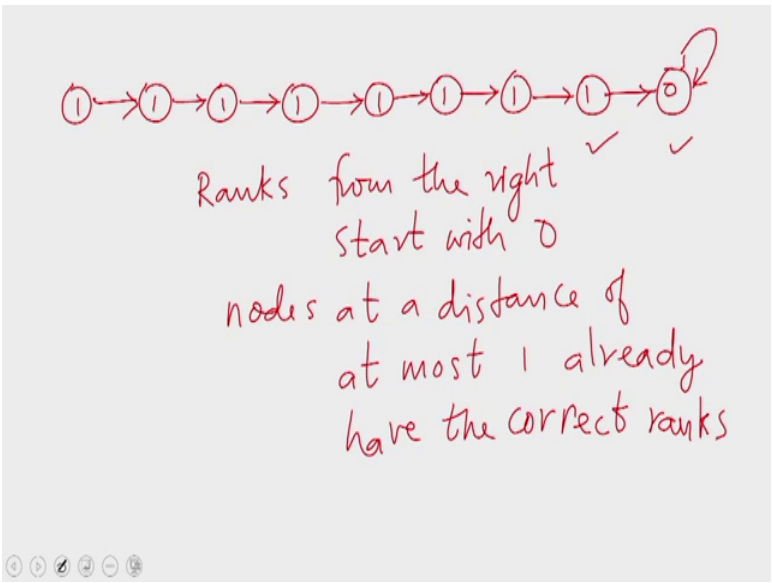
When you run through the list, you exit the list, by that time all the nodes in the list would have been ranked. So, the time taken would be order of n, which means when you design a parallel algorithm, you would expect the cost of the algorithm to be order n is the algorithm is to be optimal, but that is a challenge indeed. So, this is the problem that we attempt to solve.

(Refer Slide Time: 20:06)



So, let us say, we are given an array of size n and we have n processors to begin with we assume that there are n processors, that is one processor per node in the list. So, let us assume for now, let me assume that the model is CREW PRAM, that is you can have concurrent reads, but exclusive writes are expected and let me also assume that n is a power of 2 for easiness of exposition. You can easily see how the algorithm can be generalized for non powers of 2. So, let me consider the logical diagram of a list. All the pointers are, let us say flowing from the left to right, left to the right and the last node, let us say points to itself.

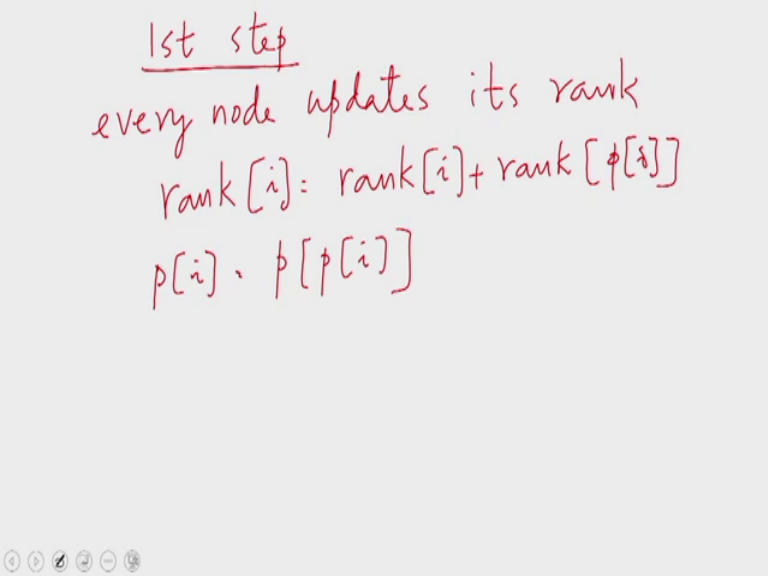
(Refer Slide Time: 21:01).



So, the last node in the list is the only node, which points to itself. Therefore, you can uniquely distinguish the last node by checking for this condition, if the parent pointer is to itself then that node is the last node. Let us say we initialize the rank values of the nodes in this fashion. The last node gets a rank value of 0, every other node gets an initial rank value of 1.

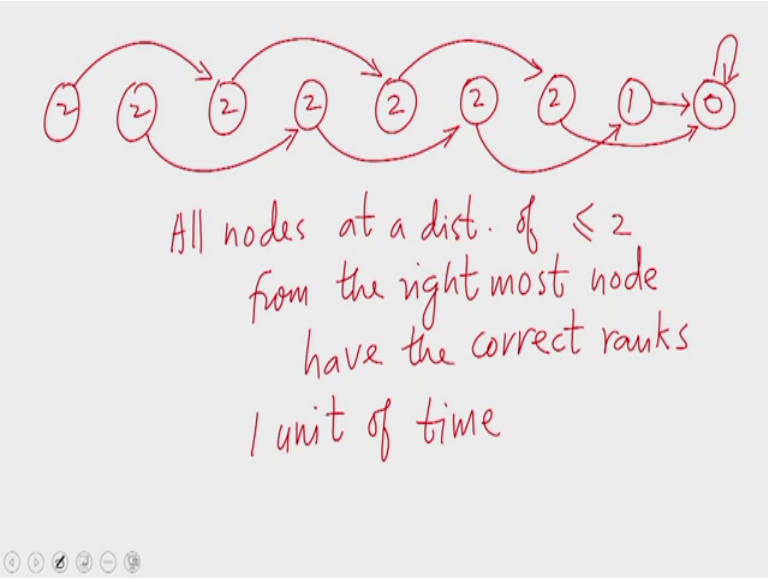
So, you can now see that the last node as well as the next node, the node to the left of it have the correct ranks, if the ranks are computed from the right end and we also assume that the ranks start with 0. So, the rightmost node is going to be given a rank of 0, the next node is going to be given a rank of 1 and then the next node will be given a rank of 2 from that perspective the last two nodes already have the correct ranks. So, we can say that nodes at a distance of at most one already have the correct ranks. So, once again the initial ranks are assigned in this fashion. The last node in the list that is the rightmost node in the list is given a rank of 0. Every other node is given a rank of 1, then the algorithm proceeds in this fashion, in the first step every node updates its rank.

(Refer Slide Time: 23:20).



So, what the node does is to update the rank in this fashions, the rank of a node i is its present rank plus the rank of its parent. After updating the ranks of every single node in this fashion what we do is this, we replace the parent pointer with a pointer to the grandparent of this node. That is every single node will adopt the grandparent as the parent. So, let us see what happens to the logical diagram of the list now, if you do this.

(Refer Slide Time: 24:26).



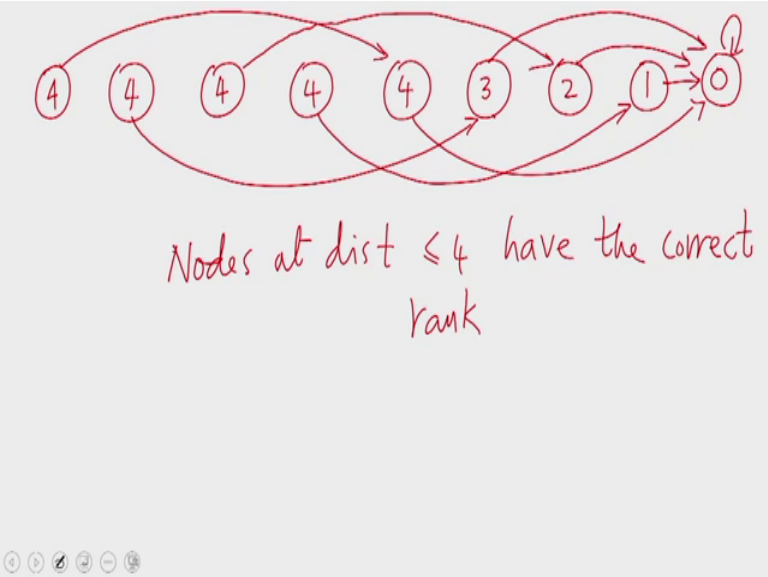
The last node continues to point to itself, its parent does itself therefore, its grandparent is also itself therefore, the pointer reallocation will not make any difference to it, its initial rank was 0, it continues to be 0. The node to the left of it was pointing to the last node, its initial rank was 1 and its present rank will also continue to be 1 whereas, this node which was pointing to the node to the, node to its right will have its rank updated to 2. Its previous rank was 1 and it was pointing to a node of rank 1. This 1 plus 1 will be stored as the new rank of this node and it is pointer, the parent pointer will now be to the parents parent. So, it will end up pointing to the rightmost node.

At every other node, we find that the rank values updated to 2. This is, because all these nodes were pointing to nodes of rank 1, the node of the, the rank of the parent will be added to the rank of the node itself. Therefore, all these nodes will now, have a rank of 2 and all of them will adopt the grandparent as the new parent, the earlier grandparent will now be the new parent. So, this is how the parent, the pointers will look like at the end of the second step.

So, here we find that all nodes at a distance of at most 2 from the last node have correct ranks node and we have taken only one step to achieve this that is, because all the processes are operating simultaneously. There is one processor located at every single, every single node in the list and all the nodes are updated simultaneously. So, at the end

of the first step, this is what the list look like, it looks like. Then in the second step we repeat the process.

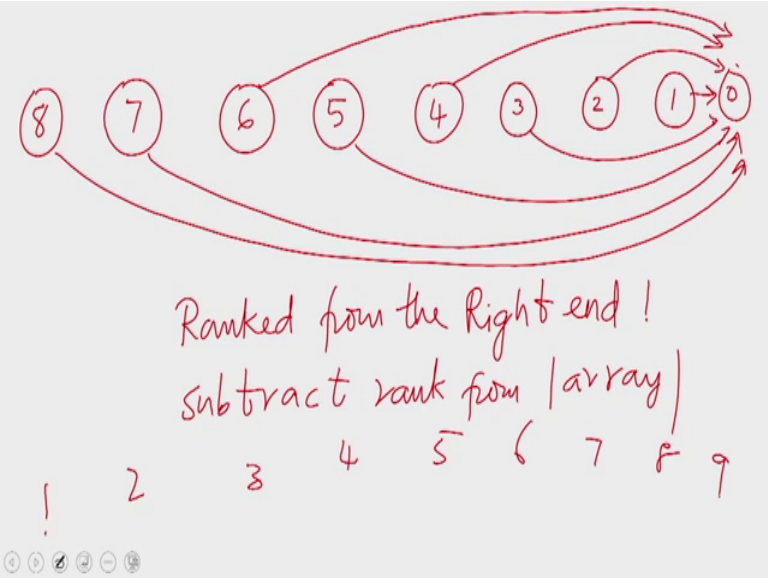
(Refer Slide Time: 27:21).



So, here there is no change to the rightmost 3 nodes. They will have ranks of 0 1 and 2, but the fourth node from the right had a rank of 2 before and it was pointing to a node of rank 1, which is the second rightmost node here. Therefore, now, it will be pointing to the last node and will be holding a rank of 3, the node to the left of it had a rank of 2 before and it was pointing to a node of rank 2. Therefore, its rank will be updated to 4 and it will end up pointing to the last node.

Node rank 2 will be pointing to the last node as well. So the node, the sixth note from the right will also have a rank of 4 and so do the nodes to the left of them and they would all be pointing to the previous grandparents now. So, once again we find that nodes at a distance of 4 or less have the correct ranks now. That is from the previous step, the number of nodes with the correct ranks has doubled.

(Refer Slide Time: 29:19)



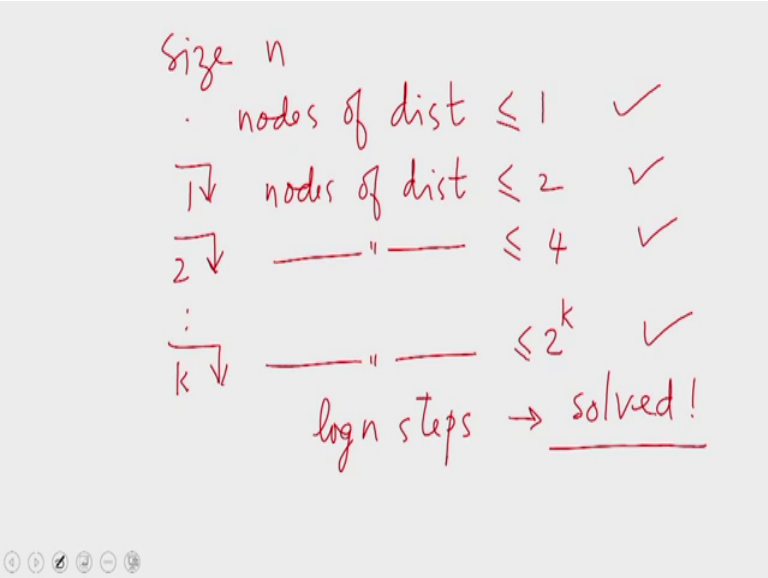
Now, going on to the final step, for this particular list. This list has at most 9 nodes. Now, we find that these nodes do not, the nodes with rank 4 or less at the right end. Do not change their ranks, but all the remaining nodes will now, upgrade their ranks like this. The node to the left of it will get a new. We will get a rank of 5, because at the end of the previous step it was pointing to a node of rank 1 and node, the node to the left of it will get a rank of 6, because it had a rank of 4 and it was pointing to a node of rank 2.

Here, the rank will be 7, because it had a rank of 4 and it was pointing to a rank of, node of rank 3 and the right left most node will get a rank of 8, because it had a rank of 4 before and it was pointing to a node of rank 4.

Now, in, at the end of the previous step every node had the left most node as its grandparent. Therefore, now at the end of the third step, every node will end up pointing to the last node. So, at this point we can say that the list has been ranked from the right end, but the ranks start from 0, if you want the ranks to start from 1 and also the left end all you have to do is to subtract, the found ranks from the value of, from the size of the array.

So, in this case the array has 9 elements. So, if you subtract the values, you find that the left most node has a rank of 1, the second left most node has a rank of 2 and so on.

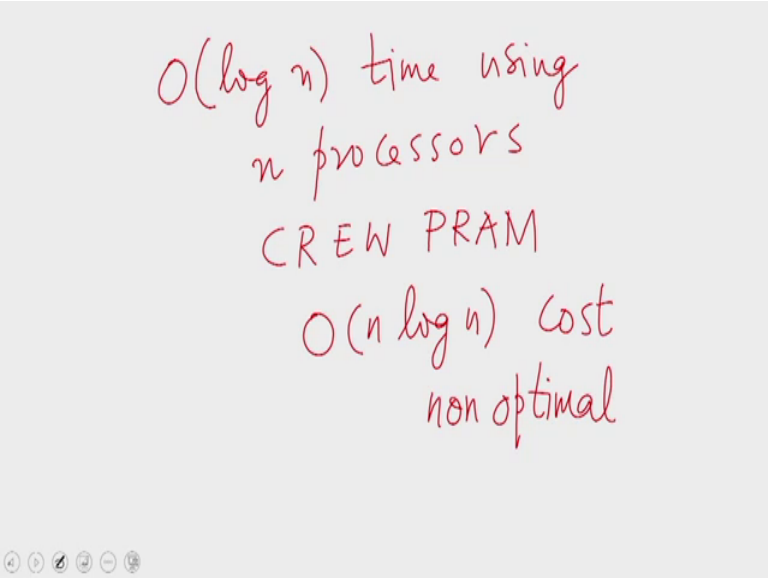
(Refer Slide Time: 32:05).



So, now the linked list is indeed ranked. So, in general when you are given a list of size n initially nodes of distance at most 1 are ranked correctly. This is at before the first step, after the first step nodes of distance less than or equal to 2 from the right end are correctly ranked. After the second step nodes of distance less than or equal to 4 from the right end will be correctly ranked continuing like this we can say that after the kth step nodes at distance less than or equal to 2 power k from the rightmost node will be correctly ranked.

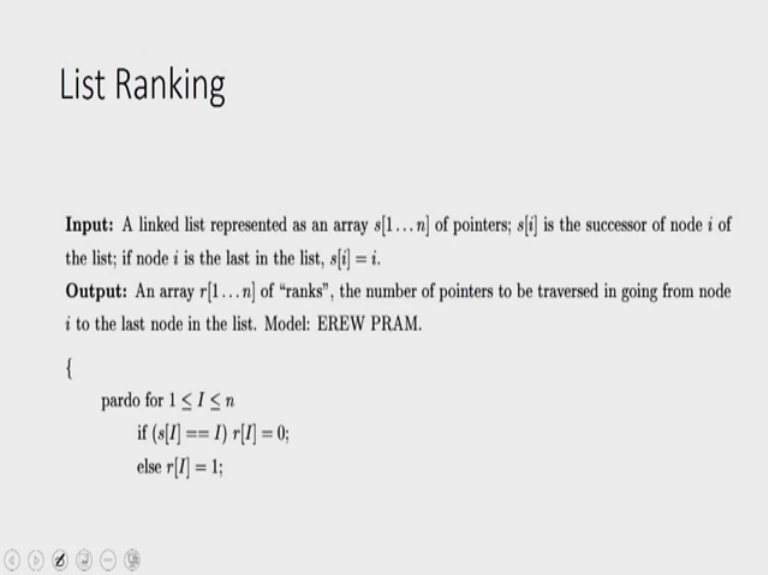
So, if you have a list of size n given in an array of size n, if you continue this for log n steps, then every node is correctly ranked. So, the problem will be solved. So, what we find is that, the algorithm runs correctly in order log n time using n processors, on a CREW PRAM.

(Refer Slide Time: 33:22).



So, the cost of the algorithm is order of n log n, which is not optimal. So, this is a non optimal algorithm. So, let us take a look at a formal specification of the algorithm now.

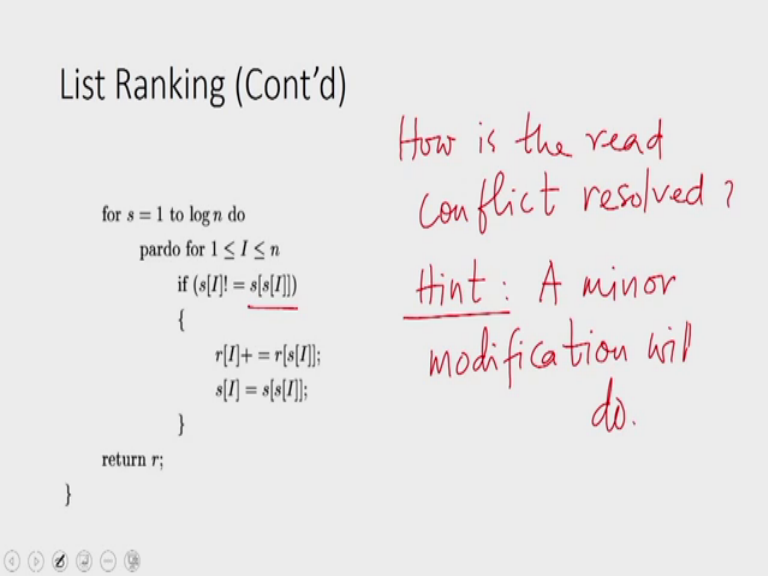
(Refer Slide Time: 34:12)



So, the input to the algorithm is a linked list represented as an array of pointers, s of i is the successor of node i. This is what we were calling the parent, parent or successor, mean the same thing here in this context. If node i is the last node in the list; then the successor of i is i itself. The output is an array of ranks, the rank of node i will be given in the ith position of the ranks array.

So, right now we have assumed a model of CREW. We shall see how to change this to an EREW PRAM algorithm. So, first what we do is this, in parallel for every node we assume that there is one processor at every single node, every node will check whether its successor is itself.

(Refer Slide Time: 35:11).

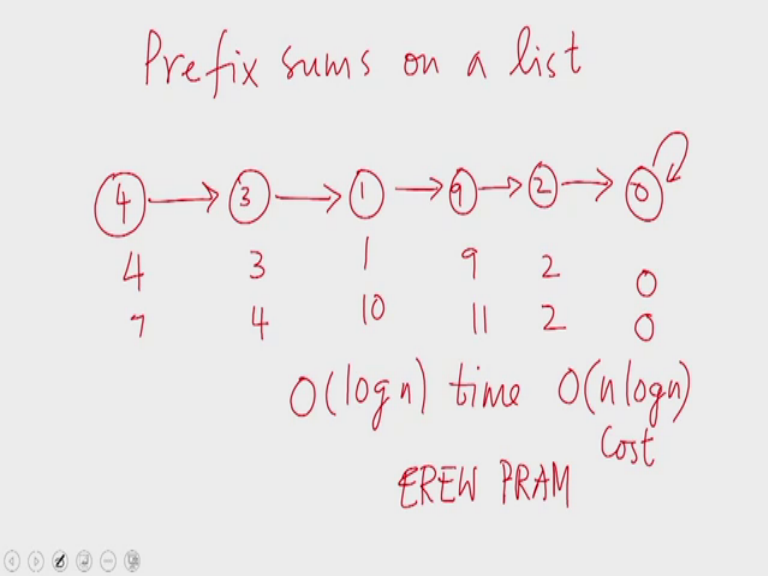


If it is not the case then it adopts a rank of 1, if it is indeed the case then it adopts the rank of 0, which means the logically last element in the list will get a rank of 0, every other element will get a rank of 1 and then repeatedly for log n steps, we do this in parallel for every single node i, if the successor of the node is not the same as the successors successor, then the rank of the node is updated. The present rank of the successor will be added to the present rank of the node and this will be the new rank of the node and then the successors, successor will be adopted as the new successor.

This is repeatedly done for log n steps and then finally, the array r, the array of ranks is returned. So, this is what the algorithm is, but the way it is specified, it does not work for EREW PRAMS, because there are concurrent reads. Where do the concurrent reads happen? It happens here, when the nodes check whether the successors successor is the same as its successor. They are all invoking the successors successor, but then there could be several nodes, which are pointing to the rightmost node. Now, all of them will be accessing the rightmost node simultaneously, causing a read conflict. How do we resolve this read conflict?

Once this read conflict is resolved, the algorithm will work correctly on any EREW PRAM. So, I am leaving this as an exercise for you, figure out how this can be done. I will give you a hint, a minor modification of the code will. Now, let us consider a few applications of list ranking. The first one is finding prefix sums on a linked list.

(Refer Slide Time: 37:17)



Let us consider the logical diagram of a list, let us say the nodes of the list have certain integers. Let us say we want to find the prefix sums over these integers from the right end. How do you find prefix sums given the list in an array form. We can use the same algorithm that we have just seen, only that the initialization has to be now different.

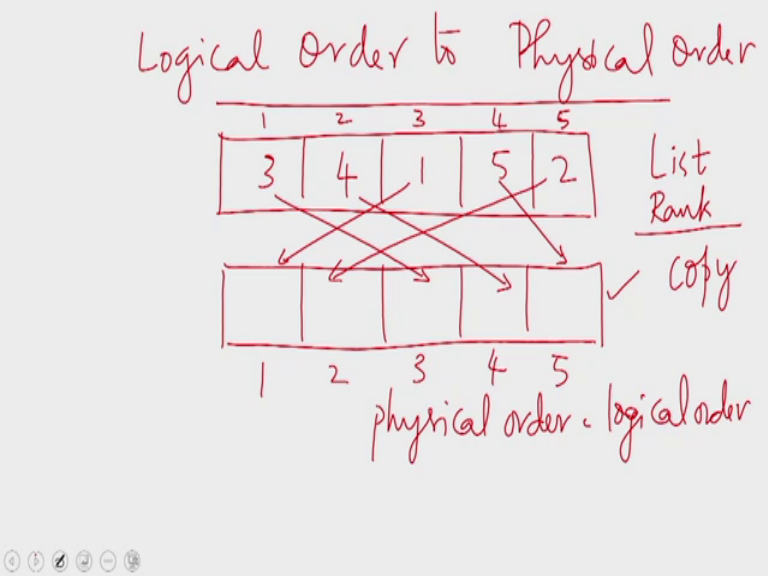
You would initialize the array with these values making sure that the rightmost node will have a value of 0. So, we can always add node to the right end, which is a dummy node, pointing to itself and holding a value of 0 and then every other node in the list will contain the given values. This is exactly analogous to the initialization that we had in the list ranking algorithm that is continue through the steps of pointer jumping. What we find us that initially every node will add the successors value to itself. So, the left most node will have a value of 7 now, the second left most node will have a value of 4, the third one will have a value of 10 and then 11 2 and 0.

So, we find that the prefix sum values are already correct for the 3 rightmost nodes, which are the nodes at a distance of 2 from the right end. So, continuing in this fashion after log n steps, we find that the prefix sums are computed at every single node in the

list. So, if you want to compute the prefix sums from the left end the direction of the pointers have to be changed. You will have pointers moving from right to the left and then if you do pointer jumping, you would get prefix sums from the right from the left end.

So, we find that the prefix sums on a list can be found in order of log n time for a cost of order of n log n on an EREW PRAM. So, that is one application of list ranking, the other application is converting a list from its logical order to physical order.

(Refer Slide Time: 40:18)



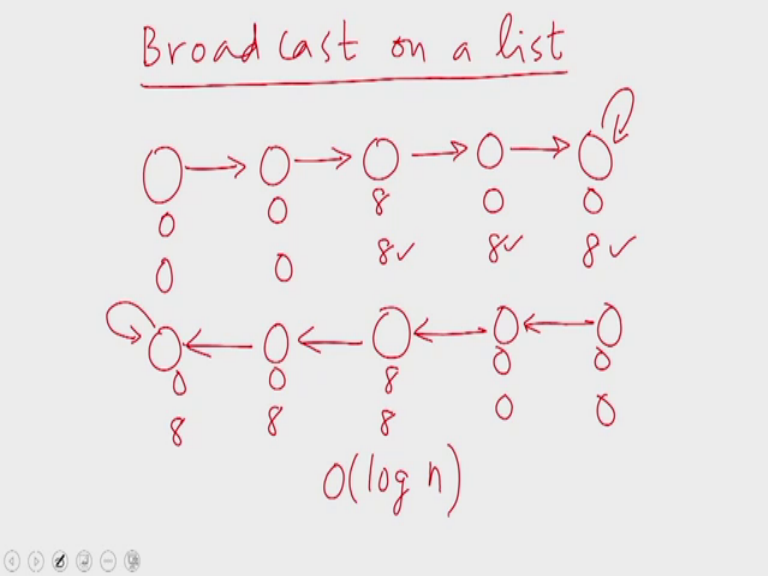
So, let us say we are given a list, the same example that we saw at the beginning of the topic, at the physic, first physical location we have the logically third node, the second physical location we have the logically fourth node and so on and then let us say, we find the ranks of the node using our list ranking algorithm. So, the ranks are 3 4 1 5 and 2.

So, once we have found the ranks of the node, we can rearrange the nodes in the logical order by taking another array of exactly the same size. So, in this case we take an array of size 5, where the locations are numbered from the left and again and in the original array we have now, let us say 5 processors 1 processor per element. So, the processor which is sitting on location one sees that it is of logical rank 3 and therefore, will copy this element into the third location. The processor sitting on the second physical element realizes that its rank is 4 and will copy it into the fourth location. So, the physically third element will be copied into the first location in the new array, the physically fifth fourth

element will be copied into the fifth location and the final element will be copied into the second location.

So, now in the new array, we have the list arranged in the logical order that is in the new array, the physical order and the logical order are the same. So, for this conversion all that we need to do is to invoke list ranking and then copy. The copying can be done in order one time, if you have 1 processor per element our model is EREW, but since each processors reading its own location and then copying into an exclusive location, there is no concurrent read or concurrent write therefore, an EREW PRAM is enough, in order one time you can do the copying. So, the total time taken is still order of log n.

(Refer Slide Time: 43:30)



So, converting of a list from the logical order into physical order can be done in order of log n time at a cost of order of n log n. Similarly, broadcast on a list, this is analogous to the broadcast in an array that we saw at the beginning on the lecture given a list in which an element wants to broadcast its value to every other node on the list.

We can initialize every node with a value of 0 except this node and do a prefix sums, then the prefix sum values would be, now the value 8 has been broadcast to the right, that is the 3 nodes at the right end have received the broadcast value, but the nodes to the left have not receive the broadcast value for that what you have to do is to change the direction of the pointers and the node, that is without a predecessor will be pointing to itself.

Now, the list has been inverted and then initialize the value 0 at every node and perform a list ranking. Do a prefix sums once again. Now, the nodes to the left of the given node will be having the value 8. So, the value has been broadcast over the entire list, again the time taken is order of log n for a cost of order of n log n and the model is the same EREW PRAM.

(Refer Slide Time: 45:34)

And again as before we can generalize this to finding prefix or suffix maximum or minimum all for the same complexity order of log n time and order of n log n cost for the same model and all its applications can be similarly adapted from the array setting to the list setting. So, that is, that is it from the sixth lecture. Hope to see you in the next lecture.

Thank you.