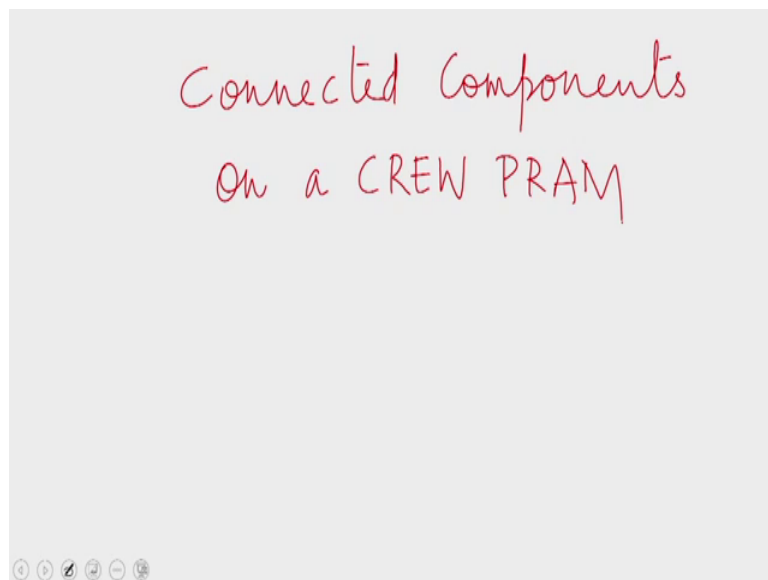


**Parallel Algorithms**  
**Prof. Sajith Gopalan**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture - 22**  
**Connected Components (CREW)**

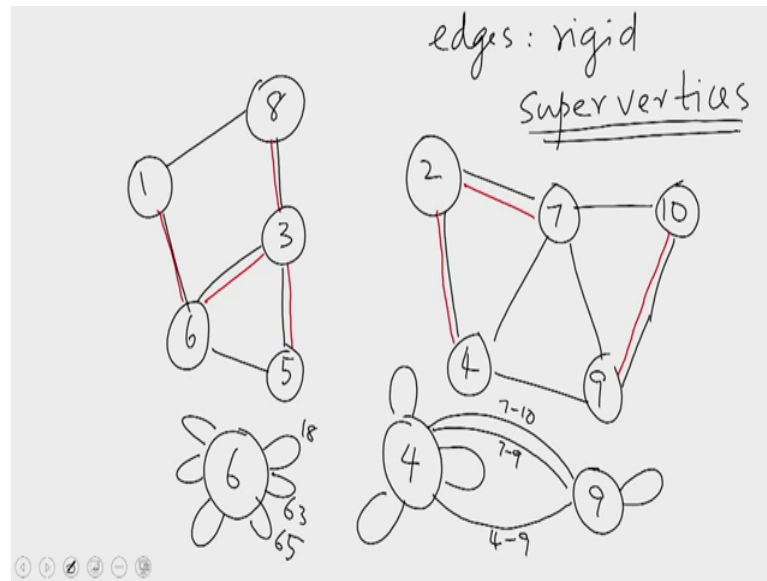
Welcome to the 22nd lecture of the MOOC on Parallel Algorithms. In the previous lecture we studied an algorithm for finding the connected components of a graph and this algorithm ran in order of  $\log n$  time using  $n$  plus  $m$  processors on an arbitrary CRC PRAM, today we shall see a connected components algorithm that runs on the CREW PRAM.

(Refer Slide Time: 01:00)



So, concurrent writes are not allowed here that is our problem for today. First let me exemplify this with a graph.

(Refer Slide Time: 01:25)



Let us say we have a graph with the multiple components. So, let me number the vertices. So, there is a 10 vertex graph and these are the edges of the graph. So, what we do is, this we define a collection of trees in the graph. These trees are defined by choosing a set of edges in the graph. For example, let us say we choose the edge 1 6, 6 3, 3 8 and 3 5. So, that is a tree which spans this component completely.

In the other component let us say we have multiple trees. So, here we have a tree which spans 2, 7 and 4 and let us say we have another tree that spans vertices 9 and 10. Now, what we do is this. We assume that our original edges are all rigid. The original edges are let us say, rigid to begin with and let us assume that the vertices attract each other. Now, imagine that the selected edges, that is those edges that are marked in red have suddenly become elastic. These edges were rigid to begin with, but now they have become elastic. Then what you find is that all these vertices collapse along these edges. For example, vertices 1 and 6 collapse to become one single vertex, 6 and 3 collapse along the edge to become one single vertex and so on.

So, what you find is that this one single component collapses into a composite vertex. Let us name it using one of those vertices. So, let us say vertex 6 represents 1, 8, 6, 3 and 5, but then when this collapse happens there are several edges to it for example, the edge 1 8, the edge 6 3, the edge 6 5, the all the other edges. There are 6 edges all of them become self loops of this one single composite vertex.

Now, let us see what happens to this other graph. Here also, edges here also we have red edges collapsing. So, vertex 2 combines with vertex 7 and vertex 4 that is because the edges 2 to 4 and 2 to 7 collapse. So, we have one single composite vertex that represents 2, 7 and 4 let me denote it by 4, then there are 3 edges between them 2 to 7, 7 to 4 and 2 to 4. So, there are now 3 edges from 4 to itself on the other side we have this edge 9 to 10 collapsing into one single vertex, let me call that vertex by 9 and there is the 9 to 10 edge turning into a self loop.

But then unlike in the first component here we find that there are certain edges that are between these two composite vertices. 4 here is a composite that stands for 2, 7 and 4, 9 here is a composite which stands for 9 and 10. What we find is that there are black edges, the original edges that move between the components. For example, there is an edge from 7 to 10. Now, that becomes an edge between the composite 4 and the composite 9. There is an edge from 7 to 9, which also becomes such an edge. And then there is an edge from 4 to 9.

So, these are the 3 edges that go from the composite vertex 4 to the composite vertex 9. So, after the edges have been collapsed the graph looks like this. So, what we have exactly done is this we take the graph we assume that all the edges in the graph are rigid those are the black edges. And then we assume that all the vertices attract each other this is just for visualization and then let us grow a grow trees on these components. Let us we choose certain edges of the graph in such a way that the edges together will define a forest, so that each tree in the forest will be completely within a component of the graph.

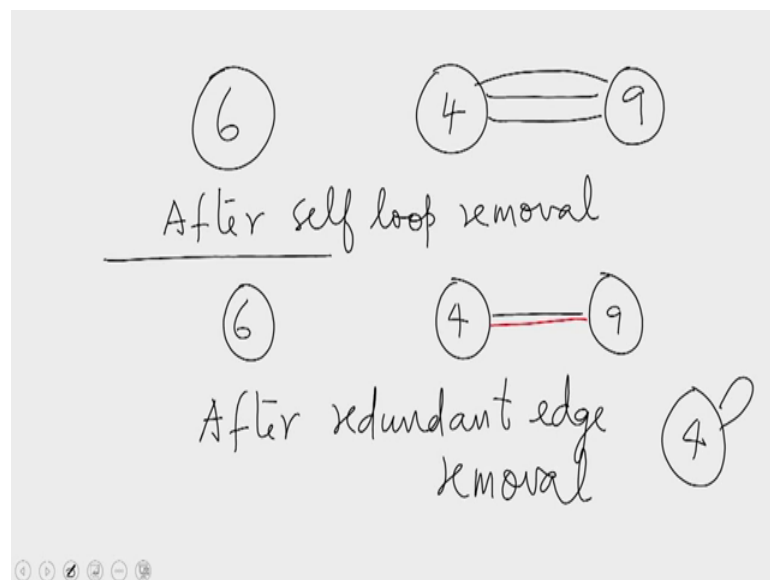
So, here there are two components in the graph, from the first component we get one single tree, one single tree that spans the whole of the component, from the second component we get two trees. The first tree spans vertices 2, 4 and 7, the second tree spans vertices 9 and 10. Then what we assume is that all these chosen edges which were rigid initially become elastic all of a sudden. And since the vertices attract as I said before, you would see that the vertices collapse along the chosen edges. So, we get what are called super vertices. When vertices collapse into one another we get what are called super vertices.

So, we have now 3 super vertices 6, 4 and 9, these super vertices are composite vertices then the original edges now become either self loops. Like in the case of the first

component every single edge within that component has now become a self loop. An edge from a vertex to itself of course, their labels differ for example, edge 1 8 has now become an edge from 6 to 6, but you can label it as one 8 because it was the original edge between 1 and 8. In the second component we have two super vertices now denoted 4 and 9.

There are 3 edges between these, these are respectively 7 10, 7 9 and 4 9 and the original graph. And then there are self loops to that is edges 2 to 7, 7 to 4 and 4 to 2 are self loops at vertex 4 these are edges from 4 to itself and the edge from 9 to 10 has become self loop at vertex 9. So, this is the graph we have obtained.

(Refer Slide Time: 08:17)

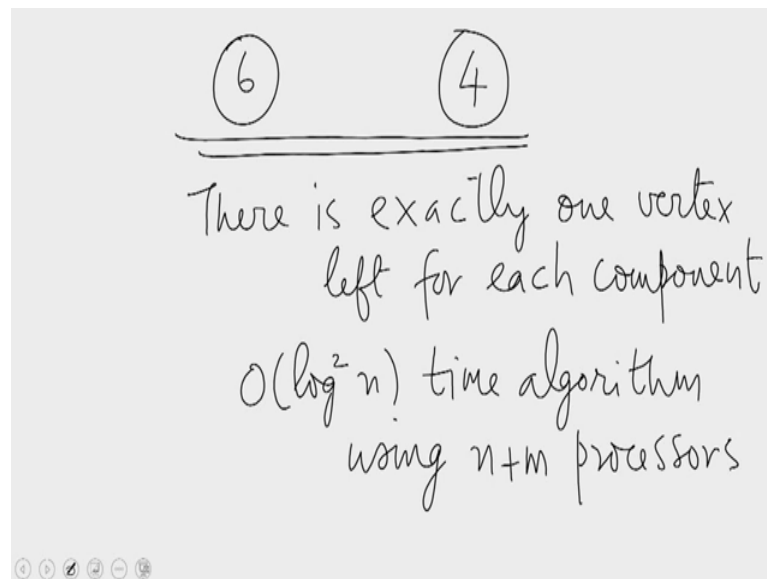


Now, we can clean up this graph by removing all the self loops. At vertex 6 we have only self loops there is no other edge, at vertex 4 we remove all the self loops, and at vertex 9 we remove the self loop, but then we find that between 4 and 9 there are 3 edges. So, this is how the graph looks like after self loop removal.

And then the next step is to get rid of all the redundant edges, that is what we want is that between a pair of vertices there should be only one single edge there should not be multiple edges. Now, in this example we find that between 4 and 9 there are multiple edges, we will replace them with a single edge. So, after the redundant edges are removed the graph shrinks to this form. So, this completes one iteration.

So, after one iteration, we find that the size of the graph has reduced, now we have fewer vertices, and then we iterate; once again we choose for each component of the graph several trees for every vertex, we want either a parent or an edge or a child. So, let us say there is only one edge here that edge is chosen to represent a tree. So, now, we have a tree involving vertices 4 and 9. So, what we find is at 4 and 9 collapse into one single vertex with a self loop and then we remove the self loops and the graph reduces to this.

(Refer Slide Time: 10:29)

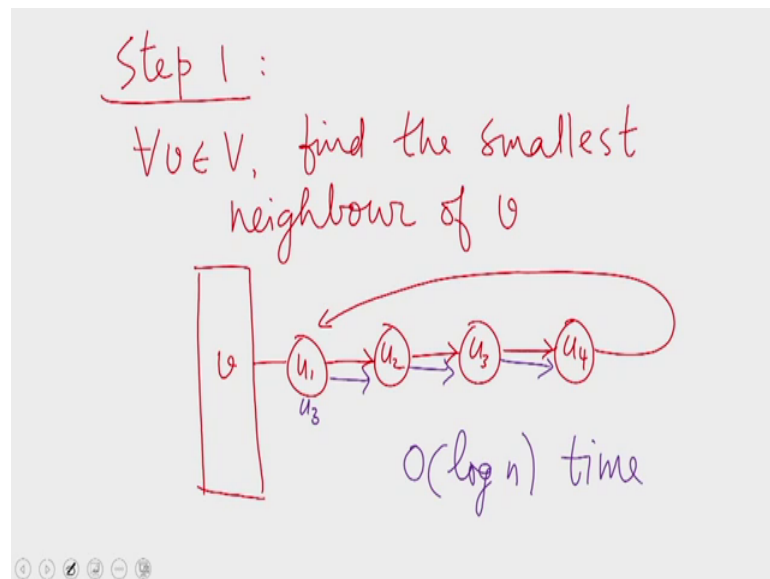


At this point we hold the algorithm because now there is exactly one vertex left for each component. So, that is the gist of the algorithm. Given the graph we choose a set of edges, so that every vertex either gets a parent or a child which means we have a collection of trees in the graph. The trees are defined in such a way that a tree is completely within a component. So, a component could be partitioned into multiple trees, and then what we do is to shrink all the trees to one single super vertex.

And then we will have redundant edges, there will be self loops any edge that was within the tree will be a self loop and any edge that was between trees will be now between two super vertices. But then it could be that even though the original graph was a simple graph, now there are there could be multiple edges between super vertices. These are redundant edges. Then we clean up the graph by removing the redundant edges and the self loops and then we are ready for the next iteration.

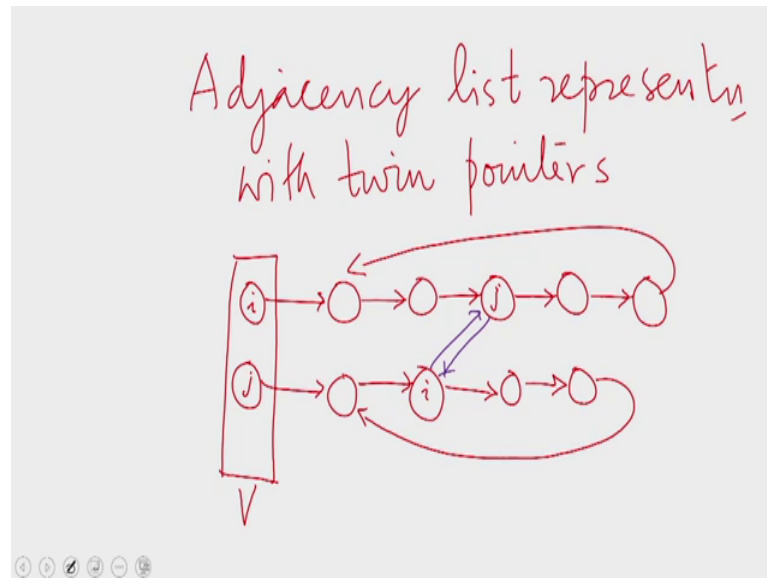
This will guarantee that the number of vertices reduces by a factor of at least two in every iteration. Therefore, we will require at most  $\log n$  iterations to finish the algorithm. If we managed to show that each iteration can be executed in order  $\log n$  time then we would have obtained an order  $\log^2 n$  time algorithm using  $n + m$  processors. So, that is a high level description of the algorithm. Now, let us get into the details of the algorithm.

(Refer Slide Time: 12:45)



So, in step one what we do is this. Of course, here we assume that, here we assume that the graph is given in adjacency list representation with twin pointers.

(Refer Slide Time: 12:55)



What that means is that, we have an array of vertices and for each vertex  $i$  we have an adjacency list the list of all vertices that are adjacent to  $i$ . We will assume that this is a doubly linked list and circular to. So, if vertex  $j$  is adjacent to  $i$ , somewhere in  $i$  is adjacency list there is an entry corresponding to  $j$ . Similarly, if you look at vertex  $j$  you find that in the adjacency list of  $j$  somewhere there is an entry for  $i$ . These two entries are called the twins of each other and we assume they point to each other, these points are called twin pointers. So, we assume that we have the adjacency list representation of the graph with twin pointers already given to us.

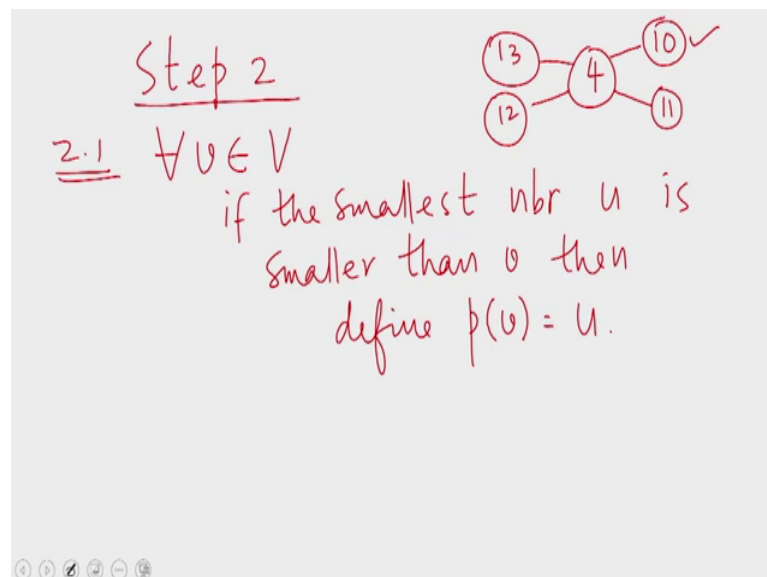
Then in step one of the algorithm what we do is this for every vertex  $V$  find the smallest neighbor of  $b$ . So, we assume that vertices are consecutively numbered from 1 to  $n$  and for each vertex  $v$ , now we are finding the smallest numbered neighbor of that vertex. But how would you find the smallest numbered neighbor of  $v$ ? We can do this by using pointer jumping on the adjacency list of vertex  $v$ . So, for each vertex  $v$  we have this adjacency list. So, let us say these are the neighbors of vertex  $v$ . From the pointer jumping algorithm we know that when we have one (Refer Time: 15:35) every node of the list then we can use pointer jumping to essentially compute a prefix sum over some initial values that have been distributed over the vertices.

In the list ranking algorithm, we put a 1 on the last node and a 0 on every other node sorry the other way round, we put a 1 on every node other than the last one on which we

put a value of 0 and then we perform a prefix sum. So, here essentially we will do a prefix minimum. The prefix minimum will end up giving us the smallest value at the last node. So, if you do prefix minimum from the right end which is essentially the suffix minimum the first node in the list will end up getting the smallest value. So, by pointer jumping over the adjacency list the size of which is at most  $n$  minus 1 which will be the vertex degree of vertex  $v$ ; so, if you do pointer jumping on this in order of  $\log n$  time you will be able to find the smallest vertex here. Of course, we can do this without destroying the original pointers that can be done by making a copy of all these pointers and then doing a pointer jumping using these new pointers which are marked blue in color.

So, we do pointer jumping using the blue pointers and when the pointer jumping is over. Finally, we will have the smallest value here let us say  $u_3$  is found to be the smallest here at vertex  $u_1$  and then we can discard all the blue pointers and go back to the original red pointers. So, in this way without destroying the original pointers we can find the smallest neighbor of vertex  $v$ . And when we do the simultaneously for every single vertex  $v$  would have found the smallest neighbor of every vertex  $v$  simultaneously. So, this can be done in order of  $\log n$  time.

(Refer Slide Time: 17:34)



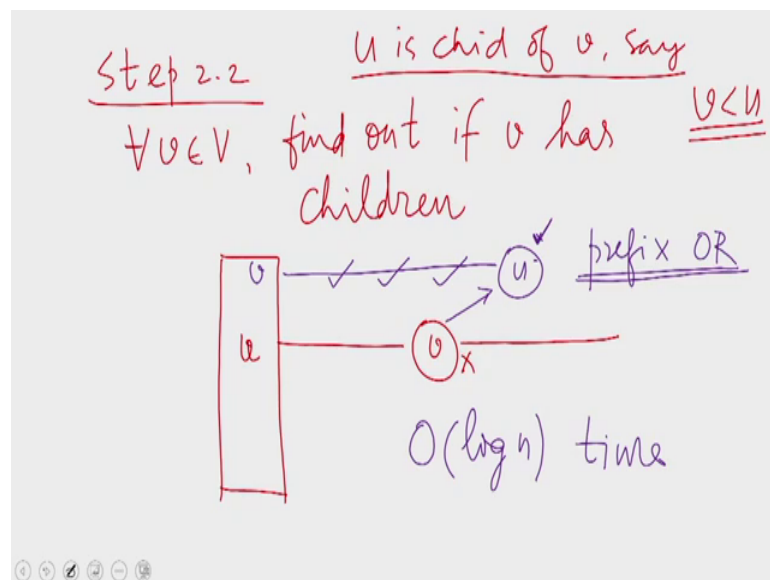
And then in the next step, now we have found the smallest neighbor for every vertex. What we do is this for every vertex if the smallest neighbor  $u$  is smaller than  $v$  then



define  $p$  of  $v$  as  $u$ . That is now we are defining a parent pointer for vertex  $v$ . This is defined as the smallest neighbor provided that that smallest neighbor is smaller than  $v$ .

If the smallest neighbor is not smaller than  $v$  then; that means,  $v$  is a local minimum let us  $v$  does not have any smaller vertex. For example, if we happens to be 4 and its surrounded by let us say 10, 11, 12 and 13 then the smallest neighbor is 10 which is larger than 4. So, in this case 4 will not be hooking to vertex 10. So, vertices we will get some vertices we will get parents now, a vertex that has a smaller neighbor we will definitely get a parent. It will be choosing the smallest of the smaller neighbors as its parent. So, this is step 2, 1; the first part of step 2.

(Refer Slide Time: 19:08)



Then in step 2 2, for all  $V$  find out if  $v$  have children. For every vertex we want to find out if  $v$  has children, and we want to do this on a CREW PRAM remind you in a CRCW PRAM arbitrary model that was easy if a node has children we only have to let the children come and write at the parent, but that cannot be done here because we are not allowed to use concurrent rights. So, how do you find out if  $v$  has children? We can use the adjacency list representation and its twin pointers for this purpose.

Let us say  $u$  is a child of  $v$ . If  $u$  happens to be a child of  $v$  of course, in the adjacency list of  $u$  there is an entry for  $v$  and this entry has already been found out by  $u$  in step 2 1,. In step 2 1, we find for each vertex the smallest neighbor of  $u$  and choose it as the parent. Now, here what we have assumed is that  $u$  has adopted  $v$  as its parent, which means  $v$  is

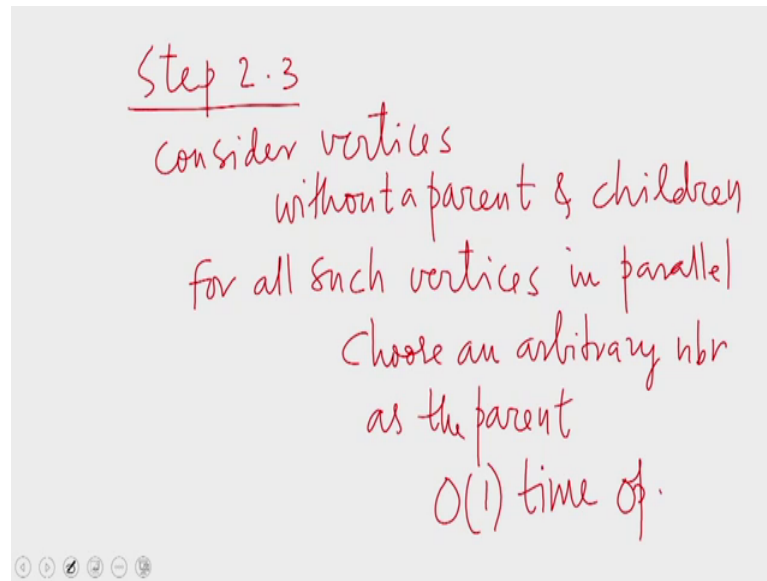
the smallest a smaller neighbor smallest neighbor of  $u$ ,  $v$  is the smallest neighbor of  $u$  and that happens to be smaller than  $u$ . So,  $v$  is smaller than  $u$ . That is why  $u$  adopted  $v$  as its parent.

Now, in the adjacency list of  $u$  there is this entry for  $v$  which  $u$  already knows the address of therefore, what  $u$  does is to go to this entry and mark it to indicate that  $v$  is now my parent. Then in the next step the processor which is sitting on this adjacency list entry can go to its twin and inform the twin which happens to be the entry of  $u$  in the adjacency list of  $v$ . So, the twin is now informed of this mark that is this mark is copied on to the twin of course, you can put a different mark to indicate that this is a child there is  $u$  is a child of  $v$ . So, if  $v$  has multiple children, now in the adjacency list of  $v$  multiple tick marks will appear.

Therefore, to detect if  $v$  has a child all you have to do is to check if there is a tick mark on this adjacency list. Now, for each vertex  $v$  if we perform a prefix or which can be done using pointer jumping one round of pointer jumping, if we perform prefix or over this array then we will know if  $v$  has an entry with a tick mark in it. So, this can be done in order of  $\log n$  time. Again that is if in step 2 1, you had chosen  $v$  as its parent you will now go and inform  $v$  that I have chosen you as my parent and that informing will be done in the adjacency list entry of  $u$  within the adjacency list of  $v$ , that can be accessed through the twin pointer from the entry  $u$   $v$ .

This is being done simultaneously by every single vertex. So, if  $v$  happens to have multiple children all those children will be marking their corresponding adjacency list entries in vertex  $v$ . Therefore, all we have to do is to check whether  $v$ 's adjacency list has tick marks for every single  $v$ . This can be done using a prefix or over the adjacency list entries because you can take these marks as one and the absence of a mark as 0. This can be done using one round of pointer jumping which again we will take order of  $\log n$  time; given that we have one process of our every adjacency list entry of the graph. So, in order of  $\log n$  time we have found out if  $v$  has children.

(Refer Slide Time: 23:48)

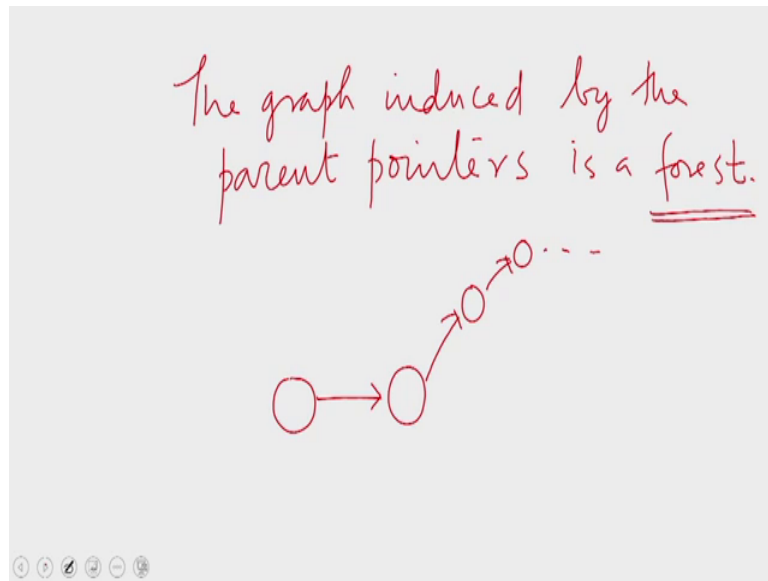


Now, in step 2 3, consider vertices without a parent and children. So, remember in step 2 1, we did not succeed in finding a parent for every vertex this was a counter example. If a vertex was surrounded by all larger vertices then it is a local minimum and therefore, will not choose a parent in the step. So, all local minima are without parents after step 2 1, every other vertex has chosen a parent. Then in step 2 2, which check if a vertex has children in particular all local minima that were left out in step 2,1, now discover that discover whether they have children or not.

So, those who fail in the second step 2 that is they in step one they the local minima all failed to get parents in step 2 2, some of them fail to get children to that is some of them discover that they are without children. So, at this point at the beginning of step 2 3, we have these vertices that are local minima and are without children. So, we consider all such vertices. And in parallel for all such vertices what we do is these for all such vertices, in parallel choose an arbitrary neighbor as the parent. So, step 2 3, is an order of an operation. If a vertex has discovered that it is a local minimum and also that it is without children then it can pick the first element in its adjacency list as the parent.

So, this is an order one operation order one time operation. So, steps 2 1, 2 2, 2 3 altogether can be done in order of  $\log n$  time and this ensures that every vertex gets either a parent or a child.

(Refer Slide Time: 26:20)

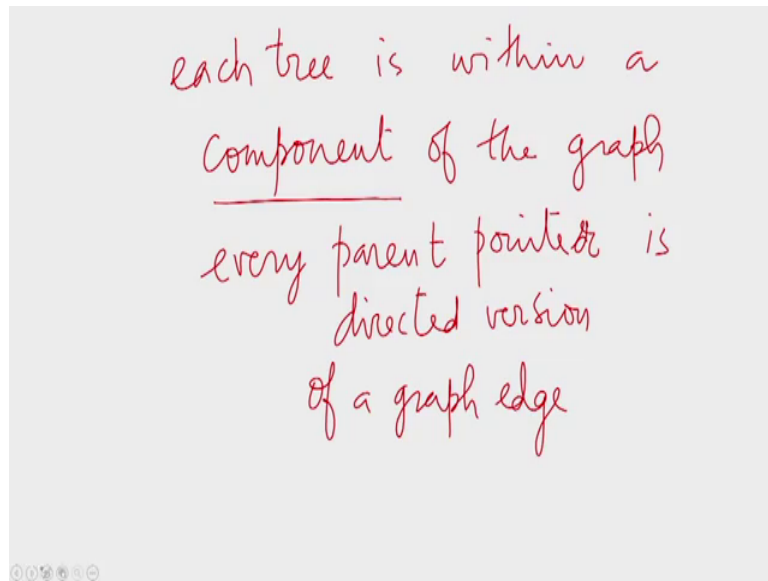


Now, I claim that the graph induced by the parent pointers is a forest which is a collection of several pieces. Now, why is this a forest? This is because we cannot be forming cycles. Once again let us go back to step 2.1, in step 2.1 we define parent pointers that are from larger nodes to smaller nodes. So, here a local minimum failed to choose a parent because all its neighbors were of larger values. So, apart from local minima all are choosing the smallest of its neighbors as the parent, but this smallest happens to be smaller than  $v$  itself therefore, every pointer that we define in step 2.1 happens to be from a larger vertex to a smaller vertex.

Therefore, at this point we cannot have a cycle. Every edge is from a larger vertex to a smaller vertex. So, we cannot have a cycle. So, at the end of step 2.1, we do not have a cycle. In step 2.2 we are not defining any new pointer we are merely finding out whether a vertex has children or not then in step 2.3 we consider all vertices that are now isolated, they are without parents and without children. So, when you have such a vertex what we do for those vertices was to pick an arbitrary neighbor as the parent. So, that vertex was pointing to a smaller neighbor which was pointing to a smaller neighbor and so on.

Now, this vertex is without children. So, there is no incoming pointer to it is arbitrarily hooking to some neighbor, therefore, we cannot form a cycle once again, therefore, we get a collection of trees.

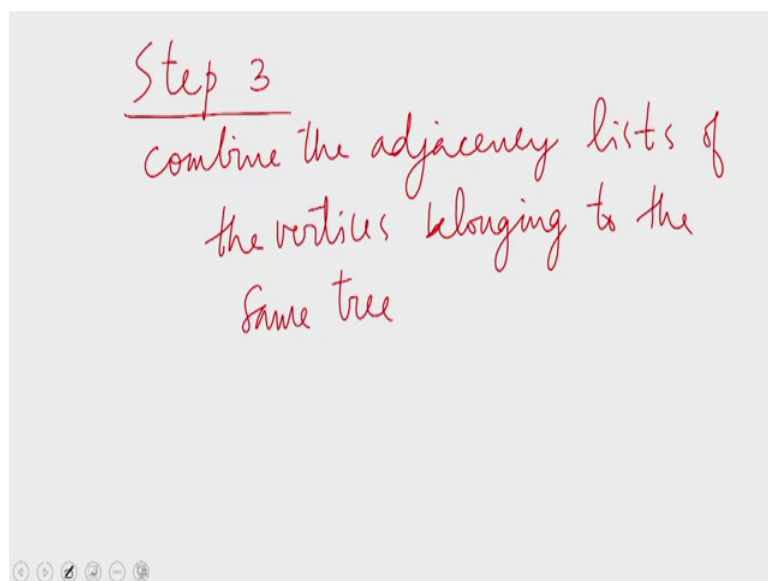
(Refer Slide Time: 28:20)



each tree is within a  
component of the graph  
every parent pointer is  
directed version  
of a graph edge

Now, we can claim that each tree is within a component of the graph that is because every parent pointer is a directed version of a graph edge that is we have defined parent pointers using the graph edges. Therefore, every tree that we form must be necessarily included within a component of the graph. So, this is exactly as in the example that we saw at the beginning of the lecture. We have chosen some edges these edges will form a forest and every tree in the forest is completely within a component. So, that is what we have done in step 2 2.

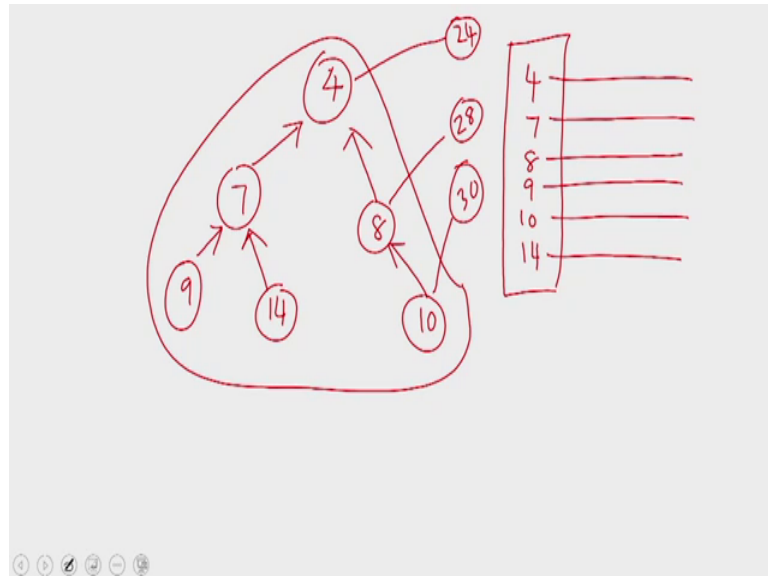
(Refer Slide Time: 29:40)



Step 3  
combine the adjacency lists of  
the vertices belonging to the  
same tree

Now, the second step is to for the third step is involves combining the adjacency list of the vertices belonging to the same tree.

(Refer Slide Time: 30:20)



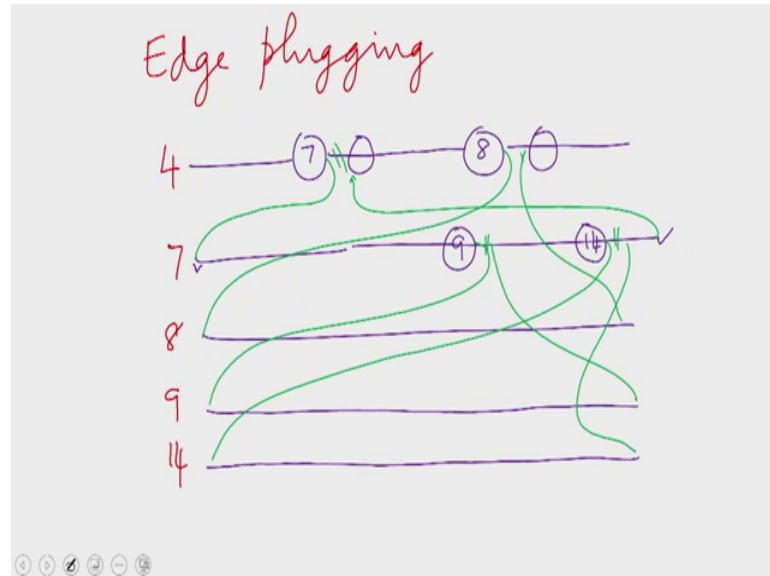
Let us take an example tree. Let us say this is a tree we have. We want to combine the adjacency list of all these. So, in the adjacency list representation we have this adjacency list corresponding to 4, 7, 8, 9, 10, 14. Now, all these vertices have joined together to form a single tree. What this means is that all of them belong to we have already identified there all of them belong to the same connected component, and they are all going to fuse together into one symbol super vertex. Then all the adjacencies of these should come together.

For example, 4 must be adjacent to some vertex let us say 24 somewhere else, 8 is adjacent to let us say some vertex 28 using non-tree edges and let us say 10 is adjacent to 30. Now, 24, 28, 30 etcetera might belong to different super vertices after the step. Therefore, what we find is that this single super vertex, must now be adjacent to the super vertices super vertex containing 24, the super vertex containing 28, the super vertex containing 30, etcetera.

Now, we will be meaningfully able to deal with all these adjacencies only if we collect all these adjacencies together in one place. That is to begin the next iteration we should get the graph in a cleaned up form the same cleaned up form with which we started this iteration. For that we have to collect all the adjacency list entries of this one single super

vertex and then we have to remove the redundant edges and the self loops out of that So, the first step of that is collecting all the adjacencies together. So, how do we do that?

(Refer Slide Time: 32:35)



We can do this using what is called edge plugging. This can be done like this here we find that vertex 4 has 2 children, 7 and 8. So, let us consider the adjacency list of 4 in this we have 7 and 8 somewhere, likewise, likewise we have the adjacency list of 4 and 7 and 8. These are circular linked lists. So, I have not shown the cycling part of it, that is the last node of this adjacency list of course, will be pointing to the first node.

Now, what we do is this. Consider the pointer going out of 7. So, there is a vertex here right after 7 which is some other node that is adjacent to 4. What we do is this take this adjacency list of 7, remove the back pointer that is the last pointer is pointing to the first the last node is pointing to the first node here. 7 anyway have access to this node and let us assume it also have access to this node the last node. So, we take this list the adjacency list of vertex 7, we hold it by the two ends of it and then we take it and insert it between the entries corresponding to 7 and the next one in the adjacency list of 4. That is we thread the pointers in this fashion from here we define a pointer to this point and this last point will end up pointing to this vertex.

So, what we have managed to do is to insert the adjacency list of 7 entirely within the adjacency list of 4. So, this is now a remote pointer. Similarly, I consider the successor of vertex 8 here, and the adjacency list of 8 is spliced in between those two nodes. So, now,

the adjacency list at 4 will look like, this it starts from the first node comes to 7 and then goes to the beginning of the adjacency list of 7 and then goes all the way through the adjacency list of 7 and then goes to the successor of 7 in the adjacency list of 4, continues with the adjacency list of 4 comes to 8, and then goes to the beginning of the adjacency list of 8 goes through the adjacency list of 8 and then goes to the successor of the adjacency list of 8 in the adjacency list of 4 and then continues with the adjacency list of 4.

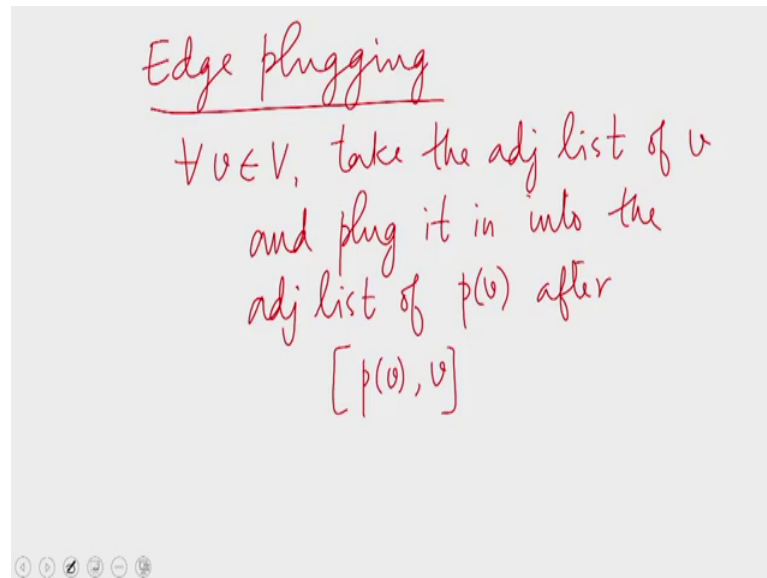
So, this has been done by the processors at 7 and 4. 7 and 4 now identify that their parent is 4 therefore, each of them will do this operation. The processor which is sitting at vertex 7 can hold the two ends of the adjacency list of 7 and it can splice it in between the adjacency list entry corresponding to 7 in the adjacency list of 4 and the next entry. At the same time vertex 8 will be splicing its adjacency list in the second position. So, this is what is done by vertices 7 and 8.

At the same time let us consider the children of 7 and 8, 7 has two children 9 and 14. So, let us consider the adjacency lists of 9 and 14. So, within 7 we have some location where 9 appears and some location where 14 appears. So, now exactly as before we will be the processes that is sitting on vertex 9 will be splicing in the adjacency list of 9 right after the entry corresponding to 9 in the adjacency list of 7. This is done after removing the pointer between 9 and its successor and the pointer between 14 and its successor.

So, now, the adjacency list of 9 has gone completely within the adjacency list of 7 which at the same time is getting spliced in into the adjacency list of 4. So, all these adjacency lists have now become reachable from the beginning of the adjacency list of 4. Likewise, we will also be doing for the children of 8, 8 has only one child which is 10. So, let us consider the adjacency list of 10 here. Likewise, we will be doing in further adjacency list of 10 as well. So, what we do is to splice in the adjacency list of 10 within the adjacency list of 8. Therefore, what we find is that all the adjacency list of all these vertices, now become a part of the adjacent a list of 4.

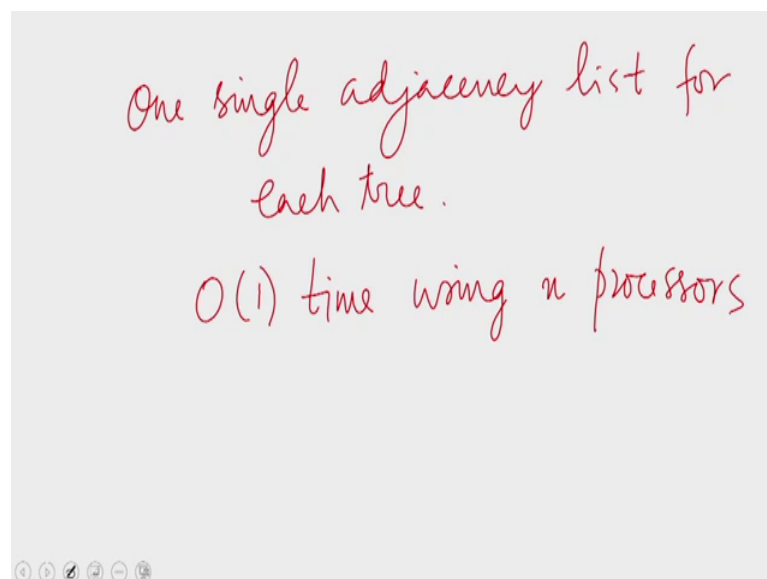


(Refer Slide Time: 37:42)



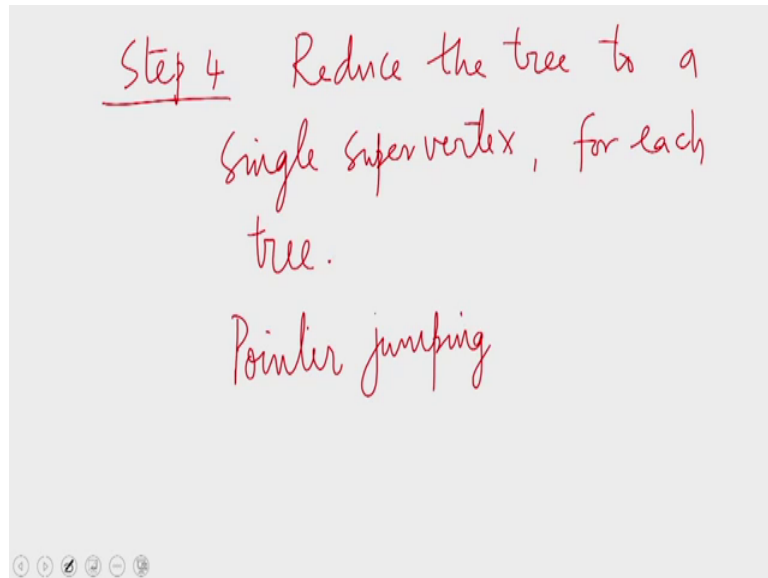
So, this is what we call edge plugging. So, in edge plugging formally what we do is this. For all  $v$  take the adjacency list of  $v$  and plug it in into the adjacency list of  $p$  of  $v$ , after  $p$  of  $v$  which happens to be the adjacency list entry corresponding to  $v$  within the adjacency list of  $p$  of  $v$ . So, this entire adjacency list of  $v$  is plugged in into the adjacency list of  $p$  of  $v$  and this is done by every single vertex  $v$ . Once we do this as we saw in the example what it happened what happens is that the adjacency list of all the vertices belonging to the same tree will be collected within the adjacency list of the root.

(Refer Slide Time: 39:02)



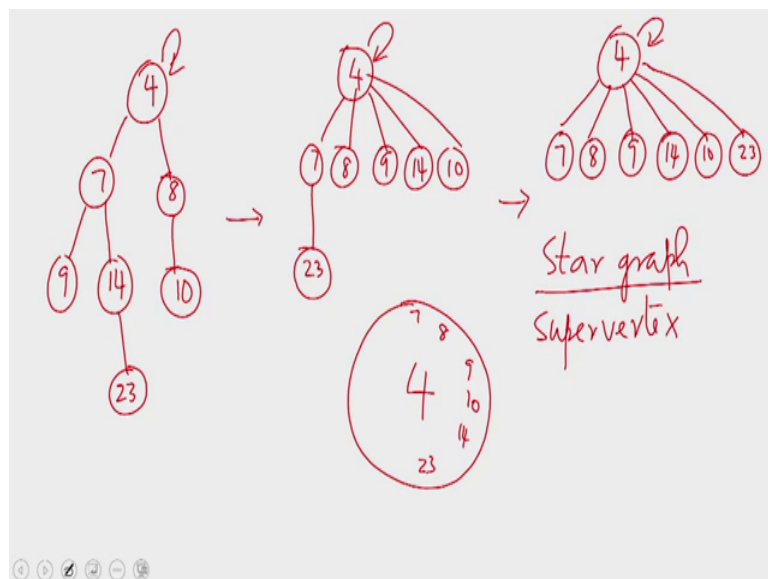
So, at this point we have one single adjacency list for each tree. We can do this on order one time using  $n$  processors. So, that is what we did in the third step we have collected all the adjacency lists of the vertices of the tree together at the root.

(Refer Slide Time: 39:46)



Now, in step 4 we want to reduce the tree to a single super vertex. This we want to do for each tree. How do we do this? We can do this using panda jumping again.

(Refer Slide Time: 40:32)



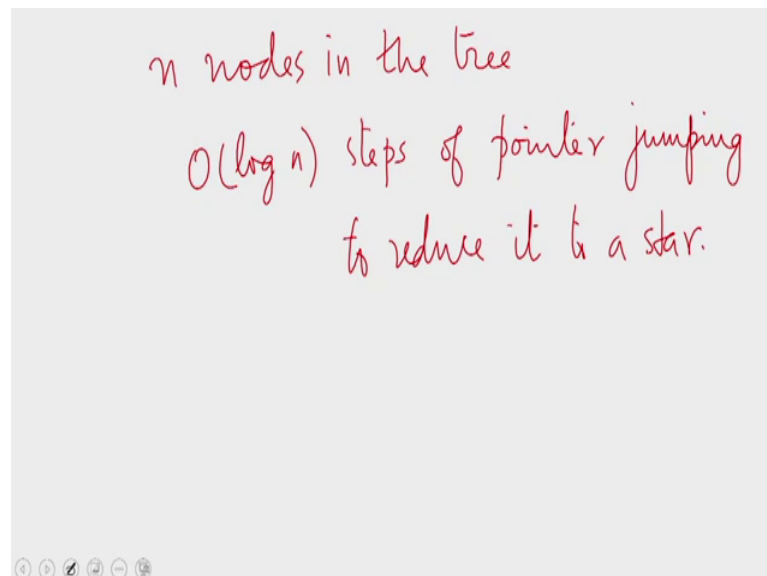
For example, when we have a tree of those form pointer jumping involves adopting the grandparent as the parent. So, after one step of pointer jumping the tree will look like

this, the grandchildren of 4 will adopt 4 as its parent. So, 9, 14 and 10 will now adopt 4 as its parent. Vertex 23 will adopt 7 as its parent, because 7 is its grandparent in the original tree. So, after one round of pointer jumping the tree will look like this [nose]. Of course, here we assume that the parent of 4 is itself.

In the second step of pointer jumping once again every node is adopting the grandparent as its parent. So, here 23 is the only node with a true grandparent which is 4. Every other node has the parent equal to the grandparent, so their pointers will not change. So, what happens is that after the second round of pointer jumping all vertices, all of them end up pointing to vertex 4 and 4 is pointing to itself. At this point the tree has reduced to a star graph. A star graph is what we call a super vertex.

So, when we form a star graph all these vertices that is vertices 7, 8, 9, 14, 10 and 20 3 all of them know that they are going to fuse with vertex 4. Essentially, they are adopting vertex 4 as their representative. So, we can assume that this is a super vertex which is going to be named 4 and vertices 7, 8, 9, 10, 14 and 23 have all fused with vertex 4. So, they have all identified with vertex 4. So, when we form a star graph of this form, we say that we have formed a super vertex with the root as its identifier.

(Refer Slide Time: 43:13)



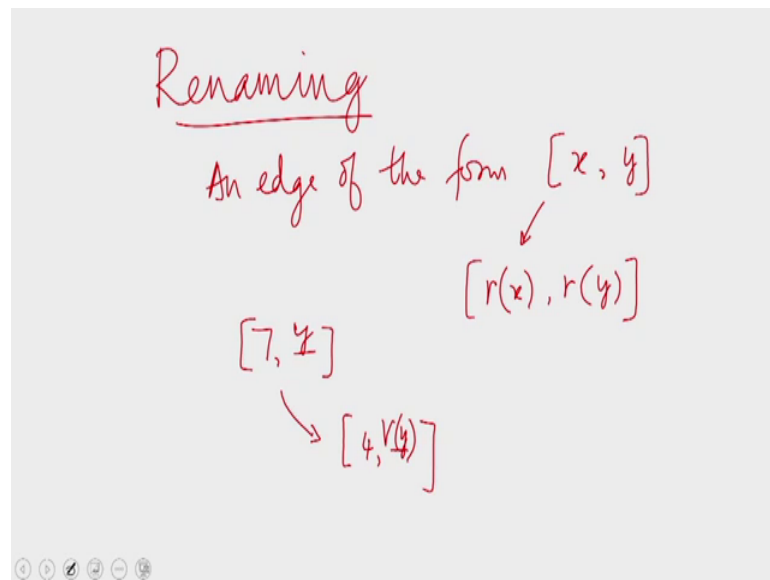
So, as we have seen we have  $n$  nodes in the tree. A tree is only a small part of the graph, so it can have at most  $n$  nodes. Therefore, you require order of  $\log n$  steps of pointer jumping to reduce it to a star. So, if you have one processor for every single vertex the

time taken to reduce the tree and early to a star is order of  $\log n$  time. And this is going on simultaneously for every single tree. So, if you spend order of  $\log n$  steps all the trees in the graph will reduce to a star graph which means we would have shrunk the entire graph into super vertices.

So, now, we have achieved two things. In the previous step we collected all the adjacency lists of all the vertices belonging to the tree together at the root, and then in further order  $\log n$  steps we have now shrunk every tree to a star graph. So, now, every leaf of the star graph can assume that it is out of the graph only the root will participate in the further operations of the algorithm. They have essentially identified themselves with the root. Now, those vertices are all going to lose their identity they will all be renamed with the name of their root.

For example, 7, 8, 9, 10, 14 and 23 have gone and fused with vertex 4 therefore, now they do not have separate identities all of them will be renamed as 4, which means in the 5th step where we will be taking renaming what we do is this.

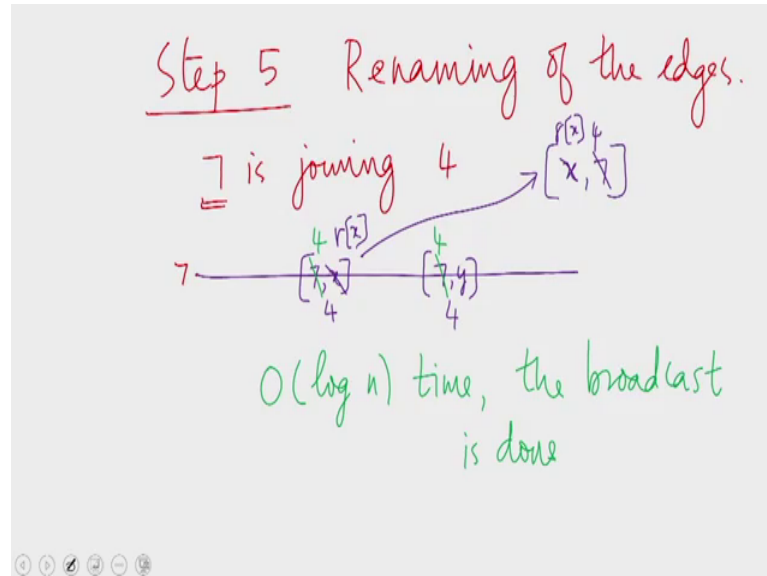
(Refer Slide Time: 45:05)



An edge of the form  $x y$  will have to be replaced with an edge of the form  $r$  of  $x$ ,  $r$  of  $y$ , where  $r$  of  $x$  is the root of the tree containing  $x$  and  $r$  of  $y$  is the root of the tree containing  $y$ . For example, here 4 is the root of our example tree. So, if we had an edge of the form 7 and something this will have to be renamed as 4 and something. Here of course, we should be if this is  $y$  here we should be filling in  $r$  of  $y$  the root corresponding to the node

y, the root of the tree containing node y. So, this is how we are going to rename the edges of the graph. So, this is what we do in the next step.

(Refer Slide Time: 46:17)



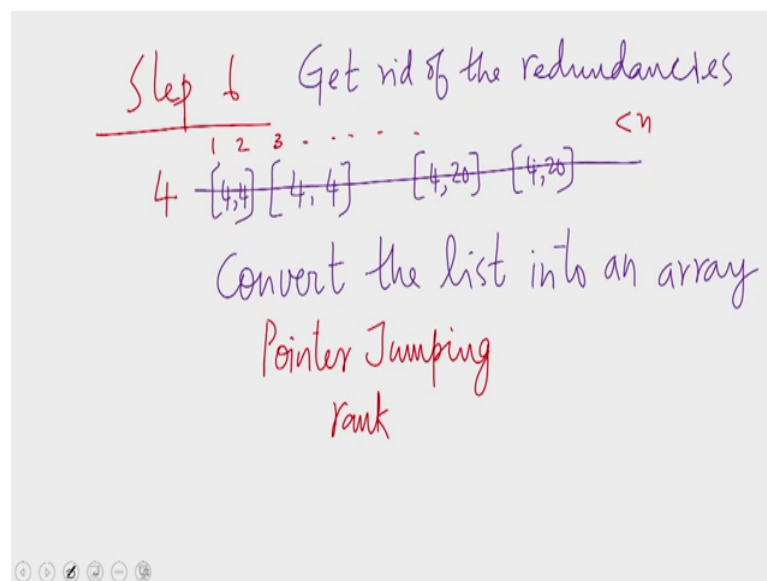
So, in step 5 we do the renaming of edges. Now, what we assume is that even though edge plugging has been done the original pointers are retained and therefore, every vertex still has, every vertex that is going out of the graph still have access to its original adjacency list. So, for example, here we find that 7 is joining 4 and 7 is going out of the graph, that is once this is done 7 will be out of reckoning for the rest of the algorithm. But 7 still has access to its original adjacency list and 7 knows that it is going to identify with 4 therefore, what 7 does is to go through its adjacency list and inform every single vertex that it is now going to be renamed with 4.

So, there are entries of the form 7 x, 7 y, etcetera on this list every one of those entries will be informed that the new name for 7 is 4 which means those entries should now rename themselves as 4 x, 4 y, etcetera. So, this involves broadcasting of a piece of information over a linked list. We know that pointer jumping can be done you used for broadcasting of a piece of information over a linked list. So, using that fact we find that in order of log n time the broadcast can be done. Every vertex can broadcast its new name to every single adjacency list entry in order of log n time by performing one round of pointer jumping. So, now, entry 4 x has found out that its new name is entry, entry 7 x has found out that its new name is 4 x the second component has not changed yet.

Now, it can go to its twin, this can go to its twin which is  $x$  7 and change 7 to 4 here to, by that time this  $x$  would have been replaced by  $r$  of  $x$  by the processor that is sitting at  $x$  that processor would at this point be crossing over and coming here to change this  $x$  to  $r$  of  $x$ . So, both the twins are getting renamed simultaneously, but still there is no concurrent right because the processor that has already updated  $x$  to  $r$  of  $x$  is now coming over here, when the processor here is going over there. Therefore, when 7 is getting rewritten by 4 in the twin of 7  $x$  the  $x$  is getting rewritten by  $r$  of  $x$  by the processor that is at location  $x$  7. So, without concurrent right we can in this way rename all of the adjacency lists of the graph.

Now, this we are doing using the original pointers in the adjacency list, but then after h plugging we have a new set of pointers which thread the adjacency list into the root of the the;

(Refer Slide Time: 49:50)



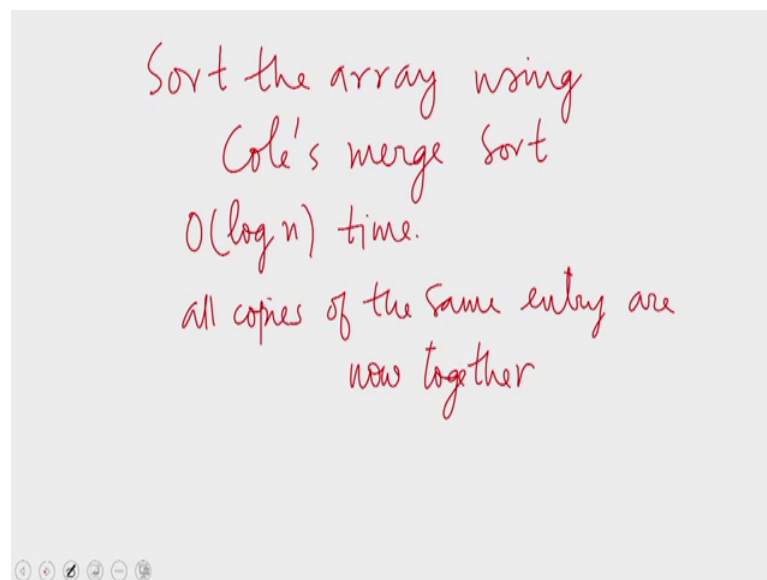
So, if you look at the adjacency list of a vertex now, we find that corresponding to vertex 4 in its adjacency list there are multiple entries. For example, for every self loop we have entries of the form 4 4 and there could be multiple entries like this. So, there could be multiple self loops and multiple entries like this.

Now, the next step is to get rid of the redundancies. In step 6 we try to get rid of the redundancies. So, how do we do this? For this first what we have to do is to convert this linked list into an array. This of course, a familiar operation; once again we can use

pointer jumping for this, to rank the list once you rank the list we know which is the first entry, which is the second entry and so on. The maximum length is  $n$ , there are at most  $n$  minus 1 neighbors to every vertex. So, using pointer jumping we rank the list.

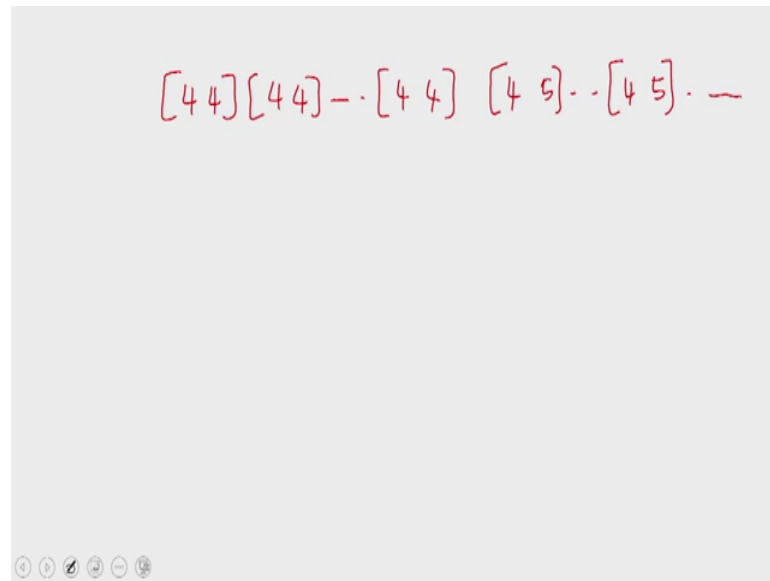
Once we have ranked the list, we can take an array and copy these elements into the array. So, the first element will copy itself into the first location of the array, the second element will copy itself into the second location of the array in this way we can get a physical representation of the list which is identical to its logical representation this is something that we have seen before. So, by doing this we will get all these entries in the into an array.

(Refer Slide Time: 52:07)



Once the entries are in an array, we sort the array using Cole's Merge sort. Since, we have one processor for each element of the array this will take order of  $\log n$  time, but when we do this what we gain is that the redundant elements are all coming together. For example, all copies of the same entry are now together.

(Refer Slide Time: 53:07)



What I mean is that when I look at the adjacency list of 4 all entries of the form 4 4 are together, then all entries of the form 4 5 for example, are together and so on. So, now, we have not managed to get rid of the redundancies, but we have managed to bring all the redundant elements together in one place.

We have to remove all the self loops, and out of the redundant edges we have to keep exactly one and remove the rest. Now, that becomes easy because every single set is now together. How to do this? And make sure that the iteration will run in order of  $\log n$  time on the whole, we shall see in the next lecture. So, that is all from this lecture. Hope to see you in the next.

Thank you.