Data Structures and Algorithms

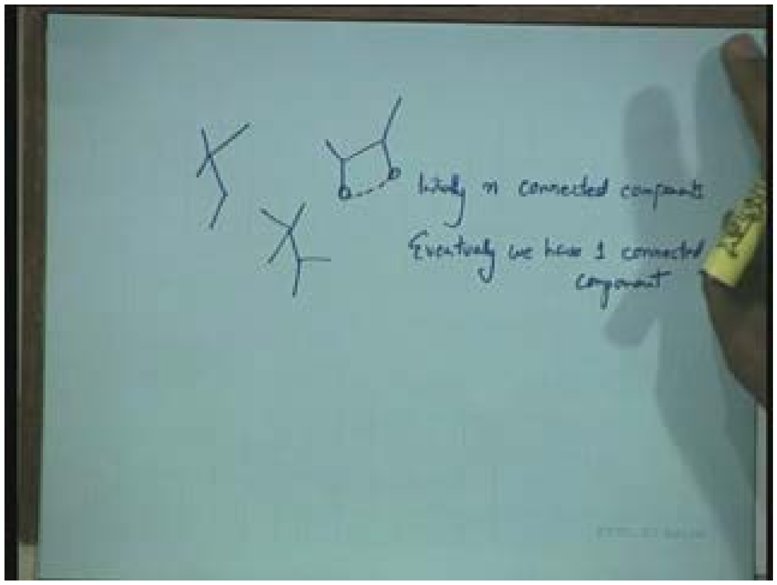
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 32

First class we looked at the Kruskal's algorithm for computing minimum spanning trees. We were, we saw what the algorithm was and what we were doing was, to complete algorithm, we have to figure out, how to detect if there is a cycle that gets formed when we add an edge into the current set of edges that we have already went by.

So, if you recall what we have said was that we would try and maintain the collection of connected components. So, I am going to revise part of that but we are going to today, look at the data structure, for being to do that, that is called the union fine data structure. So, let see, where we were as far as Kruskal's algorithm was considered. We said, I have already picked a set of edges.

(Refer Slide Time: 3:27)


Initially n connected components
Eventually we have 1 connected component

Recall that the set of edges would always form a forest, it would be a collection of trees. There could be no cycle that is already existing. So, these are the set of edges that have already been picked in Krukal's algorithm. Now, when I am trying to add a new edge, I have to check if it forms a cycle or not.

Suppose, this is the new edge being added, this forms a cycle and 1 way of detecting whether a certain edge would form of a cycle or not, is to check, if its 2 end points lie in the same tree of this forest. The tree is the same as the connected component and so, it suffices to check whether the 2 end points of an edge, line the same connected component. This is where we were, at the last class.

So, we have to somehow maintain our collection of connected components. So, as the algorithm proceeds - we have discussed this in the last class - in number of connected component reduces by 1 with every step.

Initially, we started off with n connected components. Initially, we had n connected component and eventually, finally, we have only, how many connected component will we have? 1. This is how the algorithm proceeds. So, I am going to abstract this problem out and capture it as a problem on maintaining a collection of disjoint sets.

So, what is the setting now? I have a universe of elements. Let say, I have e_1 e_2 $e_{n.}$ Let us say, these are n elements in my universe. Initially, each of these elements is a set in itself. Now, the following operation, so, this is a collection of disjoint sets that I have. Initially, I have the collection of disjoint sets, at each stage I am going to have a collection of disjoints.

So, what we are trying to do is to maintain a collection of disjoint sets under the operations of, let us see, what are the operations, we are doing, going to be doing on the disjoint sets. So, what do these elements correspond to, in the case of Kruskal's algorithm? But, what are they initially? Initially, these e_1 $e_{2,}$ what are e_1 e_2? These would be the vertices, the initial vertices.

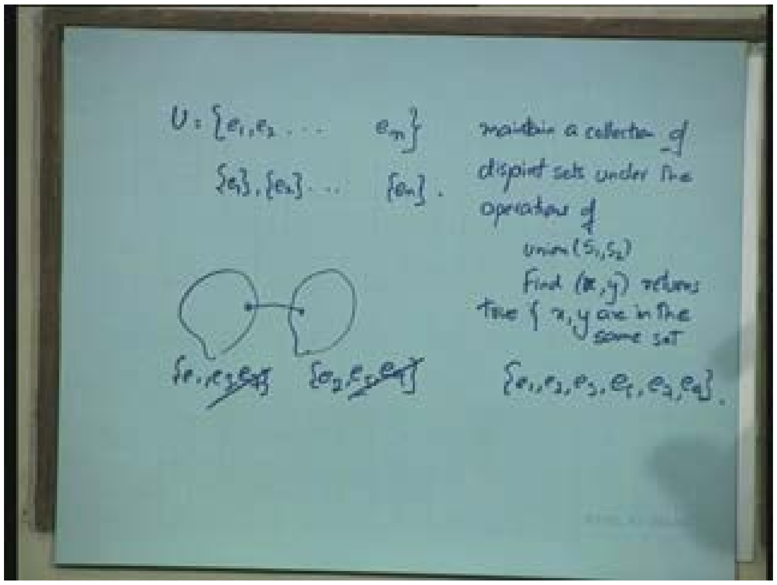
Now, what are the operations that we have to do on this collection of disjoint sets? 1 operation is union, the operation of union has to be down when I have 1 connected component, I have another connected component and the edge that I add, runs between these 2 connected components. Then, the resulting thing would be, this entire thing could be, 1 connected component and it should get reflected here, by taking the union of the corresponding sets.

So, I have to do an operation of union that is 1 operation under which I have to maintain this collection of disjoint sets. So, what do I mean by that? Suppose, this collection have e_1 e_3 e_7 in it and this, sorry, this set has e_1 e_3 e_7 in it, this set has e_2 e_5 and $e_{9,}$ then after the union, I should not have these 2 sets in my collection. But these 2 sets of the collection should get replaced by 1 set which is e_1 e_2 e_3 e_5 e_7 and $e_{9.}$

The other operation that I have to do is, so, given an edge, I have to look at the 2 end points of the edge and determine if they belong to the same connected component or not. So, the end points of the edge would be the 2 elements and I have to check given to 2 elements, whether the line same set or not. So, we will call that an operation of find.

So, what does find do? Let say, find takes as parameters 2 elements, x and y and returns true, if x comma y are in the same set. What should it return? Well, this is not a completed description of find. If x and y are not in the same set, what should find return? It should return the 2 sets in which those 2 elements lie. Why? Why should it return the 2 sets? Because, then we need to do the union on those 2 sets, exactly. So, union should take as parameters, 2 sets, let say s_1 and s_2 and should take the union. This is what we require of our data structure.

(Refer Slide Time: 7:39)



$U = \{e_1, e_2 \ldots e_n\}$

$\{e_1\}, \{e_2\} \ldots \{e_n\}.$

maintain a collection of disjoint sets under the operation of

union(S_1, S_2)

Find (x, y) returns true if x, y are in the same set

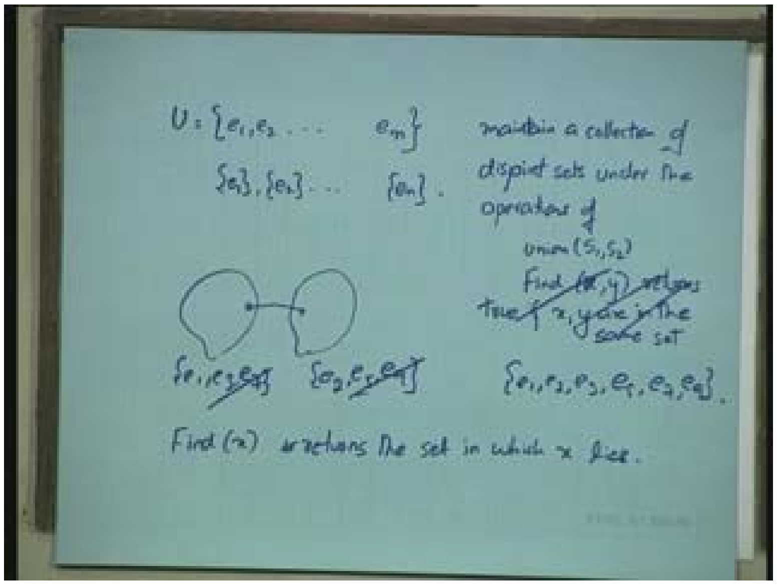
$\{e_1, e_2, e_3, e_4, e_5, e_6\}.$

$\{e_1, e_3, e_5\}$
$\{e_2, e_4, e_6\}$

So, this is not an accurate description of find, I will have to modify it. But you understand the need for returning the sets in which the 2 elements lie. So, instead of find x comma y, let me have just an operation, called find x, returns the set in which x lies. There is a unique set in which x lies, because our collection of sets is always a partition of universe. So, there will be a unique set and you want to return that set.

Then how will we implement this operation? We will do find x, we will do find y, if the 2 sets are the same then we will conclude that they are forming a cycle, if those 2 sets are different, then you would take the union of those 2 sets. Yes, everyone following? So, if I were to write down Kruskal's algorithm now, it would look like this. So, with this, assuming that these 2 operations exist, it would look something like this.

(Refer Slide Time: 8:22)



$U = \{e_1, e_2 \dots e_n\}$ maintain a collection of disjoint sets under the operations of

$\{e_1\}, \{e_2\} \dots \{e_n\}$.

union (S_1, S_2)

Find ~~(x, y) returns true if x, y are in the same set~~

$\{e_1, e_3, e_5\}$ $\{e_2, e_4, e_6\}$ $\{e_1, e_2, e_3, e_4, e_5, e_6\}$.

Find (x) returns the set in which x lies.

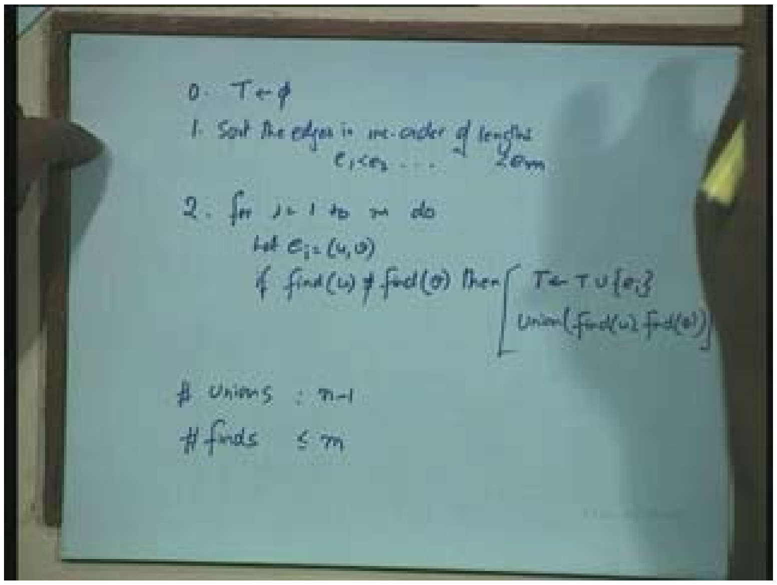
So, what were the steps of Kruskal's algorithm, first step? Sort the edges. Sort the edges in increasing order of length. Let say, this ordering is e_1 e_2. Please do not confuse between this e and the element e that I had in the previous slide. This e corresponds to edges and the previous e is element. So, keep these separate. Now, what should I do? For i equal to 1 to m do.

I pick an edge, what is the edge? e_i, I am considering an edge e_i. So, let e_i equal to u comma v which means u and v are 2 end points of the edge e_i. What is it that I have to do? If find u equals find v, then it forms a cycle. Then, we do not have to do. So, I should really do, not equal. If find u is not equal to find v, then, T is T union e_i. So, I should have some T and initialised to null and what else should I do? Union: find u, find v.

Of course, I need to initialize this collection of disjoint sets. So, I would have, when I create this collection of disjoint sets, it will get initialized to, what will it get initialized to? Singleton, so, each element in the collection would be singleton vertex. So, this would be what the procedure will look like now. We need to understand, what kind of data structure to keep for, to find and what kind of a data structure to keep for maintaining the collection of disjoint sets, so that these operations can be done very quickly.

How many times do we do the union operation? How many times do we do the find operation? Number of unions, because, every time I do an union, I include an edge into my tree. How many edges can be there in my tree? Exactly: n minus 1. So, I will have exactly, n minus 1. How many finds will I have? For every edge I consider, I have to do 2 finds. In the worst case, how many edges will I consider? All the m edges, no more. So, number of finds is less than or equal to m. Let us make it. Not equal to 2 m.

(Refer Slide Time: 12:17)



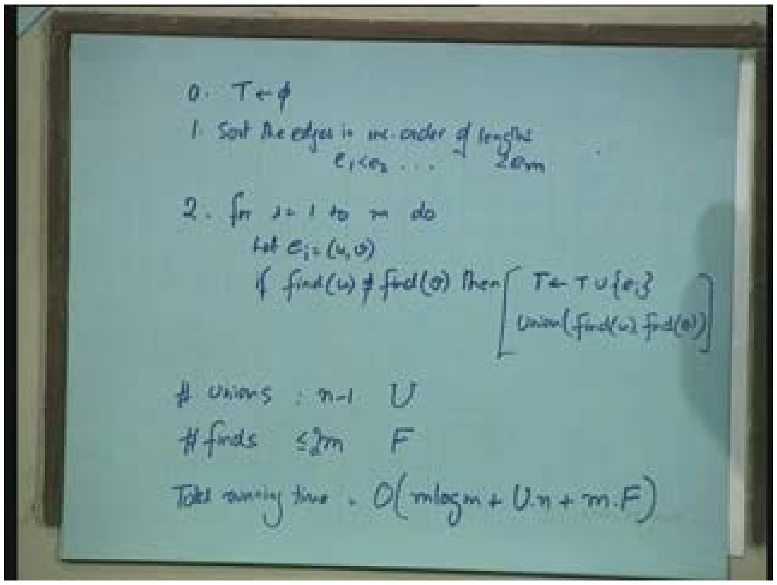
```
0. $T \leftarrow \phi$
1. Sort the edges in inc. order of lengths
        $e_1 < e_2 \ldots \leq e_m$
2. for i = 1 to m do
        let $e_i = (u, v)$
        if find(u) $\neq$ find(v) then [ $T \leftarrow T \cup \{e_i\}$
                                          Union(find(u), find(v)) ]
```

unions : $n-1$

finds $\leq m$

I have said, for i equals 1 to m. But, you know, you can always break out of this procedure, the moment you form a tree. So, if you form a tree before, you can break out of this procedure, of this for loop. So, you will keep it less than or equal to 2 m. So, what will be the total running time of this procedure then? If this operation; let say takes u times and this takes f time, then what is the total running time? Anyone? This step will take m log, m time, plus u times n, plus, m times f. So now, we have to find out a good data structure. By good view, mean, which would do the u and which will have the small u and small f. Is everyone comfortable with this?

(Refer Slide Time: 13:17)



```
0. $T \leftarrow \phi$
1. Sort the edges in inc. order of lengths
        $e_1 < e_2 \ldots \leq e_m$
2. for i = 1 to m do
        let $e_i = (u, v)$
        if find(u) $\neq$ find(v) then [ $T \leftarrow T \cup \{e_i\}$
                                          Union(find(u), find(v)) ]
```

unions : $n-1$ U

finds $\leq 2m$ F

Total running time = $O(m \log m + U.n + m.F)$

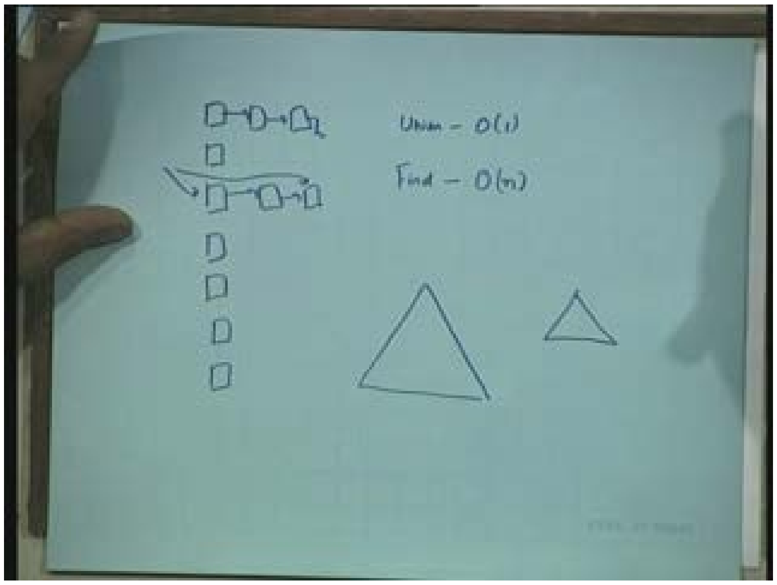
Are there any questions to this point? Can someone suggest a data structure to me? How will we maintain this collection of disjoint sets? Linked list, let say, liked list. You are at the end of the course, but, you cannot think beyond liked list. So, how will we have? We will have as many linked list as the number of sets, is the qutei. So, 1 linked list for each set.

No, let us complete this first. How much time union take and how much time did find take? Union can be done in constant time. (hindi) So, let say, we keep track of the front and the end of each liked list. If we do that, then I can combine 2 linked lists in constant time. Union will not take too much time. But, how much time did find take? Order size of the liked list, which in the worst case could be end. Now, is that good? We were to do this, we will take m, n time which is too large.

We are looking for the time complexity or something like, n log m. If we looking for something like, n log m, this quantity should be no more than log m and this quantity should also be no more than log m. We can permit it to be as large as log m. So, this is not too good at data structure. Someone had another idea. What was your idea? A tree? What will you do with the tree? How will you use a tree? Heap? What will you do with a heap?

How much time does it take to merge 2 heaps? Order n? Height of the heap, why? Why does it take, if I have 2 heaps, why does it take order H time to merge them? Order smaller, no. Number of elements in the smaller heap, but that could be as large as n by 2. What else? So, we will have a new data structure … and what will a new data structure be? The sets, I will just show this thing to you and then you will understand what is happening.

(Refer Slide Time: 16:32)



So suppose, my universe was a, b, c, d, e, f, 6 elements, simple. So initially, recall that my, what are the sets in my collection? The singletons: a, b, c, d, e, f. So, I have a, I have b, I have 1 node for each of these 6 sets. Now, suppose you say, union, the set containing a, so, you will say something like, union find a comma, find b. The set containing a and the set containing b. So, what I am going to do is to make 1 of them, so, each of these nodes has only 1 point … or a reference. So, it has a data field and 1 reference field.

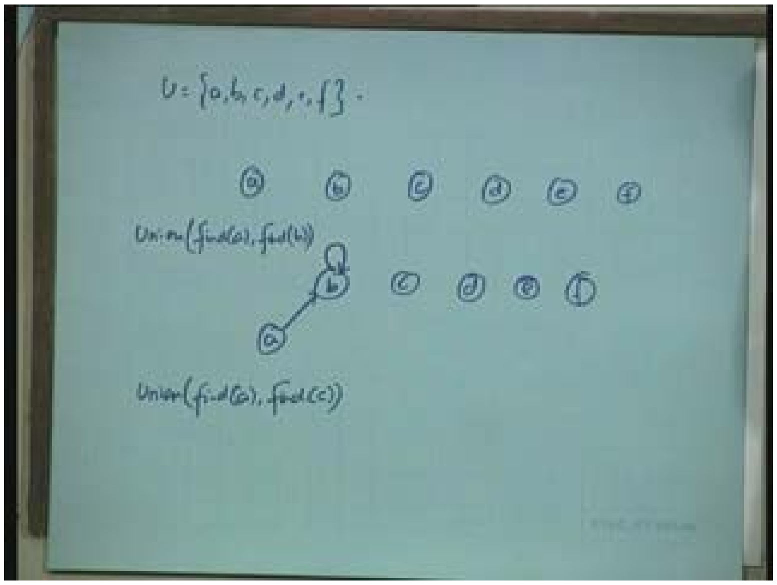
So, I will make 1 of these guys point to the other. So, at the next step I will have, when after I do this, this is what my collection looks like. These of course, remain like as they are. (hindi) Suppose, you were to say the same thing, union, find a comma find c. When I say find a, I will start from a and keep going up, till I head the root. So, this is now the root.

How do I know it is the root? Because, its pointer points to itself or it is null or whatever. Yes, when we do the union, you will understand this, you know as, as we proceed. So, this is what the trees look like. So, each one of them is a tree. Right now, this tree has only 1 node in it. But, this tree has 2 nodes in it and now the pointers are going up. You just have parent pointer … When I say find a, I will start from a and keep going up the tree till I reach a node, let say, (hindi) parent point is null or it is pointing back to itself. I think at that point, I know it is a root.

And so, this says that find a, that the element a, is in the set whose root or in which b is. When I come to b, this is the element, this is the set in which, what we are doing is that for each set, how do we represent a set? So, each set is represented by 1 of the elements in a set which is in this representation, it will be the root of the set.

So in some sense, all the element of the set, elect a leader and this is the leader of that set. So, a and b are the only 2 elements of the set and the leader is b. So, when I say find a, it return to me, reference to this node which says that this is the root of the set to which, in which element a lies. When I say find b, what will it return? The same thing and then I can compare those 2 and can determine that they are in the same set or not. So, what find a, find b returns are the roots of the corresponding trees. You will understand this as you proceed.

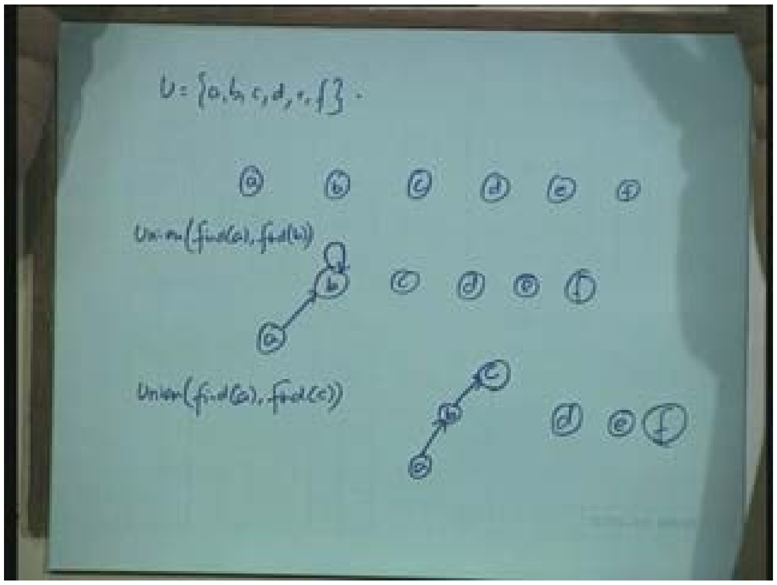
(Refer Slide Time: 21: 06)



What union does is, it takes the root of these 2 trees and links them up and makes 1 point to be other. For instance, here, I might decide to make b point to c. In which case, my new representations would look like this. Now, if I were to do a find a, what will be returned when I do the find a? A reference to c, a reference to this node and when I do the find b, what is returned? A reference to the same node. So, I can compare these two, I can return c or I can return a reference. Actually, it is best to return a reference because then that can be used by the union operation.

What do you mean? c was not alone? If c was not alone, I understand what you mean, if c was not alone. We will come to when c was not alone. So, all you are doing is, taking the roots and merging them. So suppose, at this point I did another operation which was union, find d find e. So, what will I do? I will link up the roots for d and e.

(Refer Slide Time: 21: 42)



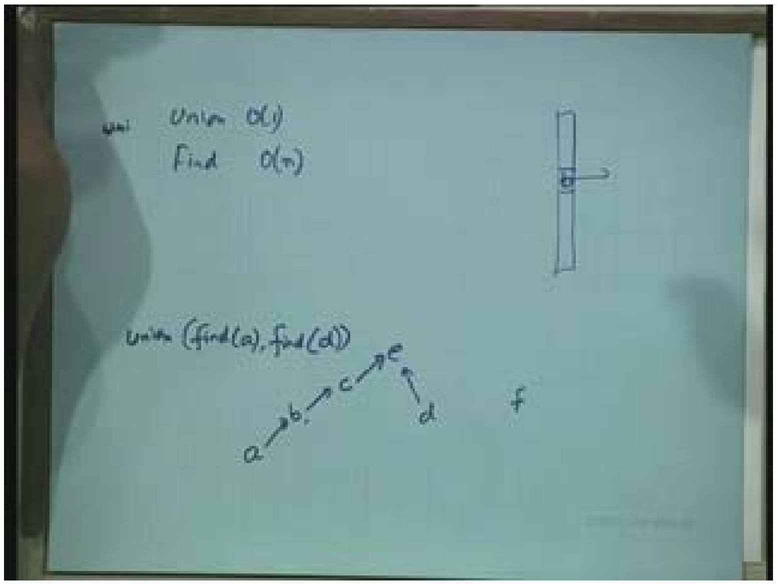
So, let say, I decide to make d point to e. Now as you can see, d and e are not alone. If I decide to do an operation like, let me keep this picture there and if I do, let say, union or let me write it down here. Union, find a comma find d then, what does find a return? It returns a pointer to c, d returns a pointer to e. I need to link up these 2 nodes. So, I can make c point to e or make e point to c, whichever I please. Let say, I decide to make c point to e. This is what I would get there. And of course, f would be sitting on its own. Is everyone understands, what the procedure is?

You understand what the find operation is? What does find operation do? It starts from the element and keeps tracing the pointers up, till it hits the roots. Exactly, so, you have a list of vertices. When I have an edge, I have its 2 end points, I have the vertices and from that vertex list, I must have the reference to this node. Recall, you have a data structure for your graph in which you have an array.

Suppose, I had an adjacency list of presentation, so this array would contain my list of vertices. I could have another reference from here to this node here, for every, so, this was node vertex b, try to have another reference from here to here, so that I can access the state. This is just referring to this particular node and so this will always remain the same. So, what is the problem with this implementation? How much time does union take?

So, union takes now as input, references of root nodes. And so, all it as to do is to modify 1 pointer, 1 reference to point to the other, to refer to the other. So, union takes order 1 time but, how much time does find takes? Find could take a lot of time because it might go through a very long ways to reach the root, in the worst case. Can you construct a sequence of unions in which this would happen? Write at the side first to merge a b, then a c, then a d, then a e, and a f and if you were to doing the union in this order then, things would go back.

(Refer Slide Time: 25:38)

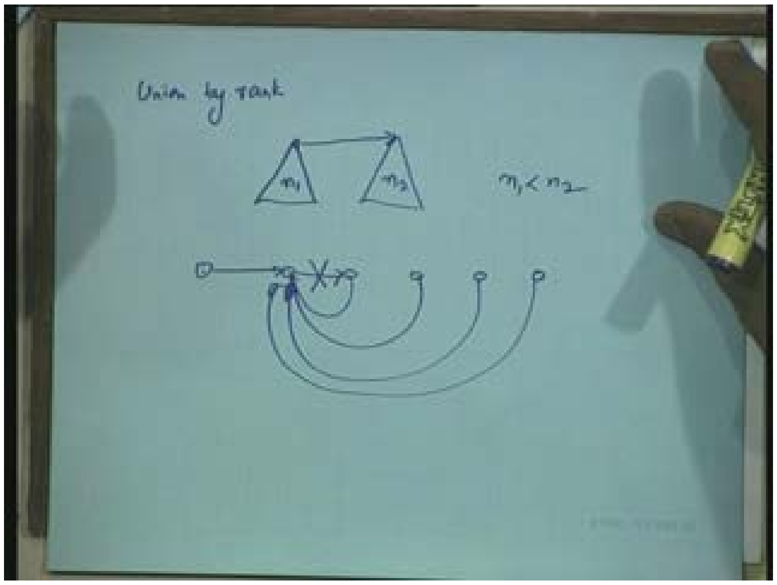


We have to do the unions in a more clever manner. Find? No, we will do the union in a clever manner. So recall that when we were linking elements, we had that option, either link make 1 point to the other or point, make the other point to the first. So now, we will exploit that.

So, I am going to use a rule called union by rank in which if I have 2 trees and suppose, this has n_1 nodes in it and this has n_2 nodes in it. Then, I will make the lighter tree point to the heavier one. So this, without loss of generality, let say, n_1 less than n_2. Then, I will make this point to that. We will make the lighter point to the heavier. Now, you will not have a this kind of a scenario in which you know, if you have this, let say, 6 elements; first you made this point to this and then, when you are trying to combine this and this, you will not make this point to this anymore.

What will happen now? You will make this guy point to this and now, this tree has 3 nodes in it. So, if this combines this, you will make this point to this and this point to this and this point to this. So that now, what is the height of this tree that I get? 1 only, and so, find will take very little time. (hindi) So, we have to see that if we use this root, what can be the height of the tree in the worst case. How high the tree become? How high can the tree become if we use this root?

(Refer Slide Time: 28:24)



Anyone? Login, why? You are trying to construct the worst case. But that need not be the way we do things. That might not to be worst case. How will you argue that this rule of union by rank will lead to trees which have height?

So, what is the claim we want to make? No, not the height is minimum. A tree with n_1 or n, n_1 nodes let say, has height less than or equal to log of $n_{1.}$ Suppose, I have to make, prove this claim, at tree with n_1 nodes, set any point if I have a tree with n_1 nodes in it, it has height at most log of n_1 (hindi) by induction.

Good. So, let us use induction. I am not going to write down the proof formally, but I will tell you what the procedure is. So, I am combining 2 trees: one, n_1 nodes, the other, n_2 nodes. Without loss of generality, let us say, n_1 is less than or equal to n_2. Let us assume that the induction hypothesis is true till this stage of my procedure. That means (hindi) height is less than or equal to log of n_1 and (hindi) is less than or equal to log of n_2 (hindi) Everyone with me?

Now, we have to show, as a consequence of this I will get a new tree with how many nodes in it? n_1 plus n_2 could be the number of nodes in the new tree. So, I have to argue that its height is no more than log of n_1 plus $n_{2.}$ Let see whether that is true.

So, what are the 2 cases? n_1 strictly less than $n_{2.}$ What will be the height? No, what will be the height? The height could be, so, height of resulting tree is either the height of this tree, so we have done this. It is either the height of this tree or it is the height of this tree plus 1.

So, height of resulting tree (hindi) height of resulting tree is less than or equal to max of h_2 comma h_1 plus 1. It can take a value of h_1 plus 1 also. (hindi) the height of resulting tree fine, equals max of h_2 fine (hindi) Now let see, this is the height of the resulting tree.

If this value equals $h_{2,}$ there are 2 possibilities: either this value equal to $h_{2,}$ but h_2 less than log of n_2 which is less than log of n_1 plus $n_{2.}$ The other possibility, as this quantity equals h_1 plus 1 which is less than or equal to log of n_1 plus 1 which is equal to log of 2 times n_1 which is less than or equal to log of n_1 plus $n_{2,}$ because n_2 is greater than or equal to $n_{1.}$ Actually, we have not used the fact that n_1 is strictly less than $n_{2.}$ Have we used that fact? So, it will become equal, but that is okay. So, I do not really need this.

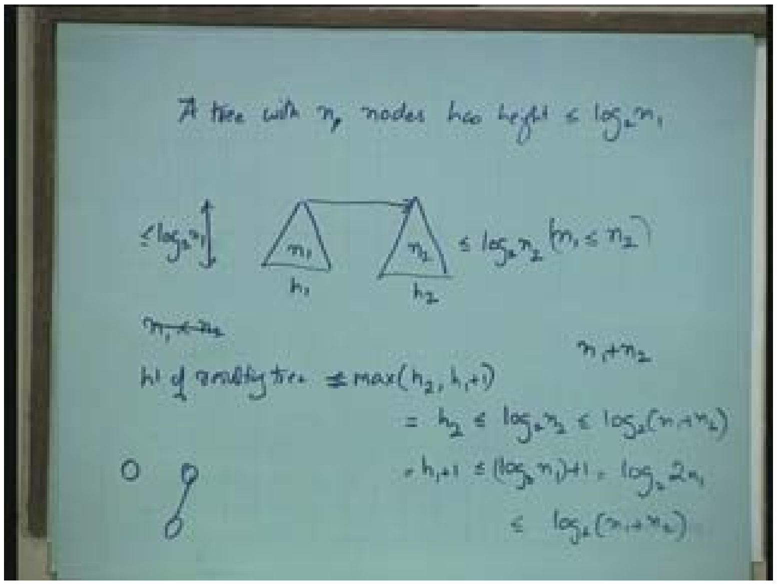
What I was said? One of the trees is, has lesser or equal to number of nodes than the other, if they are equal, actually you can connect it in any way. So, I made 1 point to the other. This, by induction hypothesis this height is at most log of $n_{1,}$ this is height at most log of $n_{2.}$

What is the height of the resulting tree? It is either the height of this tree or it is height of this tree plus the 1. If it is the height of this tree then, it is log of n_2 which is less than or equal to log of n_1 plus $n_{2.}$ If it is this tree plus 1, height of this tree plus 1, then it is log of n_1 plus 1 which is log of, the same as log of 2 times n_1 which is less than or equal to log of n_1 plus $n_{2,}$ because n_2 is larger than $n_{1.}$

This is the base case 2. When n equals 1, the height becomes 0. Let us define the tree of 0 with the only 1 node, as having a height zero. (hindi) If n equals 2, this becomes 1 which is okay. When we have 2 nodes in the tree, it has height 1 then, by definition. (hindi) So now, what are we saying? If this is, if the tree has only 1 node in it, this is height 0. If this is the case, that is height 1. So, I am counting the number of edges on the longest path from the, from one of the leaves to the root: counting the number of edges and not the number of nodes.

Everyone with me, so, what does this show? Is this the complete proof? What am I doing in induction on? Number of nodes in the tree, so, I am assuming that the statement is true for all nodes of a certain number, less than a certain number, I can say, it is true for this when I link this and I get a tree with larger number of nodes, it will continue to be true.

(Refer Slide Time: 36: 14)


A tree with n_1 nodes has height $\leq \log_2 n_1$
$\leq \log_2 n_1$
n_1
n_2
h_1
h_2
$\leq \log_2 n_2$ $(n_1 \leq n_2)$
$n_1 + n_2$
ht of resulting tree $\leq \max(h_2, h_1+1)$
$= h_2 \leq \log_2 n_2 \leq \log_2(n_1+n_2)$
$= h_1+1 \leq (\log_2 n_1)+1 = \log_2 2n_1$
$\leq \log_2(n_1+n_2)$

Great, this is called union by rank. Rank meaning, the rank is the number of nodes in the tree. You can also do a union by height. As in, you can keep the, make the shallow tree point to the tree with larger height. That could also work. Let see why?

What am I doing now? This is a tree of height h_1, this is tree of height h_2. h_1 less than or equal to h_2. I do this, I make root of h_1 point to the root of h_2. I am just showing you 2 alternative ways to do the same thing. Now, what should my induction statement will be? How will I proof, what is the claim I should try to make? What holds true?

So, a tree of height h; what should I try and proof? A tree of height h has at least, 2 to the h nodes. This is converse of that claim. There we were talking of a tree with so many nodes of height at least log of h.

Here, if the height is h, then it has at least 2 to the h. At most, it has height at most log of n_1, here we would write that the tree of height h has a large number of nodes in it, at least, 2 to 3 h. So, that means that you can never have a tree of height more than log of n, because if the height of the tree was more than log of n, then it will have more than n nodes in it. It is not possible because, there is only n node in the graph to be … in a collection to be … So, that will place a log in bound on the height of any tree.

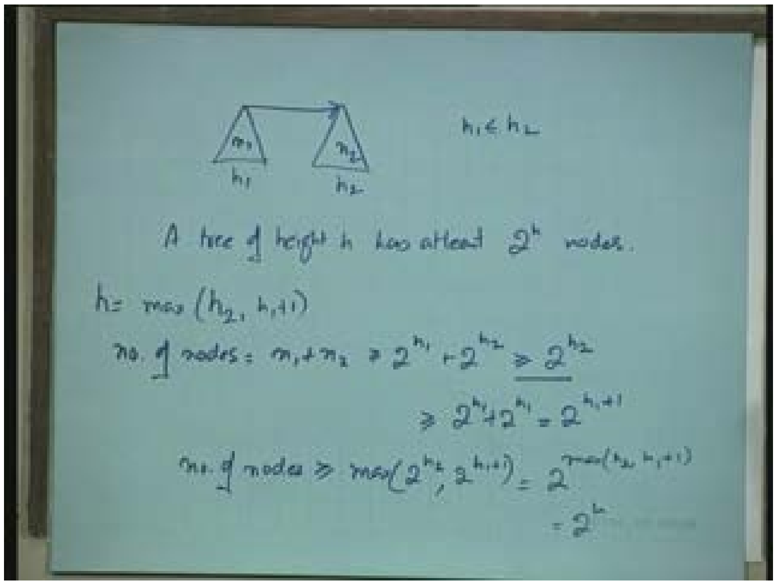
How will we prove this? Once again by induction, suppose, claim is true for all trees of height up to a cretin number and then when I do this linking, the resulting tree has height h_2 max of h_2 comma h_1 plus 1 once again. h_1 plus 1 will occur only if h_1 equals h_2.

Now, this is the height of the new tree. Let say, h is the height of the new tree, this is this. Why is this h, why is the number of nodes, now, what is the number of nodes in the new tree? Equals n_1 plus n_2; number of nodes in this tree plus the number of nodes in this tree.

Now, we know that n_1 is at least 2 to the h_1, n_2 is 2 to the h_2. So, this quantity is greater than or equal to the 2 to the h_2. That is one way of thinking of it and since, h_2 is more than h_1, this is also greater than or equal to 2 to the h_1 plus 2 to the h_1 which is equal to 2 to the h_1 plus 1. So, the number of nodes is grater than or equal to max of 2 to the h_2 comma 2 to the h_1 plus 1. I can also write it as, max of this is 2 to the max of h_2 comma h_1 plus 1 which is 2 to the h. (hindi) Proofs are very similar, if you look at it carefully. You are just turning them around, both of these schemes can be used to do the union.

What do they both ensure? Why does this ensure that the height is more than, no more than log n? because, if a tree has a height more than log n (hindi) 2 to the, a tree of height log n will have 2 to the log n which has n node in it already. If it has height more than login n, if it will have more than n nodes, which is not possible.

(Refer Slide Time: 41:47)



$h_1 \leq h_2$

A tree of height h has atleast 2^h nodes.

$h = \max(h_2, h_1+1)$

no. of nodes $= n_1 + n_2 \geq 2^{h_1} + 2^{h_2} \underline{\geq 2^{h_2}}$

$\geq 2^{h_1} + 2^{h_1} = 2^{h_1+1}$

no. of nodes $\geq \max(2^{h_2}, 2^{h_1+1}) = 2^{\max(h_2, h_1+1)}$

$= 2^h$

So, from this argument, no tree will have height more than log n and this argument, we have already said directly that no tree has height more than log n, log of the number of nodes in the tree. Since the maximum number of nodes in any tree is at most n, no tree has height node the log n. So now, given this, how much time does union takes?

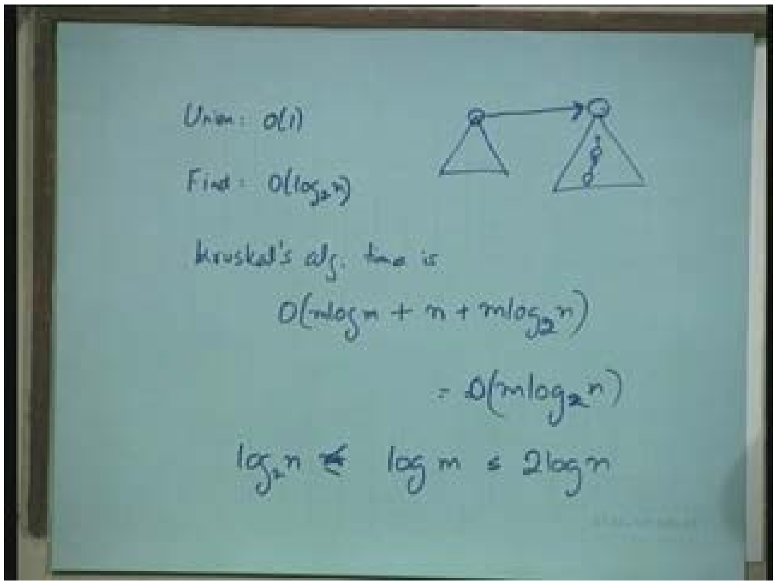
Constant time? Yes, so something has to be done more and what is it that has to be done? In the root node, we have to keep the track of, either the height or the number of nodes, whichever it is. And, this is an information which is not difficult to maintain because, when I do an union, this value is updated to either the height, if you maintain the height it becomes the max or if you are maintaining the number of nodes, you just add this quantity that.

So, how much time does union take now? Constant time because, you just have to update this variable and do this reference update. How much time does find take now? I start from a node and keep moving up the tree. Since the height of the tree is never more than log n, find takes no more than log n time.

Total time taken by Kruskal's algorithm then becomes, time for sorting, m log n plus, how many unions did we say we need? At most n unions, every time I include an edge, I need 1 union. Since, each one of them is taking constant time, this is just order replay. How many finds do I need? For every edge, I may need 2 finds.

So, m times log n. What is this? m log n (hindi) What is log m? (hindi) n square? (hindi) value is 2 log n and minimum value (hindi) in the connected graph? log of n. So, log m and log n are the same things. log m is theta log n. They are in the constant. So, whether you write log n here or log m here, it is immaterial. They are the same quantities.

(Refer Slide Time: 45:26)



Union: $O(1)$

Find: $O(\log_2 n)$

kruskal's alg. time is

$$O(m\log m + n + m\log_2 n)$$

$$= O(m\log_2 n)$$

$$\log_2 n \le \log m \le 2\log n$$

So, I will, the next thing I want to do is to show you 1 way of improving the union find data structure. We saw the method, one way so, this data structure we said if you do union by rank then the time required for find improves. It becomes order log n. You can further improve the time required for find by using the technique called path compression whose analysis, we are not going to do in this class. But, you will do learn in your algorithms course.

What is the technique? So, it is just the following. You have your tree in which you are doing a find. (hindi) So, you went up the tree like this to do this find. Now the question is, why do not we do something at this point, so as to improve the performance of future finds?

You might have to find this node again? Yes, we will modify these things. We will make this parent pointer, point straight to the root and not just this guy, everyone on this path. Why are we doing this? Anyone? because now, this guys become closer to the roots.

What is, I am not drawing rest of the tree but now, you can see that this node is this, this, let me put a dot here. This guy, connected to just 1 link to the root. Let me put a cross here and this last 1, without anything is this. Of course, there are sub trees hanging down here. They will continue to hang down here, because there could be other nodes pointing towards this so they will continue to. But when I later, so, what are we doing as a consequence of this? We are bringing the nodes closer to the root, thereby, reducing the height of the tree and that reduces the number of, the time required to a find. No, the root is not changed to this tree. The root of this tree remains as before. Union by height, you are worried about if the union procedure requires height.

The root of this tree will point to the root of other tree. No, these will not change then, these will not change then, we are not going to change these now. If I take the union of this with someone else that is okay. I did that, I am not going to change this pointer. So, if you are opting this procedure, then union for the union operation, you should not work with height. But, with number of nodes: you understand why? because we are changing the height of the tree by doing this.

We might be changing it in a manner which might be hard to recompute. After I do this compression, may be this was a node, there was leaf here which was the farthest from the root. When I change this, the height of the changes, but how do I now compute the new height of the tree in constant time?

Very difficult, so we use the union by number of nodes. That is the procedure we are going to use. So, the metric will be which tree has lesser number of nodes. The tree with lesser number of nodes will be made to point to the tree with larger number of nodes or the root of the tree with less than number of nodes will be made to point to the tree, the root of the tree with larger number of nodes. Everyone follows this? questions?

What we doing in? What are we doing in path compression? What we are doing in path compression is that, we say, when I am searching for this node, when I doing a find operation on this node, in any case I am going to travel all the way up. I am going to traverse this link and I am going to traverse this link and this link and this link.

Why do not I do something now, which will make future easier for me? Future finds easier, lesser time, so, what I do is, that once I do this, I will may be make another pass of this and now change the pointers to directly point to there. We do not know the root, we do not know when to make the, no, but when I am doing the find, I need to know where the root is that is why I am doing a find in the first place. How do I know, the root point there. Only when I reach the root, I could know where the, what the root reference is. So, I need to make a second path to update.

Every time we do a find, we will update the pointer. Then there is no need. So, for here instance I did not change this. This is no point, what does it mean to change the pointed this. This we will change because, that gives us, the next time the picture is this, we will look at this picture and decide what to change and what not to. So, again when I go throw

here, what is the point in changing it, because it is already pointing the … This becomes the picture, I do not have this picture any more in the, the next type. How do we directly go to the nodes in this tree? We keep a reference.
For every node, when I am keeping the node in this union find data structure, the node corresponds to a vertex of my graph. I do not just create the union find data structure as the standalone entity. I have to link it up with my graph somehow. For every vertex, I have 1 node here, I keep a kind of cross reference from the vertex in my linked list data structure or the adjutancy list data structure for the, in which I have the vertices, I would have the reference to the corresponding node in the union find data structure. So that when I have to reach a particular vertex, I can follow that reference and find out which connected component that vertex belongs to (hindi)

So, with that we are going to end today's class on the union find data structure and we are going to discuss another algorithm for computing minimum spanning trees in the next class.