

Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 02
Lecture - 33
Union-Find Data Structure Using Pointers

In the last lecture, we saw an array based implementation of the union find data structure, in which the find operation to constant time and we use an amortized analysis to claim that the union operation over a sequence of m unions would take $m \log m$ time. So, effectively each union was a logarithmic time operation, amortised over the entire sequence. Now, we will see a more sophisticated implementation which has even better complexity.

(Refer Slide Time: 00:29)

Union-Find data structure

- A set of elements S **partitioned** into subsets, or **components**, $\{C_1, C_2, \dots, C_k\}$
 - Each s in S belongs to exactly one C_j
- Support the following operations
 - **MakeUnionFind(S)** — set up initial components, each s in S is a separate singleton component $\{s\}$
 - **Find(s)** — returns the component containing s
 - **Union(C, C')** — merges the components C and C'

So, recall that the union find data structure keep track of a partition of a set S and supports three operations. One is the make union find which creates a trivial partition, where each elements belongs to it is own, each element is in a single term partition named by itself. So, we use the same names for the partitions and the elements, so each s in a partition called S each small s . Find tells us which partition a given element belongs to and union combines two components or two partitions into a single one.

(Refer Slide Time: 01:01)

Implement with arrays/lists

- Implement Union-Find using array
Component[1..n], lists **Member**[1..n] and array **Size**[1..n]
- **MakeUnionFind(S)** is $O(n)$
- **Find(s)** is $O(1)$
- Amortized complexity of each **Union(k,k')** is $O(\log m)$ over a sequence of m operations

So, the array based implementation uses an array called component to tell us which component each element belongs to, it keeps an array of list to tell us which elements belongs to each components. So, it is a kind of reverse mapping, for each component k it tells us which members of the set belong to that. And finally, it keeps track of the size of each component, because we merged components by taking smaller once and relabeling them with the name of the bigger once. So, make union find was an order n operation, find was an order one operation and the amortized complexity of the union operation was $\log m$ for a sequence of m operations.

(Refer Slide Time: 01:37)

Implementing with pointers

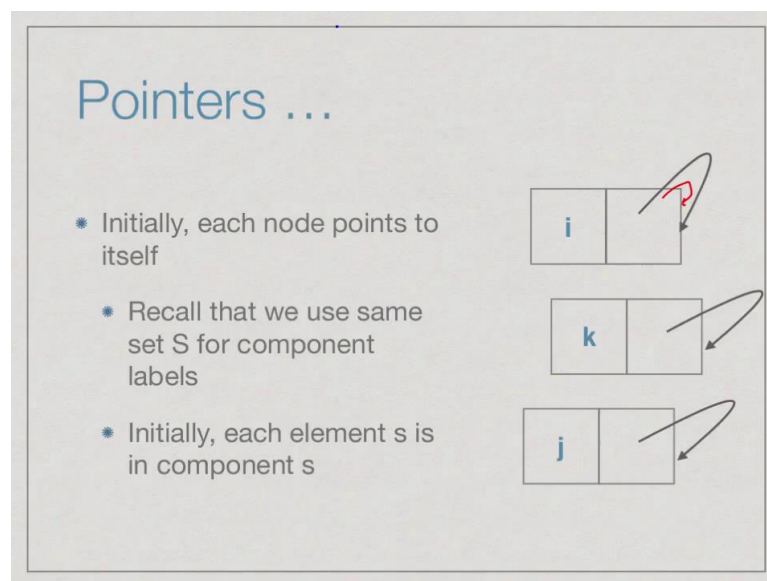
- Each element of the set is a node with two fields
- Name: the name of the element
- Label: pointer to the set containing the element
- Recall that we use same set S for component labels

The diagram shows two nodes, each with a 'Name' field and a 'Label' field. The top node has 'i' in the Name field and its Label points to the bottom node. The bottom node has 'j' in the Name field and its Label points to itself. Handwritten red text indicates 'i ∈ component j' and 'j ∈ component j'.

So, we will now have a different representation, we will use nodes with pointers. So, each element of the set will now be represented as a node, the node will have two parts, it will have the name which is the element that you are keeping track of and a label part which keeps track of its component, so it is a pointer to the component. Now, remember that the names of the components are the names of the elements. So, if we have a name here for example, we have name j it says that this node represents the element j and its label points back to itself, so it says this is j belongs to the component j.

Here on the other hand, we have the element i and through this we go to the first node which points to itself and this set that a i belongs to component j. So, i belong to component j, so we are read a kind of we are making heavy use of the fact that the names of the elements and the names of the components are the same. So, we are kind of in some context, we are interpreting the name field as name of the element and when you following pointers, we are interpreting the name field as a name of the component.

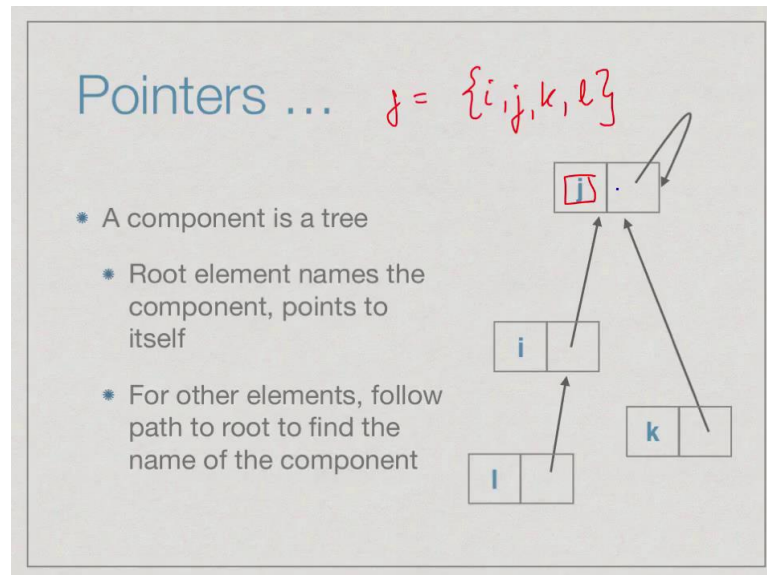
(Refer Slide Time: 02:55)



So, make union find is the initialization, so what it does is, it will set up each component as a single term containing an element of the same name. So, i belongs to the components i, so there is a node i which points to itself, then there is a node k which points to itself indicating this is a component called k containing k, this is the components called j containing j. So, we have n such components 1 to n, so we have n

such nodes each of which contains the name in the first component and a pointer to itself in the second component saying itself single terms component.

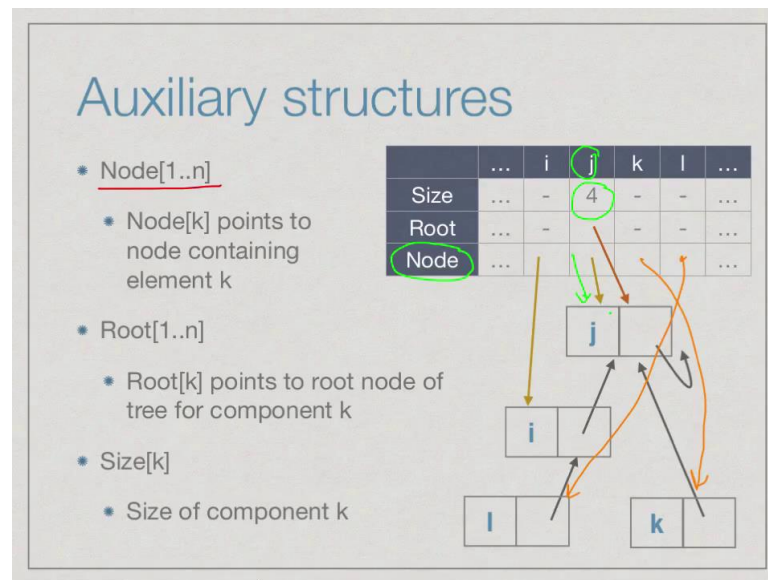
(Refer Slide Time: 03:31)



So, after a period of time we will see that when we merge node, these pointers will point instead of themselves it will point to other things. So, here for example, could be a component containing i, j, k and l and the structures says that the name of the component is actually j. So, this is the component j and it contains i, j, k, l, so you can see that the element i has a pointer not to itself, but to another nodes. So, if you follow this we find that it is pointing to an element j which points to itself, so it is name of the component of i is j.

Similarly, for l if we start from l, l points to i, i points to j and so on, so we will see that this is how we will use to this is a strategy, we use to implement find. We have to start with the node and go up the path and this will be a tree, so every path will end up the root and the root will be the name of the component.

(Refer Slide Time: 04:25)



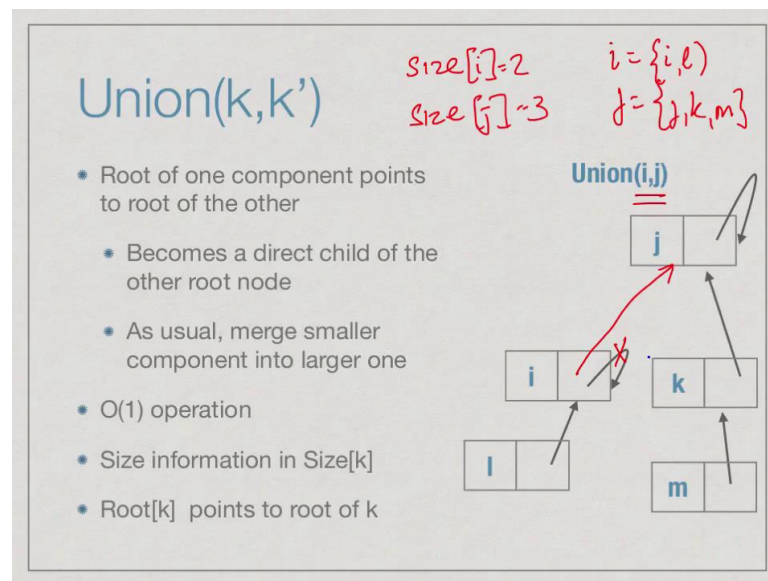
So, just to help us navigate this tree a little easier, let us assume that we have the following additional arrays of pointers and numbers. So, we will keep track first of all of where each node is. So, we have an array called node here and each element of the array is a pointer to the node containing that element. So, the entry i says node i is here, entry j says the node j is here and so on. So, there will correspondingly be an entry which says that node k is here and node l is here and so on.

So, every element in our actual set will have a pointer from this node array to the actual node in the tree, when the set of tree is set of components, the actual node where that value lies, then we need to know which of the components which are currently active. So, here in this particular segment i, j, k, l which are the component to j, there use to be a component to i, i when i was initialize the use to be a component k, the use to be component l. But, these names are no longer in use, because these components have a got merged directly or indirectly with j.

So, right now among i, j, k, l only j remains, so we say that the root of j is, the component j is the node containing j and for all the other labels i, j, k and l, we say that i, k, n, l there is no component called i there is no component called k, so we just have a blank same, same there is no root for these things. So, if you want to find the root of this component, go to the node i and then work your way up to it.

Finally, we will use a similar strategy as before in terms of merging by size, so we will just explicitly keep track of the size of each component. So, here it is says that size of this components i, j, k, l is 4 because there are currently 4 elements. So, we have this node pointer which points to each element in the set where it currently lies, which node contains that element. We have a root pointer which tells us for each component which is currently active in our partition, which is the root of that partition and then we have the size which tells us the size of each component.

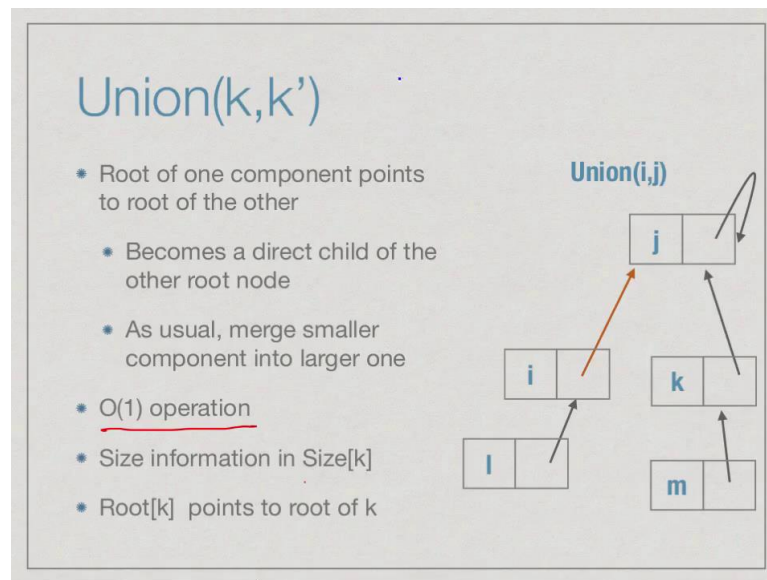
(Refer Slide Time: 06:29)



So, when we do union we basically do this merging of the smaller one to the larger one. So, here we have two components, we have component j which is j, k and m and we have component i which is i and l. So, now we want to do the union of i and j, so now we first will somewhere we will have recorded that the size of i is 2 and the size of j is 3. So, having recorded this, we have to now therefore merge i into j.

So, what this means is that this edge which says that i is a root of the string has to be destroyed and instead we have to put a link like this and merge these two trees and to one larger tree.

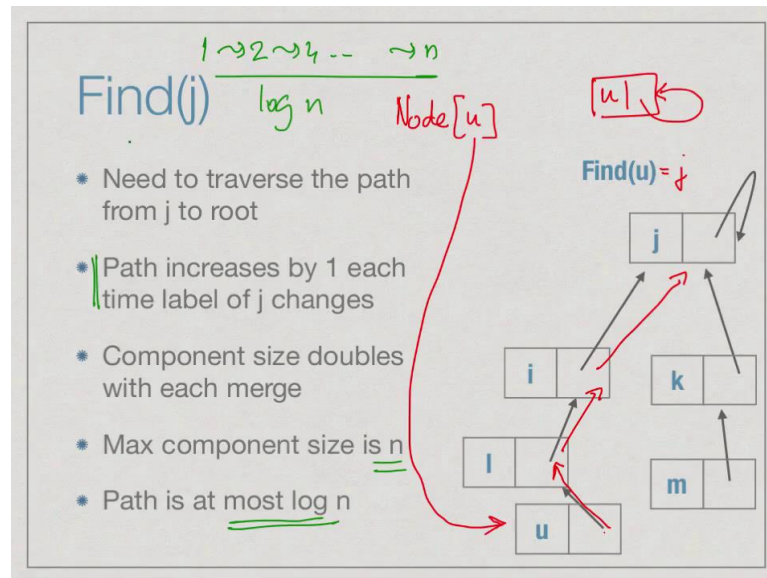
(Refer Slide Time: 07:26)



So, if we do that this is the resulting tree, now we have a single component called j of course, we would have updated the size of j to be the size of i plus the whole size of j. So, now it will say 5 instead of 3 and notice that this whole operation is an order 1 operation, because we take a fix number of steps to find out the sizes of the two components to find the roots of the two components and decide which root to massage into which other root.

So, whether to make i point to j or j point to i, so this is an order 1 operation and that is because we have the size information tell us the size, we do not have a go through the component to find it size and we have this root information it directly tells us where the roots of the two partition are. So, which is we have to go directly to that place and make one point to the other or vice versa.

(Refer Slide Time: 08:09)



Now, find on the other hand as we saw requires us to start from the value that we want to check and work away up to them. So, supposing we say find of you, then remember that will have this node array and so node of you will say that the node containing u is here. So, we will start there and then we will follow this path, so we will say this points here and this points here and this points here and finally, when we reach a node which point to itself that is the name.

So, then we will say therefore, it find of u equal to j, so this takes time proportional to the path that we traverse from the node to the root. Now, remember initially we had a path for u which looks like this, so initially every node has a path of length 1 to itself sent that it is it same root. Now, why does is path change it changes, because of the union operation, every time we do a union 1 node which use to point to itself, now points to another node and everything below it as it is path increase by 1.

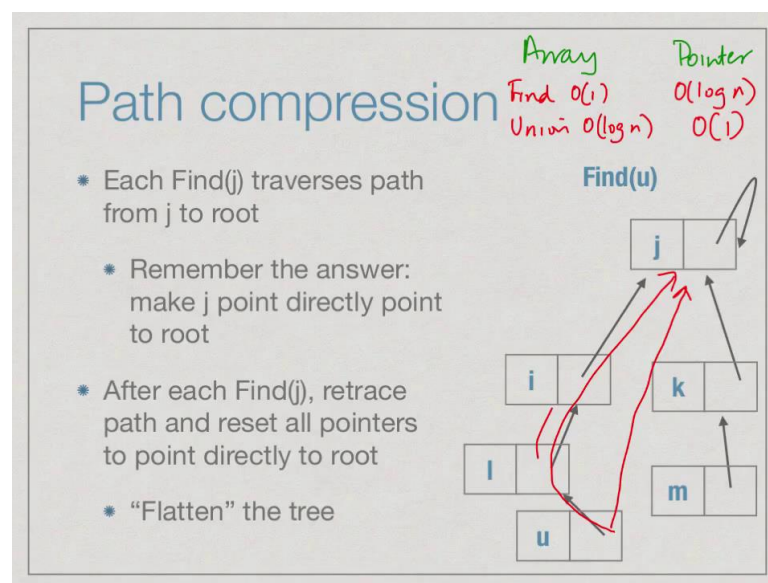
So, therefore, the path increases by 1 every time the label of j changes, if you going to go from a find of j from j to the root, how much time this takes changes by 1 each time j changes it is label. But, as before we are doubling the component size, which time we merge. So, which time we do an union, then label containing j contains quizzes many element as a did before.

Now finally, the best you can do is to combine all the nodes in your graph into a single component, the maximum component size is n. So, therefore, we start with a component

to size 1, then we double it to 2 and we double it to 4 and eventually we cannot go faster. So, this whole thing can be at most $\log n$ steps, so therefore $\log n$ times the label of j can change, each time the label changes, because we take the root of another component attach it here, the path link increasing by 1.

So, the path length increases 1 plus 1 plus 1 $\log n$ times, so the path is at most $\log n$. So, with this analysis we can see that, because we are merging smaller components and to larger components find will take $\log n$ time.

(Refer Slide Time: 10:29)

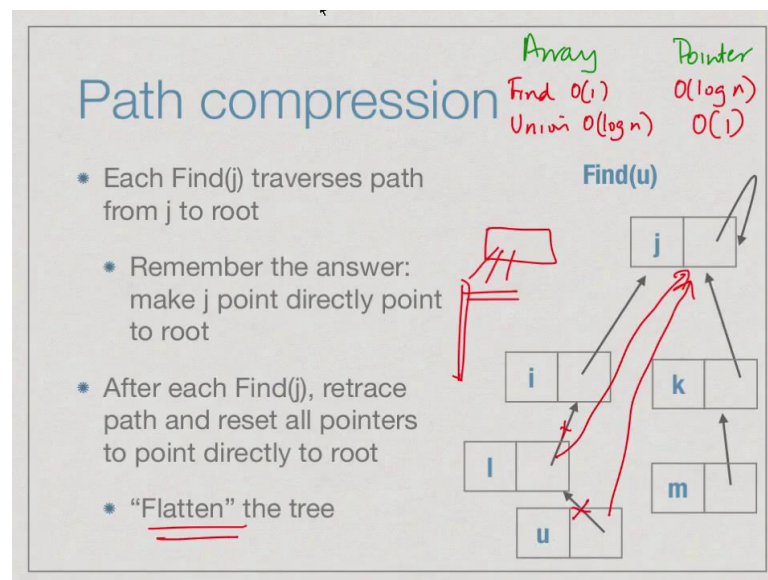


So, we are now at a place where we are dual to the earlier data structure, in the earlier data structure with arrays. So, we had find was a constant time operation and we had is amortized analysis that set that union was $\log n$, now we have this pointer base think with says that union is order 1, but find is order $\log n$. So, it is use that we have got essentially as similar kind of structure, it set that we are reverse the complicity of the two operations.

Now, turns out that we can do something much more cleaver an actually make it even more efficient to find. So, we can keep the order one of course, for union, but order $\log n$ can actually be reduced. So, the key is that once we have traversed this path from u to j , we know is some sense that not only do you know that the path containing to component containing u is j , we know the component containing l is j you also know the component containing i is j .

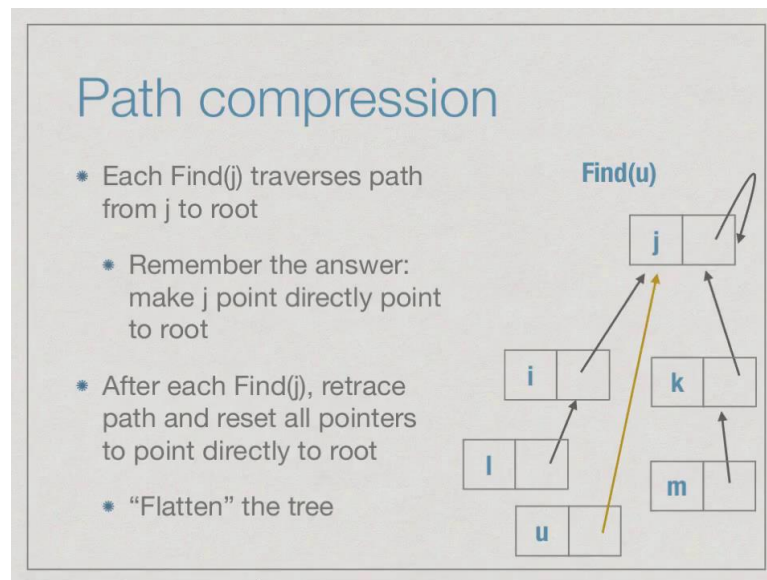
So, why do not we make use of this information and not recomputed each time we come back. So, what we will do is, we will do the usual find j, so we will traverse the path and then we will go back from j to the root and say now I know this is j. So, let me bypass is whole in may be point directly to j, so I have remove the earlier things.

(Refer Slide Time: 12:02)



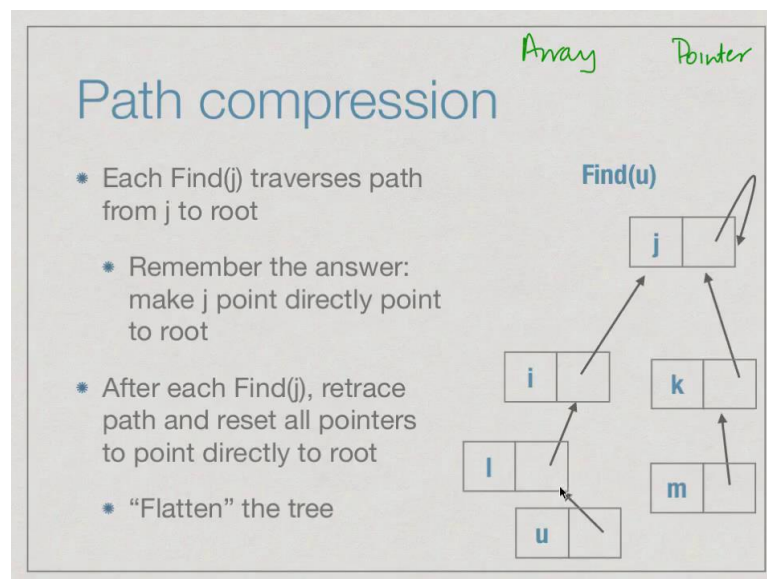
So, I will remove this edge and directly go, similarly before I remove of the go there. So, I will say I can remove this edge directly, so for each step I can remove until I finally, reach the one with just below the root. So, basically now I will end up having a flat tree along this path, this path it use is look like this it not been flattened out like this. So, every think now has a directed access to the path, so we have flattening the tree.

(Refer Slide Time: 12:26)



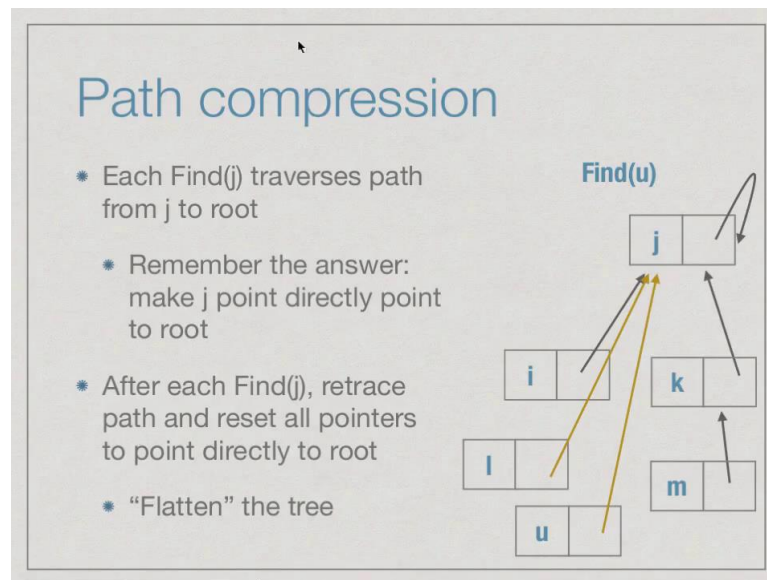
So, if you want to look at the pictorially, so I start with this.

(Refer Slide Time: 12:36)



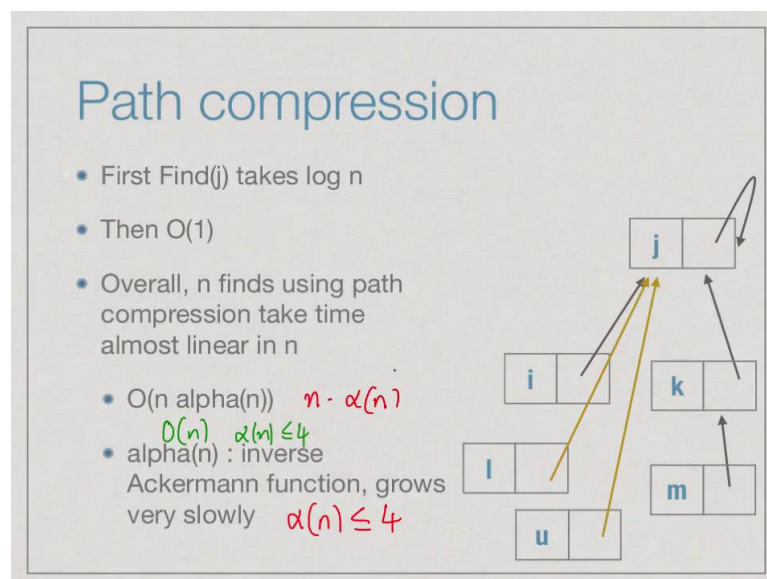
And then if I look at the current situation, then I will replace this edge from u to l as an edge from u to j directly.

(Refer Slide Time: 12:42)



Then, I will replace the edge from l to i as an edge from l to j directly and so now I am end up this flat tree. So, now subsequent look up, so I am going to take only one step from u and l.

(Refer Slide Time: 12:52)



So, the first find takes $\log n$ time, then it takes constant time, so now you can actually show I mean is this is not an easy calculation. So, we are have not going to try and attended now, but after this you can show that just like we can do an amortised analysis for in the earlier array based implementation to get an overall $\log n$, $m \log n$, cost $m \log$

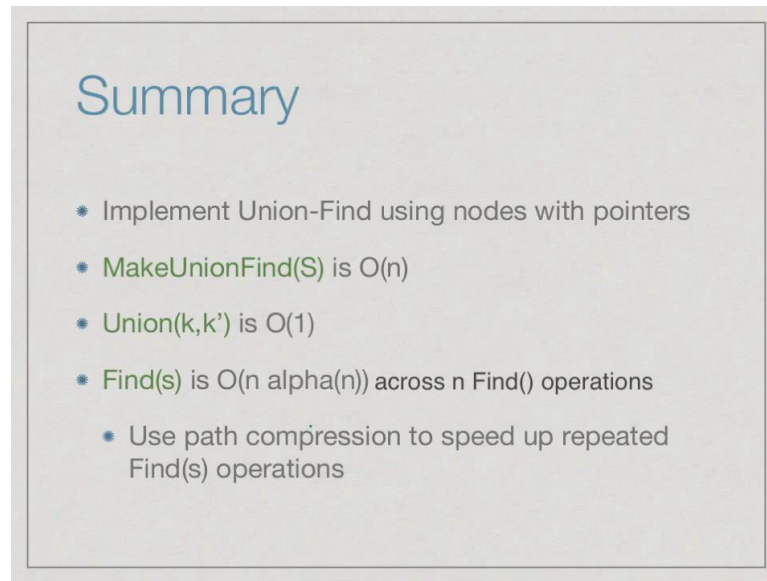
m cost for m union operations. Here, you can actually show that a total of n finds will actually take something almost linear in n.

Almost linear in n is actually n times of function called alpha of n and this alpha of n is something which is the inverse of the Ackermann function, the Ackermann function is something which grows very fast. So, it is a function which grows faster than any explanation, now this is like the log of the inverse Ackermann, I mean the log of Ackermann. So, you know that for the exponential function, the inverse is the log. So, that is why log n is nice for us, it is slow log of 1000 is 10, log of 1 billion is 20, 1 million is 20, log of 1 billion is 30 it was very slowly, because in other directions growing fast 2 to the power 10, 2 to the power of 20, 2 to the power of 30 goes very fast.

So, similarly Ackermann goes very fast, so if you take what is the corresponding log function, for what value is this the Ackermann inputs it grows very slowly. In fact, it is claim that alpha of n is at most 4 for any value of n that you are likely to encountered in a practical computing problem that you can solve.

So, you can think of this has been something like order n, because we know that alpha of n is less than equal to 4. So, this is like 4 n, so path compression gives as a drastic reduction from n log n which we would get without path compression to something which is linear. So, therefore, now by moving to this pointer base think, we have actually achieved a considerable saving compare to the more direct array based implementation that behave in the last lecture.

(Refer Slide Time: 14:48)



Summary

- Implement Union-Find using nodes with pointers
- **MakeUnionFind(S)** is $O(n)$
- **Union(k,k')** is $O(1)$
- **Find(s)** is $O(n \alpha(n))$ across n Find() operations
 - Use path compression to speed up repeated Find(s) operations

So, to summarise if we implement union find using nodes with pointers, then we can do of course, the initial set of the make union find and linear time, union now becomes a order 1 operation. Because, we just directly access the two roots and merge this smaller root with the bigger root and find now if you use this path compression trick, although and principle it is $n \log n$ for n operations, it is $n \alpha n$ for n operations if we use path compression.