Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

Week: 06

Lecture 26 Kruskal's Algorithm 2

We will continue our discussions on Kruskal's algorithm. We have seen at a high level how the Kruskal's algorithm works. We have arranged the edges of the graph in the increasing order of their weights and considered them one after another in that order. The hope is the cheaper edges would be considered first and included and costlier edges would be considered later and if it forms a cycle, they are rejected. okay so in this way we are collecting a set of edges we need to first prove that the set of edges form a tree which we have seen already and we have to prove that it forms a minimum spanning tree, the edges that are collected in A, A is the name of the set that we had given is indeed forming a minimum spanning tree. So let us look at the proof, okay.

The proof is again by contradiction A is going to have n minus 1 edges let x1 x2 x n minus1 be the edges in A in the increasing order of their weights okay. Remember our assumption that all edge weights are distinct, if this is not forming a minimum spanning tree, we are going to arrive at a contradiction. So assume that there is a minimum spanning tree with a cheaper cost, therefore weight of T dash is less than weight of T and let y1 y2 y nminus1 here there is a 2 y1 y2 be the edges of T dash again in the increasing order of their weights.

So we have two sequences of edges with the increasing order of their weights, okay. A is x1 x2 x n minus 1 and in T dash we have y1 y2 etc y n minus1 these are not same.

$$A = x_1\ x_2 \cdots x_{n-1}$$

$$T' = y_1\ y_2 \cdots y_{n-1}.$$

If these two sequences are same then T dash equal to T we are assuming that they are not same, T dash has a cheaper cost, that means this sequence will differ at some point. So, let j be the point at which they differ that is x1 equal to y1, x2 equal to y2, x3 up to x j minus 1 equal to y j minus 1 but x j is not equal to y j

$$x_1 = y_1, \quad x_2 = y_2$$
$$x_{j-1} = y_{j-1},$$
$$x_j \neq y_j .$$

The first position where the sequence differ okay it should differ somewhere. Since xj was chosen greedily, that is after choosing x1 etc up to x j minus1, xj is chosen as that edge that has got the cheapest possible weight.

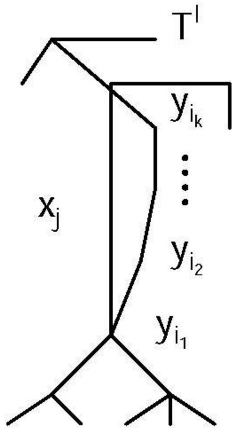
Therefore, any other edge will be costlier, that is the reason why we have this property wxj is less than wyj.

$$x_j \neq y_j \text{ and}$$

$$x_j = y_j \text{ for } 1 \leq i \leq j-1$$

Now in order to force a contradiction we are going to construct a spanning tree T double dash that is going to be cheaper than T dash. Note that T dash is a minimum spanning tree, if I am able to construct something cheaper than that that is a contradiction to the minimality of T dash, right. So that will be a contradiction. Therefore, this construction if I demonstrate it leads to a contradiction, we will recap that again first let us construct the T double dash in the following way, recall our discussion earlier you have a spanning tree you have a non-tree edge added and when you add a non-tree edge a unique cycle is formed.

We know that xj is not equal to yj, so this is the tree T dash



xj is not in the tree since xj is not in the tree when you add a unique cycle is formed in T dash so this is xj plus T dash. Here is the cycle the unique cycle that you see so that is going to have certain edges, the edges of T dash are labeled y1, y2, y3 and so on. So let yi1, yi2, yik these are the edges in the cycle C okay, yi1, yi2, yik. If this is a proper subset of y1 to yj minus1, Suppose this is completely contained in y1 to yj minus1 that means y i1 equal to xi1, y i2 equal to xi2 that is because xj x1 equal to y1, x2 equal to y2 up to xj minus1 equal to yj minus1 because of this property,

$$y_{i_1} = x_{i_1}, y_{i_2} = x_{i_2}, \cdots, y_{i_k} = x_{i_k} \text{ because } y_1, \cdots y_{j-1} = x_1, \cdots x_{j-1}$$

if these edges are subset of this they will match the corresponding x value right. Therefore, we have the following situation all these things are also x i1, x i2 and so on therefore you have xj, x i1, x i2 etcetera x i we have used the notation k.

xik they are all edges of a tree okay this yik is xik, yi2 is xi2, yi1 is xi1. this is xj now you see that there are some edges in T all these things are in T and they form a cycle that is not possible right. Because edges of T are acyclic T is a tree no subset of edge can form a cycle therefore it is not possible that you have yi1 to yik is completely contained in this that means it should contain some vertex some edge outside this edge set that means this path has got an edge yt where yt is where t is greater than or equal to j right it is not in the set therefore it must be some yt and t is greater than or equal to j it could be j or more than j. Therefore, wyt remember the edges of y are arranged in the increasing order therefore this will be greater than or equal to wyj but we know that wyj is greater than wxj because xj was greedily chosen therefore you have wyt greater than xj okay.

$$w(y_t) \geq w(y_j) > w(x_j)$$

Now consider including of xj and removing of yt okay. This is going to be another spanning tree again recall the way in which we have, suppose you have spanning tree and you add a non-tree edge it forms a cycle you remove one of the edges in that cycle you get another spanning tree. Okay, so I have included xj and I have removed yt

$$T'' = T' + x_j - y_t$$

I have included one and I have removed one of the edges in that cycle okay. So T double dash will be another spanning tree what is the cost of wT dash it is wT dash plus wxj because you have added this xj and you have removed yt therefore, minus yt but wyt is larger than wxj. So, you are adding a value and subtracting a bigger value. So, the whole thing becomes smaller therefore, wT double dash is smaller than wT dash.

$$w(T'') = w(T') + w(x_j) - w(y_t)$$

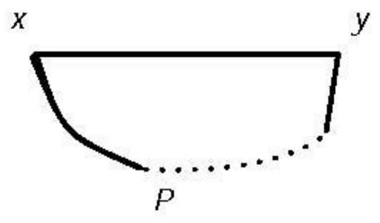
$$< w(T')$$

okay and this is a contradiction to the minimality of T dash, this what I mentioned earlier. I have now a spanning tree which is cheaper than minimum spanning tree that is not possible therefore this contradiction shows. a spanning tree smaller than with a smaller cost than the Kruskal's spanning tree is not possible, that means Kruskal's algorithm is producing minimum spanning tree okay. This completes the proof for the fact that the algorithm works correctly, right. So we keep on examining the edges in the increasing order of their weight, include some and reject some whatever you have included is forming a minimum spanning tree.

Therefore it proves the correctness of the algorithm however the implementation of that is very subtle, you have to check whether a edge that you are considering is forming a cycle with the edges that have been included already that is a non-trivial task and we are going to focus on the implementation details of how we are going to get that done okay.

Therefore the key question is when an edge is added, whether it forms a cycle with the edges that are already been considered okay. So this graph theoretic question is going to be answered in the following way. We are not going to try to physically construct a cycle, that is not needed, whether it is forming a cycle or not we want only the yes or no answer for it.

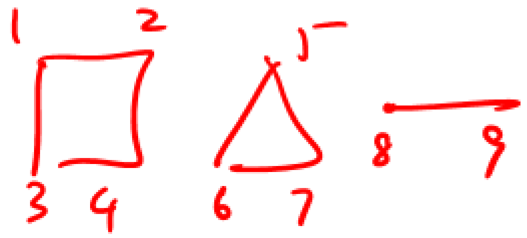
The question can be framed in an equivalent manner and this equivalent form is simpler to answer. I have a collection of edges, I have a new edge, whether it forms a cycle or not is extremely complicated to determine but there is an equivalent way. How will the new edge form a cycle with a collection you have already, there must be a path and the adding of this edge, path plus edge will be that cycle. This edge is giving end to end connection and there is a path, together it forms a cycle. Therefore if e is the edge with the end point x and y the question is, do we have a path with the edges that we have already collected connecting x and y okay.



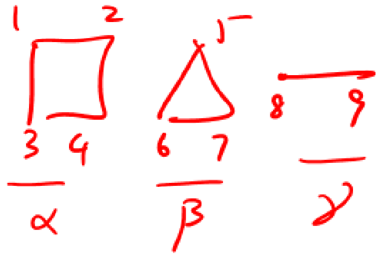
So this is the equivalent form, instead of checking if an edge e is forming a cycle we can check if there is a path connecting x and y in the edges that you have collected that is in A, A is a dynamic set, up to this point A has certain edges do we have a path connecting x and y in the edges that you have collected so far okay.

$$T = (V, A)$$

Now this is where another graph theoretic notion of connected components is helpful, what is a connected component of a graph, by connected component what we mean is all vertices in that component are connected by a path, okay. So if a graph is like this, let us say 1, 2, 3, 5, 6, 7, 8, 9, this graph has got 3 connected components. You can go from 1 to 2 you can go from 1 to 4 you can go from 3 to 2 you see that you can go from anywhere to anywhere in that cluster but you cannot go from 4 to 6 there is no path right.



So each one is a connected component okay. We can give some label, some name, we need a way to refer them in a computer program right, give some name for that and assume that we have a function that returns the name of the connected component of T, okay, whatever is the name. We will establish some naming convention but whatever is the name it should return, for example if you name this as alpha, if you name this as beta if you name this as gamma, find 6 should return beta because beta is the name of the component containing 6. Find 2, should return alpha because alpha is the name whatever is the name it should return the name of the component.



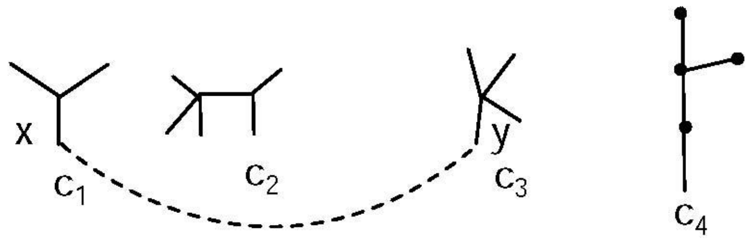
We will see how we are going to name the components etc. But if there is a name available for reference for each component, the vertex is in some component and it should return the name of that component, assume that we have one such function. You find x is not equal to find y, that means x and y are in different connected component, if they are in different connected components there is no path, if there is no path adding this edge is not going to form a cycle. Therefore I have an edge e equal to xy, x is in one component, y is in another component So x y is not going to form a cycle.

$$e = (x, y)$$

So you can see how my cycle detection test is converted into a simple test based on a function called find. If find x is not equal to find y, e equal to xy will not form a cycle with A. It will not form the cycle with the edges in A therefore I am going to include. If find x equal to find y, if x and y are in the same component, obviously there is a path that path and xy will form a cycle therefore e must be ignored that is the reason why else ignore e. If find x is not equal to find y, we are going to include, otherwise we are going to exclude, this takes care of a very simple cycle test replaced by a find function.

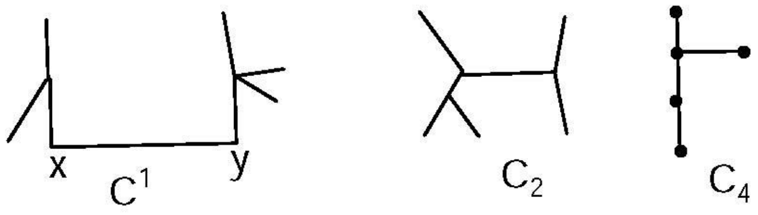
We have not yet discussed how the fine function is implemented we will come to that later but assuming that we have a fine function this cleanly handles the test of forming the cycle or not.

Now what is actually happening is after you include A, A changes A has now one more edge. Now the connected component of TA are different okay. So you look at this picture.



In this picture let us assume that this is TA, T equal to VA, A is the set of edges. This set of edges so far is forming 4 connected components, the 4 pieces C1, C2, C3, C4 are shown in the picture. Now the dotted line, it is an edge, it has been included, so A is updated by the addition of one more edge. Now the connected component C1 and C2 are fused together. Now any vertex in C1 can go to any vertex in C3 and vice versa. So that becomes one single component okay.

Therefore when you fuse those two with one component, you get the picture like this C dash.



You can see that the xy that is added has merged two components. The other components C2 and C3 are there what is C dash? C dash is C1 union C3, C1 and C3 has been fused together

$$C' = C_1 \cup C_3$$

Look at this picture here you have C1 and you have C3, C1 and C3 are put together so now the updated A has got only 3 connected components. okay. So whenever a is updated how is A is updated, by adding an edge, how do you add an edge if it is n vertices are in different components we are adding that edge.

So the n vertices are in two different components and those two components are to be put together We call that operation as union operation. So union operation where X and Y are names of two connected component, results in a new connected component consisting of union of the vertices of the components of X and Y. Put them together like this. all vertices in C1, all vertices in C3 they are put together and they are joined by XY. Now you have a new connected component okay.

This operation taking two connected components and merging into a single component is called union operation. Which connected component you are taking is given by the name of the connected component X and Y. Now armed with these two operations, we can rephrase the pseudocode that we have specified, okay. So here is the refinement of the algorithm we have seen at a very high level, okay. G is a connected undirected edge weighted graph. A is the set of edges collected by the algorithm. TVA is the subgraph of G induced by the edge set A. For a vertex x, find x returns name of the connected component of T containing x. Union XY merges the connected component X and Y into a new connected component okay and L is the list of edges in the increasing order of their weights with this notation here is a much better presentation of the algorithm that we have seen okay. A is initialized to empty set, e is the first edge of L okay.

```
Initialize:
  $A = \emptyset$,
  $e =$ First edge of $L$;
Initialize the names of the connected components of
  $T = (V, A)$
Process:
  While (NOT END OF $L$)
        Let $e = (x, y)$
        $X =$ Find $(x)$;
        $Y =$ Find $(y)$;
        If $(X \neq Y)$
            Add $e$ to $A$
            Union $(X, Y)$
        $e =$ Next $(e, L)$
Output:
  $T = (V, A)$
        \\ $T$ is a MST of $G$
```

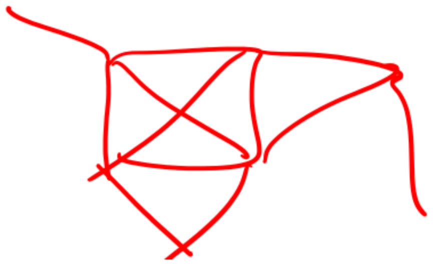
We initialize the names of the connected components how we are initializing the names we will see okay. Initially when A is empty, each vertex is a connected component this is how we have seen this is the configuration when A equal to empty, T equal to VA when A is empty no edge only vertices are there. So what are the connected components of T? Connected components of T are individual vertices. So how many connected components TA has now? n connected components okay.

This is what we have to initialize. We will see how that is done later. Assume that they are initialized and some names are given for each one of the connected components. This is initializing. How does the algorithm work, while not end of L, we are going to examine each edge let e equal to xy. X equal to find x, Y equal to find y that means X is the name of the connected component containing X, Y is the name of the connected component containing Y, if they are not same then we are going to add e and perform the union of these two connected component.

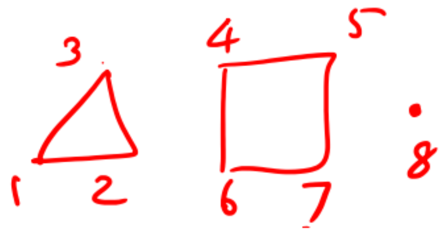
If X equal to Y, we simply ignore that edge and go to the next edge, right therefore I am not writing anything in the else part. Else do nothing therefore there is no else part if X is not equal to Y, I add that edge after adding edge to A I update the connected component information by the union operation and then go ahead, for the next edge this is e equal to next e L, it is the next edge in the list. So I am going to consider the edges in L one after another. If an edge has this property, I include otherwise I ignore and I just keep moving along L, finally I will have a A, we have seen that that A is the A we are looking for, it is forming a minimum spanning tree. So what remains to be shown is how do I implement find function, how do I implement the union procedure, okay.

So you have a set of edges and they induce a connected component, the information and processing related to the connected components are to be done, how are we going to do that, okay. So we need to process connected components of T. We have to find the name, we have to fuse the connected components. So we have to maintain and manage connected components of T. How are we going to maintain and manage the connected components of T and perform these operations? The moment I fill the detail the whole program gets completed okay.

So that is what we are going to see next, okay, how are we going to work towards this. So since connected components are components generated through the paths, okay, we will formalize our ideas in terms of the paths and reachability, okay. Two vertices are said to be connected if there is a path from x to y in G. If every pair of vertices are connected, the graph is a connected graph. This is an example of a connected graph.



You take any pair of vertices, you can find a path from one to another, example of a connected graph. We have already seen example of a disconnected graph, for example this is an example of a disconnected graph with the 7 vertices 1, 2, 3, 4, 5, 6, 7 you can have even isolated vertex 8.



So this is an example of a disconnected graph okay. A graph that is not connected is disconnected. In a disconnected graph, each maximally connected subgraph is called the connected component, this is one connected component and this is one connected component and this is another connected component. It is maximal in the sense anything outside this is not reachable anything inside this is reachable mutually reachable, 1 and 2, I can reach but 1 and 4, I cannot reach from 1 to 4, it is outside therefore it forms a maximal connected subgraphs okay.

There is a beautiful way to describe these connected components in terms of abstract mathematical discrete mathematical property called relation, okay. So let us say that, let G equal to VE is a graph,

$$G = (V, E)$$

vertex x is related to vertex y if there is a path from x to y. So we write like this, x is related to y if this property is satisfied.

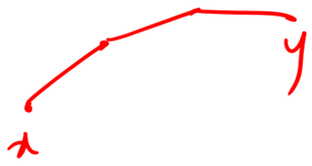
$$x\, R\, y$$

In discrete math you might have already known that a relation is an equivalence relation, if it has three properties okay reflexive, symmetric and transitive.

Reflexive means A is related to itself, x is related to itself, Do you have a path from x to itself of course trivial you do not go anywhere stay at x therefore x is connected to x

$$x\, R\, x$$

Suppose x is connected to y is y is connected to x of course yes this is an undirected graph the same path the same set of edges considered in the reverse order defines the path from y to x, if there is a path from x to y the same set of edges in the other order defines a path from y to x.

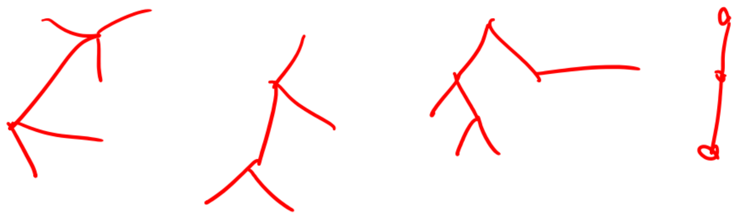


So if x is related to y, y is related to x. If x is related to y and y is related to z is x is related to z if x is related to y and y is related to z. do you have this property this is called transitive if x is related to y and y is related to z should x be related to z.

$$x\,R\,y \qquad y\,R\,z \qquad x\,R\,z$$

Put in this term if there is a path from x to y, there is a path from y to z, do you have a path from x to z ,of course you can have, simply concatenate these two you will get a walk, remove all the cycles you will get a path we are not worried about weights and other things here. It is simple reachability, if you can reach from x to y and if you can reach from y is the vertex is z obviously you can reach z from x because of this way of defining a relation okay among the set of vertices is an equivalence relation, okay. This equivalence relation is going to play a role in our context, okay.

Let us consider A, how A looks like we know that A is acyclic. Why A is acyclic, if any edge is forming a cycle we are not including it therefore, A will continue to be an acyclic structure. A may have several components, therefore each component is connected and it will be acyclic, therefore, each component is a tree, that is the reason why every acyclic graph is a collection of trees ,that is why it is also called the forest. It is an acyclic graph, no cycles. If it is acyclic and connected, there is only one piece that will be a tree. But if it is acyclic and disconnected, each one will be a tree because there is no cycle but there are several connected components. So each one will be a tree. It will be a collection of trees.



Therefore, A is always a forest. Finally A will become tree, until the final edge is added A will have several components finally A will have n minus 1 edges and A will become a single tree, but when you look into the development of A, okay it will have several trees and these trees are fused together and one big tree is formed, okay. So we have seen the relation and we have seen that the relation is an equivalence relation A is I mean vertex u is related to vertex v, if there is a path that is an equivalence relation, there is a fundamental theorem in discrete math which talks about the partition induced by an equivalence relation, several known result okay.

So if R is an equivalence relation on V, then R induces a partition on V. Every equivalence relation is going to split the underlying set into a partition. What is a partition? Partition is a way to split a set into disjoint set. It is like a partitioning of a room, you split the room into disjoint areas, non-overlapping areas, that is partitioning same partitioning of a set, splitting a set into several sets but they are all disjoint okay, we will see the formal definition of partition next but every equivalence relation is going to induce a partition on V is a basic result. Therefore A is going to induce a partition because the path relation on A is going to induce a partition of the vertex set, A is defined on the vertex set V.

Each partition is a connected component, A is acyclic, therefore each connected component is a tree okay. Therefore what A is doing is, A is creating lot of connected components which are nothing but trees. That is what A is, A consists of several trees, each tree correspond to one connected component of it. When you add an edge to A, the two connected components which are at the end vertices, are fused that is what union operation is, okay.

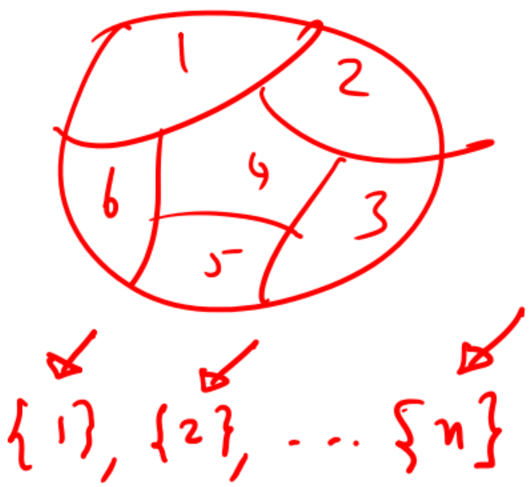
Here is the formal definition of partition. What is a k partition? By k partition, what we mean is splitting a set into k parts. Therefore, you are splitting k into S1, S2 Sk they are disjoint, so Si intersection Sj is empty, together they form the whole set,

$$S_i \cap S_j = \emptyset, i \neq j$$

union of Si is V and any two of them are disjoint this is called the partition of V.

$$\bigcup_{i=1}^{k} S_i = V$$

So partition of V here I have split V into k disjoint parts, so if this is V you can imagine a partition like this. So you have 1, 2, 3, 4, 5, 6 disjoint part, 6 disjoint sets, this is a 6 partition example right. and n partition of V is a connection of n singletons, V is this set 1 to n. So this is a partition 1 to n, you have split n into you have split V into n sets, they are disjoint, together they form V, therefore this is a n partition of V.



n partition of V corresponds to n isolated vertices, no edges, each vertex is a connected component and that is represented by individual sets okay.

Now when you are maintaining a partition you have to give a name for the set because you have to identify which set it is in, there are several sets we need a naming mechanism, name is just an identifier for an object, when there are several sets how do you identify each set. So here is a naming convention, we are going to name a set by the minimum number contained in that set, right. The minimum number in that set will act as a name for the set, in fact we may use any element because this is a partition, we can use any element, to begin with we will identify the minimum element in the set to act as the name for the set, example minimum element acting as a name for the set, element of a set acting as a name for the set, it is one way you can have other ways also but let us say minimum element is.

So you have the partition 1, 2, n. What is the name of this set? There is only one element and that is a minimum element. Therefore, the name is 1. The name of this set is 2. The name of this set is n, okay. Very simple.

Suppose I have 1 to 14 and for 1 to 14, here is a 5 partition. 5 partition, I have split 1 to 14 1, 2, 3 up to 14 into 5 sets here is the 5 set, 1, 2, 4, 7 in 1, 3, 8, 10 is in 1, 5, 6, 9, 11 is another, 12, 14 is in another, 13 is in another, like this you have 5 sets. What is the name of 1, 2, 4, 7, the minimum is 1, therefore the name of this set is 1 the name of this set is 3 because between 3, 8 and 7, 3 is the minimum, so I am going to give that as a name like this you see that the name are given and this gives rise to what is known as name array representation of a partition, okay, name array representation of the partition is associating for every element the name of the set containing it. For example what is name of 3, name of 3 is 3, 3 is in this set the name of the set is 3, what is name of 11, name of 11. Why name of 11 is 5, 11 is in the set called 5, 11 is in the set called 5.

Eg:	{1,2,4,7},	{3,8,10},	{5,6,9,11},	{12,14},	{13}
	↓	↓	↓	↓	↓
Names:	1	3	5	12	13

Name [1]=1	Name [7]=1	Name [13]=13
Name [2]=1	Name [8]=3	Name [14]=12
Name [3]=3	Name [9]=5	
Name [4]=1	Name [10]=3	
Name [5]=5	Name [11]=5	
Name [6]=5	Name [12]=12	

Therefore name so name of an element is the name of the set containing it, name of x is name of the set containing We know how to name a set. So go to the set and find the name of that set. Why name of 9 is 5? Where is 9? 9 is in the set 5, 6, 9, 11. For the set 5, 6, 9, 11 what is the name? 5 is the name. Therefore name of 9 will be 5. okay therefore this is called the name array representation.

Suppose I have name array representation here is a very simple way to implement the union algorithm and find algorithm okay. What is find? Find should return name of x, done one line that is it find x, return name x, one line code, you want the name of the set containing x, that is available in the name array, that is how we have defined the name array therefore this is done. How will you find union? Suppose A and B are two names, assume that A is less than B, how do you find a union, you have to rename all the bigger value to the smaller value, if name of i is B, change that to A right. So let us go back to this suppose I want to find the union of 3, 5 union 3, 5 so these are all the elements of 5, this is an element of 5 this is an element of 5. All of them will merge in one set. The name of the new set is 3 because 3 is the minimum of 5 and so everybody in 5 will move to 3. So what I do is name of 5, I will make it 3 name of this 5, make it into 3, name of this 5, make it into 3 name of this into 3. So you have moved all the elements which were in the set 5 into a set called 3 that is the logic implemented here.

For i equal to 1 to n, if name of i is B, change the name to A, if the name is B if the name is 5, change to 3, generalized here if the name is B change to a notice that A is smaller than B. So one pass through the array name array will change wherever there is B it will be changed to A, therefore find operation complexity. Complexity of find is order 1, there is only one step ,complexity of union is order n because you are going through the entire array and wherever B you change to A, that is order n.

okay. So we have seen a name array representation and implementation of find and union procedure in that, fitting this code over there completes a description of one implementation. We will see the details related to them in our next session.