Disjoint Data Sets

Outline

- Disjoint set data structure
- Applications
- Implementation

Data Structures for Disjoint Sets

- *A disjoint-set data structure* is a *collection of sets* $\boldsymbol{S} = \{S_1 \ldots S_k\}$, such that $S_i \cap S_j = \emptyset$ for $i \neq j$,

- The methods are:
- *find* (x) : returns a reference to S_i such that $x \in S_i$
- *merge* (x, y) : results in $\boldsymbol{S} \leftarrow \boldsymbol{S} - \{ S_i , S_j \} \cup \{ S_i \cup S_j \}$ where $x \in S_i$ *and* $y \in S_j$
  - merge({a}, {d}) is executed by a union ({a}, {d}) and update of the collection
    $\boldsymbol{S} = \{ \{ a, d \}, \{ b \}, \{ c \}, \{ e \} \}$

Application of disjoint-set data structure

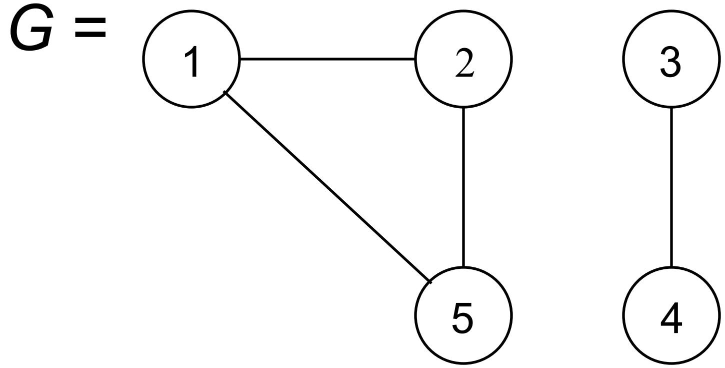
- Problem: Find the *connected components* of a graph.

1. Make a set of each vertex
2. For each edge do:
   if the two end points are not in the same set,
   merge the two sets

In the end, each set contains the vertices of a connected component.

- We can now answer the question: Are vertices x and y in the same component?

Example: Find Connected Vertices



$E = \{ (1,2), (1,5), (2,5), (3,4)\}$

merge(1,2)
$V = \{ \{1, 2\}, \{3\}, \{4\}, \{5\} \}$

merge (1,5)
$V = \{ \{1, 2, 5\}, \{3\}, \{4\} \}$

merge (2,5)
$V = \{ \{1, 2, 5\}, \{3\}, \{4\} \}$

merge(3,4)
$V = \{ \{1, 2, 5\}, \{3,4\} \}$

1. Make a set of each vertex

Set of sets of vertices
$V = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\} \}$

2. For each edge in E do:

Disjoint Set Implementation in an array

- We can use an array, or a linked list to implement the collection. We examine an array implementation only.
  - The size of the array is N for a total of N elements
  - One element is the representative of the set
  - In the array Set, each element i for i = 1,…,N has the value rep of the representative of its set. (Set[i] = rep)
  - We use the smallest "value" of the elements in a set as the representative

Using an Array to implement DS

Set = { {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8} }

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

merge ("4", "7")

Set = { {1}, {2}, {3}, {4,7}, {5}, {6}, {8} }

1	2	3	4	5	6	4	8
1	2	3	4	5	6	7	8

DS implemented as an array

find1(x)

return Set[x]; // $\theta(1)$.

```
union1(repx, repy)
  smaller ← min (repx, repy );
  larger ← max (repx, repy );
  for k ← 1 to N do
      if set [k] = larger then set [k]  ← smaller;
```

$\theta(N)$ in every case. After N-1 union operations the computation time is $\theta(N^2)$ which is too slow.

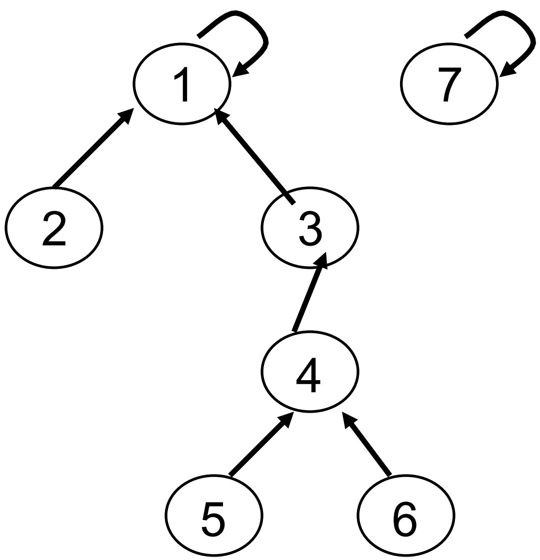
DS is implemented as an array

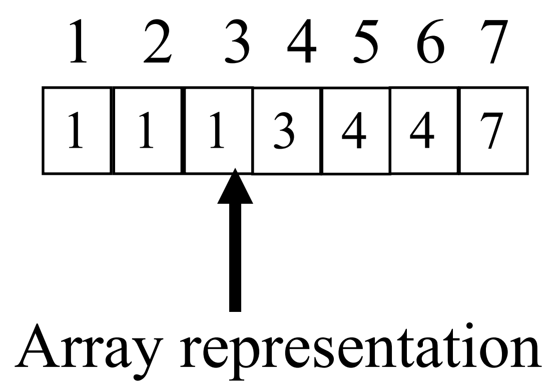
- For the following sequence of *merges* we show the resulting array

Initial array	1	2	3	4	5	6
After merge ({5}, {6})	1	2	3	4	5	5
After merge ({4}, {5, 6})	1	2	3	4	4	4
After merge ({3}, {4, 5, 6})	1	2	3	3	3	3
merge ({2}, {3, 4, 5, 6})	1	2	2	2	2	2
merge ({1},{2, 3, 4, 5, 6})	1	1	1	1	1	1
	1	2	3	4	5	6

Backward forests

- Sets are represented by “backward” rooted trees, with the element in the root representing the set
- Each node points to its parent in the tree
- The root points to itself
- Backward forests can be stored in an array

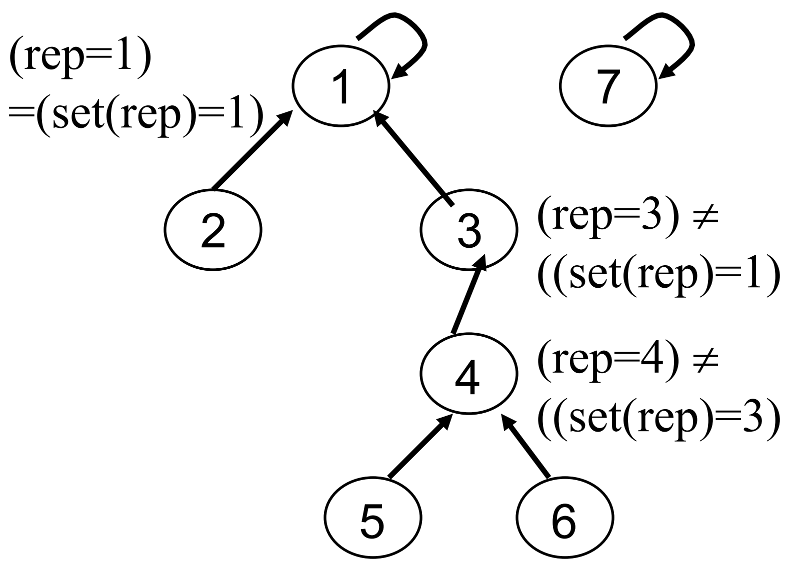




Backward forests stored in an array

```
find2(x)
  rep ← x;
  while (rep != Set [rep ])
     rep ← Set [ rep];
  return rep
```

- *find*2 is O(height) of the tree in the worst case



Example:finds2(4)

1	2	3	4	5	6	7
1	1	1	3	4	4	7

Backward forests stored in an array

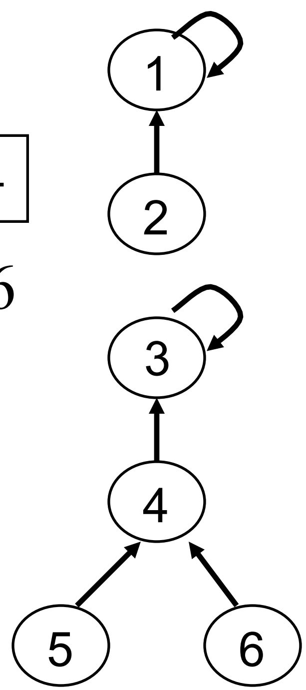
*union*2(rep*x*, *repy*).
  smaller ← min (*repx*, *repy*);
  larger ← max (*repx*, *repy*);
  set [*larger*] ← *smaller;*

- *union*2 is O(1)

Disjoint-set implemented as forests

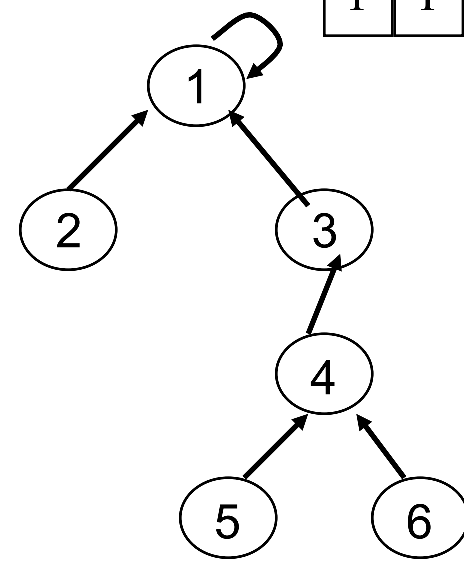
- *Example: merge2*(2,5)
- find2(2) traverses up one link and returns 1. find2(5) traverse up 2 links and returns 3.
- *union2,* adds a back link from the root of tree with rep= 3 to the root of the tree with rep=1.

1	1	3	3	4	4
1	2	3	4	5	6



⇒

1	2	3	4	5	6
1	1	1	3	4	4

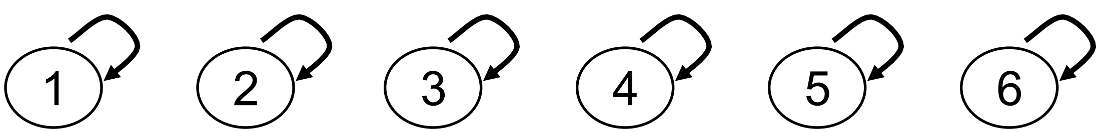


Disjoint-set implemented as backward forests

What is the worst case height?

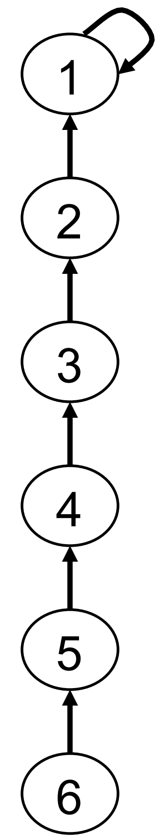
- The following example shows that *N* - 1 merges may create a tree of height *N* - 1
- Now *N* - 1 unions take a total of O(*N*) time.
- n find operations take O(n*N*) in the worst case.
- Initially:

1	2	3	4	5	6
1	2	3	4	5	6



Disjoint-set implemented as forests

- The order of execution of the "*merge*2" affects the height of the trees.
  Consider the following sequence of *merge*:
  *merge*2 ({5}, {6})
  *merge*2 ({4}, {5, 6})
  *merge*2 ({3}, {4, 5, 6})
  *merge*2 ({2}, {3, 4, 5, 6})
  *merge*2 ({1},{2, 3, 4, 5, 6})



Tree of height N -1

1	1	2	3	4	5
1	2	3	4	5	6

Disjoint-set forests with improved height

- A method to improve time by decreasing the height of the trees
- Requires another array that contains heights. Initialized to 0
- We modify *union2* to decrease the height of the trees to O(lg N) in the worst case
- ***union*3 links the root of the tree with the smaller height to the root of the tree with the larger height**
- *find2* = O(lg*N*) and *union3* = O(1)

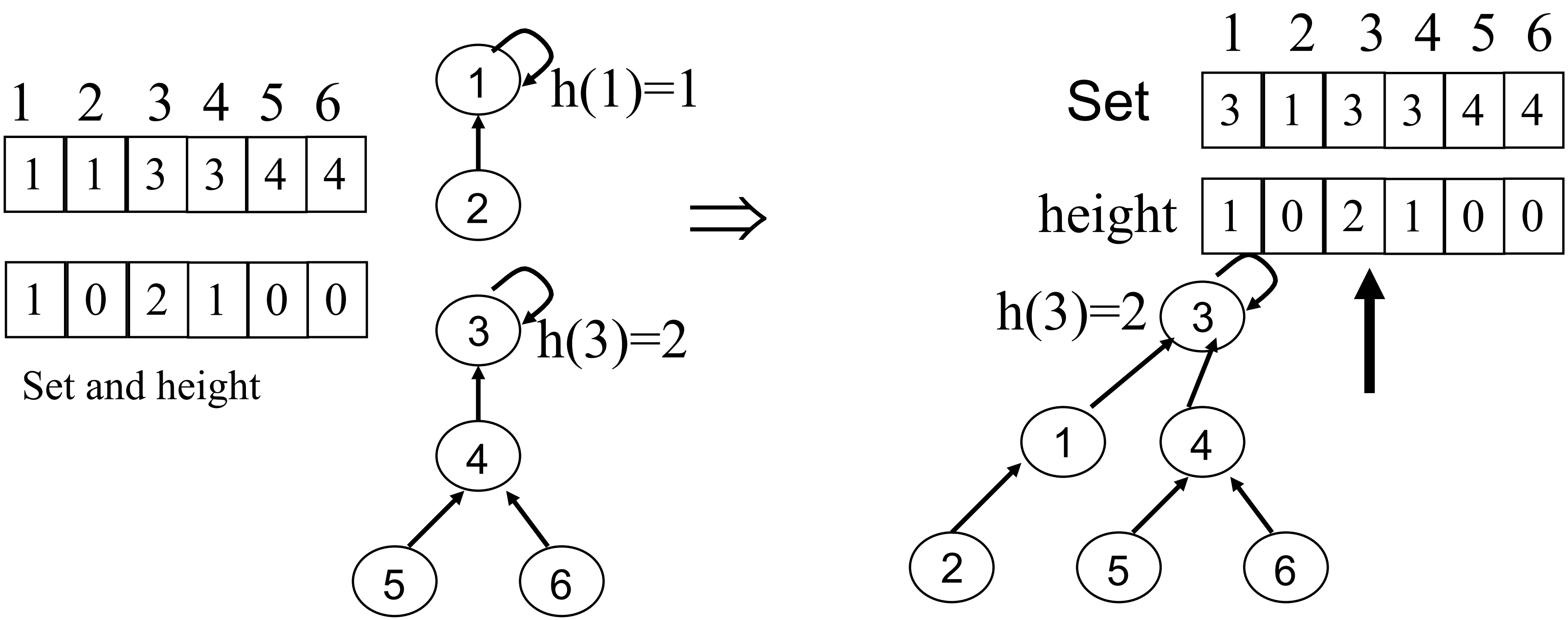
Disjoint-set forests with improved height

*union*3(rep*x, repy*)

```
if (height[repx] == height [repy])
      height[repx]++;
      Set[repy] ← repx;//y’s tree points to x’s tree
else
      if height[repx] > height [repy]
            Set[repy] ← repx  //y’s tree points to x’s tree
  else
            Set[repx] ← repy  //x’s tree points y’s tree
```

Merge with reduced height

- *Example: merge3* (2,5)
- find2(2) traverses up one link and returns 1. find2(5) traverses up 2 links and returns 3.
- *union3,* adds a back link from the root of tree of height =1 with rep=1, to the root of the tree of height = 2 with rep=3.



Disjoint-set forests also with path compression

- Another heuristic to improve time:
  - Path compression (done during *find3*). The nodes along a path from x to the *root* will now point directly to the root.
- Useful when the number of finds n is very large, since most of the time *find3* will be O(1)

Find and compress

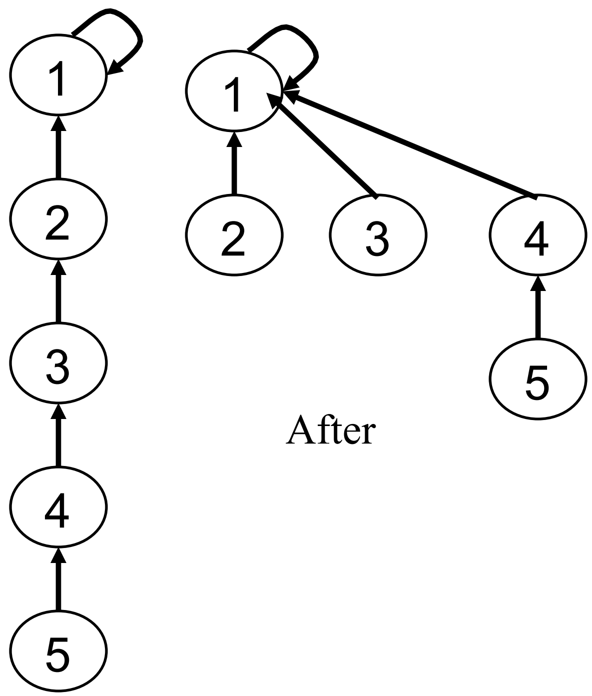
*find*3(x)

```
//find root of tree with x
 root ← x;
 while (root != Set [root ])
     root ← Set [ root];

//compress path from x to root
 node ← x;
 while (node != root)
    parent ← Set[node]
    Set[node] ← root; // node points to root
    node ← parent

 return root
```

Example: find3(4)



Time Complexity

- The worst case time to perform *n* finds and *m* unions for backward forest with improved height and path compression
  - *Approximately linear in n finds + m unions in most practical cases*
    - To be precise, it's $O((n + m)\,\alpha(n + m, n))$ where $\alpha(n + m, n)$ is *the inverse of the Ackermann function*
    - Ackermann's function grows very fast (e.g., A(2,j))
    - The inverse grows at $\lg^* n$
      - $\lg^* n = \lg \lg \ldots \lg n$ (iteratively take lg until it becomes 1 or smaller)
      - Example: lg 65566 = 4
    - Proof is beyond the scope of this class: If interested, refer to Cormen's book (the recommended text)

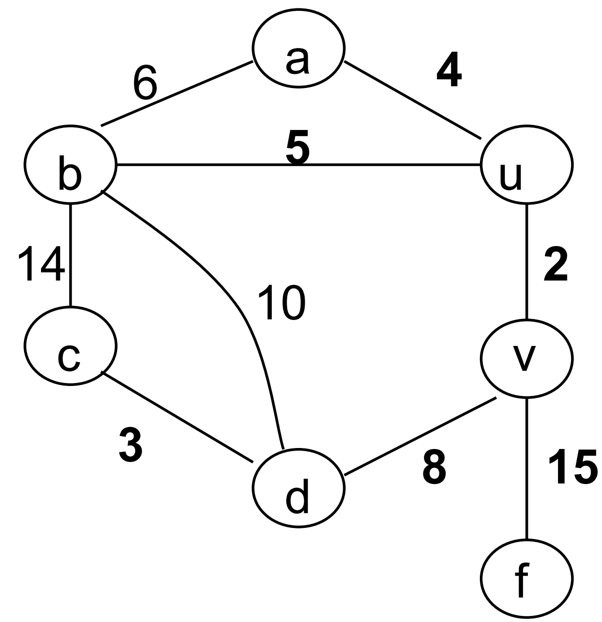
Kruskal's Algorithm

Kruskal's Algorithm: Main Idea

```
solution = { }
while (more edges in E) do
{
    // Selection
    select minimum weight edge
    remove edge from E

    // Feasibility
    if (edge creates a cycle with solution so far)
      then reject edge
      else add edge to solution

    //  Solution check
    if |solution| = |V | - 1 return solution
}
return null // when does this happen?
```



Kruskal's Algorithm:

1. Sort the edges E in non-decreasing weight

2. $T \leftarrow \varnothing$

3. For each $v \in V$ create a set.

4. repeat

5. Select next shortest edge $\{u,v\} \in E$

6. $ucomp \leftarrow find\ (u)$

7. $vcomp \leftarrow find\ (v)$

8. **if** $ucomp \neq vcomp$ **then**

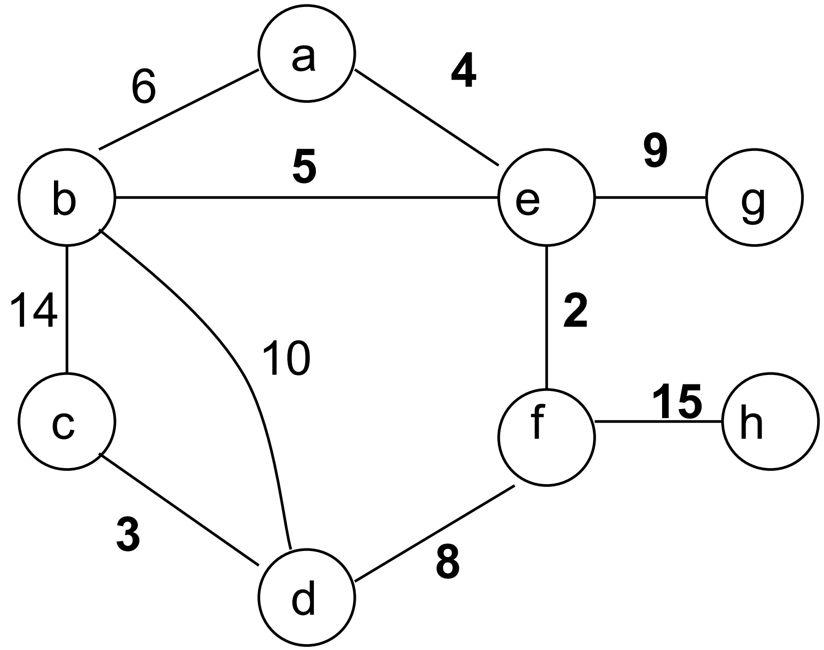
8. add edge (u,v) to T

9. $union\ (\ ucomp, vcomp\ )$

10. **until** T contains $|V|$ - 1 edges

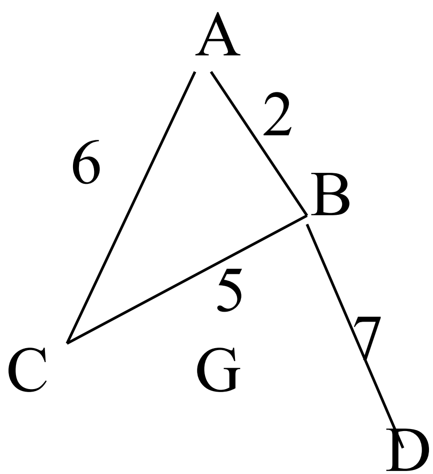
or no more edge

11. **return** tree T



C = { {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h} }
C is a forest of trees.

Kruskal - Disjoint set
After Initialization



Sorted edges T

A B 2

B C 5

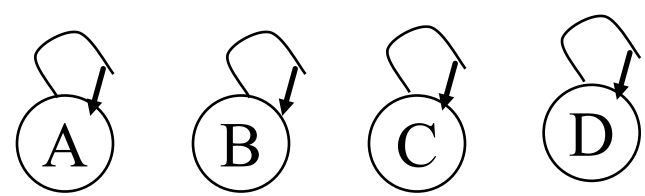
A C 6

B D 7

$1.$ Sort the edges E in non-decreasing weight

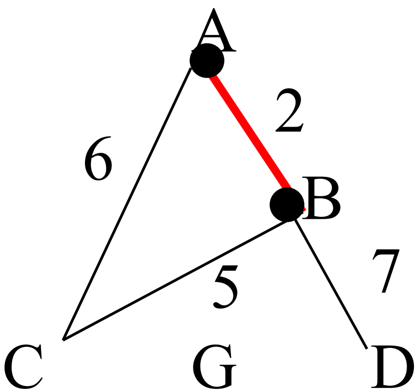
2. $T \leftarrow \varnothing$

3. For each $v \in V$ create a set.

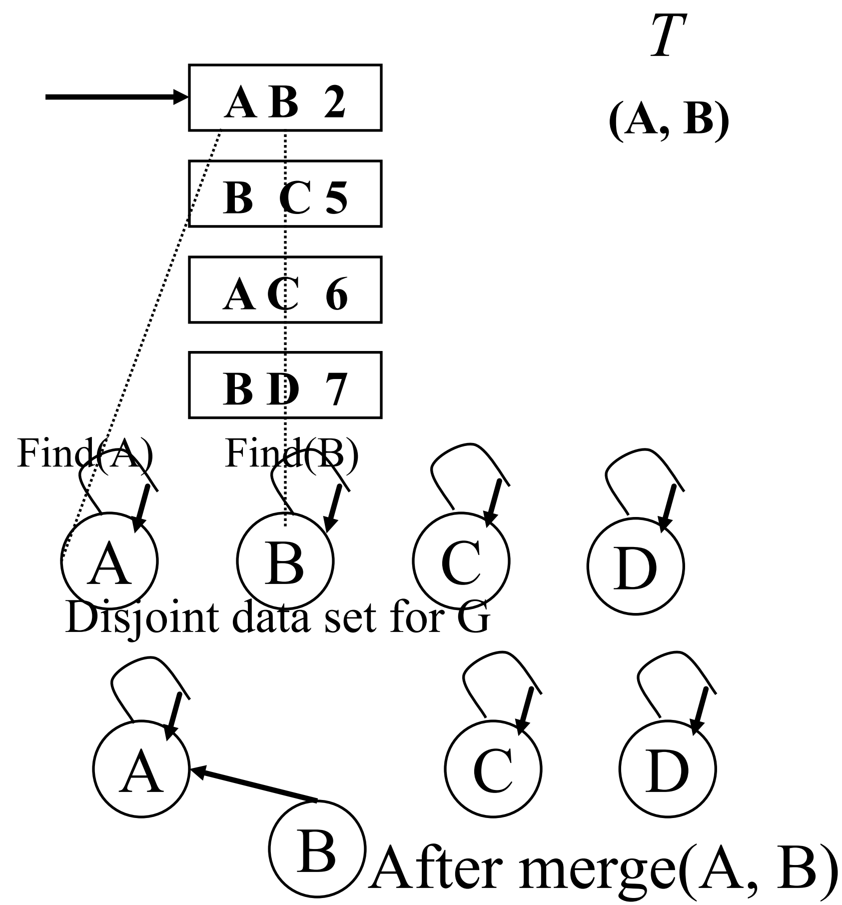


Disjoint data set for G

Kruskal - add minimum weight edge if feasible



Sorted edges



for each $\{u,v\} \in$ in ordered E

    $ucomp \leftarrow find\ (u)$

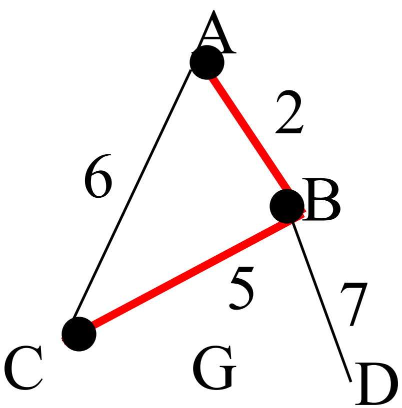
    $vcomp \leftarrow find\ (v)$

    if $ucomp \neq vcomp$ **then**

        add edge (v,u) to T

        $union(\ ucomp, vcomp\ )$

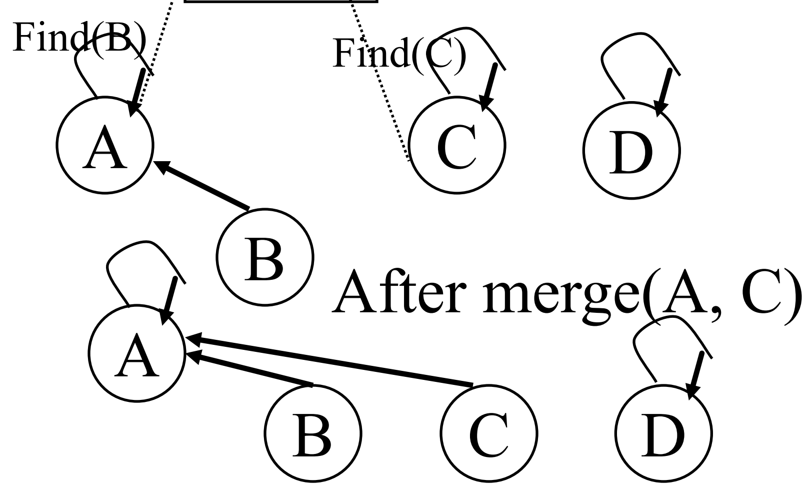
Kruskal – add minimum weight edge if feasible



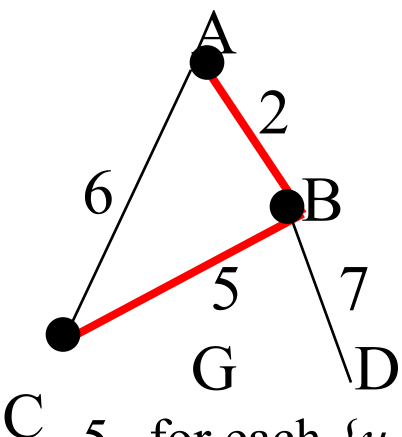

```
for each {u,v} ∈ in ordered E
    ucomp ← find (u)
    vcomp ← find (v)
    if ucomp ≠ vcomp then
        add edge (v,u) to T
        union ( ucomp,vcomp )
```

Sorted edges

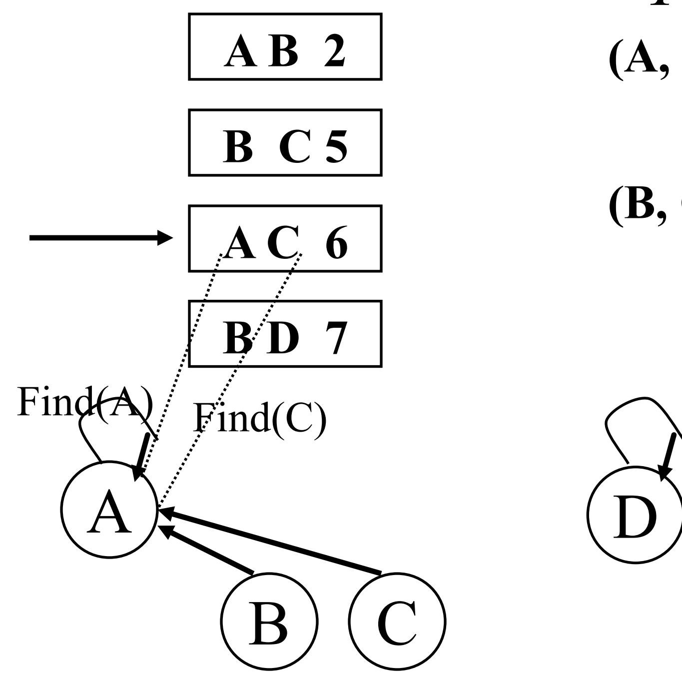
Sorted edges	*T*
A B 2	**(A, B)**
B C 5	**(B, C)**
A C 6	
B D 7	



Kruskal - add minimum weight edge if feasible



Sorted edges



T

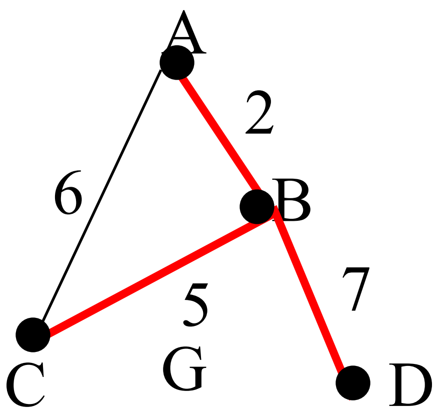
(A, B)

(B, C)

```
for each {u,v} ∈ in ordered E
    ucomp ← find (u)
    vcomp ← find (v)
    if ucomp ≠ vcomp then
        add edge (v,u) to T
        union ( ucomp,vcomp )
```

A and C in same set → Reject edge (A,C)

Kruskal - add minimum weight edge if feasible



Sorted edges

Sorted edges	T
A B 2	**(A, B)**
B C 5	**(B, C)**
A C 6	
B D 7	**(B, D)**

for each $\{u,v\} \in$ in ordered E

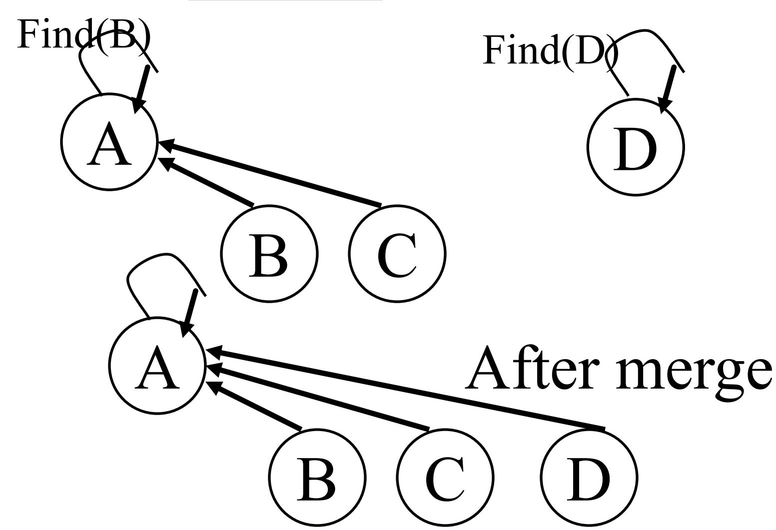
$\quad ucomp \leftarrow find\ (u)$

$\quad vcomp \leftarrow find\ (v)$

$\quad$ **if** $ucomp \neq vcomp$ **then**

$\qquad$ add edge (v,u) to T

$\qquad union\ (\ ucomp, vcomp\ )$



Kruskal's Algorithm: Time Complexity Analysis

Kruskal (G)

1. Sort the edges E in non-decreasing weight
2. T $\leftarrow \varnothing$
3. For each v $\in$ V create a set.
4. repeat
5. {u,v} $\in$ in sorted E
6. ucomp $\leftarrow$ find (u)
7. vcomp $\leftarrow$ find (v)
8. if ucomp $\neq$ vcomp then
9. add edge (v,u) to T
10. union (ucomp,vcomp)
11. until T contains |V| - 1 edges or no more edge
12. return tree T

$Count_1$ = $\Theta(E \lg E)$ for sorting

$Count_2$= $\Theta(1)$

$Count_3$= $\Theta(V)$

In the loop, there are O(E) operations on the disjoint set forest ➔

$O(E\,\alpha(E, V)) = O(E \lg E)$