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 $S_i \cap S_j = \varnothing$ for $i \neq j$,

- The methods are:
- *find* (x) : returns a reference to $S_i \in \boldsymbol{S}$ 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

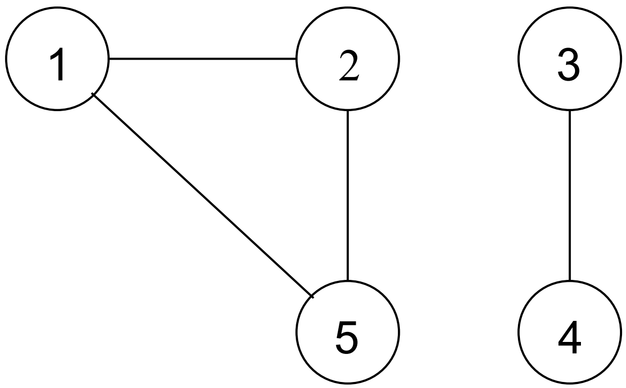
- Problem: Find the *connected components* of a graph.

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

G = 

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

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:

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

Disjoint Set Implementation in an array

- We can use an array, or a linked list to implement the collection. In this lecture 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];      // θ(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;
```

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

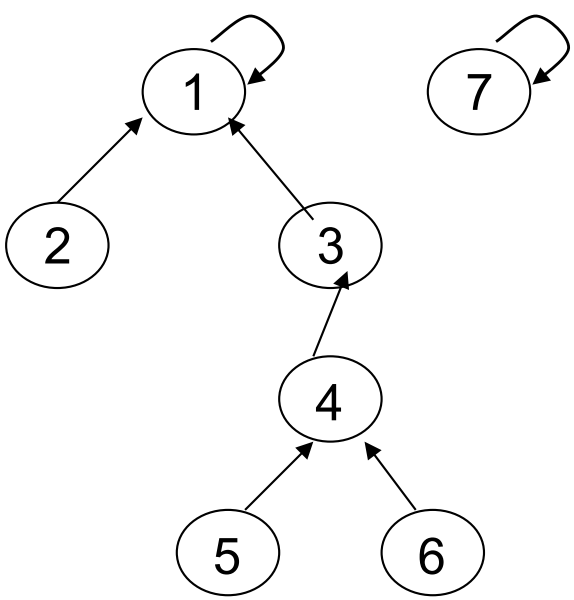
DS is implemented as an array

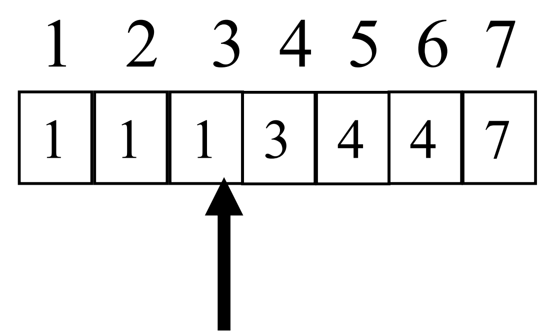
- For the following sequence of *merges* we show the resulting array

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



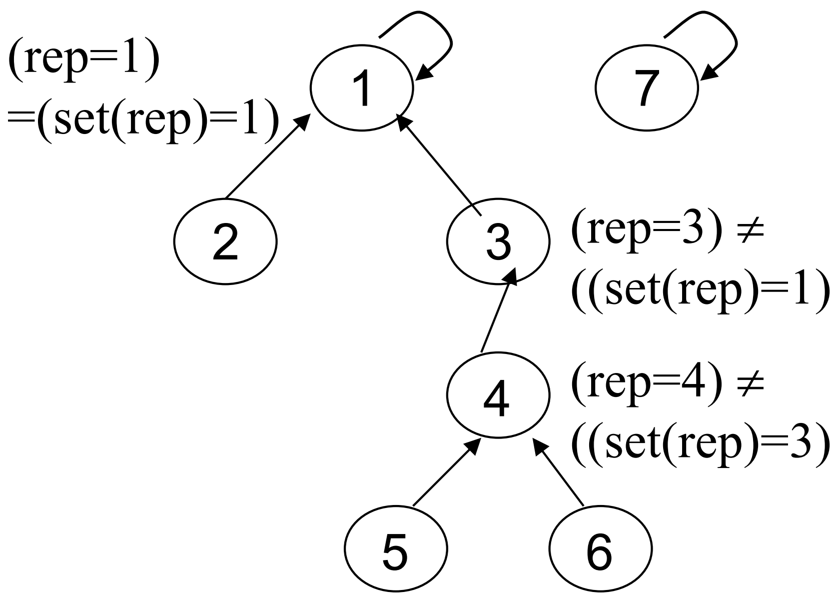


Array representation

Backward forests stored in an array

*find*2(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

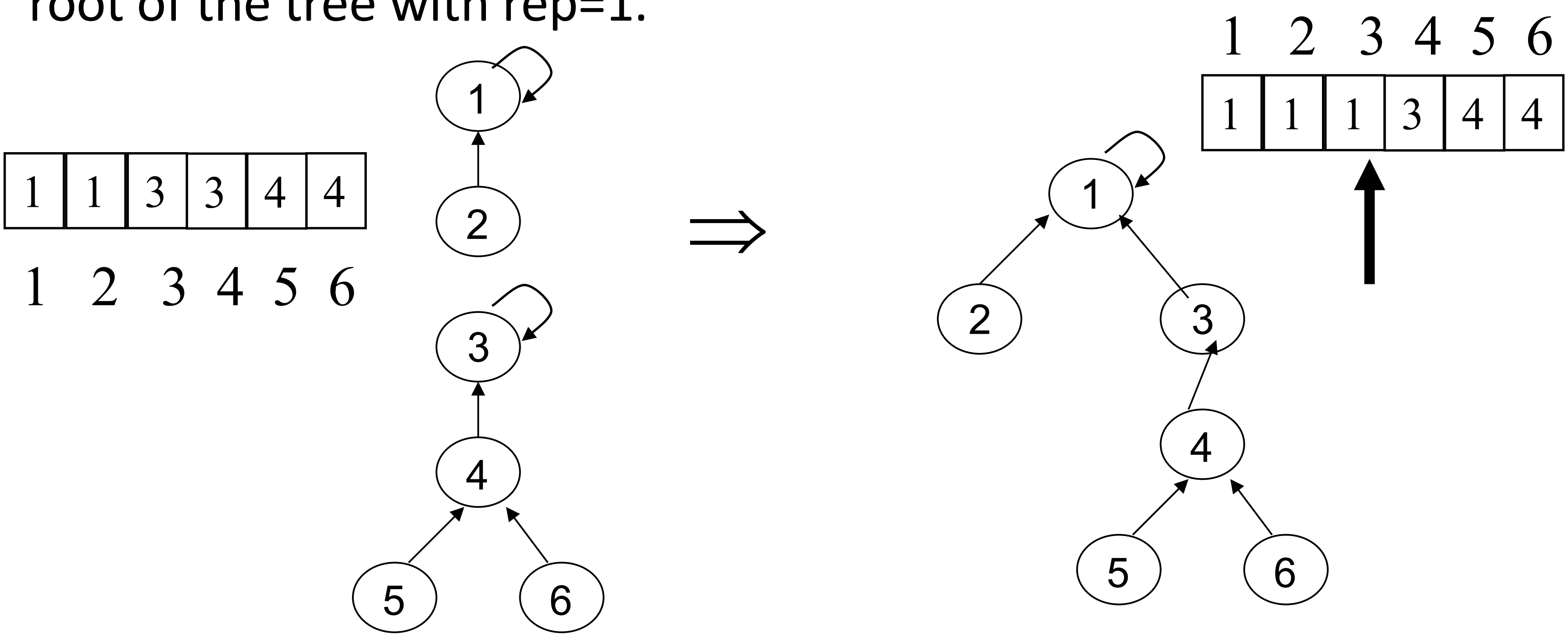
Backward forests stored in an array

union2(rep*x*, *repy*).
  smaller ← min (*repx*, *repy*);
  larger ← max (*repx*, *repy*);
  set [*larger*] ← *smaller;*

- *union2* 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.

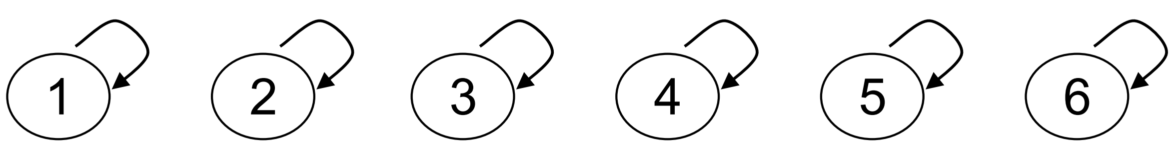


Disjoint-set implemented as backward forests

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:

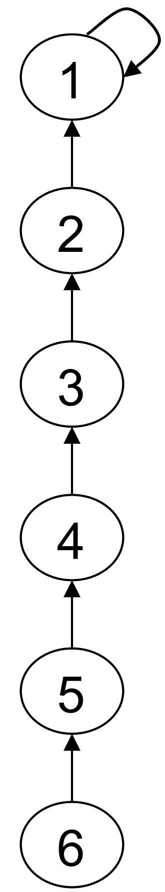
1	2	3	4	5	6
1	2	3	4	5	6



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**
- *Now 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
```

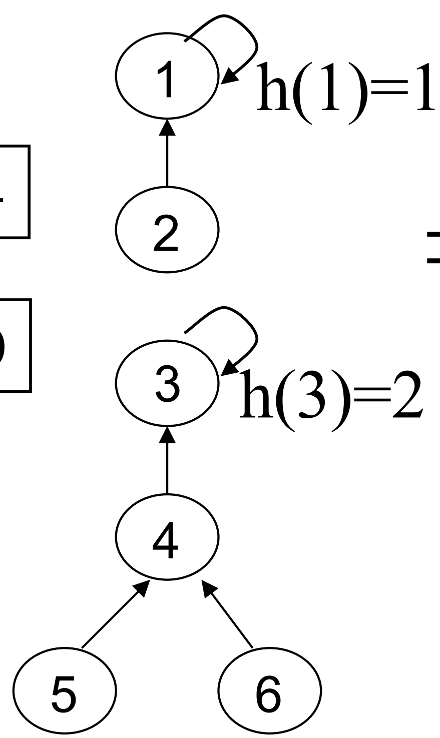
Merge with reduced height

- *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.

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

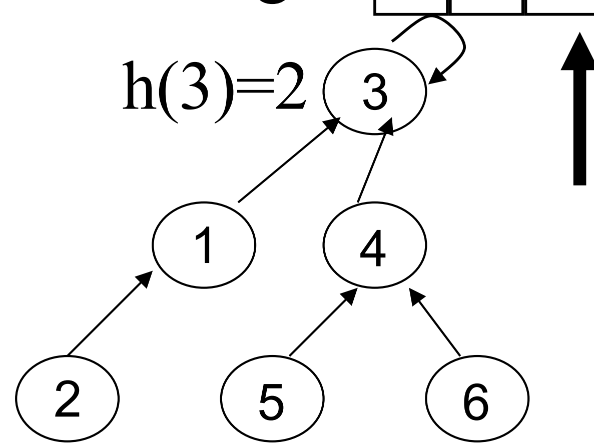
1	0	2	1	0	0

Set and height



⇒

	1	2	3	4	5	6
Set	3	1	3	3	4	4
height	1	0	2	1	0	0



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

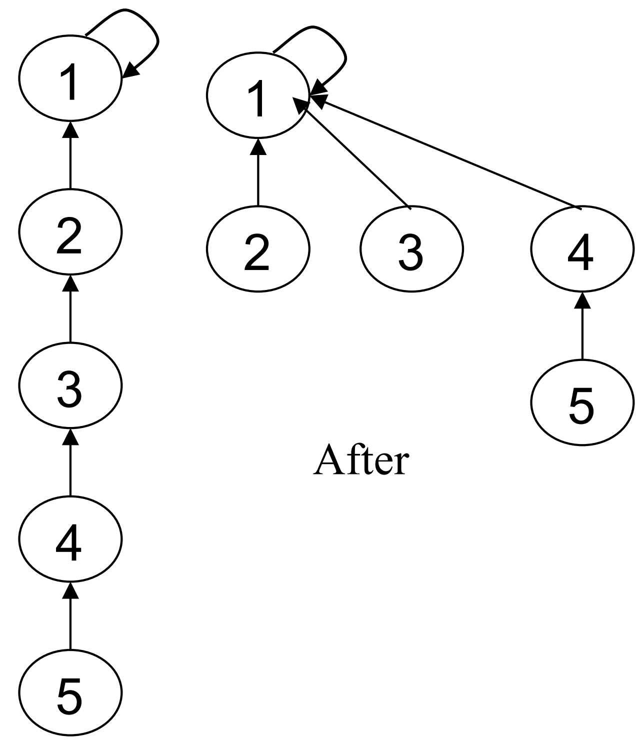
*find*3(x)

//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)



After

Summary

- 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 $\lg^* n$
    - Proof is beyond the scope of this class: If interested, refer to Cormen's book (the recommended text)

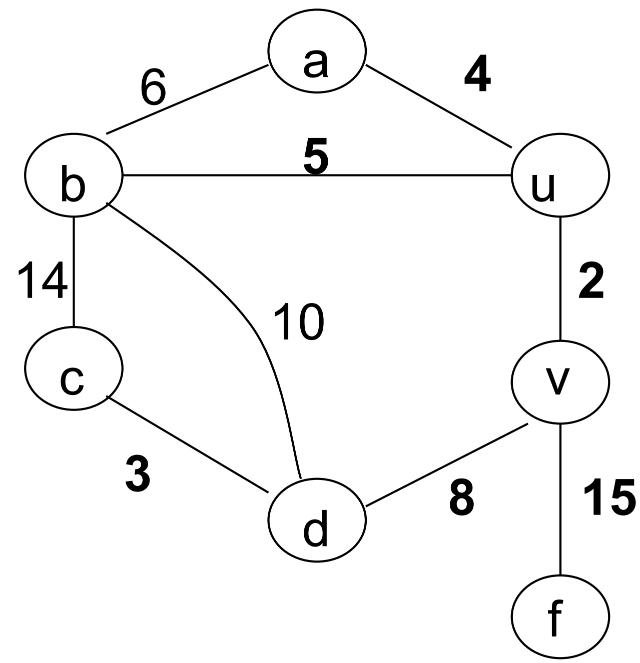
Kruskal's Algorithm

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. $T \leftarrow \emptyset$

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**

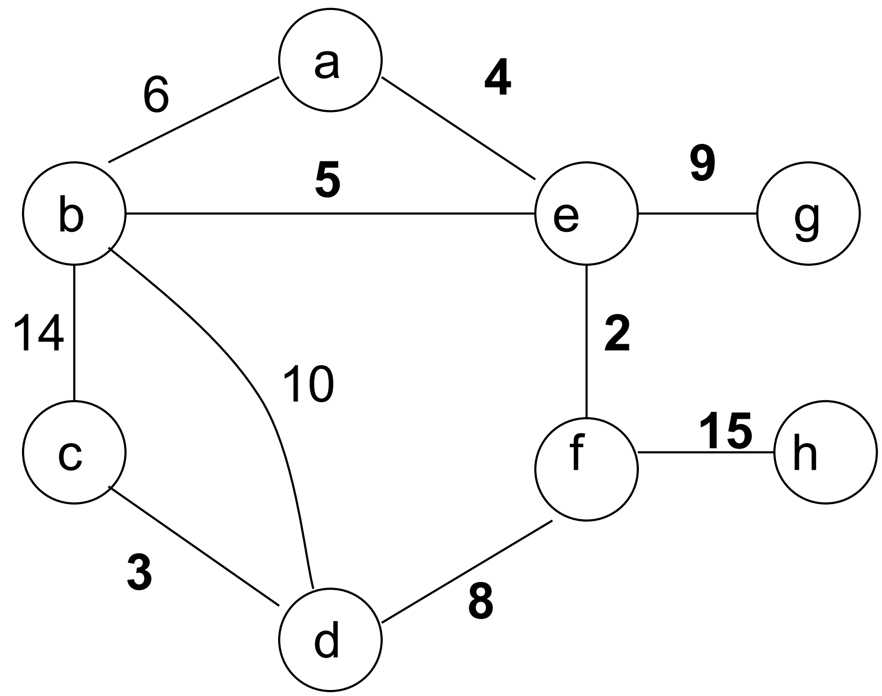
8. add edge (u,v) to T

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

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

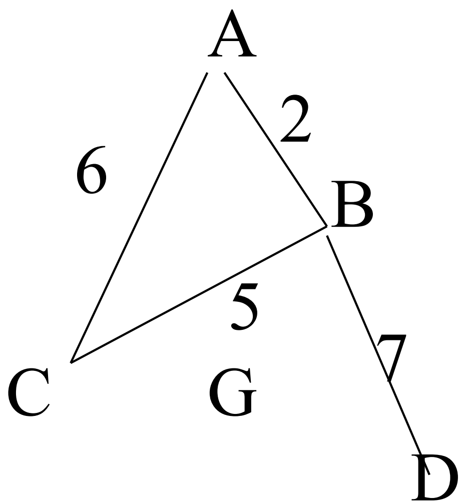
or no more edge

11. **return** tree T



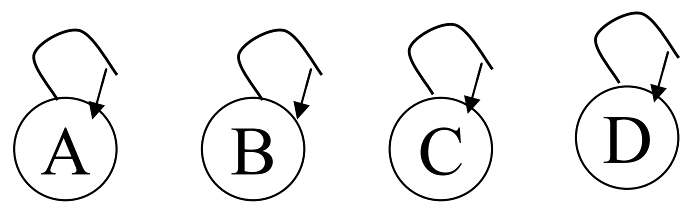
C = { {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h} }
C is a forest of trees.

Kruskal - Disjoint set
After Initialization



Sorted edges $\quad\quad$ T

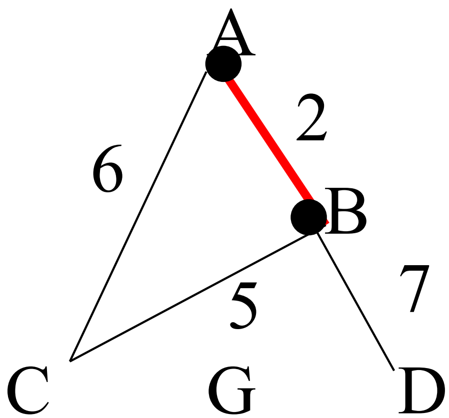
A B 2
B C 5
A C 6
B D 7



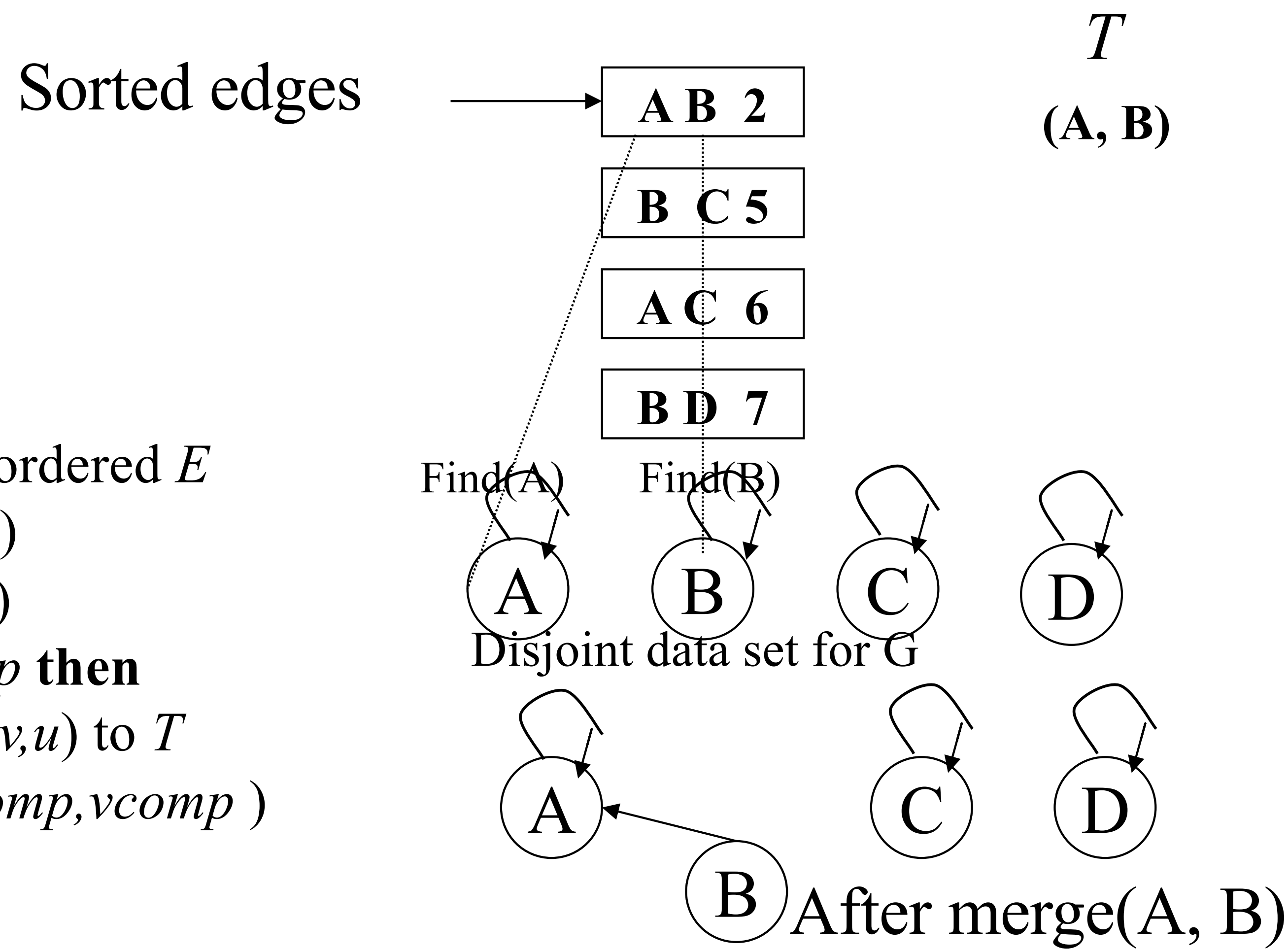
Disjoint data set for G

1. Sort the edges E in non-decreasing weight
2. $T \leftarrow \emptyset$
3. For each $v \in V$ create a set.

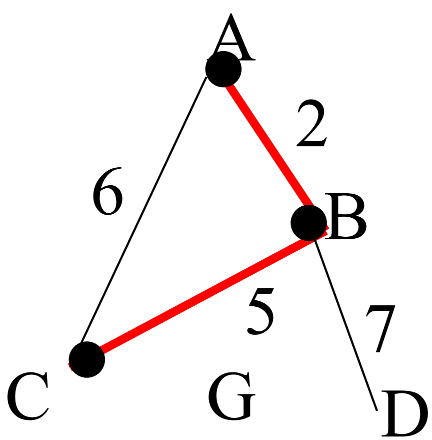
Kruskal - add minimum weight edge if feasible



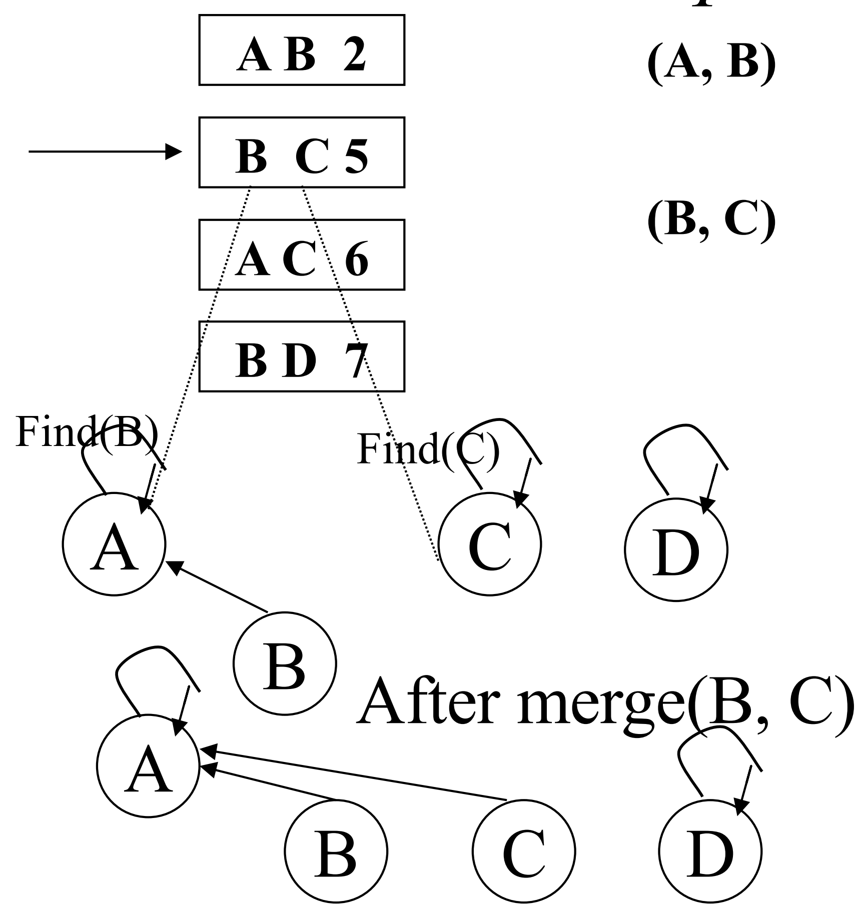
$$
\begin{array}{l}
\textrm{for each } \{u,v\} \in \textrm{ in ordered } E \\
\quad \textit{ucomp} \leftarrow \textit{find}\ (u) \\
\quad \textit{vcomp} \leftarrow \textit{find}\ (v) \\
\quad \textbf{if } \textit{ucomp} \neq \textit{vcomp} \textbf{ then} \\
\quad\quad \textrm{add edge } (v,u) \textrm{ to } T \\
\quad\quad \textit{union}(\ \textit{ucomp},\textit{vcomp}\ )
\end{array}
$$



Kruskal – add minimum weight edge if feasible



Sorted edges



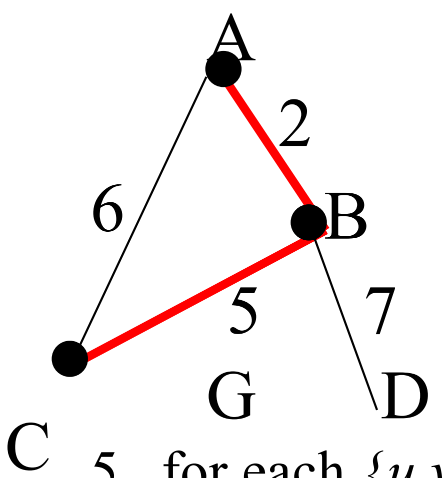
T

(A, B)

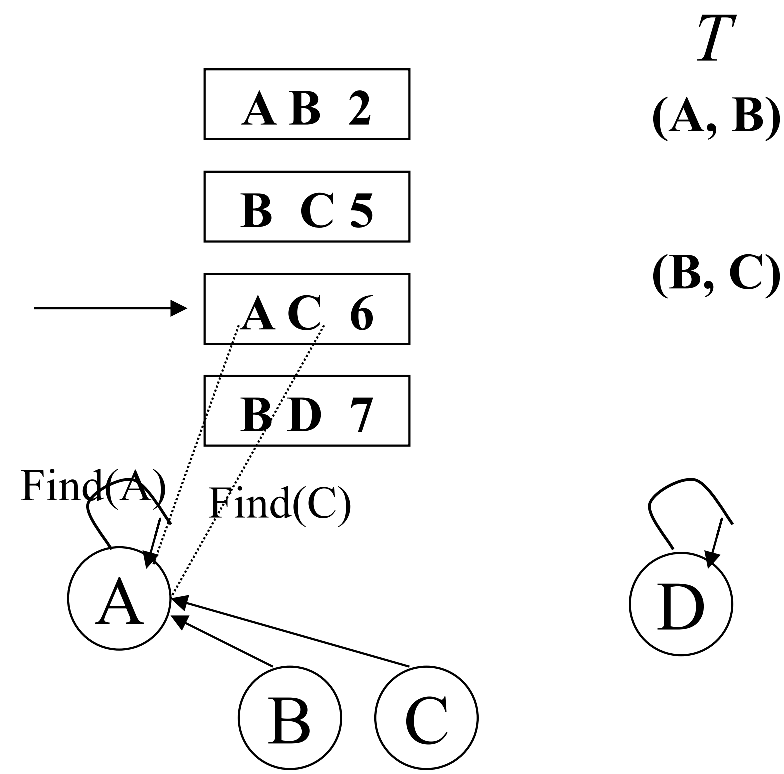
(B, C)

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



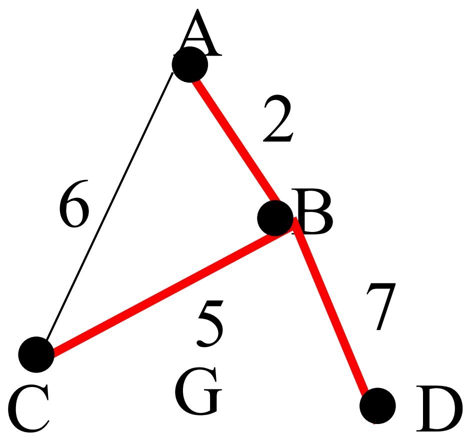
Sorted edges




```
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 )
```

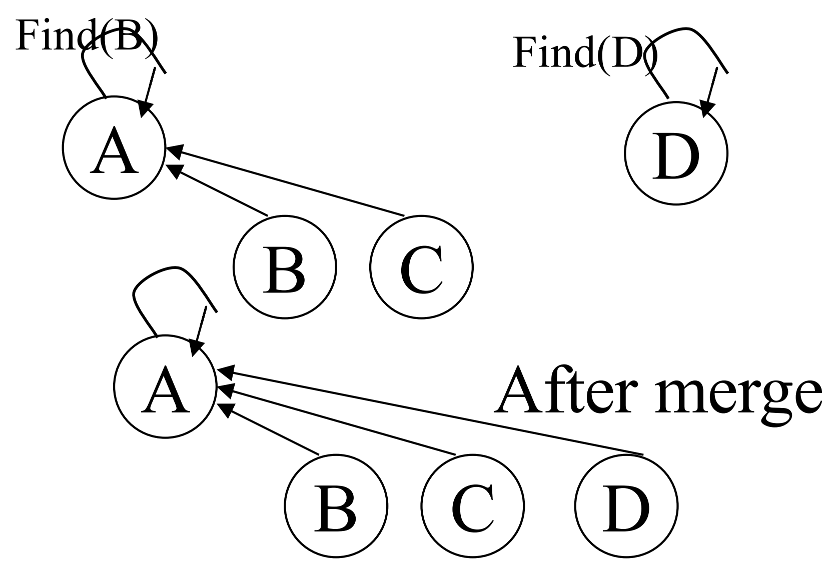
A and C in same set → Reject edge (A,C)

Kruskal – add minimum weight edge if feasible



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

```
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 )
```



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$_1$ = $\Theta(\ E \lg E\ )$

Count$_2$= $\Theta(1)$

Count$_3$= $\Theta(\ V\ )$

Count$_4$ = $O(\ E\ )$

Sorting: $\Theta(E \lg E) = \Theta(\ E \lg V)$

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

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

Lemma 1

- Let $G = (V , E)$ be a connected, weighted, undirected graph; let F be a promising subset of E.
- **let e be an edge of minimum weight in $E - F$ such that $F \cup \{e\}$ has no simple cycles. Then $F \cup \{e\}$ is promising.**

Proof of Lemma 1

- The proof is similar to the proof of Lemma 1 for Prim's algorithm.
- Because F is promising, there must be some set of edges F' such that (V, F') is a minimum spanning tree.
- If $e \notin F'$, because (V, F') is a spanning tree, $F' \cup \{e\}$ must contain exactly one simple cycle and e must be in the cycle.
- Because $F \cup \{e\}$ contains no simple cycles, there must be some edge $e' \in F'$ that is in the cycle and that is not in F. That is, $e' \in E - F$.
- The set $F \cup \{e'\}$ has no simple cycles because it is a subset of F'.
- If we remove e' from $F' \cup \{e\}$, the simple cycle in this set disappears, which means we have a spanning tree. Indeed is a minimum spanning tree because the weight of e is no greater than the weight of e'.
- Because e' is not in F, $F \cup \{e\}$ is promising, which completes the proof.

Theorem: Kruskal's Algorithm always produces a minimum spanning tree.

- The proof is similar to the proof of Prim's algorithm.
- Proof by induction on the set T of promising edges.

  1. Base case: Initially, $T = \varnothing$ is promising.
- The rest of the proof (for your exercise).