Kruskal Algorithm for creating Minimum Spanning Trees

Lecture 05.04

by Marina Barsky

Kruskal MST algorithm

Algorithm Kruskal_MST (graph G(V,E))

```
E' := edges of G sorted by weights
T : = Ø                              # collects edges of the future MST

for i from 1 to m:
    if T U {E'[i]} has no cycles
        add E'[i] to T

return T
```

Repeatedly add a minimum-cost edge
that does not create a cycle

Kruskal MST algorithm

Algorithm Kruskal_MST (graph G(V,E))

```
E' := edges of G sorted by weights
T : = Ø                             # collects edges of the future MST

for i from 1 to m:
    if T U {E'[i]} has no cycles
        add E'[i] to T

    if |T| = |V| - 1:               # we can stop once we have a tree
        break

return T
```

Stop when
N-1 edges have been selected

Running time

Kruskal_MST (graph G(V,E))

```
1   E' := edges of G sorted by weights
2   T : = ∅

3   for i from 1 to m:
4           if T U {E'[i]} has no cycles
5                   add E'[i] to T

6           if |T| = |V| - 1:
7                   break

8   return T
```

Line 1: sorting m edges by weight. O(m log m). This is the same as O(m log n) Why?

Line 3: outer for loop. O(m). We check all m edges in the worst case.
Line 4: need to find if edge E'[i]= (u,v) creates a cycle.
Find out if there is already a path from u to v in T by any graph traversal (DFS or BFS). DFS of T with n vertices and n-1 edges is O(n + n) = O(n).

Thus, total time of the for loop is O(m)*O(n) = O(mn) [O(n^3) for dense graphs]
Kruskal MST runs in time O(m log n) + O(mn) = **O (mn)**

Running time

```
Kruskal_MST (graph G(V,E))

E' := edges of G sorted by weights
T : = Ø

for i from 1 to m:
        if T U {E'[i]} has no cycles
                add E'[i] to T

        if |T| = |V| - 1:
                break

return T
```

Bottleneck: detecting a cycle

Kruskal MST runs in time **O (mn)**

Can we do better?

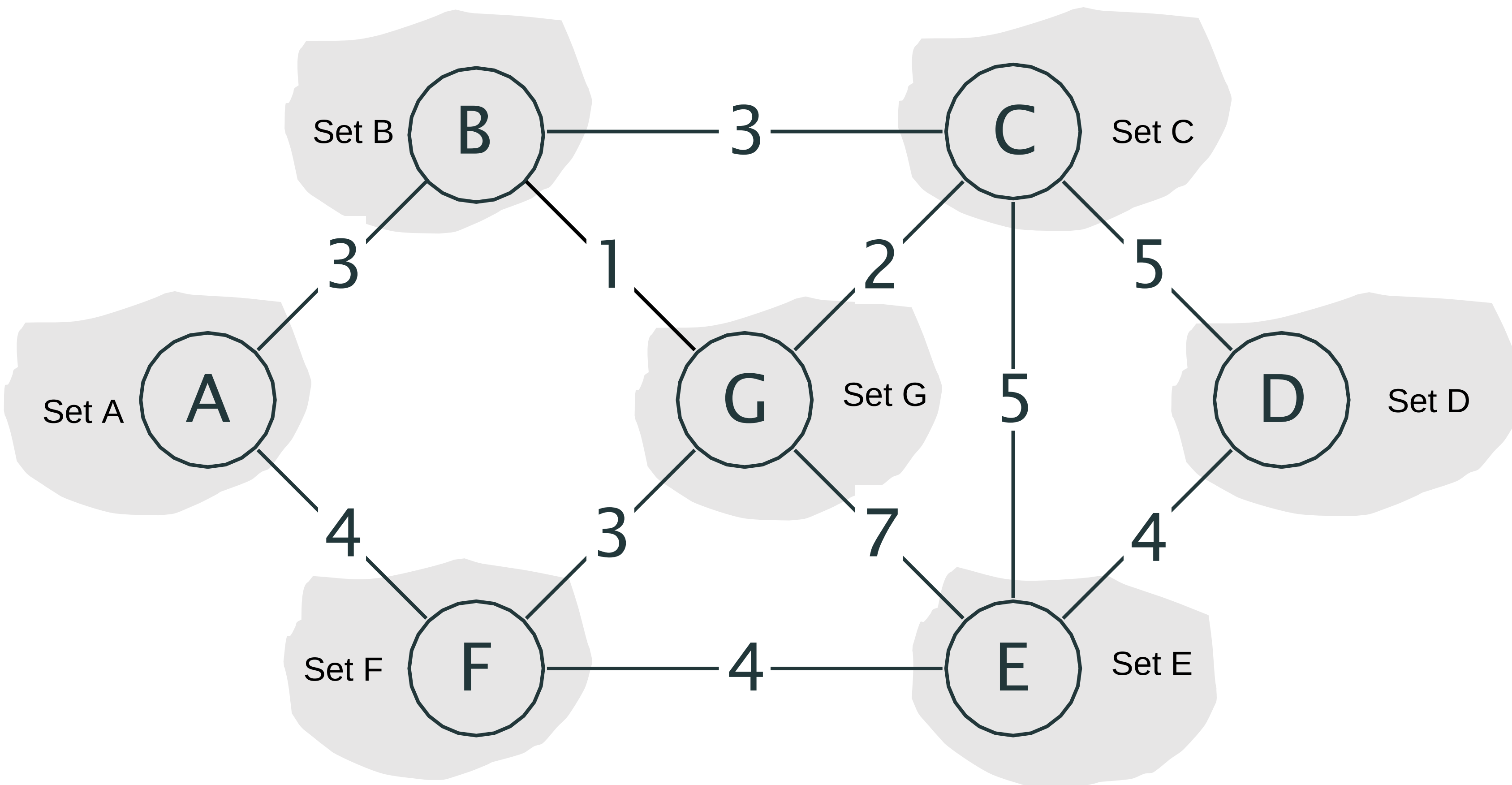
Kruskal as union of sets

We can look at Kruskal from a Set point of view

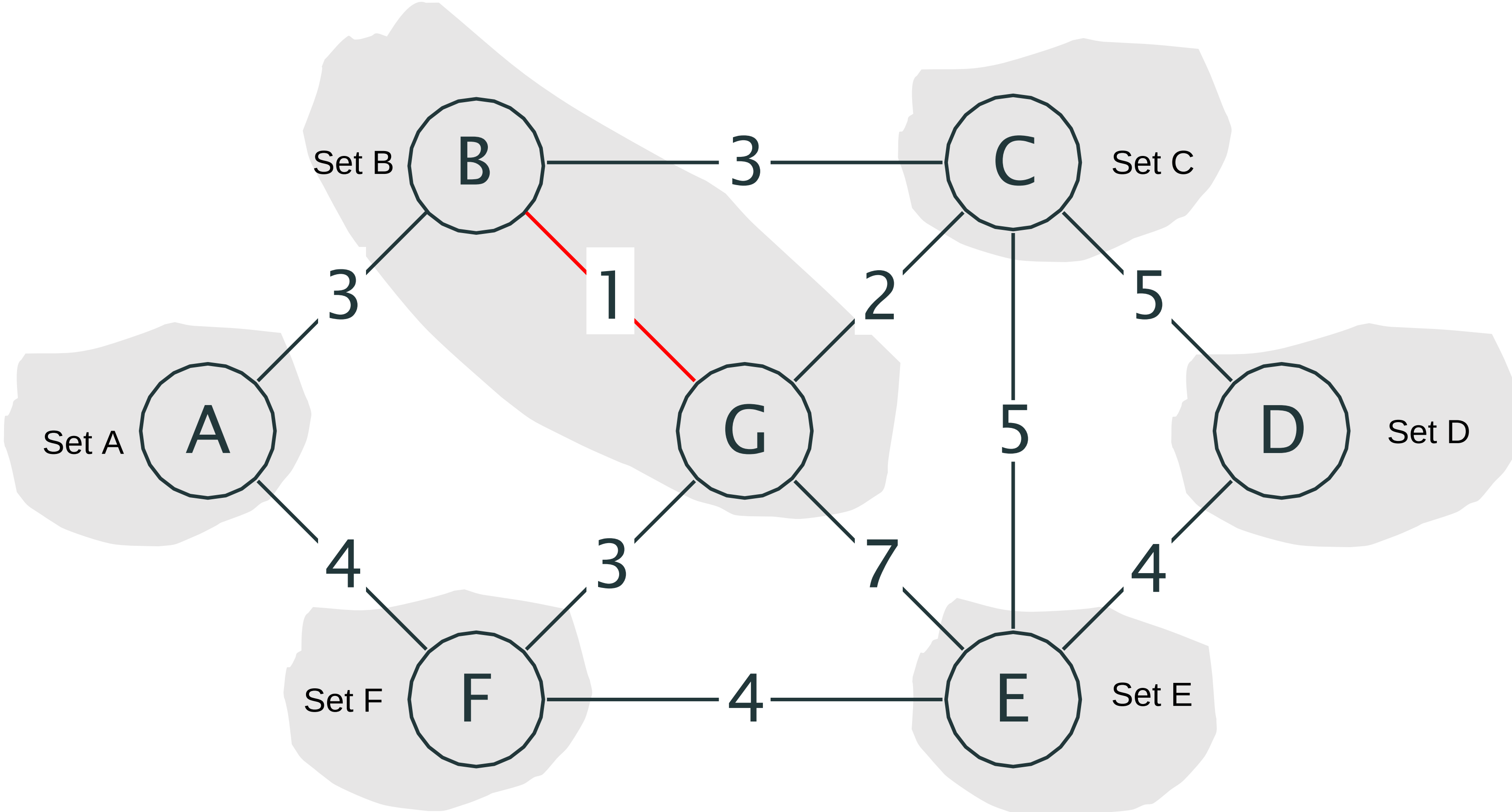
- First we have n sets: each vertex i is in its own set S_i – we need operation MAKE-SET for a single element
- Next we combine two sets of vertices S_i and S_j into one set: UNION (S_i and S_j) adding an edge (u,v) such that $u \in S_i$, and $v \in S_j$
- We do this only if $S_i \neq S_j$. We need to know if u and v are already in the same set, in the same connected component, we need to know set names for u and for v and compare them: FIND(x)

Note that all the sets are disjoint: each node belongs to a single set during the execution of the algorithm

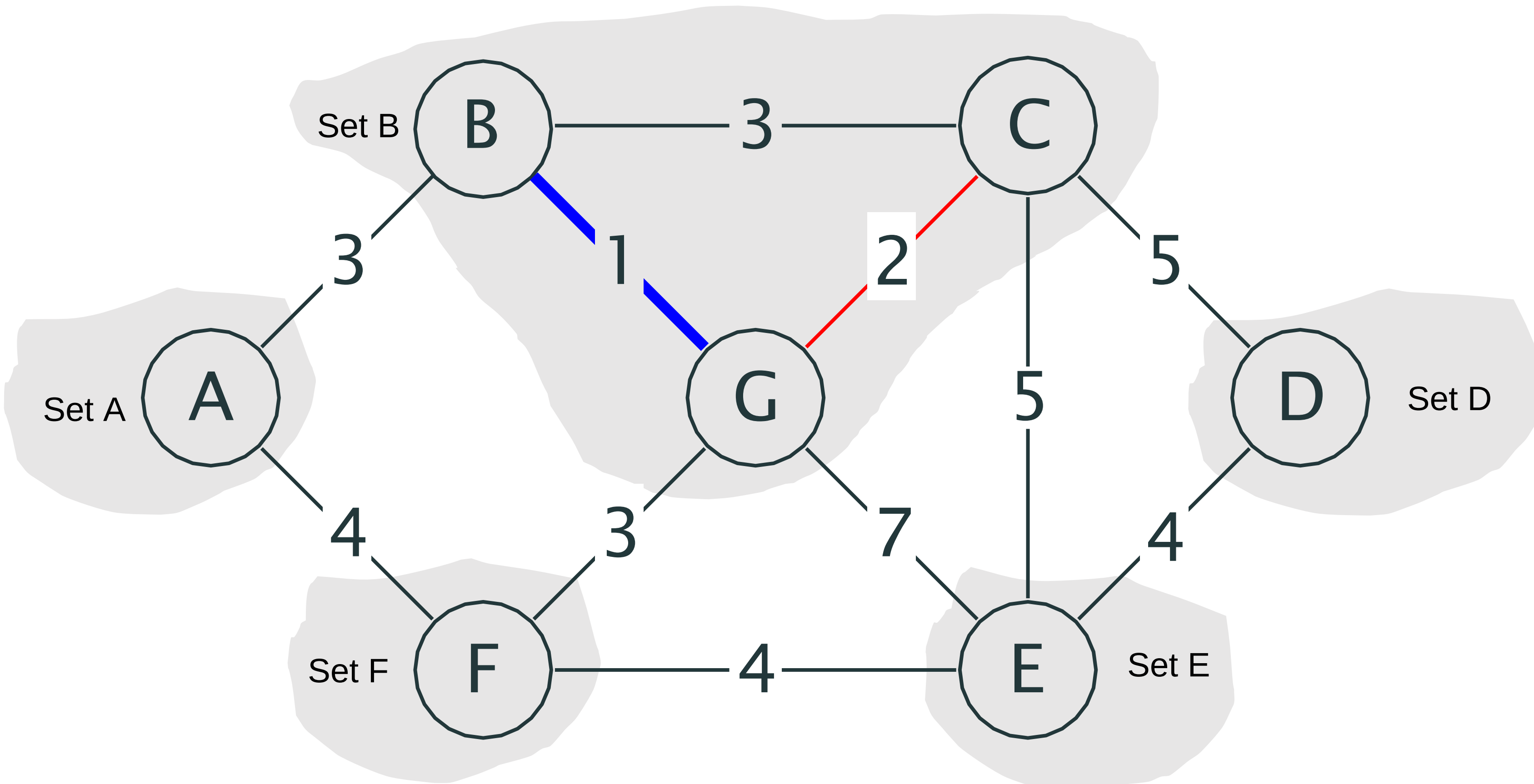
Kruskal as union of sets



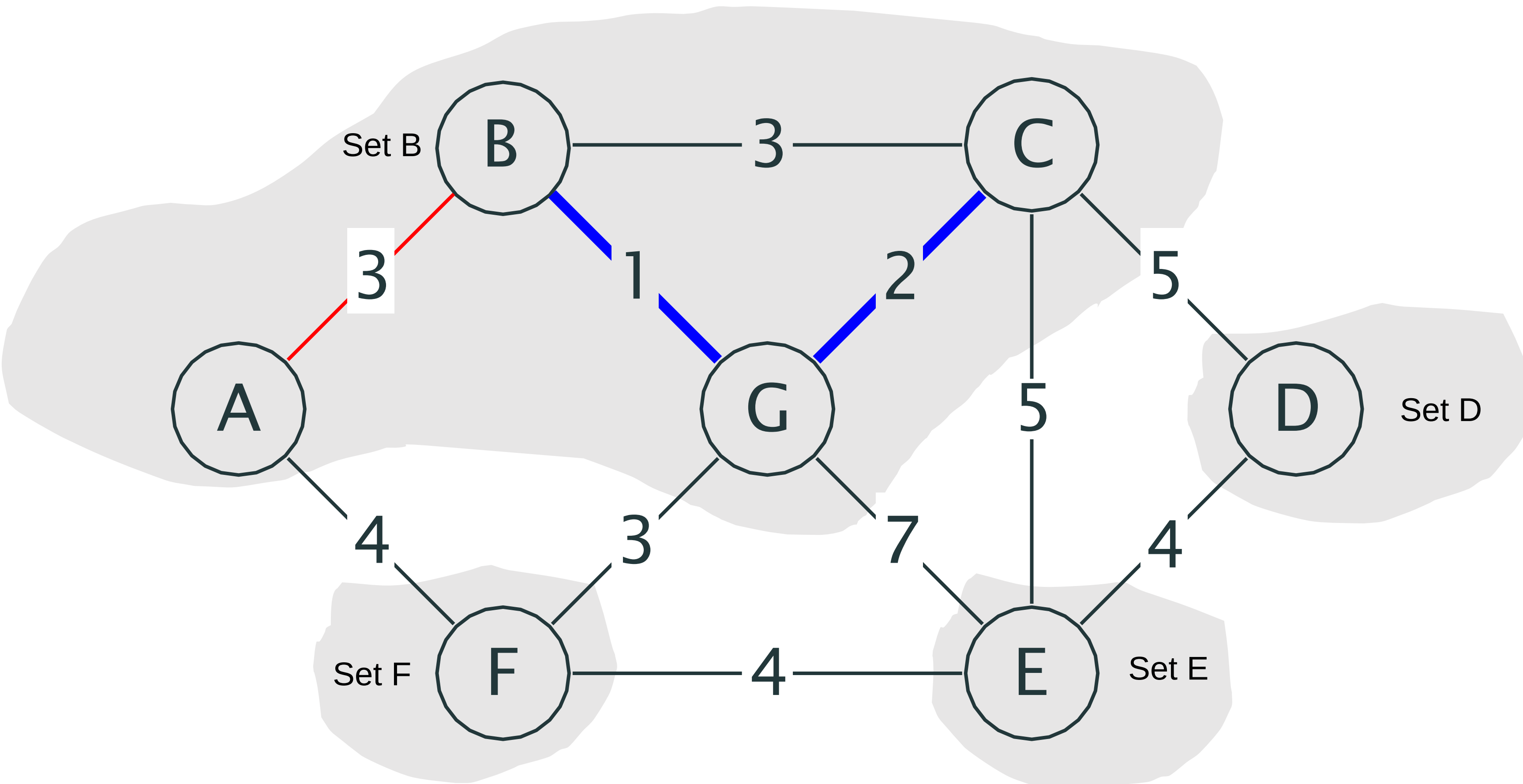
Kruskal as union of sets



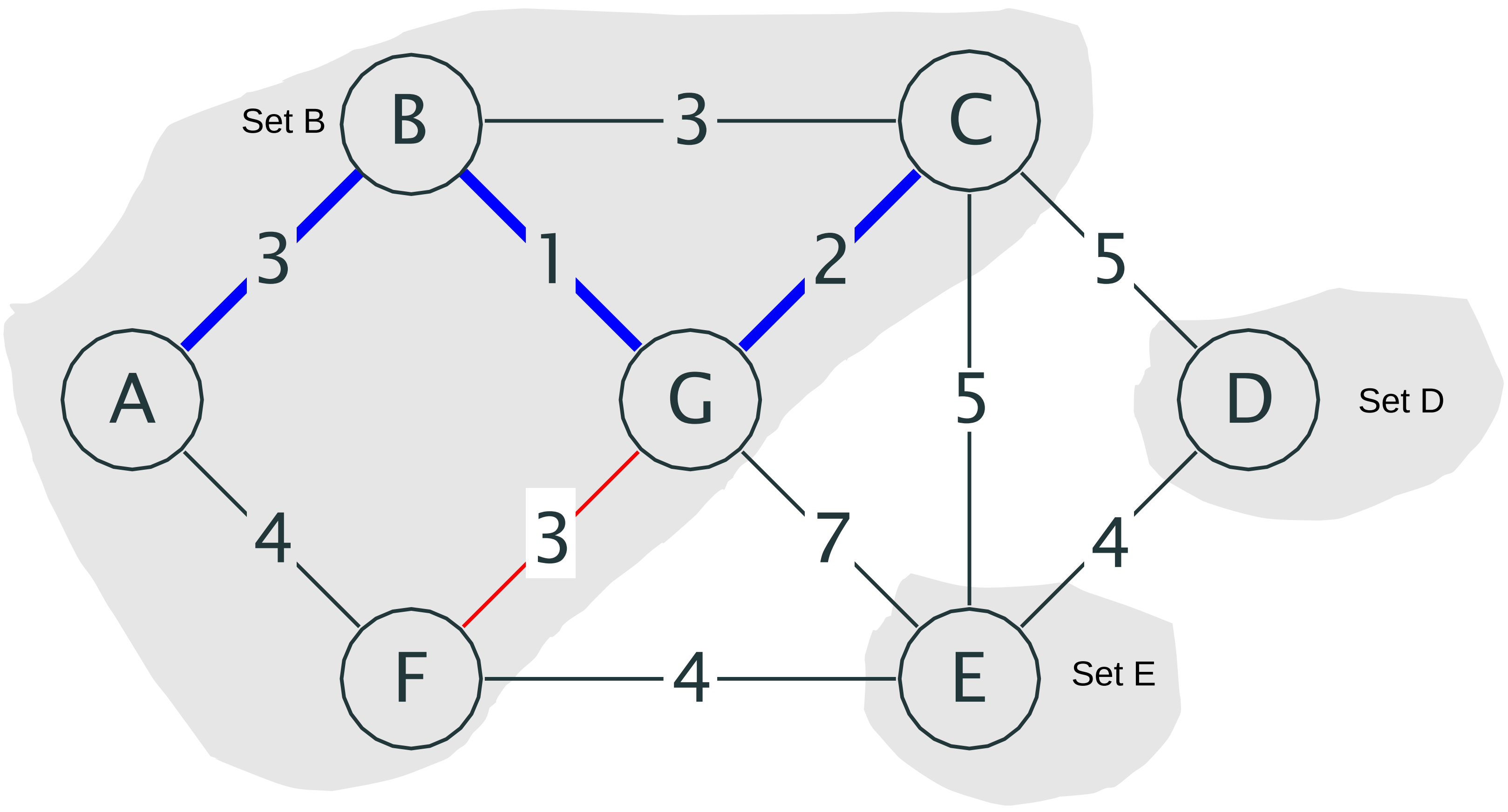
Kruskal as union of sets



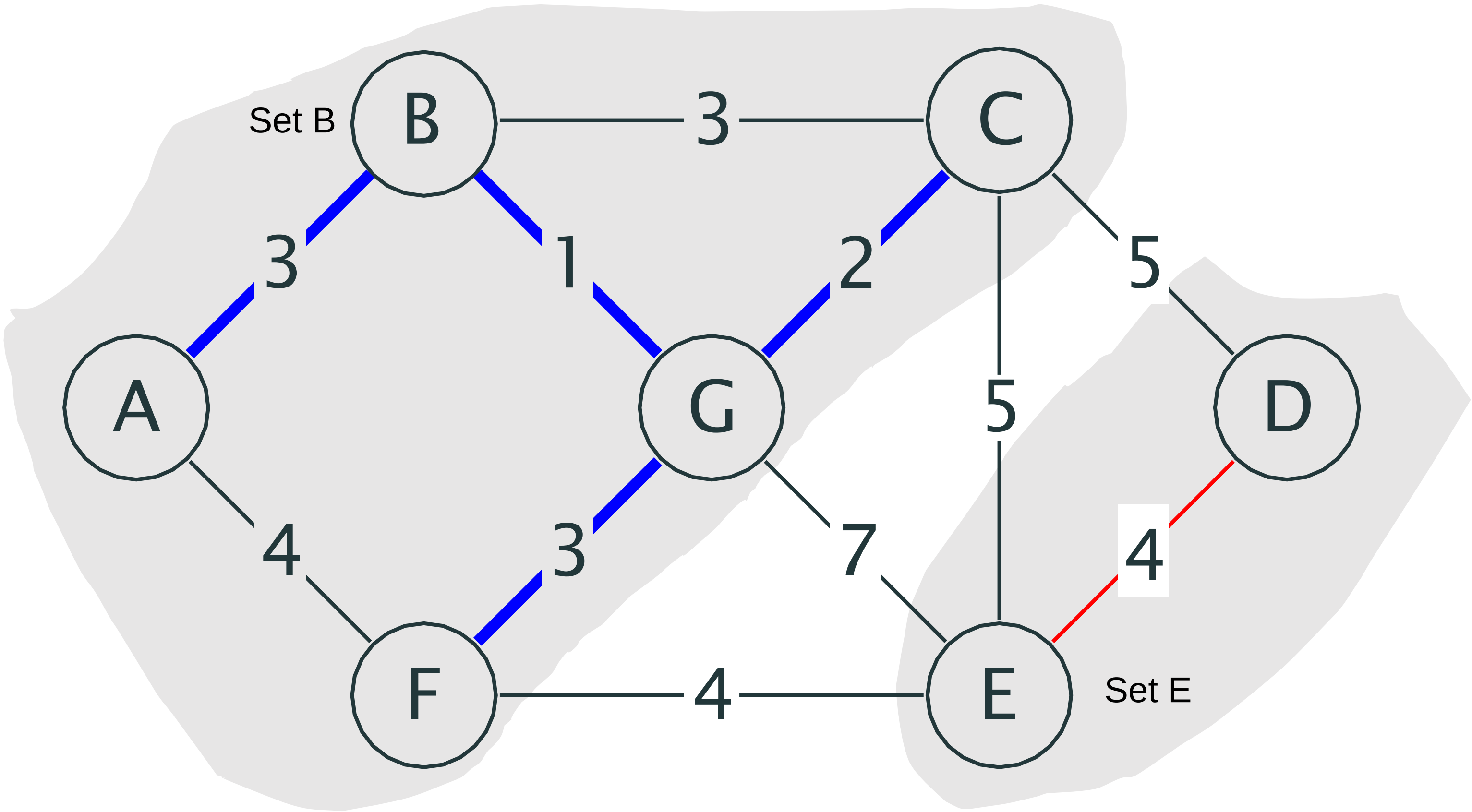
Kruskal as union of sets



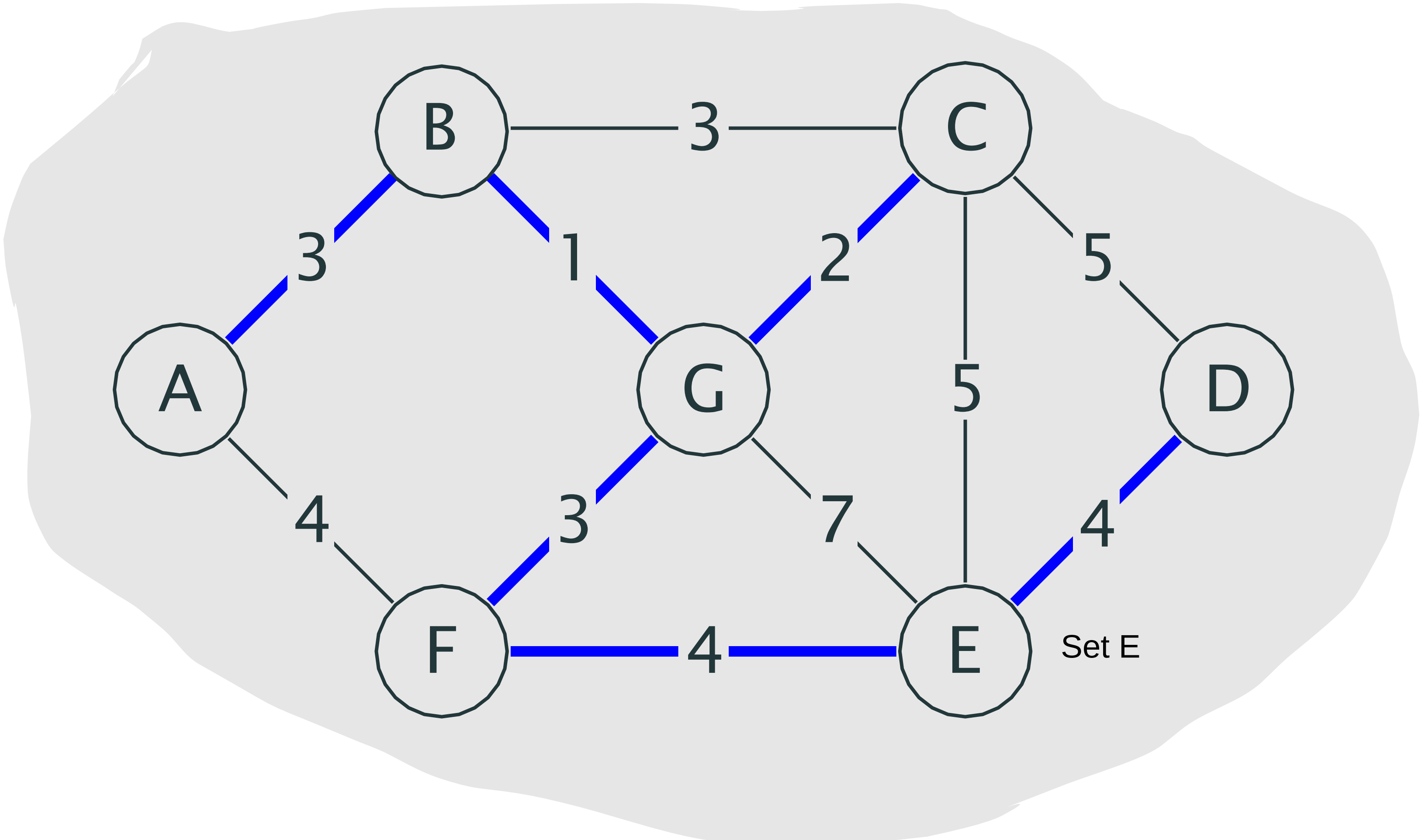
Kruskal as union of sets



Kruskal as union of sets



Kruskal as union of sets



Set spanning all vertices of G with selected edges:
MST of G

New ADT: UNION-FIND (= Disjoint Set ADT)

UNION-FIND is an Abstract Data Type that supports the following operations:

- *MAKESET(x)*: Creates a new set X containing a single element x.
- *UNION(X, Y)*: Creates a new set containing the elements of sets X and Y in their union and deletes the previous sets X and Y.
- *FIND(x)*: Returns the name of the set to which element x belongs.

UNION-FIND fits all our needs

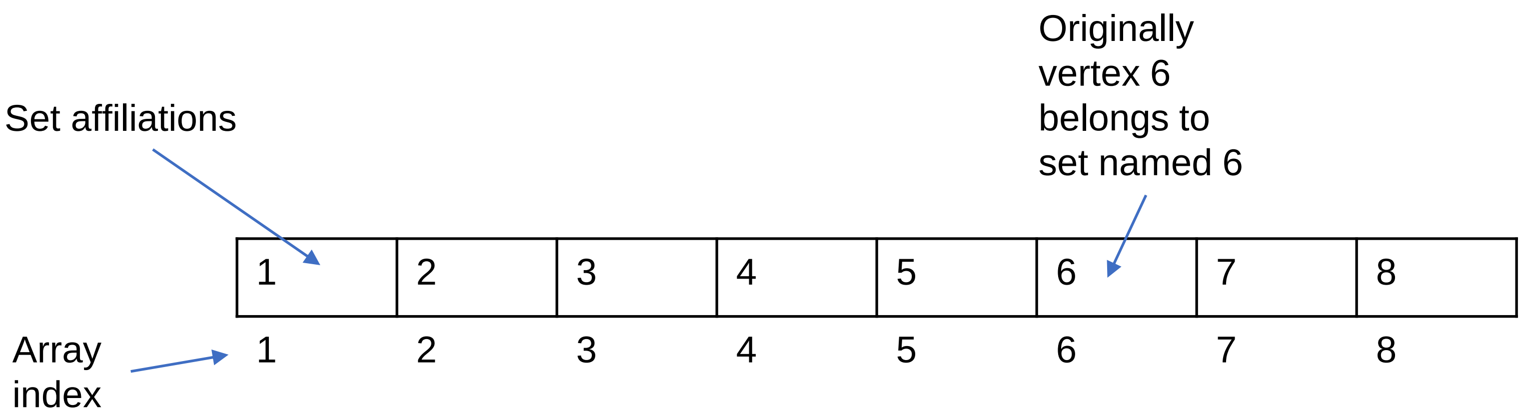
- Initially, the vertices are a collection of n sets, each with one element. We can use *MAKE-SET* n times. Each set has a different element, so that $S_i \cap S_j = \emptyset$. This makes the sets disjoint.
- To introduce a new relationship between S_i and S_j using edge (x,y), we first check whether x and y are already connected: perform *FIND*(x) and *FIND*(y) and check if they already belong to the same set.
- If they are not, then we apply *UNION*. This operation merges the two sets containing x and y into a new set $S_k = S_i \cup S_j$.

Implementing UNION-FIND: Array

- We can implement UNION-FIND using a physical array.
- We can number every vertex from 1 to n, and assume that the name of the set to which vertex *i* belongs is stored at position *i* of this array.

Array implementation: *MAKE-SET*

- For n elements, we can generate single-element sets in time O(n)
- The name of each set initially is set to the name of the element itself: which corresponds to its position i in the array



Index in this array uniquely identifies each of n graph vertices

Array implementation: **fast *FIND***

- With this representation FIND(x) takes O(1) since for any element we can find the set name by accessing its array location in time **O(1)**.

We can find set affiliation of
nodes 3 and 8 in time O(1)

Set affiliation	1	2	6	2	5	6	7	6
Array index	1	2	3	4	5	6	7	8

Array implementation: **slow *UNION***

- In this representation, to perform UNION(u, v) [assuming that $u \in S_i$ and $v \in S_j$] we need to scan the complete array and change all i's to j. This takes **O(n)**.
- A sequence of n – 1 unions required by the algorithm takes $O(n^2)$ time in the worst case.

Current set affiliations

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

Next edge to be added: (3,4)
We check that FIND(3) ≠ FIND(4)
UNION(1,2) will need to iterate over the array and replace all 2 with 1

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

Now vertices 3 and 4 belong to the same set, they are connected

Implementing UNION-FIND: Tree

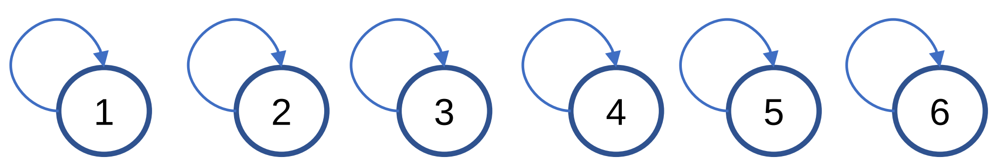
- We can implement each set as a tree, because in the tree each element has only one root, and that is where we will store the name of the set to which all elements in this tree belong.

- The tree idea is rather conceptual. We do not have to create a physical tree: we can use a *parent array* where for each node *i* we store the name of its parent in the tree.

Tree implementation: *MAKE-SET*

- To differentiate the root of the tree, let us assume that if the parent in position i is i, then node i is a root of the tree – and it also serves as a set name for all nodes in its subtrees.
- MAKE-SET creates n sets containing a single element i and in the array sets the parent of i as i. That means root (set name) of i is i.

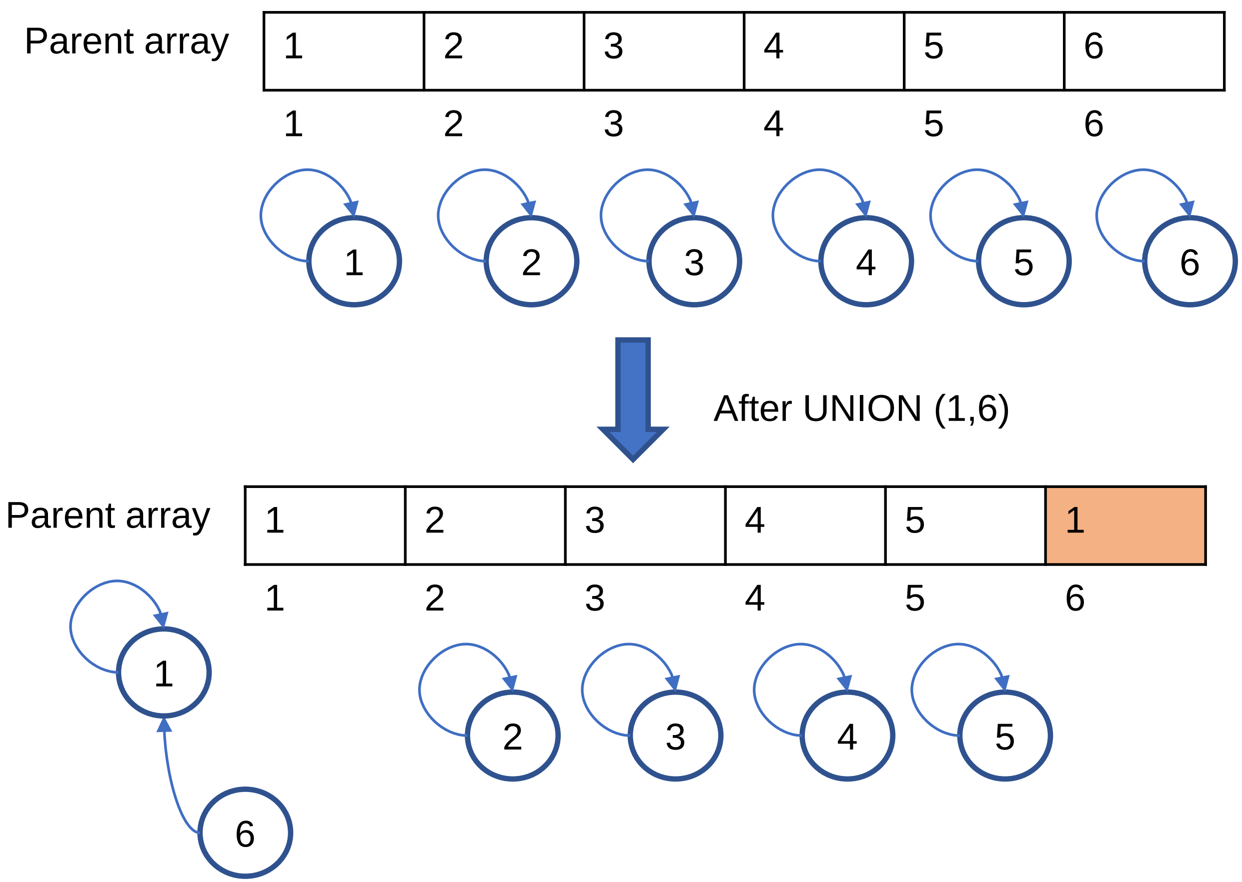
Parent array	1	2	3	4	5	6
	1	2	3	4	5	6



Create a collection of tiny trees, but still store them in the array

Tree implementation: fast ***UNION***

- To replace the two sets containing u and v by their union – update a parent of u to node v



Parent array

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

After UNION (1,6)

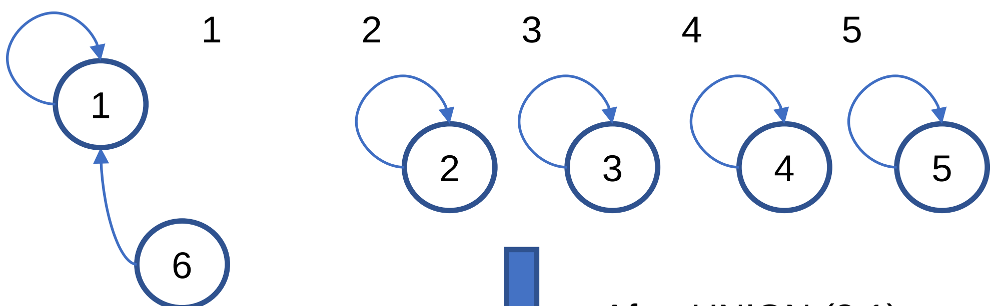
Parent array

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

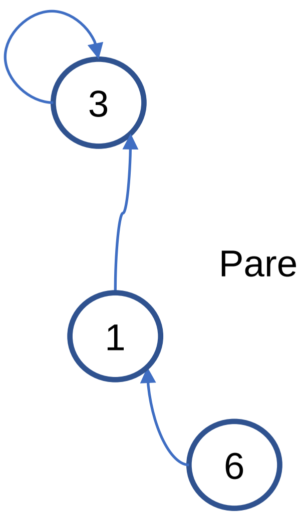
Tree implementation: fast *UNION*

- To replace the two sets containing u and v by their union – update a parent of u to node v

Parent array	1	2	3	4	5	1
	1	2	3	4	5	6



After UNION (3,1)



Parent array	3	2	3	4	5	1
	1	2	3	4	5	6

Tree implementation: **fast *UNION***

- To replace the two sets containing u and v by their union – update a parent of v to node u.
- Important to note: UNION operation is changing the root's parent only, but not the parent for all the elements in the second set.
- Therefore, the time complexity of UNION is **O(1)**.

Tree implementation: **slow *FIND***

- A FIND(x) on node x is performed by returning the root of the tree containing x.
- The time to perform this operation is proportional to the depth of the node representing x.
- It is possible to create a tree of depth n - 1 (Skewed Tree).
- The worst-case running time of a FIND is **O(*n*)** and m consecutive FIND operations take O(*mn*) time in the worst case. (not an improvement comparing to O(n) path algorithm to check for a cycle that we had before)

Fast *UNION* + Quick *FIND*

- The main problem with the previous approach is that we might get skewed trees and as a result the FIND operation takes O(n) time.
- We want to keep the height of each tree at most log n

UNION by Size

Simple heuristic:

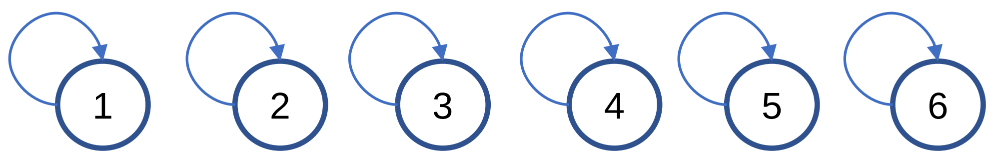
- Always make the smaller tree a subtree of the larger tree

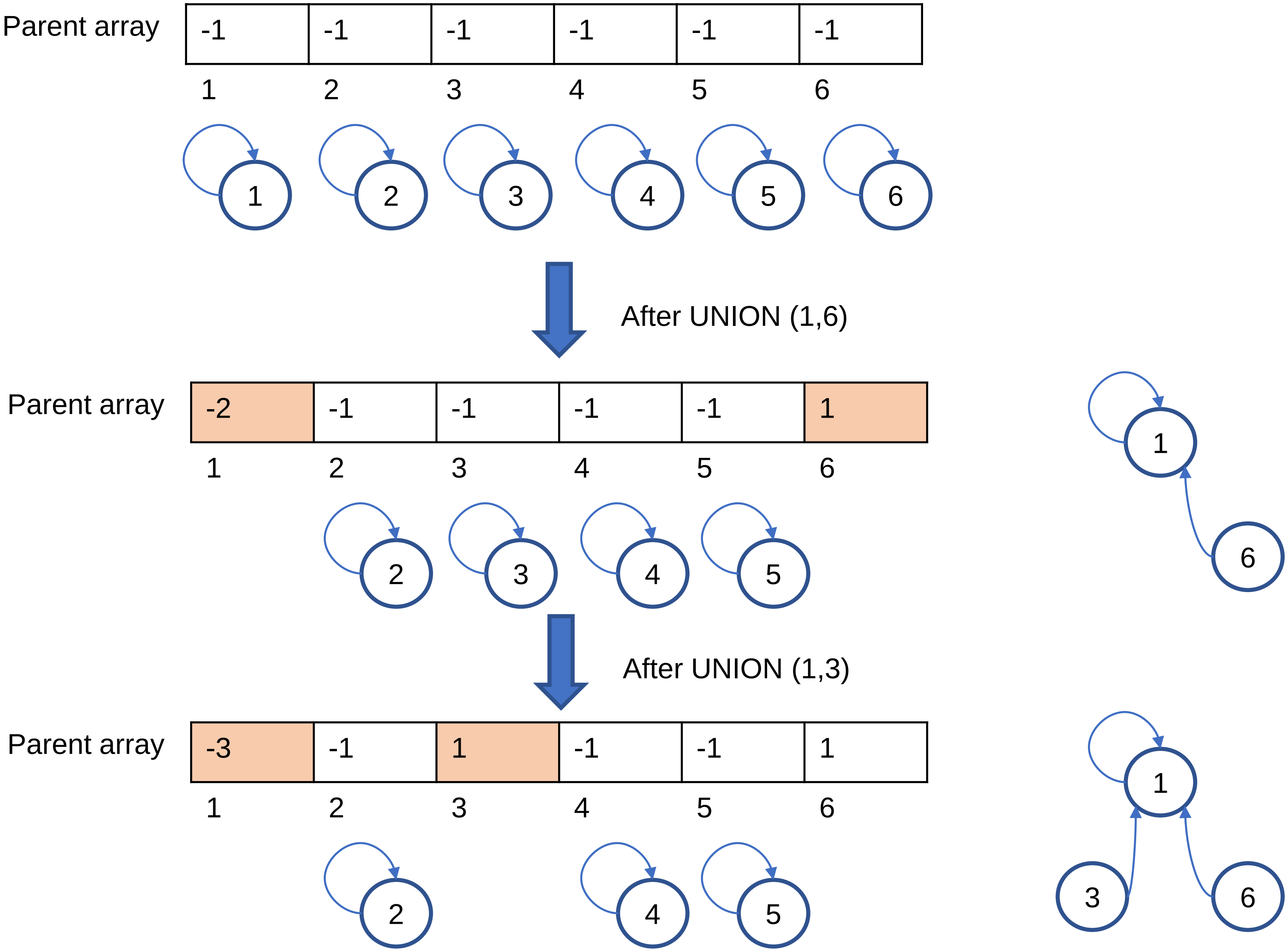
We use the same parent array

- We identify the root element of each tree by storing a negative integer representing the size of the tree rooted at node i

After n calls to MAKE-SET

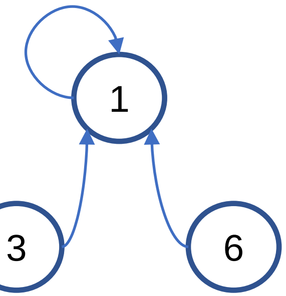
Parent array	-1	-1	-1	-1	-1	-1
	1	2	3	4	5	6




Parent array
-1 -1 -1 -1 -1 -1
1 2 3 4 5 6
1 2 3 4 5 6
After UNION (1,6)
Parent array
-2 -1 -1 -1 -1 1
1 2 3 4 5 6
2 3 4 5
1
6
After UNION (1,3)
Parent array
-3 -1 1 -1 -1 1
1 2 3 4 5 6
2 4 5
1
3 6

Parent array

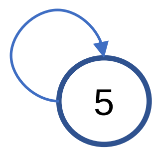
1	2	3	4	5	6
-3	-1	1	-1	-1	1

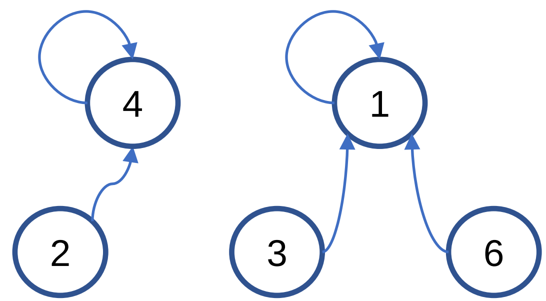


After UNION (4,2)

Parent array

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

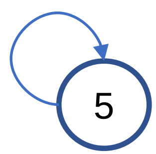


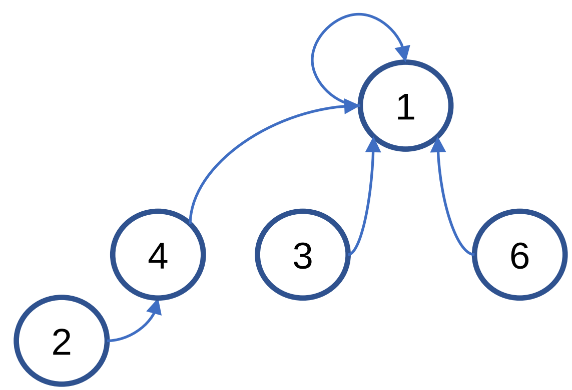


After UNION (4,1)

Parent array

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





UNION by size: **quick FIND**

- With UNION by size, the depth of any node is never more than log n. This is because a node is initially at depth 0. When its depth increases as a result of a UNION, it is placed in a tree that is at least twice as large as before.
- That means that the depth of each node can be increased at most log n times (there are at most log n UNIONs per each node).
- This gives the running time for a FIND operation as **O(log n)**
- A sequence of m FINDs and UNIONs takes O(m log n).

There are other methods that achieve the same and even better performance

- UNION by Height (UNION by Rank)
- Path Compression
- …

You do not have to know all of them for this course

Running time of UNION-FIND ADT implemented as a Tree (parent array)

Operation	
MAKE-SET(x)	**O(1)**
FIND(x)	**O(log n)**
UNION(x,y)	**O(1)**

Fast UNION – **Quick FIND**

Kruskal running time with UNION-FIND

Kruskal_MST (graph G(V,E))

```
1   E' := edges of G sorted by weights
2   T : = Ø

3   for  i from 1 to n:
4           MAKE-SET (node i)

5   for each edge (u,v) in E':
6           if FIND(u) ≠ FIND(v):
7                   T: = T U (u,v)
8                   UNION(u, v)

9           if |T| = |V| - 1:
                    break

    return T
```

Line 1: sorting m edges by weight. O(m log n).

Line 3: Making an array of size n: O(n).

Line 5: O(m) edges in the worst case.
For each edge: perform FIND O(log n) and

sometimes UNION in time O(1)

Thus, total time of the for loop is O(m log n)

Kruskal MST with UNION-FIND runs in time O(m log n) + O(n) + O(m log n) = **O (m log n)**