CSE 4303
Data Structures

Topic: Disjoint Sets

Sabbir Ahmed
Assistant Prof | CSE | IUT
sabbirahmed@iut-dhaka.edu

- **Disjoint Set** is used in scenarios where we need to *dynamically group elements* and check their *connectivity*.

- **Disjoint Set** is used in scenarios where we need to *dynamically group elements* and check their *connectivity*.

- A Disjoint Set (a.k.a *Union-Find*) maintains a collection of disjoint dynamic sets, $S = \{S_1, S_2, S_3, \dots, S_k\}$.

- **Disjoint Set** is used in scenarios where we need to *dynamically group elements* and check their *connectivity*.

- A Disjoint Set (a.k.a *Union-Find*) maintains a collection of disjoint dynamic sets, $S = \{S_1, S_2, S_3, \ldots, S_k\}$.

- Designed to efficiently perform two operations:

    - Find(x): Determines which subset a particular item x belongs to. (Can be used to check if two items are connected.)

- **Disjoint Set** is used in scenarios where we need to *dynamically group elements* and check their *connectivity*.

- A Disjoint Set (a.k.a *Union-Find*) maintains a collection of disjoint dynamic sets, $S = \{S_1, S_2, S_3, \dots, S_k\}$.

- Designed to efficiently perform two operations:

  - Find(x): Determines which subset a particular item x belongs to. (Can be used to check if two items are connected.)

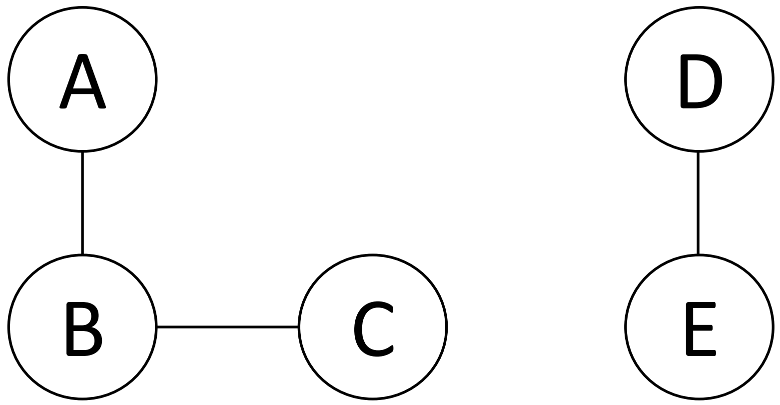
  - Union(u,v): Merges the subsets containing two items u and v into a single subset.

Let's say there are 5 people A, B, C, D, E.

A is a friend of B, B is a friend of C, and D is a friend of E.

So,

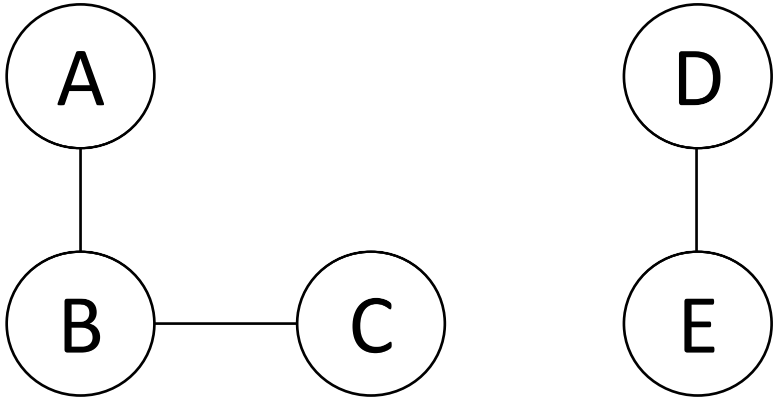
1) A, B, and C are connected.
2) D and E are connected.



Let’s say there are 5 people A, B, C, D, E.

A is a friend of B, B is a friend of C, and D is a friend of E.

So,
1) A, B, and C are connected.
2) D and E are connected.

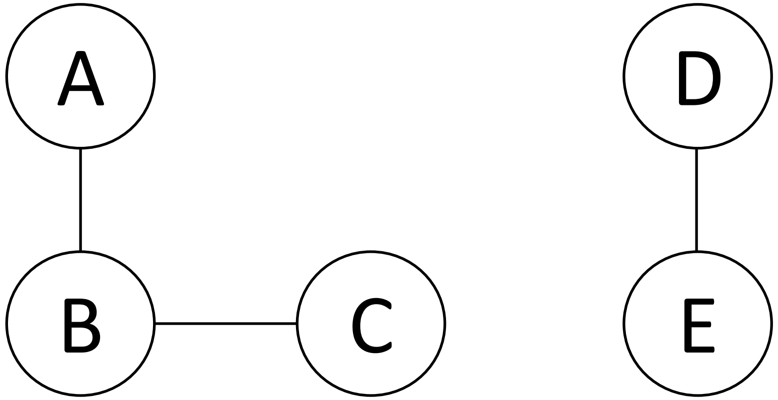


- Disjoint Set can check if one friend is connected to another in a direct or indirect way.
  (A,C are connected indirectly)

Let’s say there are 5 people A, B, C, D, E.

A is a friend of B, B is a friend of C, and D is a friend of E.

So,
1) A, B, and C are connected.
2) D and E are connected.



- Disjoint Set can check if one friend is connected to another in a direct or indirect way.
  (A,C are connected indirectly)
- Can determine the different disconnected subsets.
  (Here 2 different subsets are {A, B, C} and {D, E}.)

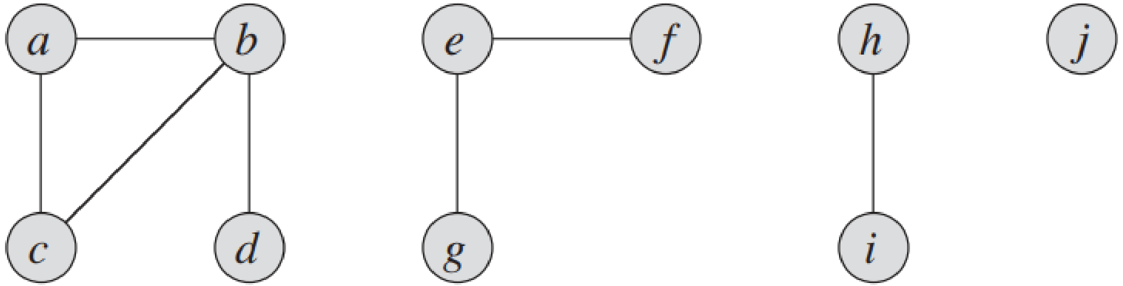
Applications:

- Can determine the *number of connected components* in an undirected graph.
- For an undirected graph, Disjoint Set can be used to *detect cycles* by checking if two vertices belong to the same connected component.
- *Minimum Spanning Tree*:

  Disjoint Set is used to check whether adding an edge to a growing spanning tree forms a cycle.

And so on…

Basics:

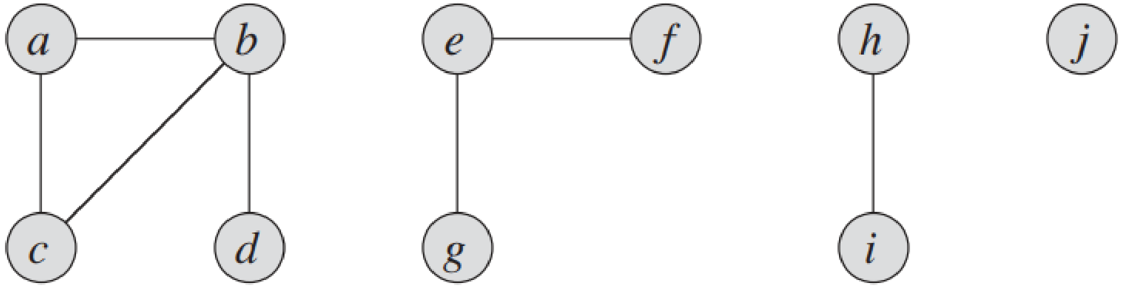


If these items are stored in a disjoint set:

$$S_1 = \{a, b, c, d\}$$
$$S_2 = \{e, f, g\}$$
$$S_3 = \{h, i\}$$
$$S_4 = \{j\}$$

Basics:

- Items that are connected will belong to the same set

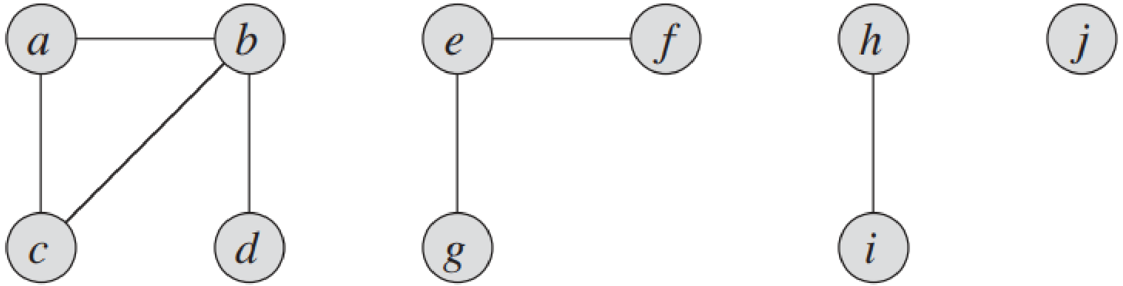


If these items are stored in a disjoint set:

$$
\begin{aligned}
S_1 &= \{a, b, c, d\} \\
S_2 &= \{e, f, g\} \\
S_3 &= \{h, i\} \\
S_4 &= \{j\}
\end{aligned}
$$

Basics:

- Items that are connected will belong to the same set
- The number of disjoint sets represents the number of groups/clusters/subgraphs

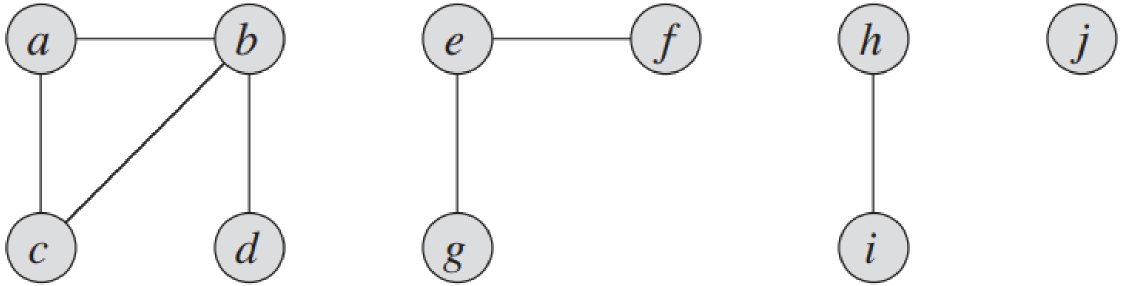


If these items are stored in a disjoint set:

$$S_1 = \{a, b, c, d\}$$
$$S_2 = \{e, f, g\}$$
$$S_3 = \{h, i\}$$
$$S_4 = \{j\}$$

Basics:

- Items that are connected will belong to the same set
- The number of disjoint sets represents the number of groups/clusters/subgraphs
- Every disjoint set will have a *representative*

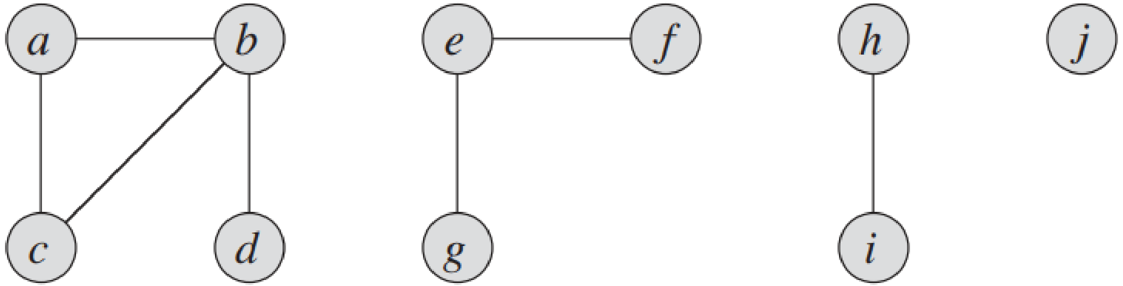


If these items are stored in a disjoint set:

$$S_1 = \{a, b, c, d\}$$
$$S_2 = \{e, f, g\}$$
$$S_3 = \{h, i\}$$
$$S_4 = \{j\}$$

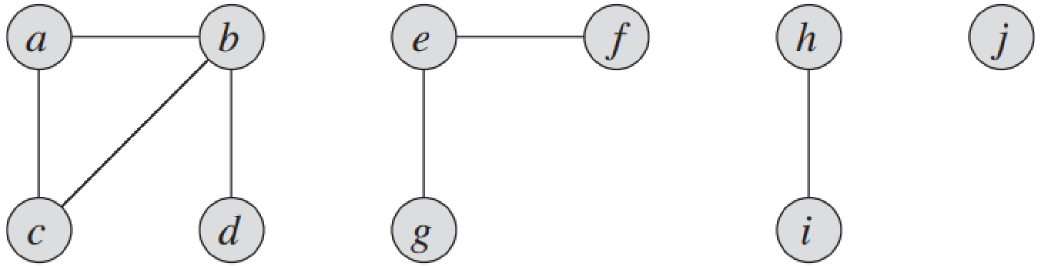
Basics:

- Items that are connected will belong to the same set
- The number of disjoint sets represents the number of groups/clusters/subgraphs
- Every disjoint set will have a *representative*
- If two nodes have the *same representative*: they belong to the *same set*.



If these items are stored in a disjoint set:

$$S_1 = \{a, b, c, d\}$$
$$S_2 = \{e, f, g\}$$
$$S_3 = \{h, i\}$$
$$S_4 = \{j\}$$

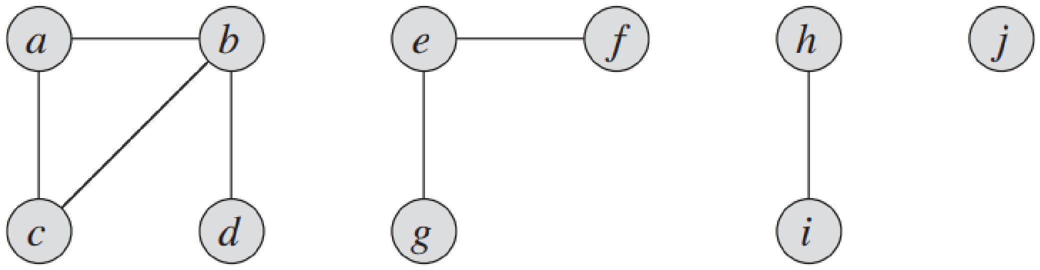


(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

Graph is a non-linear data structure with a finite number of vertices(nodes) and the edges that connect them.

Consider this Graph with 10 vertices and 7 edges



(a)

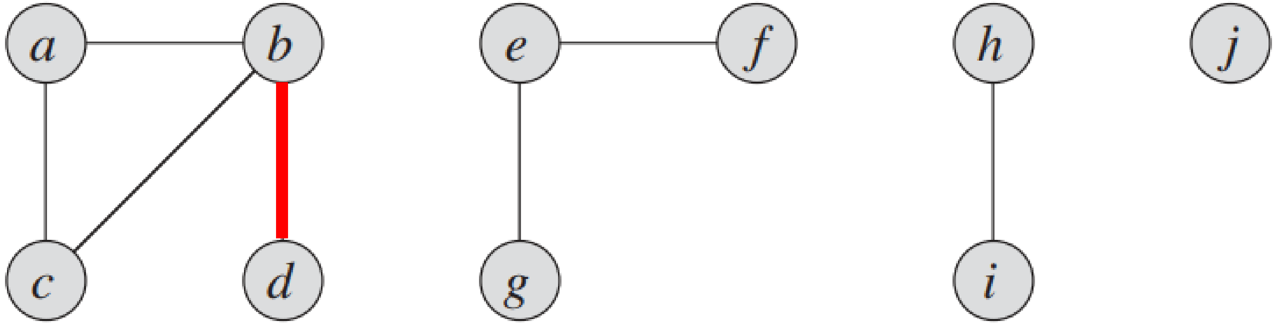
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

Graph is a non-linear data structure with a finite number of vertices(nodes) and the edges that connect them.

Consider this Graph with 10 vertices and 7 edges

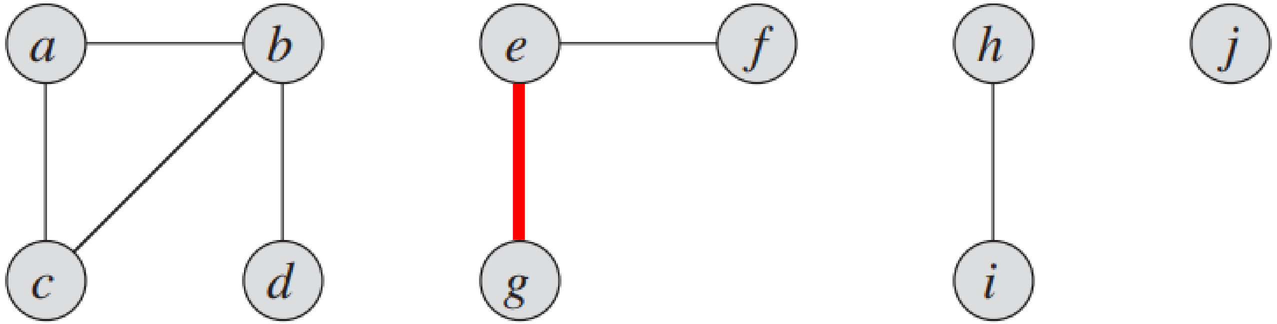
An Application of Disjoint-Set Data Structures

Determining the connected components of an undirected graph G=(V,E)



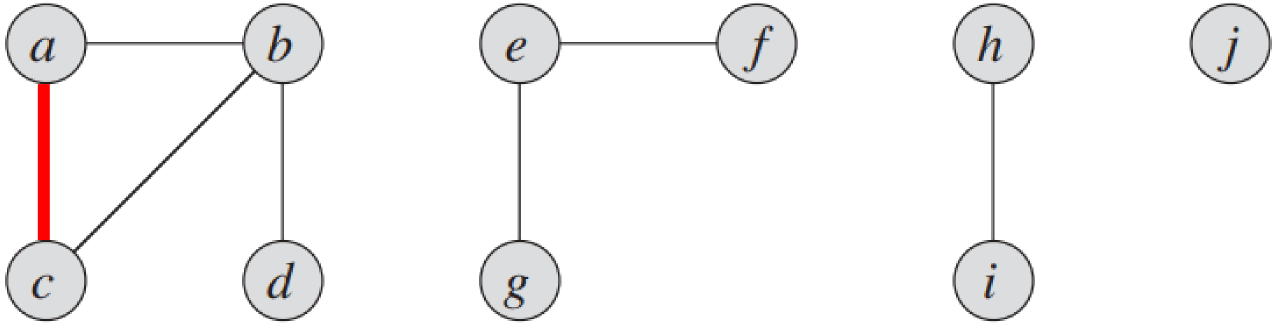
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$



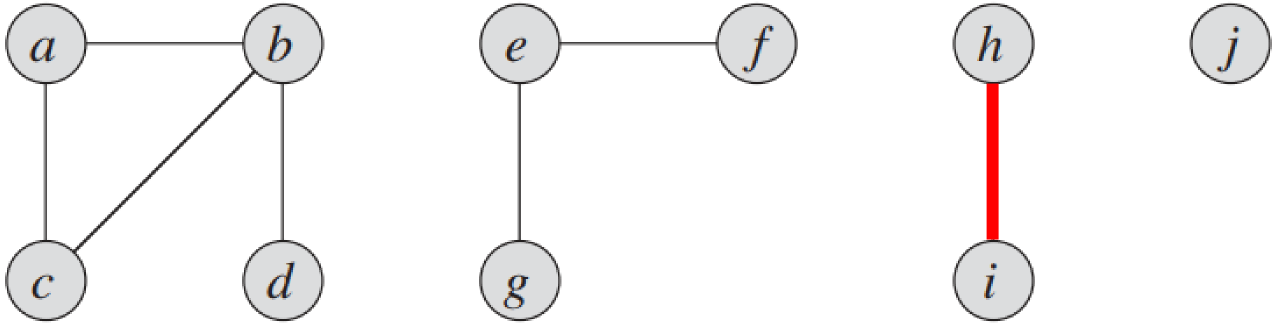
(a)

Edge processed	Collection of disjoint sets								
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$



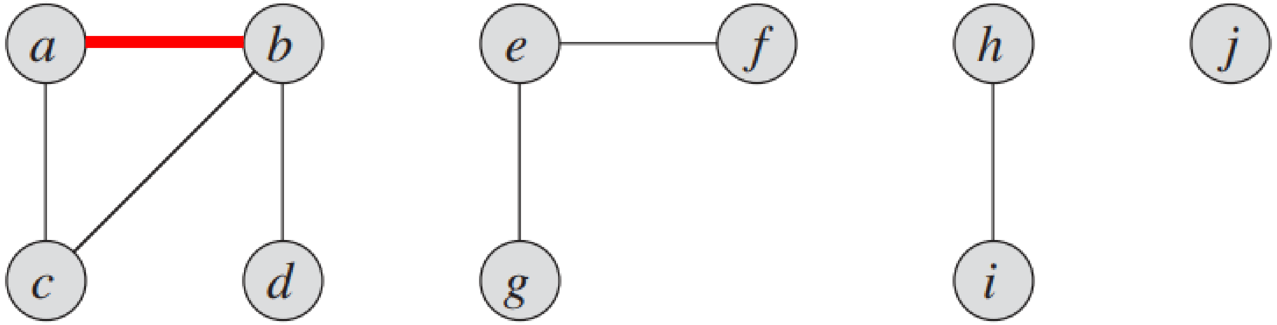
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$



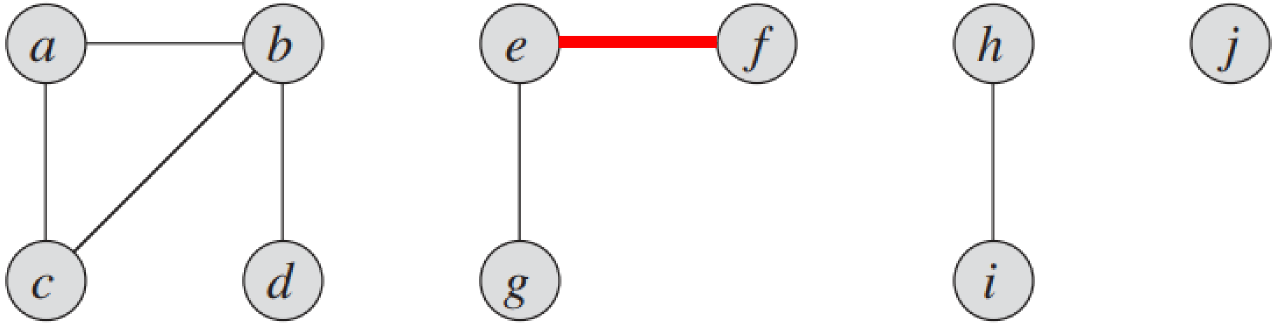
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$



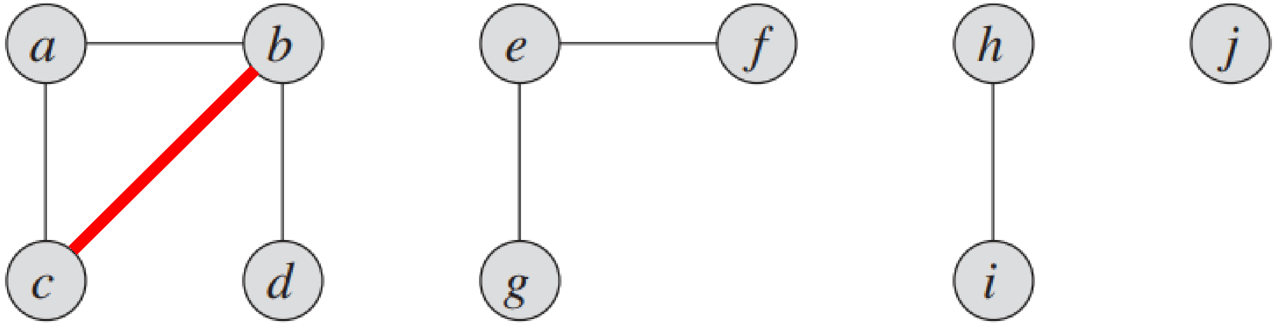
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$



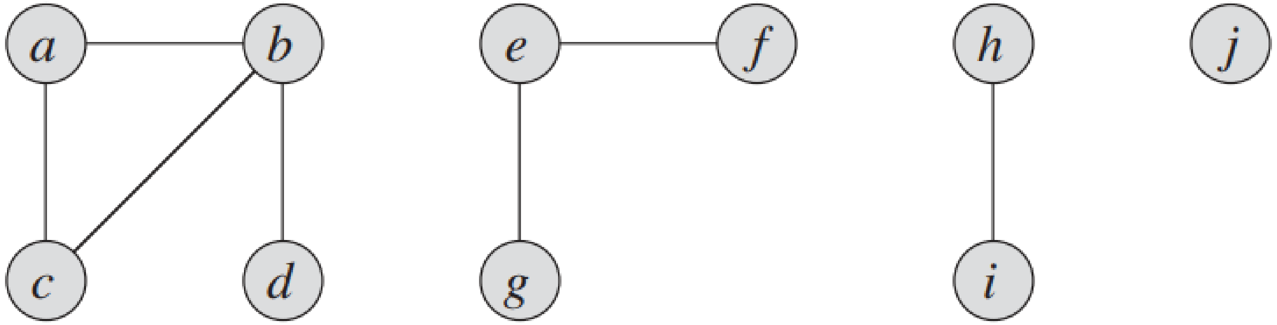
(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$



(a)

How can you detect cycle in a Graph using Disjoint Sets?

Edge processed	Collection of disjoint sets								
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Operations:

`makeSet(x)`	creates a new set with a single member x and points itself as the representative.

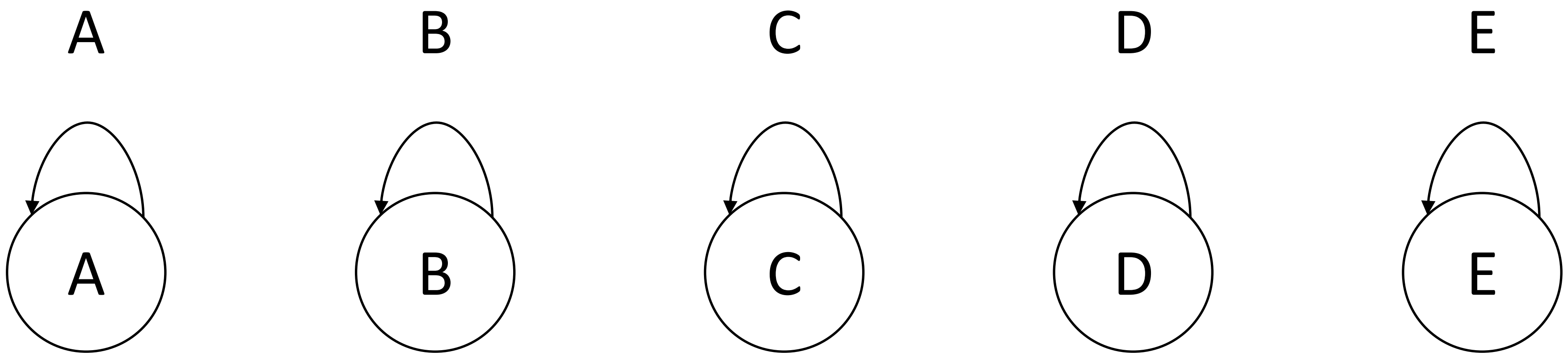
Operations:

`makeSet(x)`	creates a new set with a single member x and points itself as the representative.
`find(x)`	returns the representative of the set containing x.

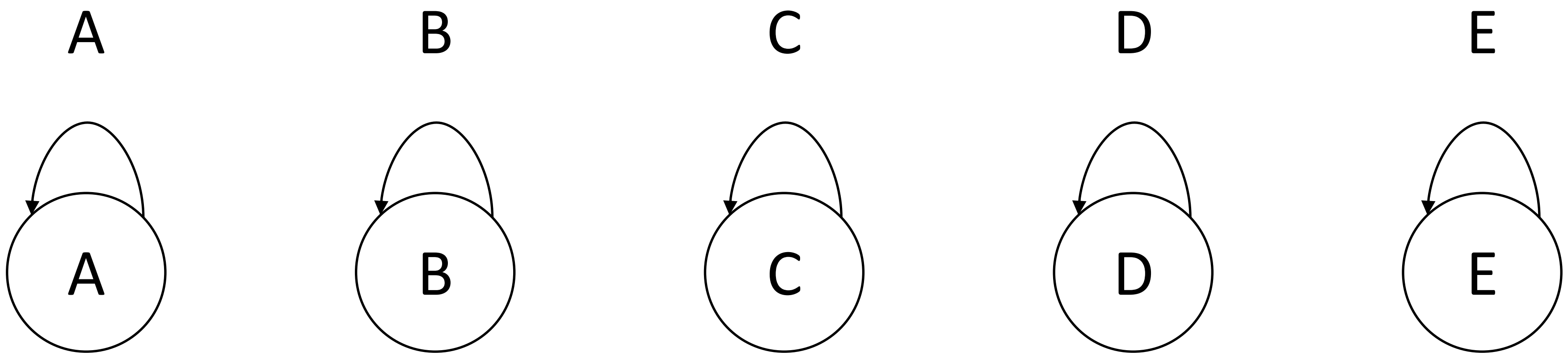
Operations:

`makeSet(x)`	creates a new set with a single member x and points itself as the representative.
`find(x)`	returns the representative of the set containing x.
`union(u,v)`	connects the representatives of two sets and forms a new set

makeset

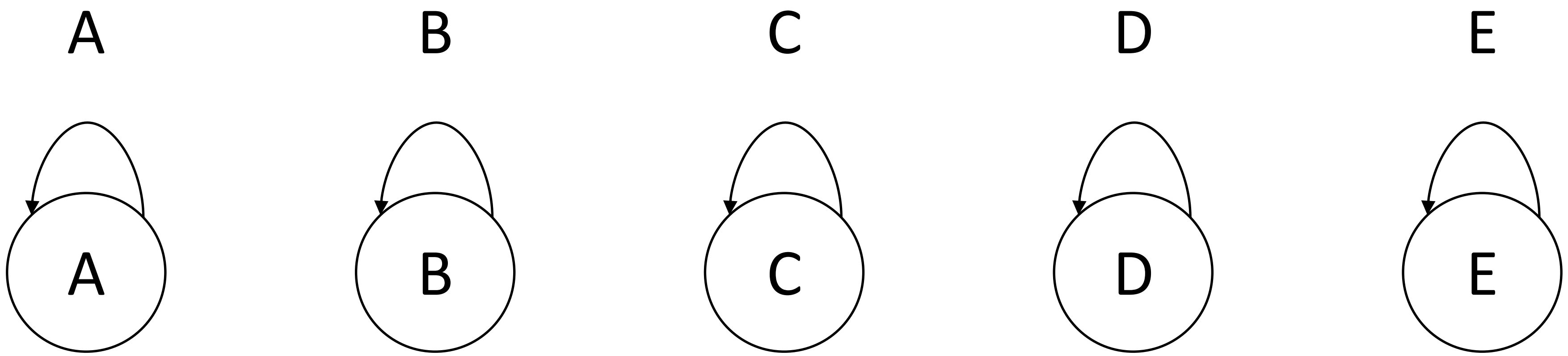


makeset



Are A,B connected?

makeset

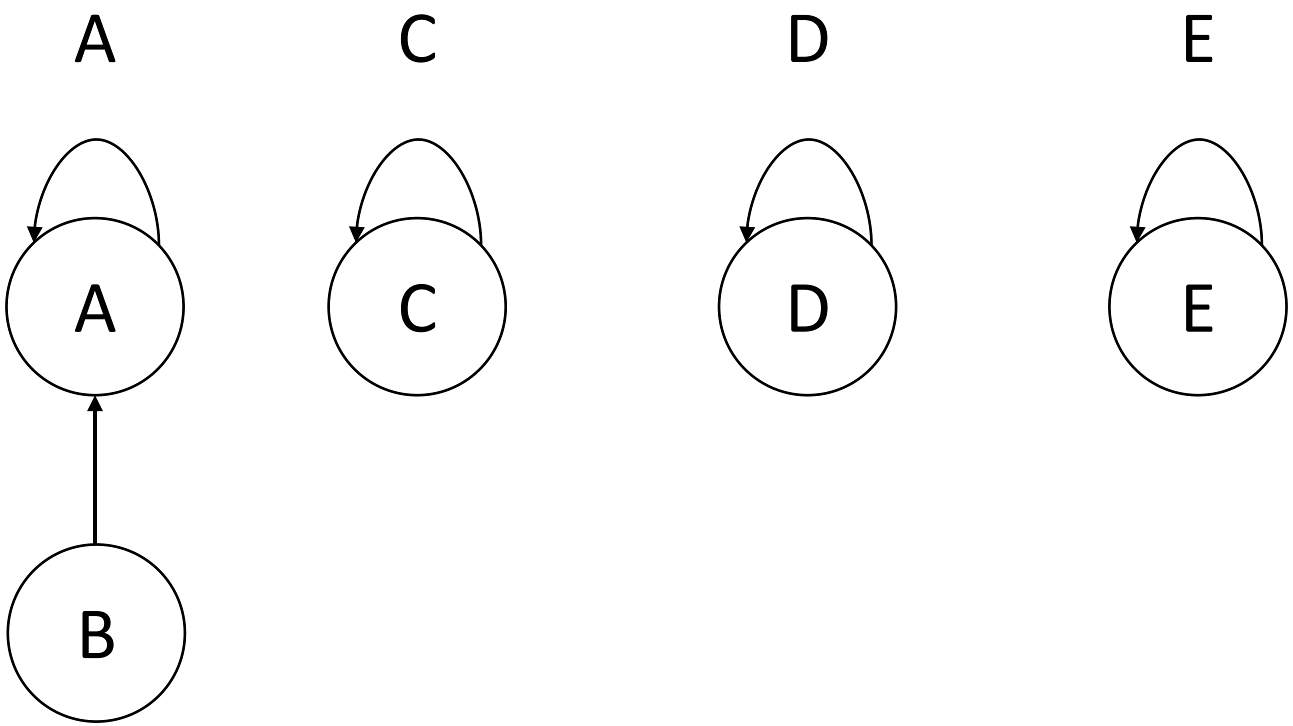


Are A,B connected?

find(A)=A

find(B)=B

A≠B --- Disjoint



Union(A,B)

find(A)=A

find(B)=B

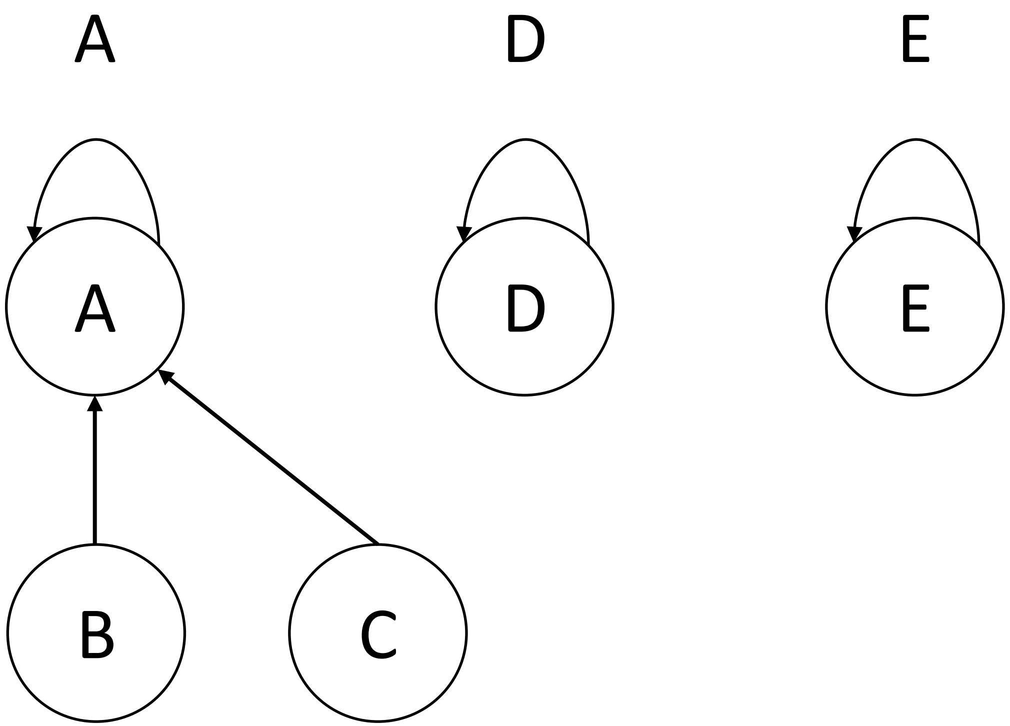
Form a new set with the representatives

Union(B,C)

find(B)=A

find(C)=C

Connect A,C

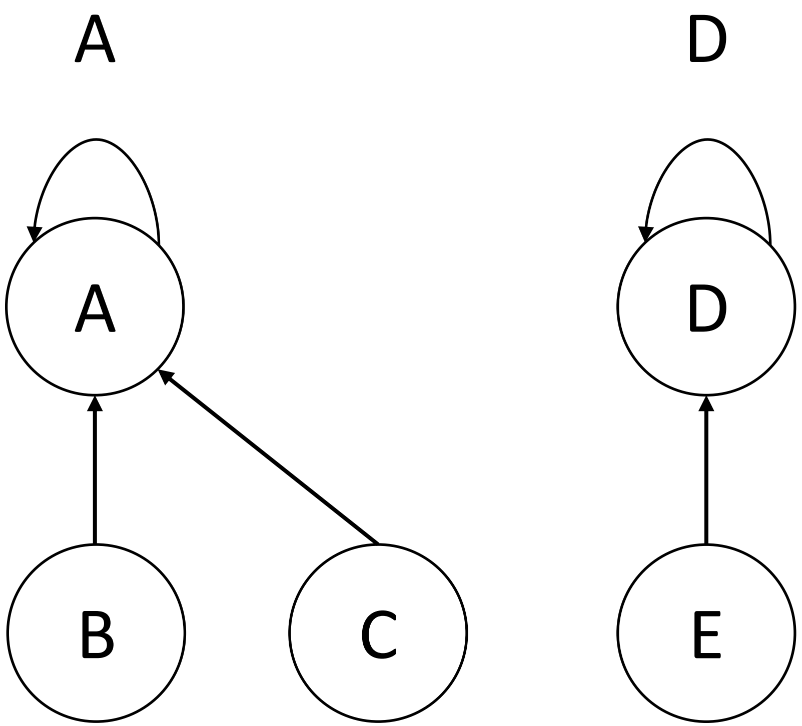


Union(D,E)

find(D)=D

find(E)=E

Connect D,E

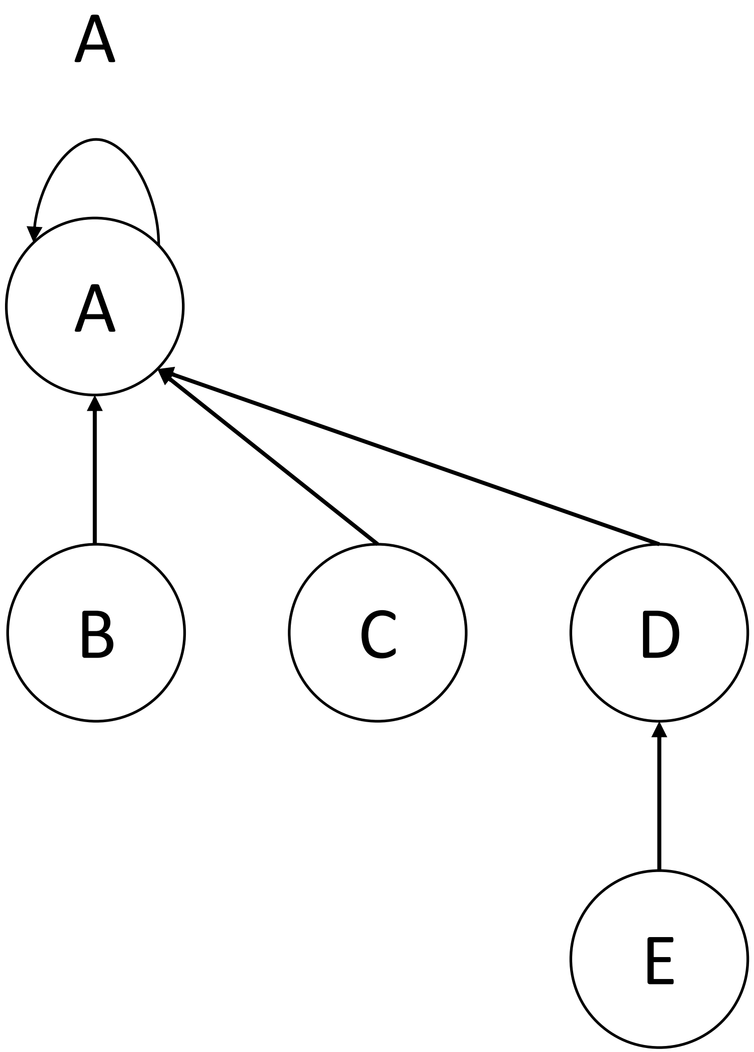


Union(C,E)

find(C)=A

find(E)=D

Connect A,D



```
int parent [#of nodes]

void makeset( u ) {
	parent[u] = u
}

void init () {
	for (i = 1 ... # of nodes)
		makeset(input[i]
}
```

```
--- find ( u ) {

     if parent [u] = u
          return u;


                    ?


}
```

```
--- find ( u ) {

    if parent [u] = u
        return u;

    else
        return find( parent[u] )
}
```

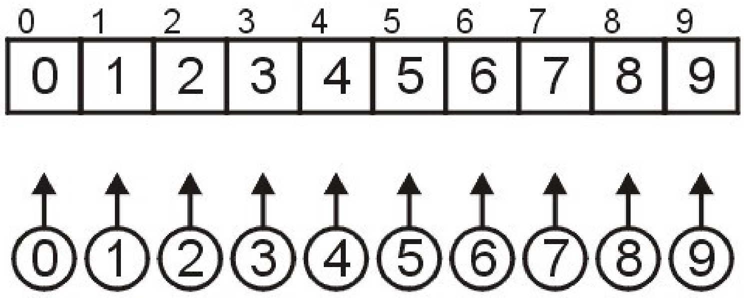
Simplest definition of **Union** operation: (sub-optimal)

```
void set_union( i, j ) {

     ri = find( i );
     rj = find( j );

                                   ?

}
```

Simplest definition of **Union** operation: (sub-optimal)

```
void set_union( i, j ) {

    ri = find( i );
    rj = find( j );

    if ( ri != rj ) {
        parent[rj] = ri;
    }
}
```

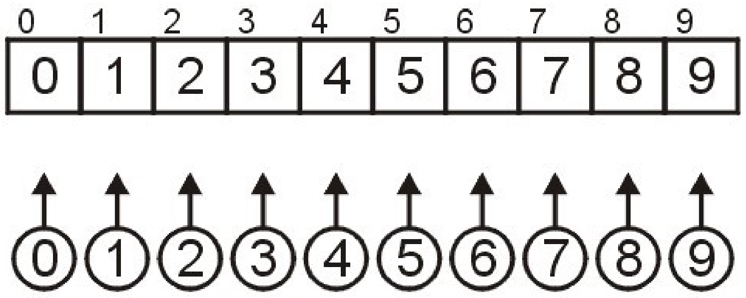
Consider the following disjoint set on the ten decimal digits:



$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}$

If we take the union of the sets containing 1 and 3

set_union(1, 3);

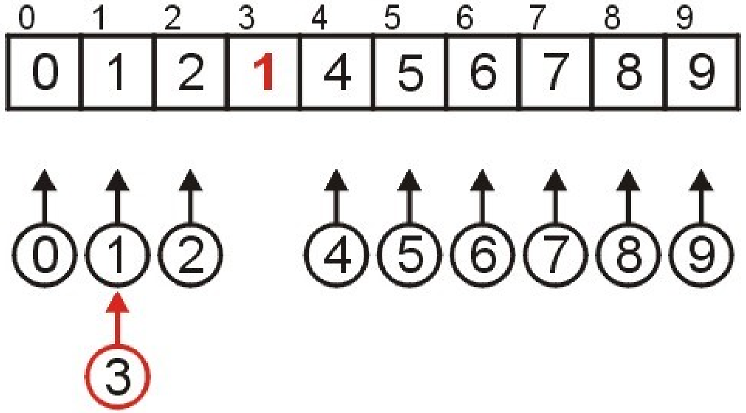


$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}$

If we take the union of the sets containing 1 and 3

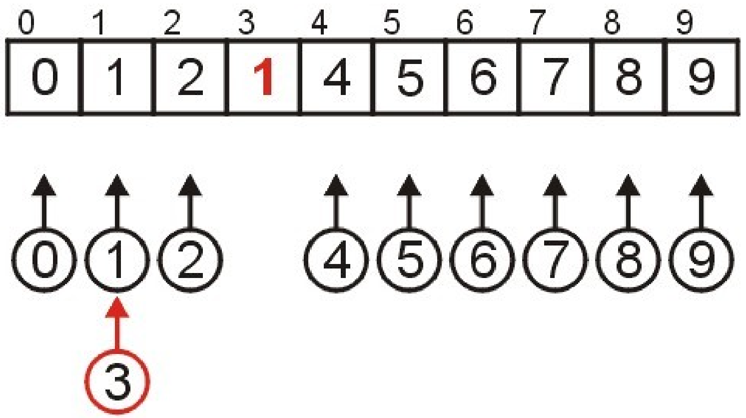
```
set_union(1, 3);
```

we perform a **find** on both entries and **update the second**



{0}, {1, 3}, {2}, {4}, {5}, {6}, {7}, {8}, {9}

Now, **`find(1)`** and **`find(3)`** will both return the integer 1

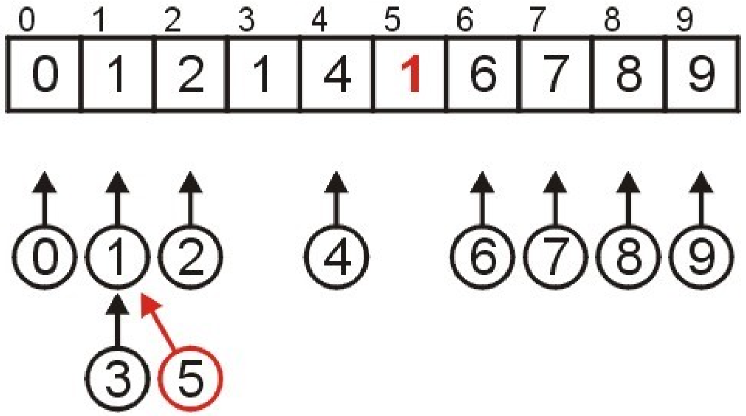


$\{0\}, \{1, 3\}, \{2\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}$

Next, take the union of the sets containing 3 and 5,

```
set_union(3, 5);
```

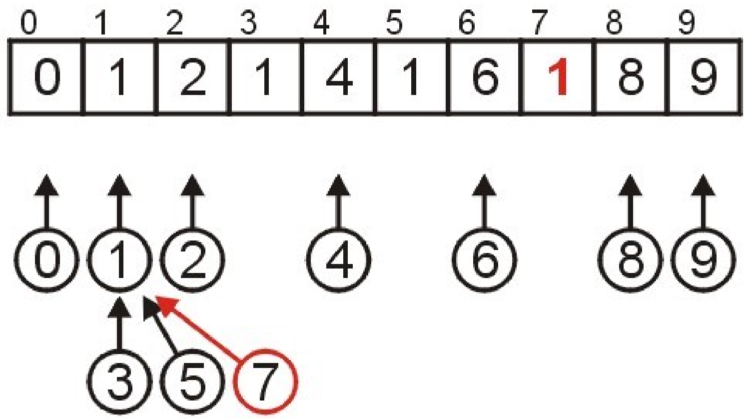
we perform a find on both entries and update the second



$\{0\}, \{1, 3, 5\}, \{2\}, \{4\}, \{6\}, \{7\}, \{8\}, \{9\}$

Now, if we take the union of the sets containing 5 and 7 `set_union(5, 7);`

we update the value stored in `find(7)` with the value `find(5)`:

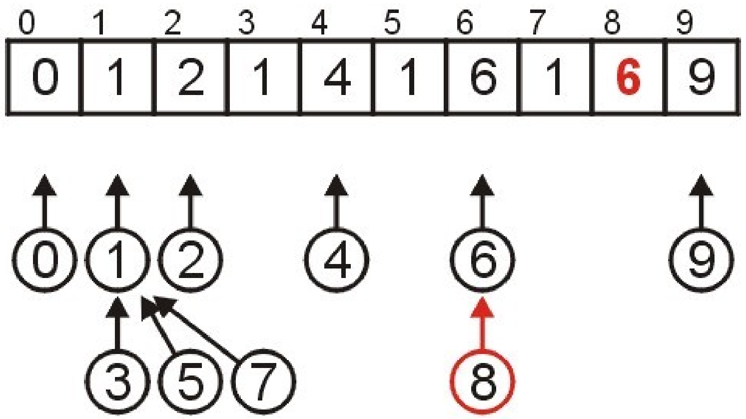


$\{0\}, \{1, 3, 5, 7\}, \{2\}, \{4\}, \{6\}, \{8\}, \{9\}$

Taking the union of the sets containing 6 and 8

```
set_union(6, 8);
```

we update the value stored in `find(8)` with the value `find(6)`:

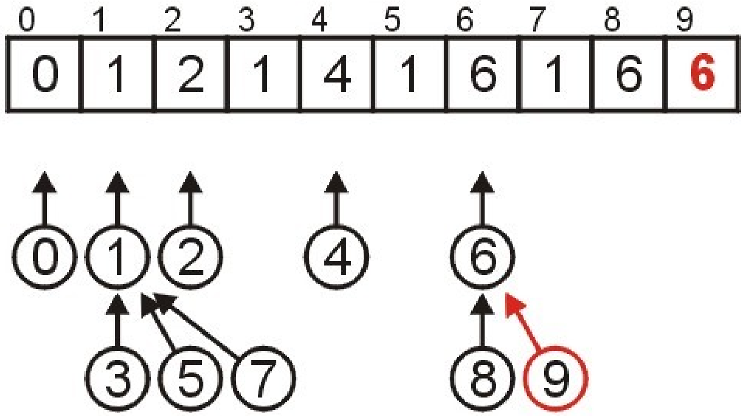


{0}, {1, 3, 5, 7}, {2}, {4}, {6, 8}, {9}

Taking the union of the sets containing 8 and 9

```
set_union(8, 9);
```

we update the value stored in `find(8)` with the value `find(9)`:



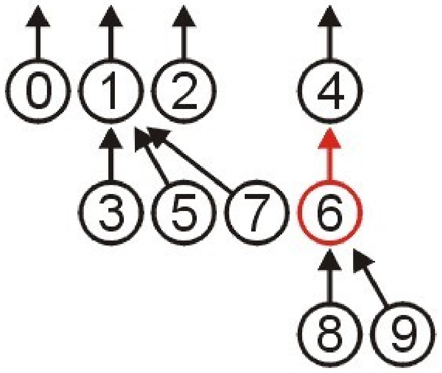
{0}, {1, 3, 5, 7}, {2}, {4}, {6, 8, 9}

Taking the union of the sets containing 4 and 8

```
set_union(4, 8);
```

we update the value stored in `find(8)` with the value `find(4)`:

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	**4**	1	6	6

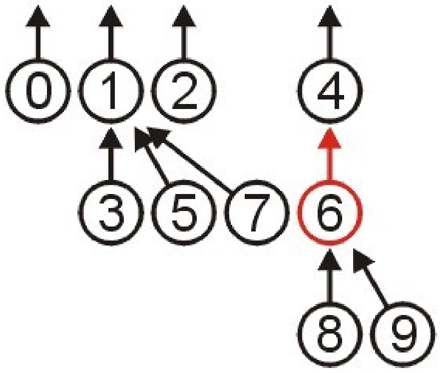


{0}, {1, 3, 5, 7}, {2}, {4, 6, 8, 9}

Taking the union of the sets containing 4 and 8
```
set_union(4, 8);
```

we update the value stored in `find(8)` with the value `find(4)`:

0	1	2	3	4	5	6	7	8	9
0	1	2	1	4	1	**4**	1	6	6



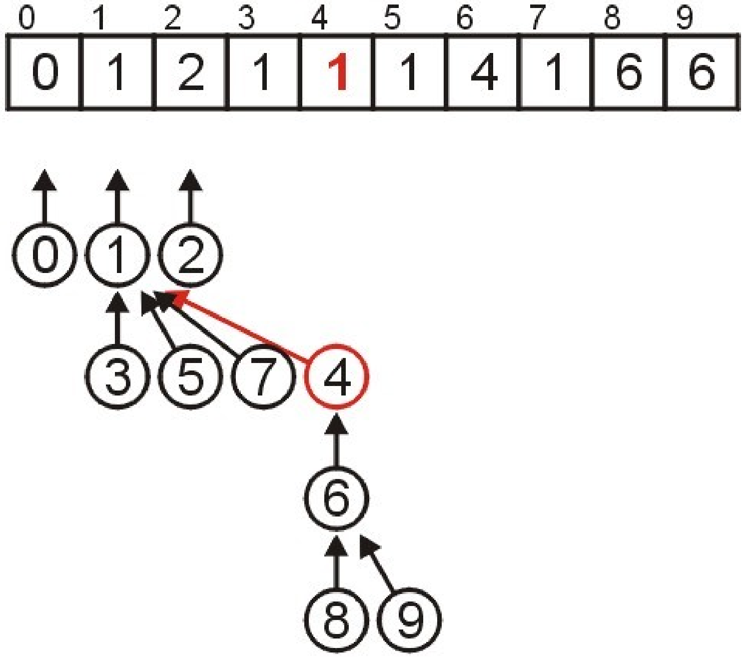
Parent of 8 didn't change, why?

{0}, {1, 3, 5, 7}, {2}, {4, 6, 8, 9}

Finally, if we take the union of the sets containing 5 and 6

we update the entry of `find(6)` with the value of `find(5)`:

```
set_union(5, 6);
```

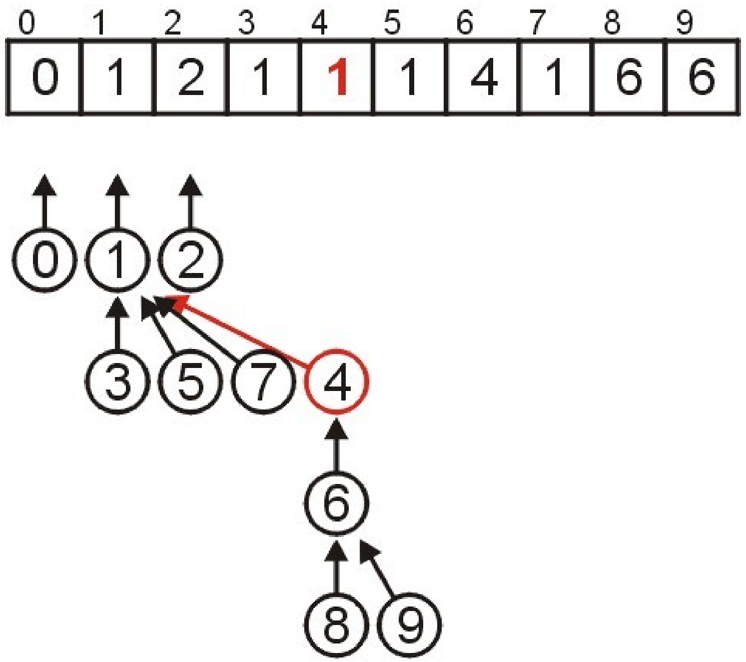


{0}, {1, 3, 4, 5, 6, 7, 8, 9}, {2}

Finally, if we take the union of the sets containing 5 and 6

we update the entry of `find(6)` with the value of `find(5)`:

```
set_union(5, 6);
```



{0}, {1, 3, 4, 5, 6, 7, 8, 9}, {2}

Observation:

If height grows, find() takes longer time.

Finally, if we take the union of the sets containing 5 and 6

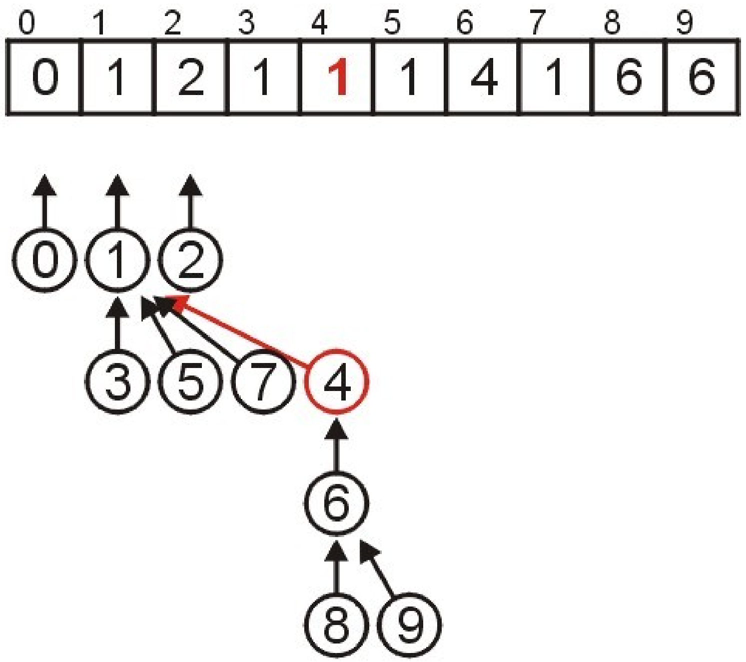
`set_union(5, 6);`

we update the entry of `find(6)` with the value of `find(5)`:

Solution

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.



$\{0\}, \{1, 3, 4, 5, 6, 7, 8, 9\}, \{2\}$

Observation:

If height grows, find() takes longer time.

```
void set_union( i, j ) {    // by size
     ri = find( i );
     rj = find( j );

     if ( ri != rj ) {
          if ( rank[ri] <= rank[rj] )
                         ?









     }
}
```

__Solution__

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.

```
void set_union( i, j ) {      // by size
    ri = find( i );
    rj = find( j );

    if ( ri != rj ) {
        if ( rank[ri] <= rank[rj] )
            parent[ri] = rj
        else





    }
}
```

<u>Solution</u>

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.

```
void set_union( i, j ) {     // by size
     ri = find( i );
     rj = find( j );

     if ( ri != rj ) {
          if ( rank[ri] <= rank[rj] )
               parent[ri] = rj
          else
               parent[rj] = ri




     }
}
```

Solution

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.

```
void set_union( i, j ) {     // by size
    ri = find( i );
    rj = find( j );

    if ( ri != rj ) {
        if ( rank[ri] <= rank[rj] )
            parent[ri] = rj
        else
            parent[rj] = ri


    }
}
```

Solution

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.

When will rank change?

```
void set_union( i, j ) {    // by size
     ri = find( i );
     rj = find( j );

     if ( ri != rj ) {
          if ( rank[ri] <= rank[rj] )
               parent[ri] = rj
          else
               parent[rj] = ri

          if ( rank[ri] == rank[rj] )
               rank[rj]++
     }
}
```

Solution

Union by Rank/Size:

During Union(x, y) operation: smaller tree (by size or rank) is attached under the root of the larger tree.

Observation

If height grows, find() will takes longer time.

Optimization*:*

Path Compression:

During the Find(x) operation, the tree is flattened by making every node on the path point directly to the root.

This reduces the depth of the tree and speeds up future operations.

Observation

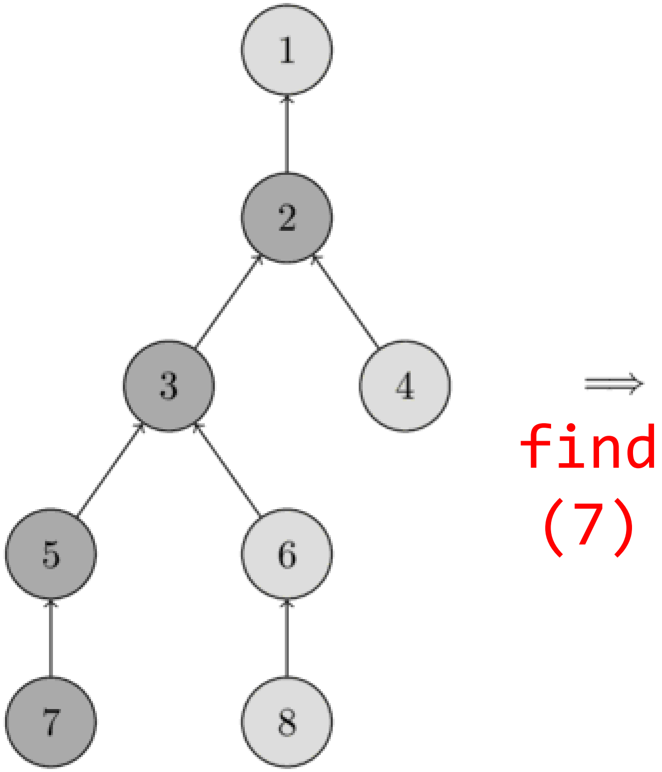
If height grows, find() will takes longer time.

Optimization:

Path Compression:

During the Find(x) operation, the tree is flattened by making every node on the path point directly to the root.

This reduces the depth of the tree and speeds up future operations.



Observation

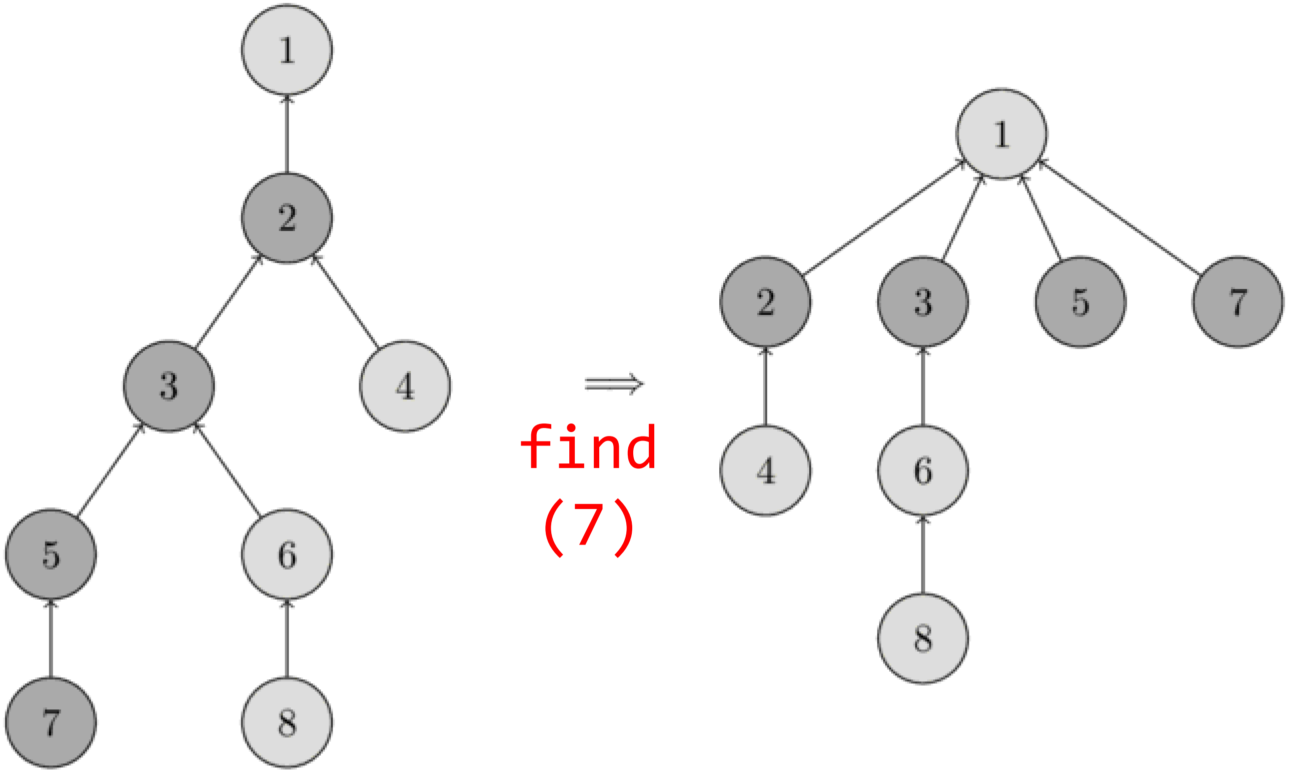
If height grows, find() will takes longer time.

Optimization:

Path Compression:

During the Find(x) operation, the tree is flattened by making every node on the path point directly to the root.

This reduces the depth of the tree and speeds up future operations.



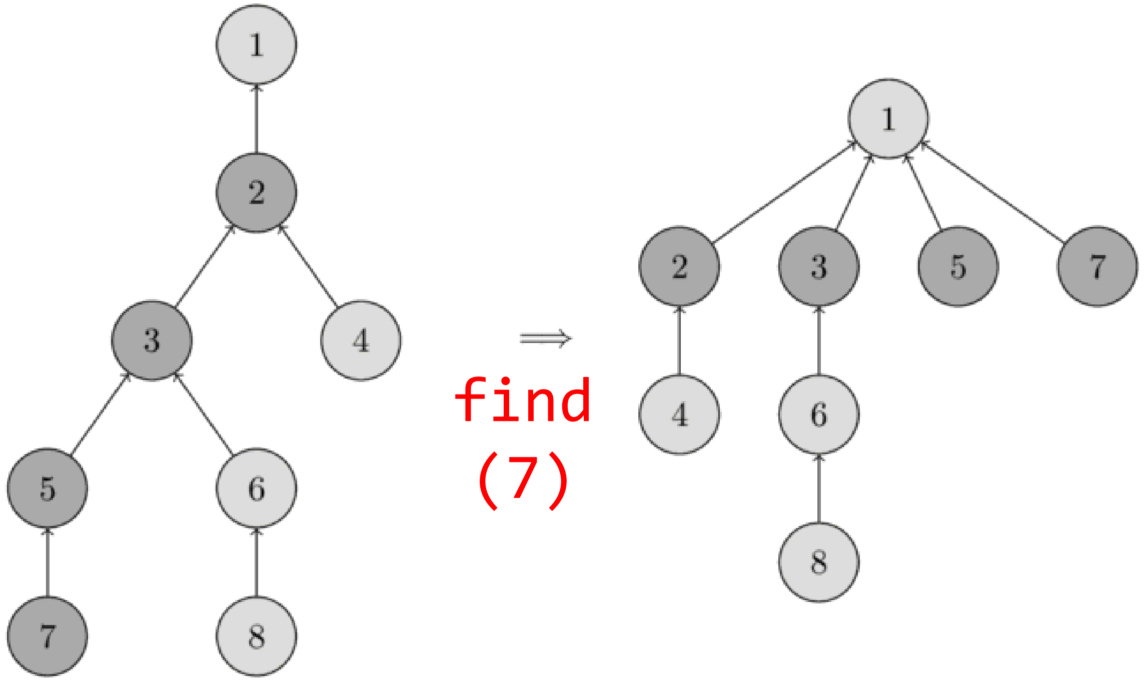
how??

```
--- find (u) {
     if parent[u] == u
          return u

     return parent[u] = find( parent[u] )
}
```

Path Compression
Simple trick

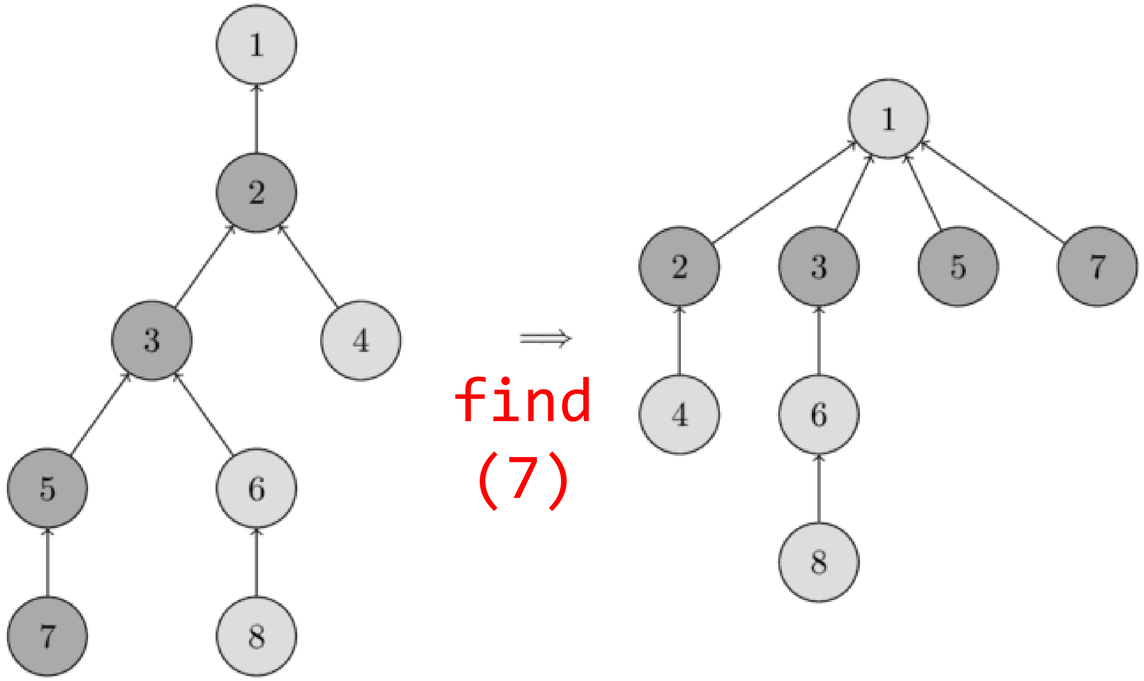
While returning from recursion, update the parent of each node

```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



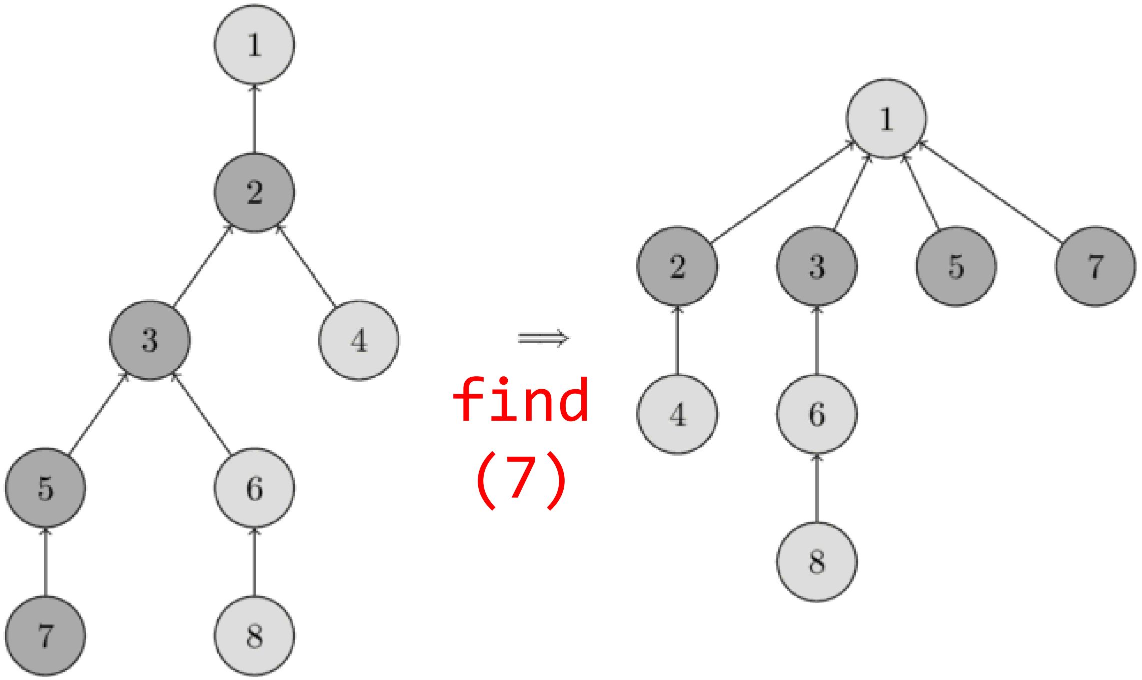
find (7)

```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



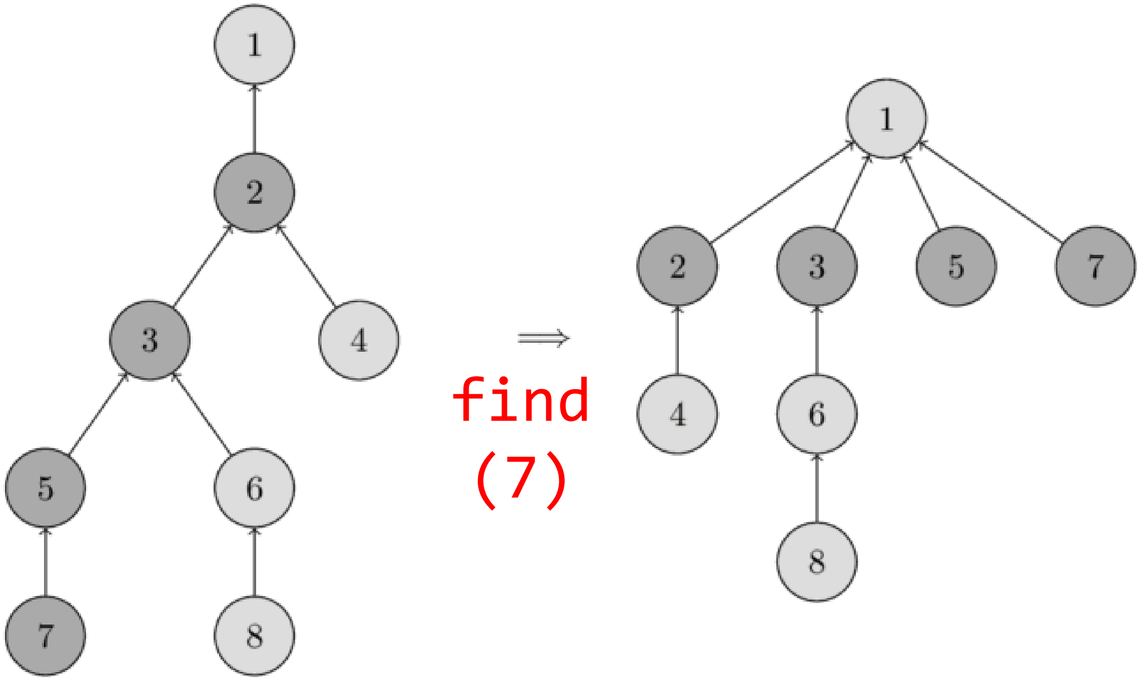
parent[7] = find (5)
find (7)

```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



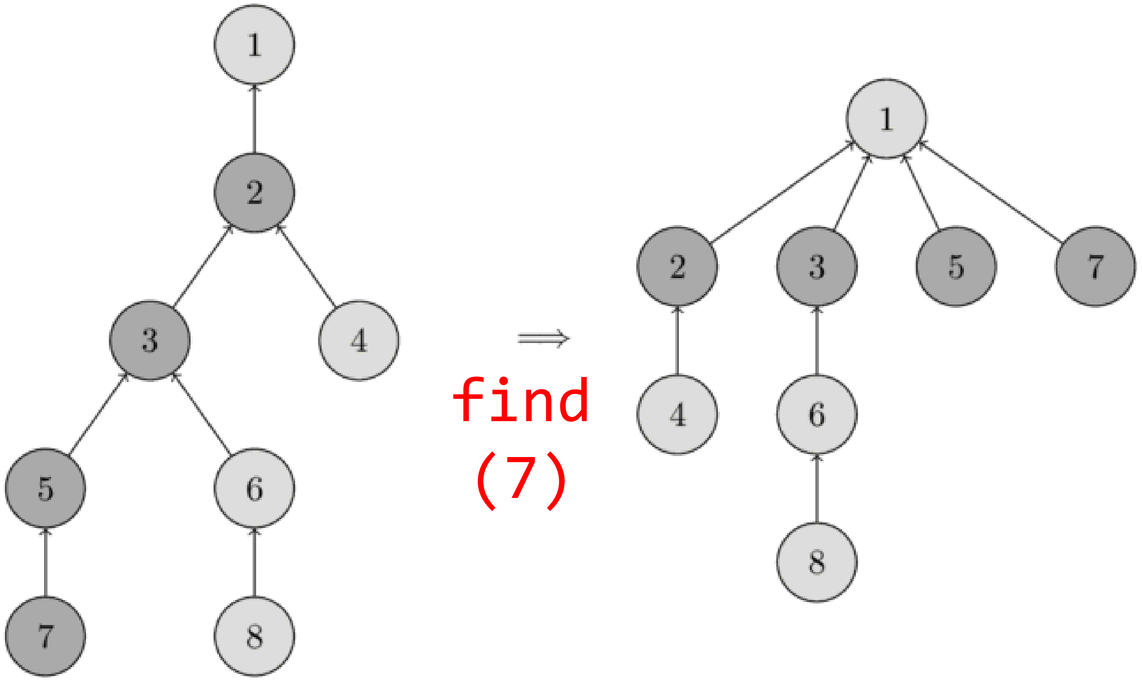
parent[5] = find (3)
parent[7] = find (5)
find (7)

```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



parent[3] = find (2)
parent[5] = find (3)
parent[7] = find (5)
find (7)

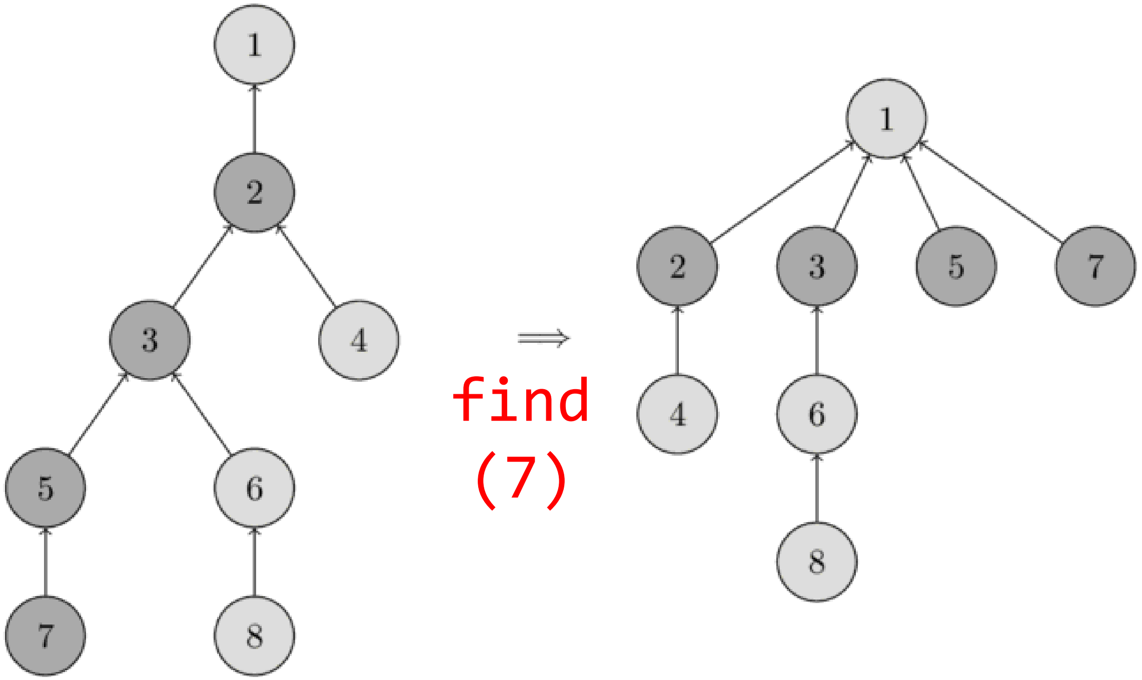
```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



parent[2] = find (1)
parent[3] = find (2)
parent[5] = find (3)
parent[7] = find (5)
find (7)

> find (1) = 1
>
> Usual find() function would directly return now
>
> But path_compression will utilize this information

```
--- find (u) {
        if parent[u] == u
                return u

        return parent[u] = find( parent[u] )
}
```



parent[2] = find (1)
parent[3] = find (2)
parent[5] = find (3)
parent[7] = find (5)
find (7)

find(1) returns 1
parent[2] = find(1) = 1
parent[3] = find(2) = 1
parent[5] = find(3) = 1
parent[7] = find(5) = 1

Application: Maze Generation

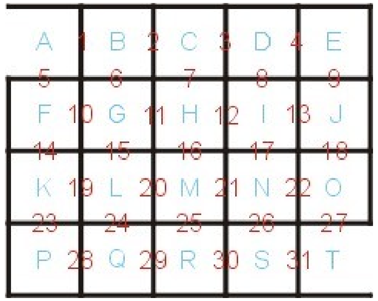
What we will do is the following:

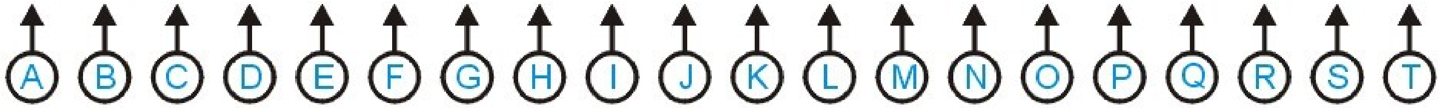
- Start with the entire grid subdivided into squares
- Represent each square as a separate disjoint set
- Repeat the following algorithm:
  - `Randomly choose a wall`
  - `If that wall connects two disjoint set of cells, then remove the wall and union the two sets`

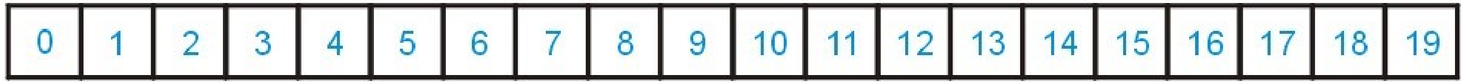
- To ensure that you do not randomly remove the same wall twice, we can have an array of unchecked walls

Application: Maze Generation

Let us begin with an entrance, an exit, and a disjoint set of 20 squares and 31 interior walls

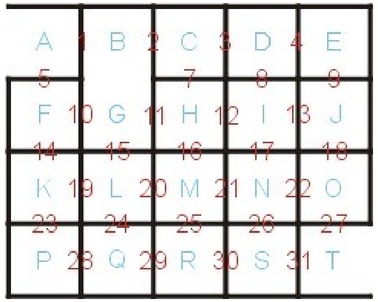


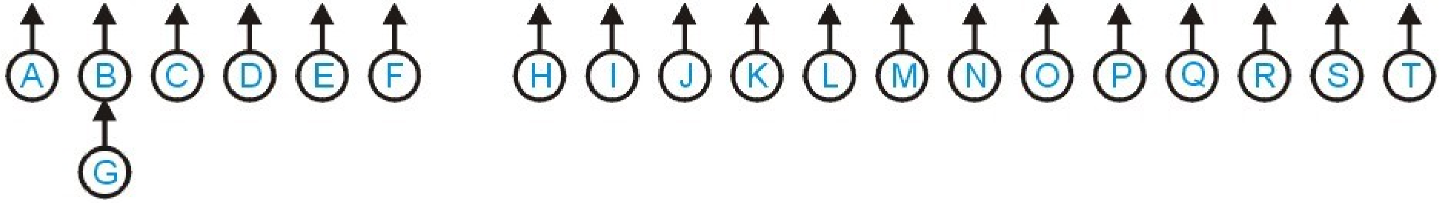




First, we select 6 which joins cells B and G

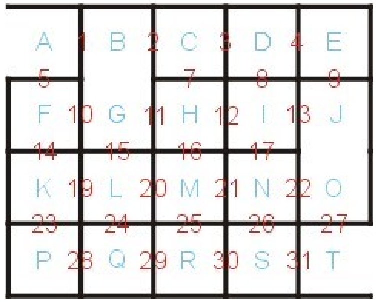
- Both have height 0

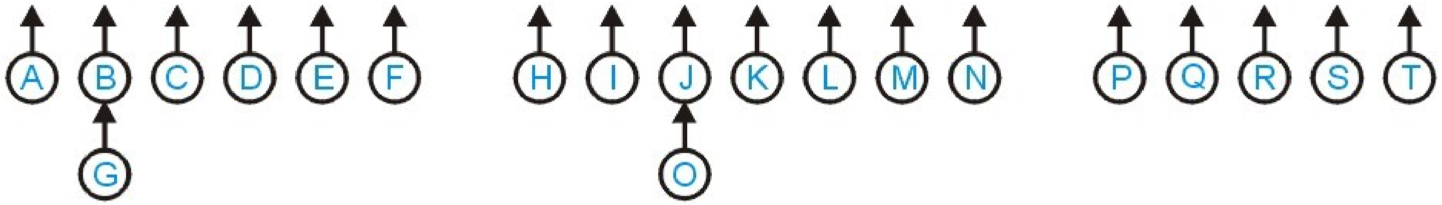




0	1	2	3	4	5	1	7	8	9	10	11	12	13	14	15	16	17	18	19

Next we select wall 18 which joins regions J and O

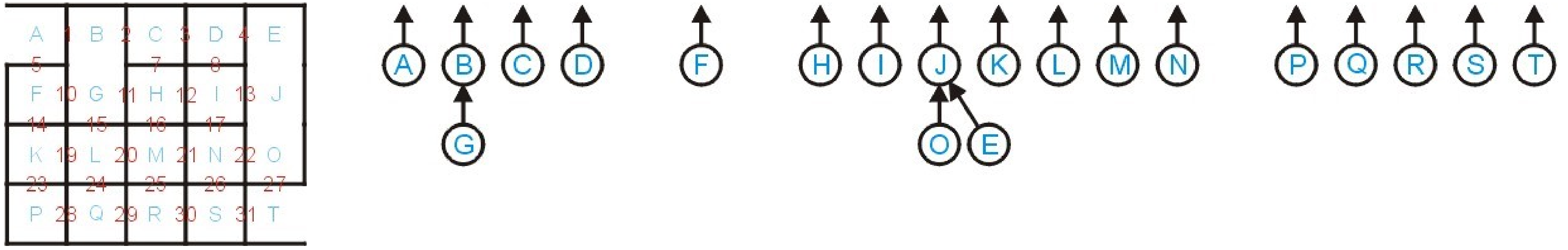




0	1	2	3	4	5	1	7	8	9	10	11	12	13	9	15	16	17	18	19

Next we select wall 9 which joins the disjoint sets E and J

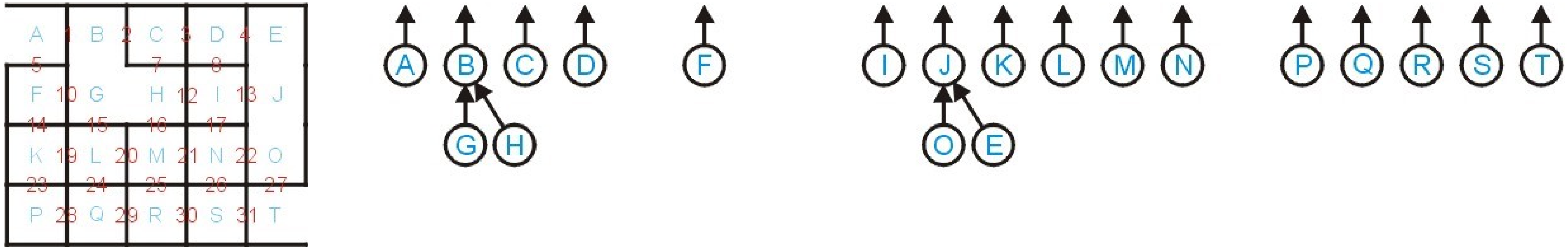
- The disjoint set containing E has height 0, and therefore it is attached to J



0	1	2	3	9	5	1	7	8	9	10	11	12	13	9	15	16	17	18	19

Next we select wall 11 which joins the sets identified by B and H

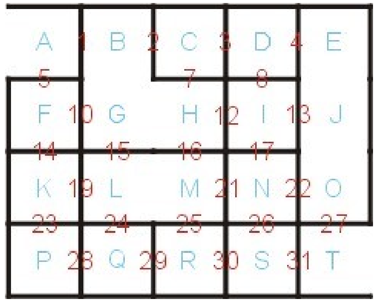
- H has height 0 and therefore we attach it to B

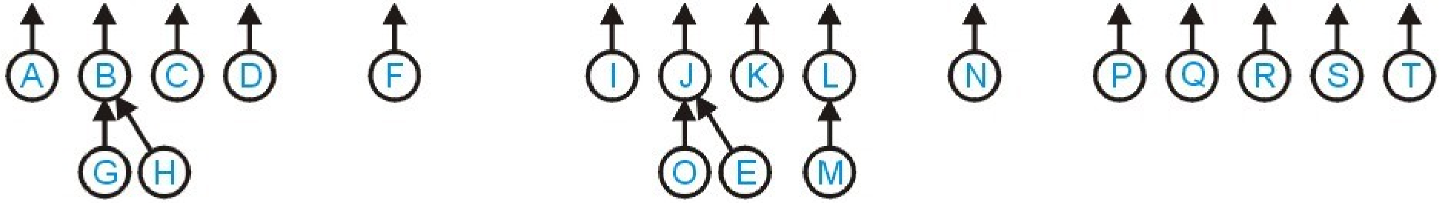


0	1	2	3	9	5	1	1	8	9	10	11	12	13	9	15	16	17	18	19

Next we select wall 20 which joins disjoint sets L and M

- Both are height 0

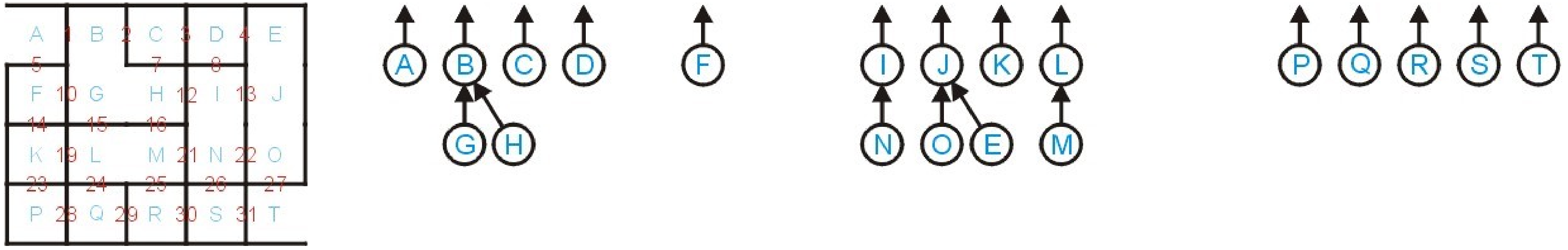




0	1	2	3	9	5	1	1	8	9	10	11	**11**	13	9	15	16	17	18	19

Next we select wall 17 which joins disjoint sets I and N

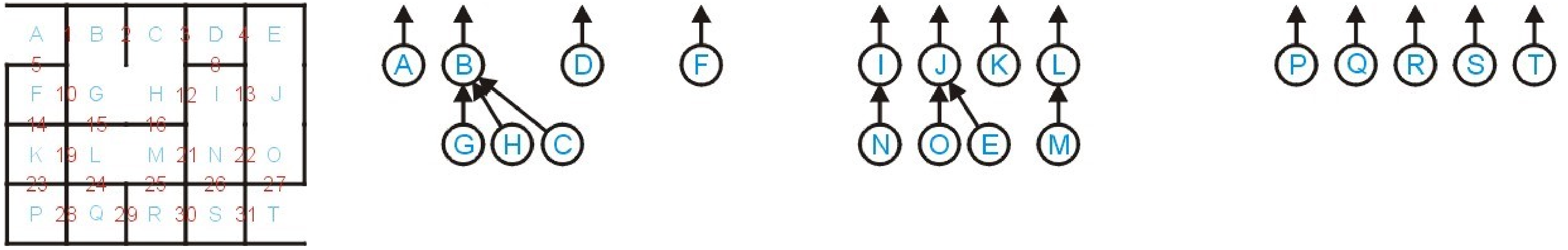
- Both are height 0



0	1	2	3	9	5	1	1	8	9	10	11	11	8	9	15	16	17	18	19

Next we select wall 7 which joins the disjoint set C and the disjoint set identified by B

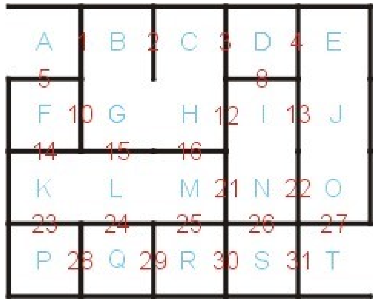
- C has height 0 and thus we attach it to B

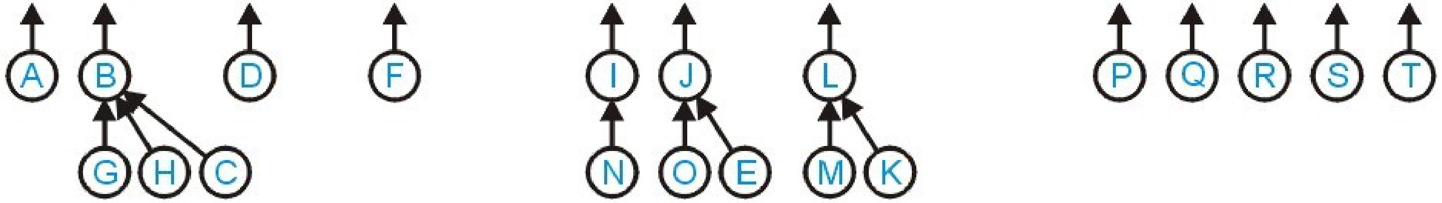


0	1	1	3	9	5	1	1	8	9	10	11	11	8	9	15	16	17	18	19

Next we select wall 19 which joins the disjoint set K to the disjoint sent identified by L

- Because K has height 0, we attach it to L

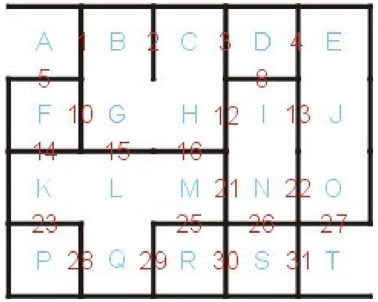


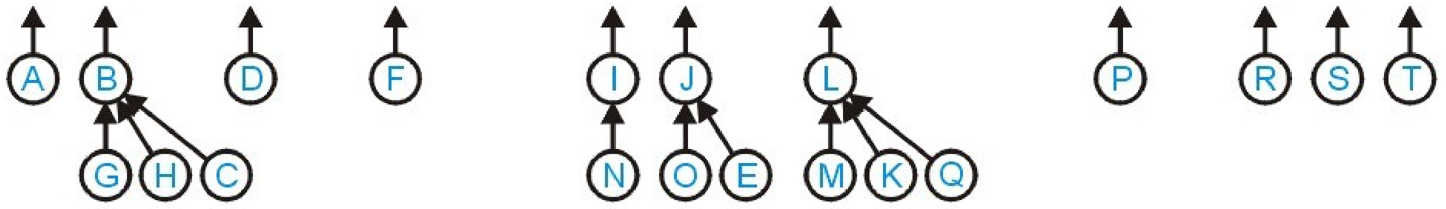


0	1	1	3	9	5	1	1	8	9	11	11	11	8	9	15	16	17	18	19

Next we select wall 23 and join the disjoint set Q with the set identified by L

- Again, Q has height 0 so we attach it to L

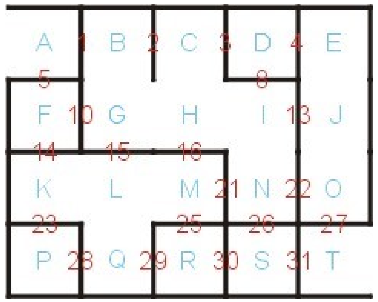


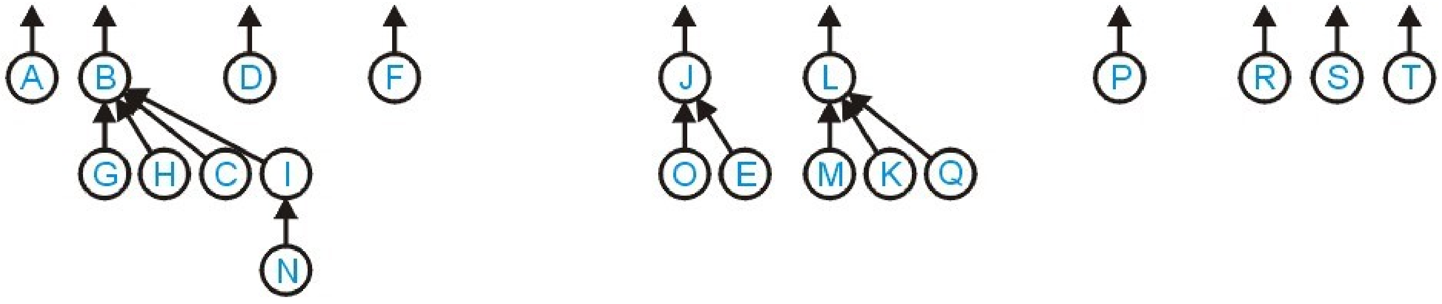


0	1	1	3	9	5	1	1	8	9	11	11	11	8	9	15	**11**	17	18	19

Next we select wall 12 which joints the disjoint sets identified by B and I

- They both have the same height, but B has more nodes, so we add I to the node B

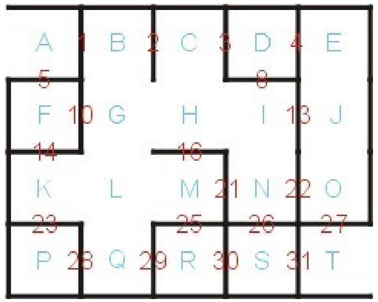


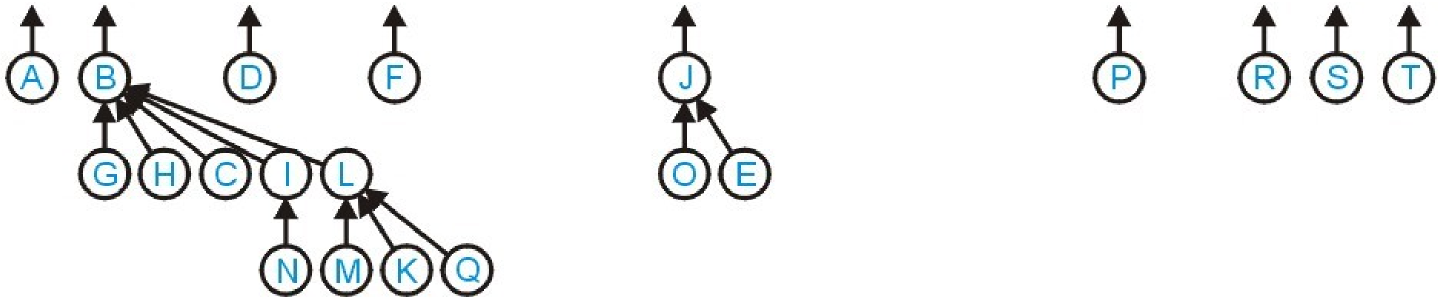


0	1	1	3	9	5	1	1	**1**	9	11	11	11	8	9	15	11	17	18	19

Selecting wall 15 joints the sets identified by B and L

- The tree B has height 2 while L has height 1 and therefore we attach L to B

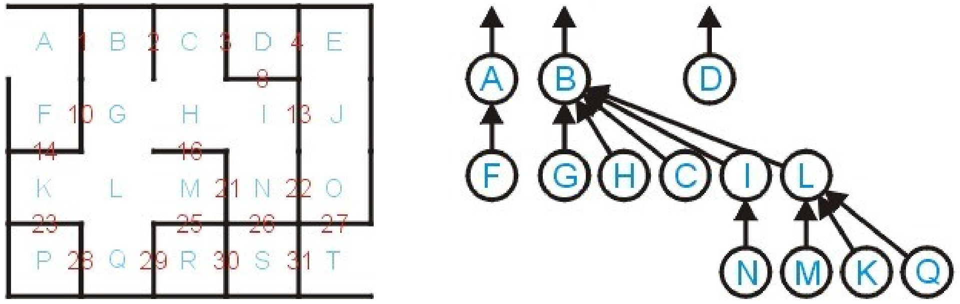




0	1	1	3	9	5	1	1	1	9	11	1	11	8	9	15	11	17	18	19

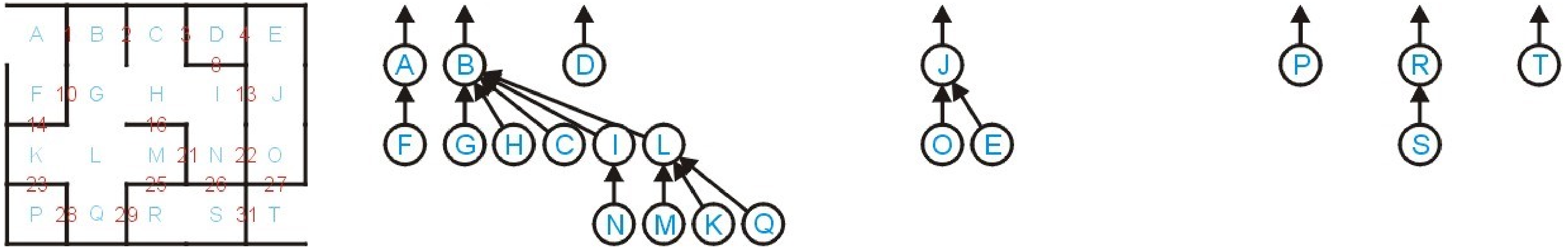
Next we select wall 5 which joins disjoint sets A and F

- Both are height 0



0	1	1	3	9	0	1	1	1	9	11	1	11	8	9	15	11	17	18	19

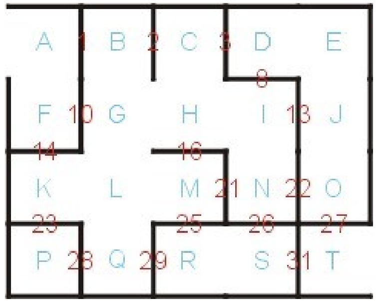
Selecting wall 30 also joins two disjoint sets R and S

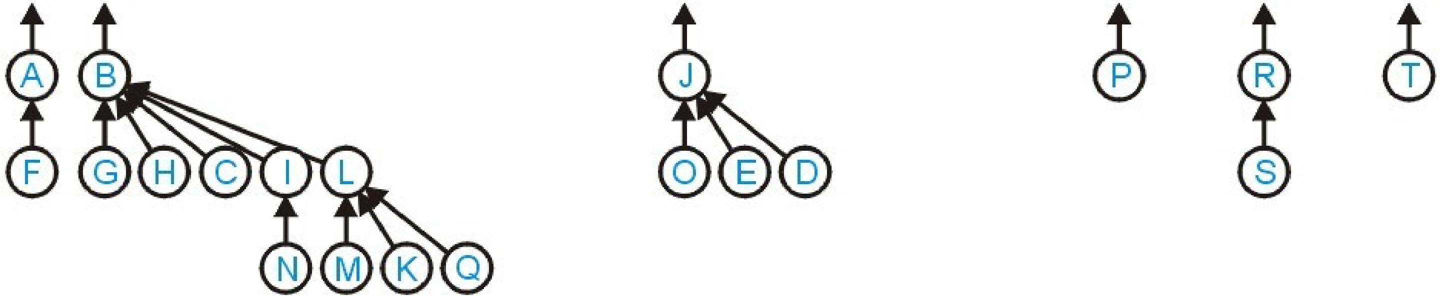


0	1	1	3	9	0	1	1	1	9	11	1	11	8	9	15	11	17	**17**	19

Selecting wall 4 joints the disjoint set D and the disjoint set identified by J

- D has height 0, J has height 1, and thus we add D to J

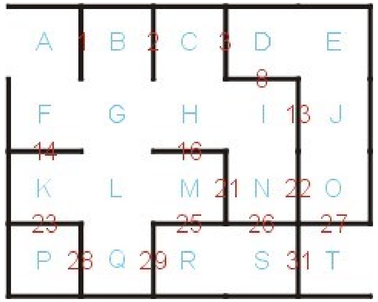


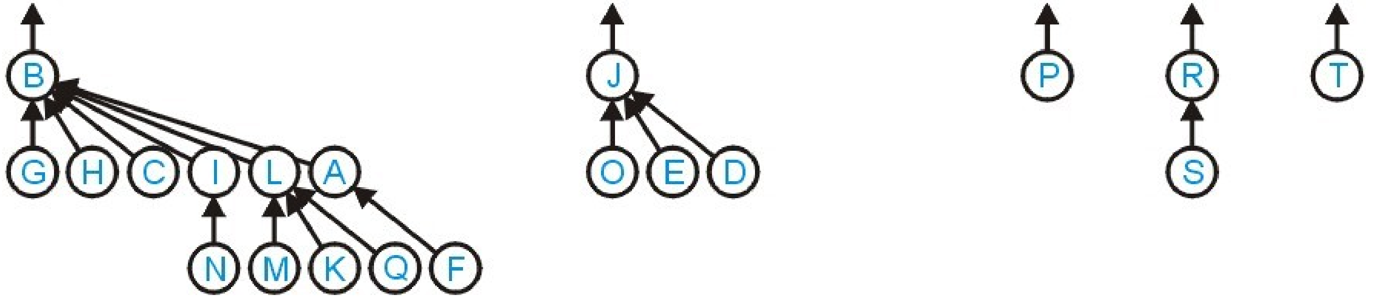


0	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	19

Next we select wall 10 which joins the sets identified by A and B

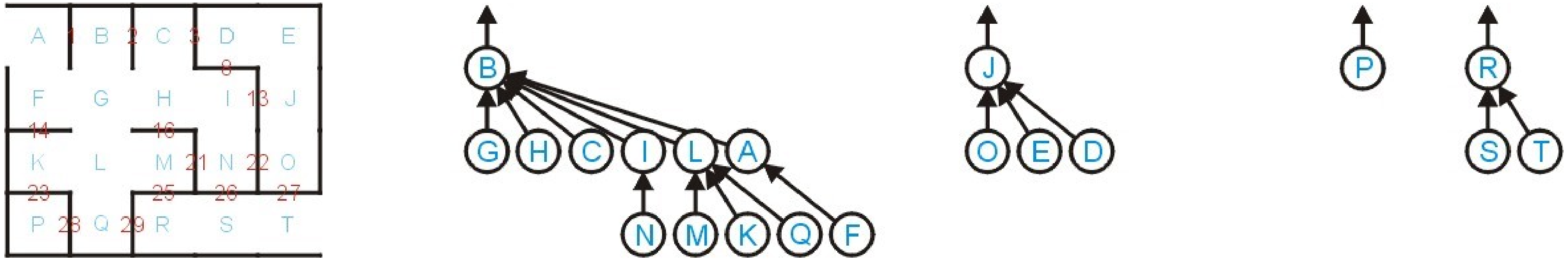
- A has height 1 while B has height 2, so we attach A to B





1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	19

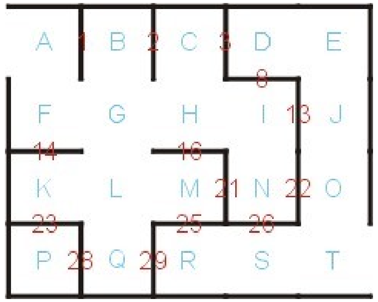
- Selecting wall 31, we union the sets identified by R and T
  - T has height 0 so we attach it to I

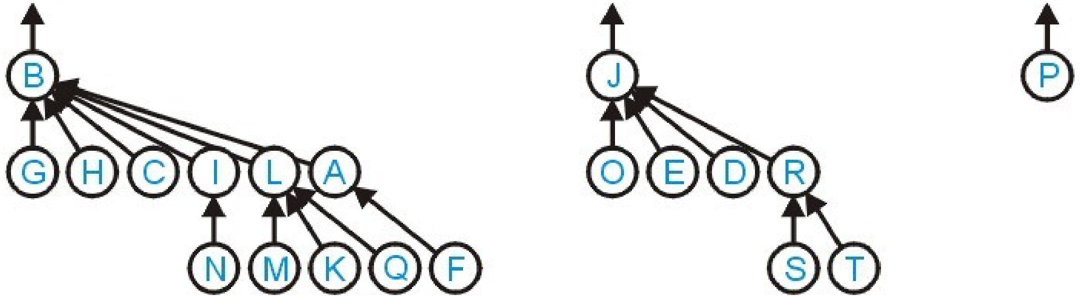


1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	17

Selecting wall 27 joins the disjoint sets identified by J and R

- They both have height 1, but J has more elements, so we add R to J

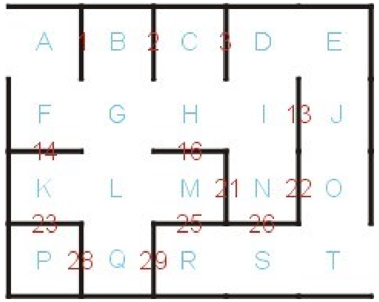


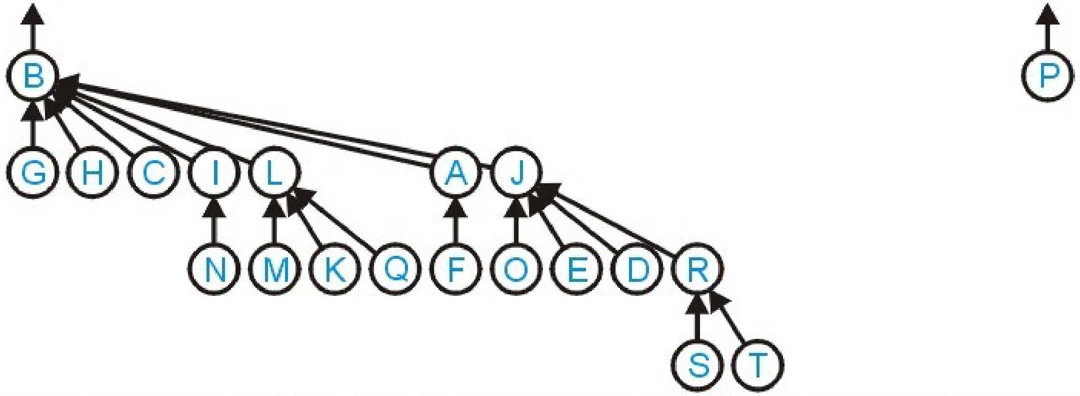


1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	9	17	17

Selecting wall 8 joins sets identified by B and J

- They both have height 2 so we note that J has fewer nodes than B, so we add J to B

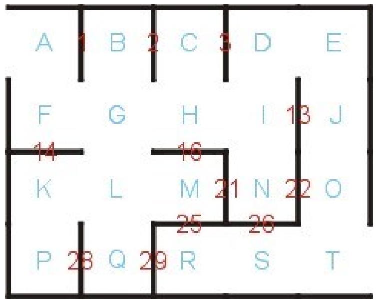


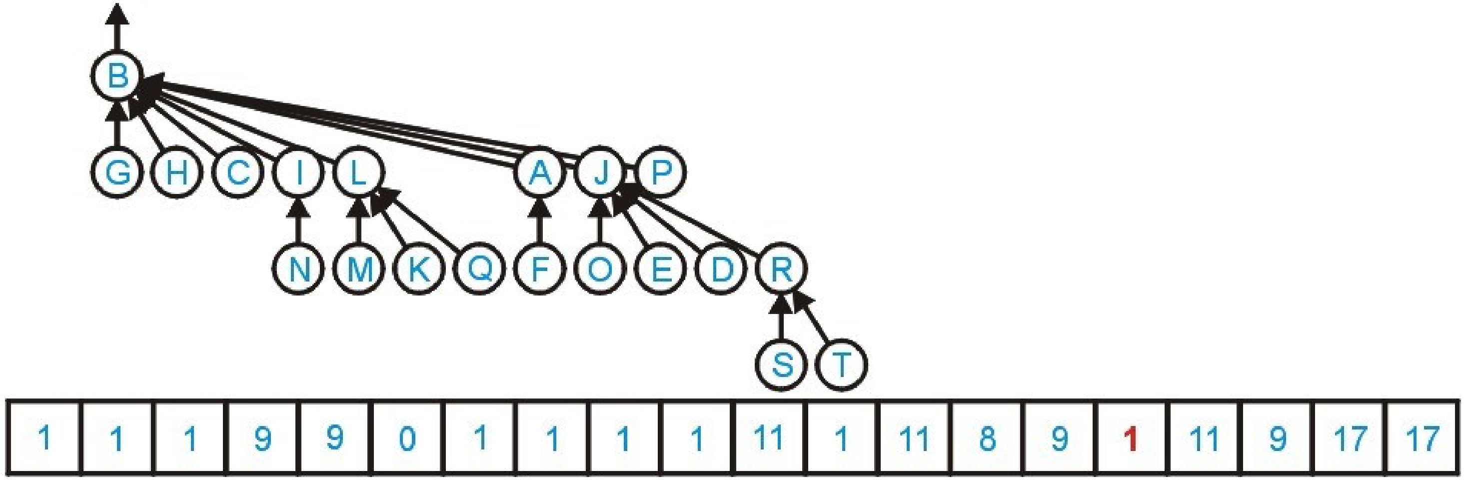


1	1	1	9	9	0	1	1	1	1	11	1	11	8	9	15	11	9	17	17

Finally we select wall 23 which joins the disjoint set P and the disjoint set identified by B

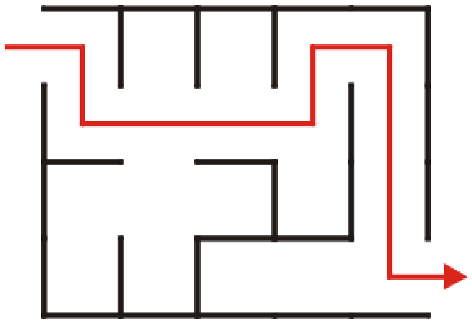
- P has height 0, so we attach it to B





Thus we have a (rather trivial) maze where:

- there is one unique solution, and
- you can reach any square by a unique path from the starting point



References

1. CLRS book Chapter 21
2. GeeksforGeeks. (2023, March 24). Union by rank and path compression in UnionFind algorithm. GeeksforGeeks. https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/
3. Ashraf, S. (2015, November 26). ডাটা স্ট্রাকচার: ডিসজয়েন্ট সেট(ইউনিয়ন ফাইন্ড). শাফায়েতের ব্লগ. https://www.shafaetsplanet.com/?p=763
4. https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

Thank you