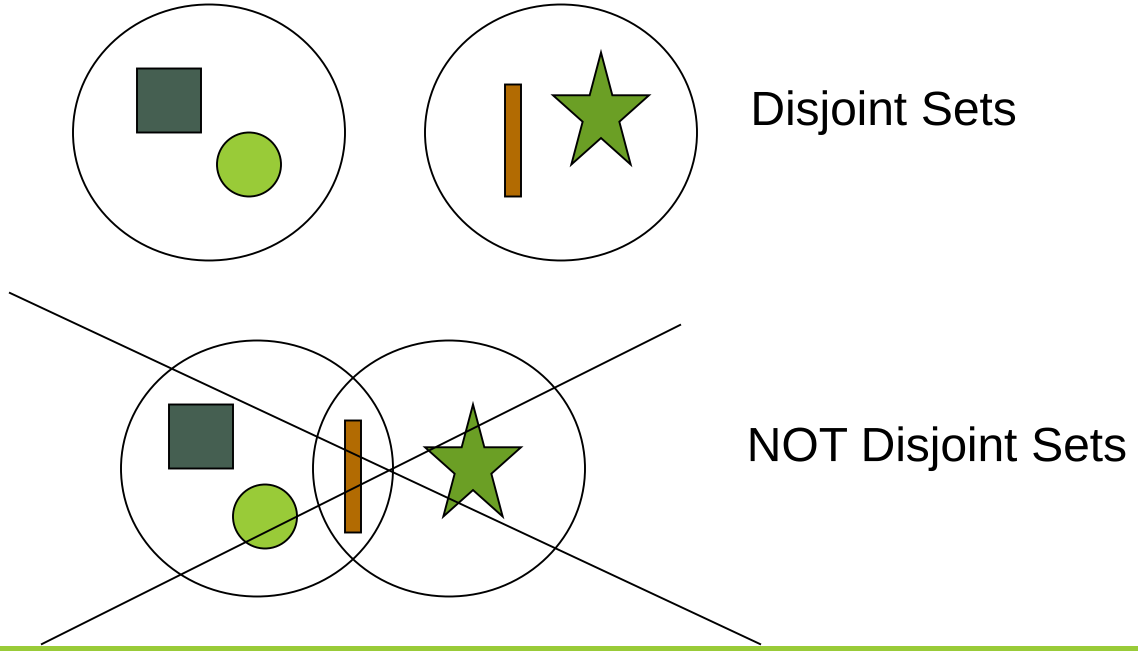
Disjoint sets

What Are Disjoint Sets?

Two sets A and B are disjoint if they have NO elements in common. (A ∩ B = 0)



What Are Disjoint Set Data Structures?

A disjoint-set data structure maintains a collection
$S = \{S1, S2, \ldots, S_k\}$ of disjoint dynamic (changing) sets.

- Each set has a representative (member of the set).
- Each element of a set is represented by an object (x).

Why Do We Need Disjoint Set Data Structures?

To determine the connected components of an undirected graph.

Operations Supported By Disjoint Set Data Structures

MAKE-SET(x): creates a new set with a single member pointed to by x.

UNION(x,y): unites the sets that contain common element(s).

FIND-SET(x): returns a pointer to the representative of the set containing x.

We will determine if two objects are in the same disjoint set by defining a function which finds the representative object of one of the disjoint sets

If the representative objects are the same, the objects are in the same disjoint set

An Application: Determining the Connected Components of an Undirected Graph

CONNECTED-COMPONENTS(*G*) //computes connected components of a graph

1 for each vertex *v* in the set *V[G]*

2 do MAKE-SET(*v*)

3 for each edge (*u*,*v*) in the set *E[G]*

4 do FIND-SET(*u*) $\neq$ FIND-SET(*v*)

5 then UNION(*u*,*v*)

SAME-COMPONENT (*u*,*v*) //determines whether two vertices are in the same connected component

1 if FIND-SET(*u*) = FIND-SET(*v*)

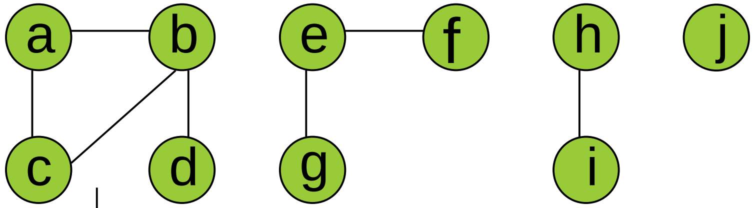
2 then return TRUE

3 else return FALSE

V[G]= set of vertices of a graph G
E[G]=set of edges of a graph G

An Application of Disjoint-Set Data Structures

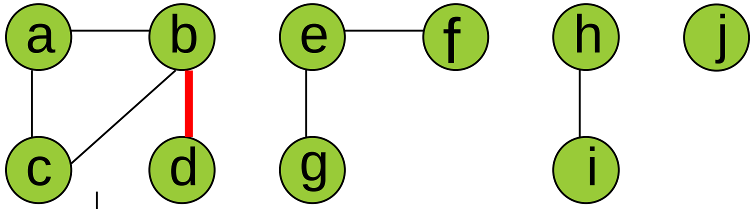
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

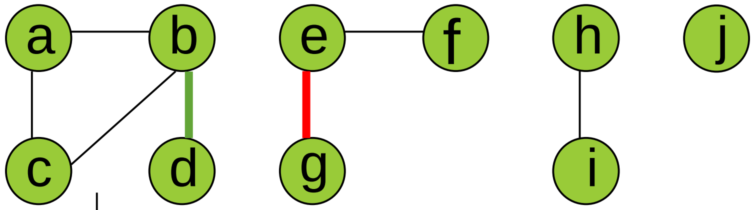
Determining the connected components of an undirected graph G=(V,E)



Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}

An Application of Disjoint-Set Data Structures

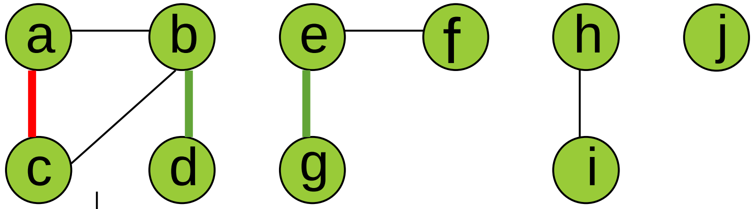
Determining the connected components of an undirected graph G=(V,E)



Initial	{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}

An Application of Disjoint-Set Data Structures

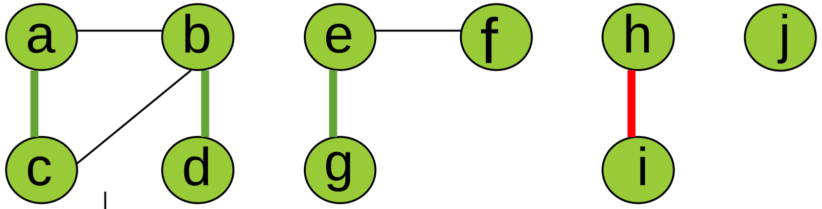
Determining the connected components of an undirected graph G=(V,E)



Initial	{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}

An Application of Disjoint-Set Data Structures

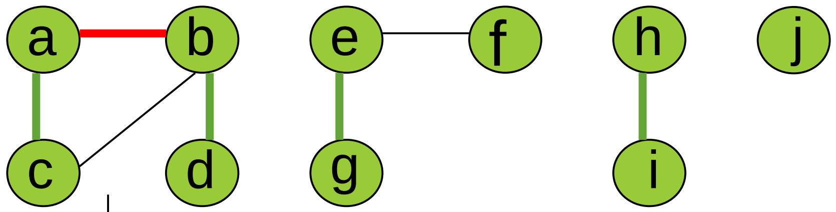
Determining the connected components of an undirected graph G=(V,E)



Initial	{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}

An Application of Disjoint-Set Data Structures

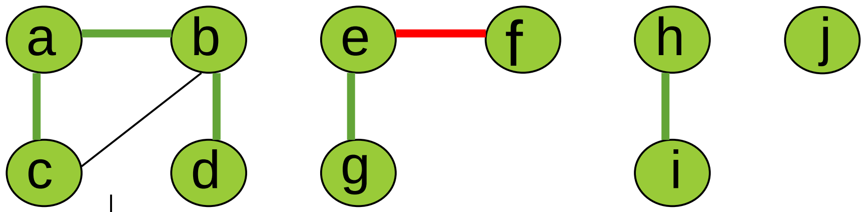
Determining the connected components of an undirected graph G=(V,E)



Initial	{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}

An Application of Disjoint-Set Data Structures

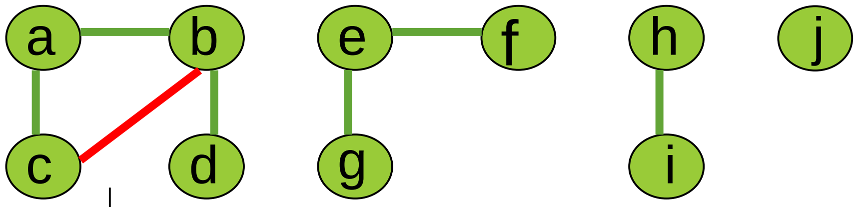
Determining the connected components of an undirected graph G=(V,E)



Initial	{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}

An Application of Disjoint-Set Data Structures

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

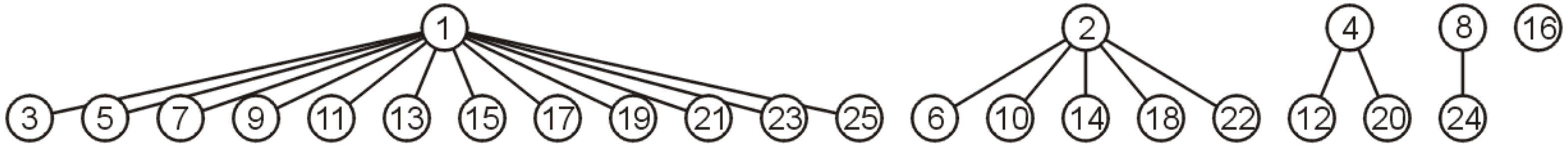


Initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}
(e, f)	{a, b, c, d}				{e, f, g}			{h, i}		{j}
(b, c)	{a, b, c, d}				{e, f, g}			{h, i}		{j}

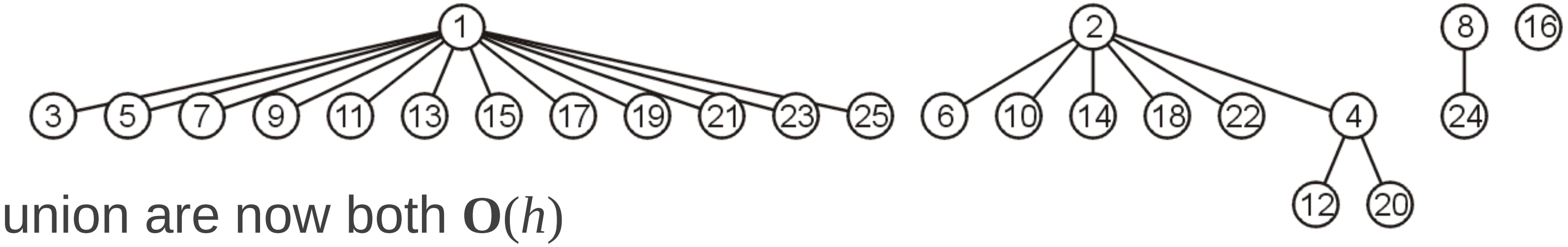
Implementation

let each disjoint set be represented by a general tree

- The root of the tree is the representative object



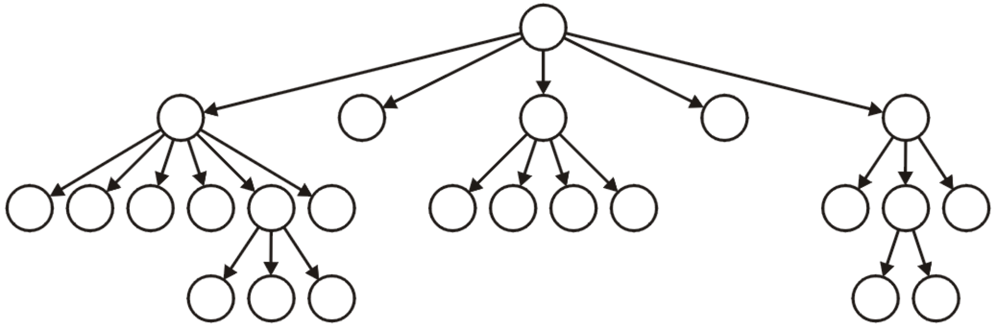
To take the union of two such sets, we will simply attach one tree to the root of the other



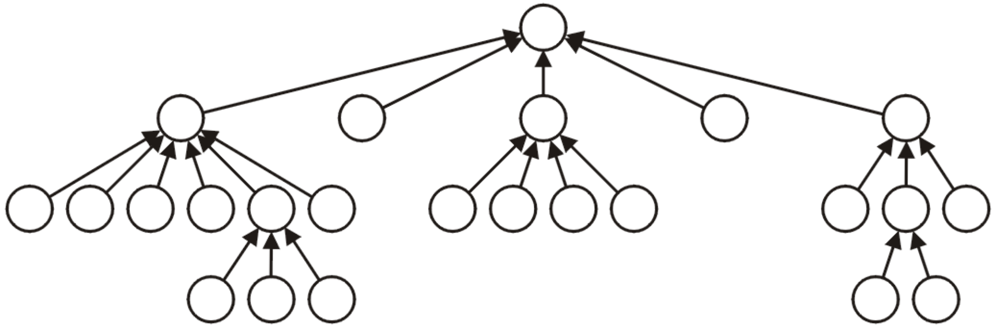
Find and union are now both $\mathbf{O}(h)$

Implementation

Normally, a node points to its children:



We are only interested in the root; therefore, our interest is in storing the parent



Implementation

For simplicity, we will assume we are creating disjoint sets the n integers

$$0, 1, 2, \ldots, n - 1$$

We will define an array

```
parent = new size_t[n];

for ( int i = 0; i < n; ++i ) {
    parent[i] = i;
}
```

If **`parent[i] == i`**, then **`i`** is a root node

- Initially, each integer is in its own set

Implementation

This function is also easy to define:

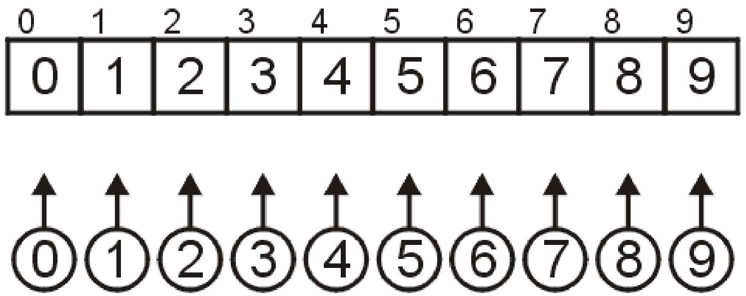
```
void set_union( size_t i, size_t j ) {
    i = find( i );
    j = find( j );

    if ( i != j ) {
        // slightly sub-optimal...
        parent[j] = i;
    }
}
```

$$T_{set\_union}(n) = 2T_{find}(n) + \mathbf{\Theta}(1)$$
$$= \mathbf{O}(h)$$

The keyword `union` is reserved in C++

Example

Consider the following disjoint set on the ten decimal digits:



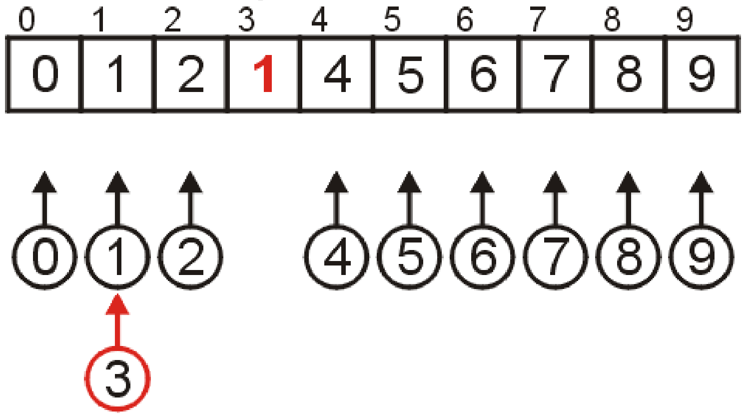
$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}$

Example

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

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

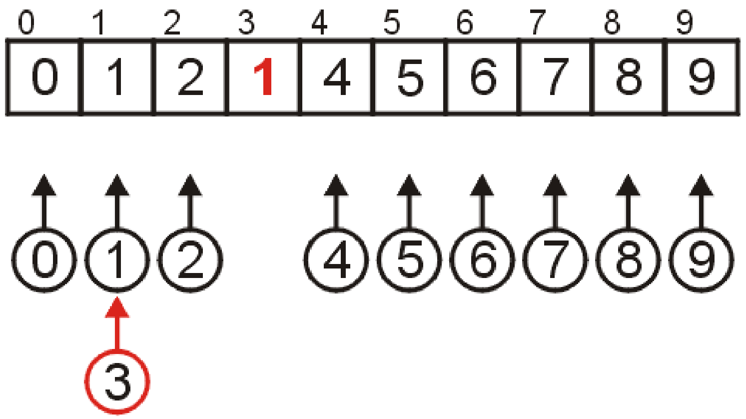
we perform a find on both entries and update the second



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

Example

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



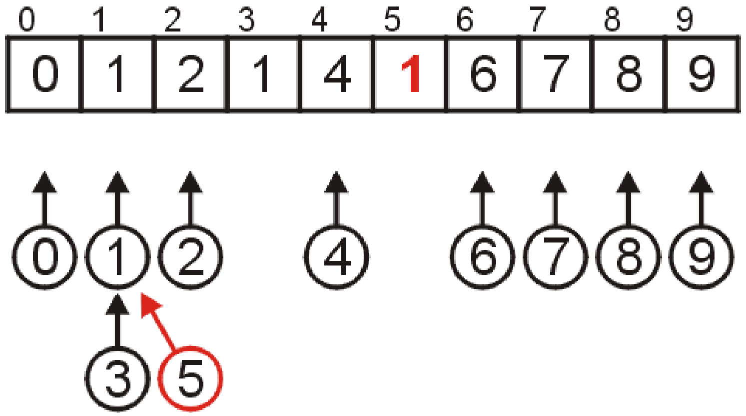
{0}, {1, 3}, {2}, {4}, {5}, {6}, {7}, {8}, {9}

Example

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

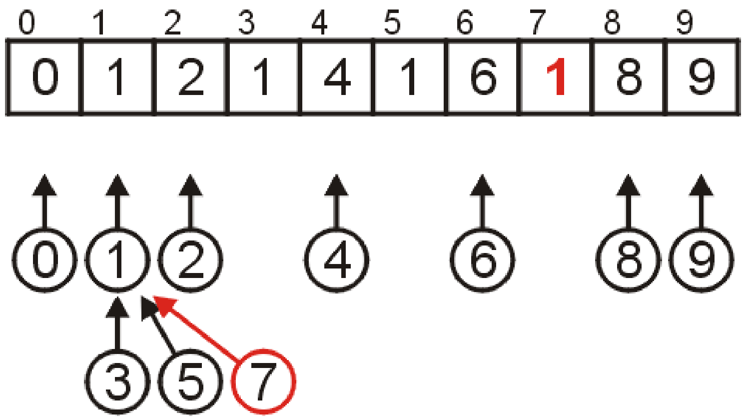


$\{0\}, \{1, 3, 5\}, \{2\}, \{4\}, \{6\}, \{7\}, \{8\}, \{9\}$

Example

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)`:



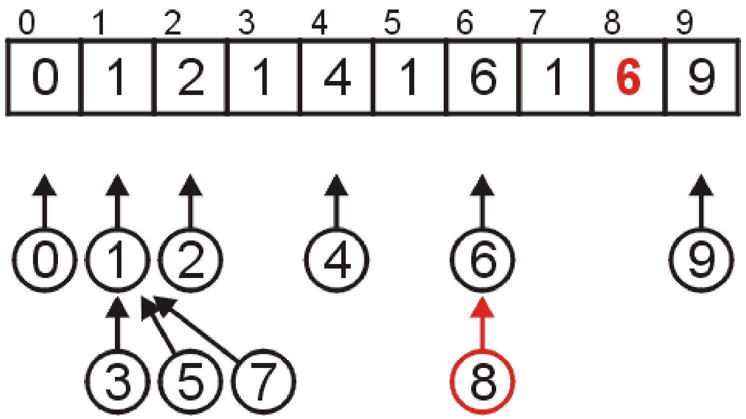
{0}, {1, 3, 5, 7}, {2}, {4}, {6}, {8}, {9}

Example

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)`:



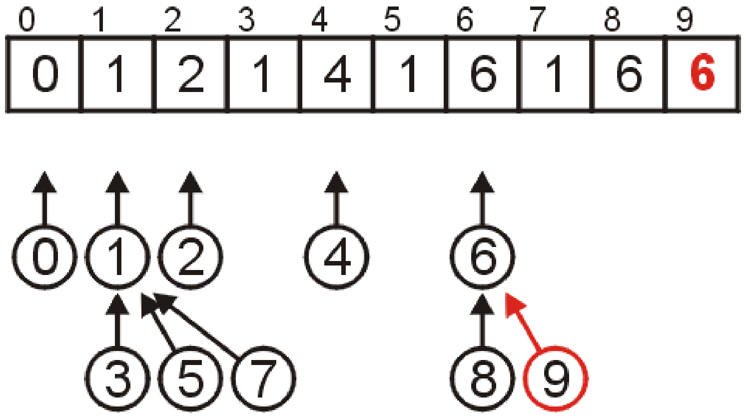
$\{0\}, \{1, 3, 5, 7\}, \{2\}, \{4\}, \{6, 8\}, \{9\}$

Example

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)`:



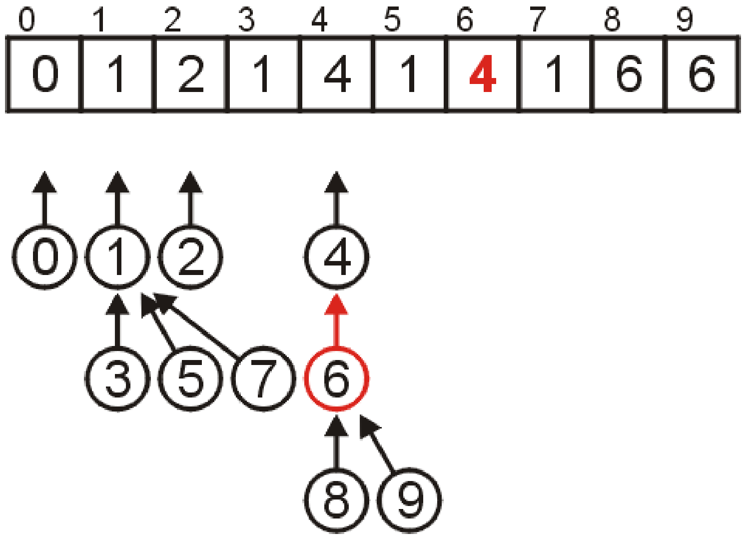
$\{0\}, \{1, 3, 5, 7\}, \{2\}, \{4\}, \{6, 8, 9\}$

Example

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)`:

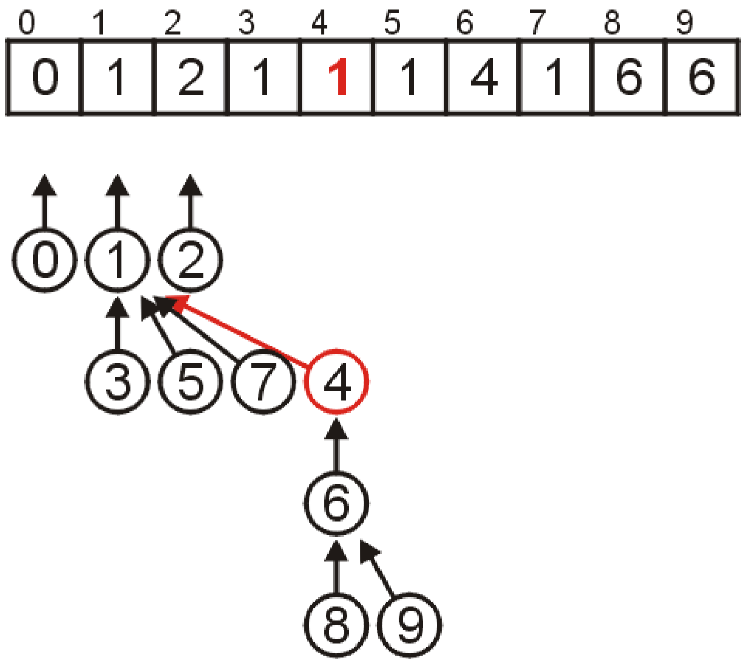


$\{0\}, \{1, 3, 5, 7\}, \{2\}, \{4, 6, 8, 9\}$

Example

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)`:



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

Optimizations

whenever find is called, update the object to point to the root

```
size_t Disjoint_set::find( size_t n ) {
    if ( parent[n] == n ) {
        return n;
    } else {
        parent[n] = find( parent[n] );
        return parent[n];
    }
}
```

Application: Image Processing

One common application is in image processing

Suppose you are attempting to recognize similar features within an image

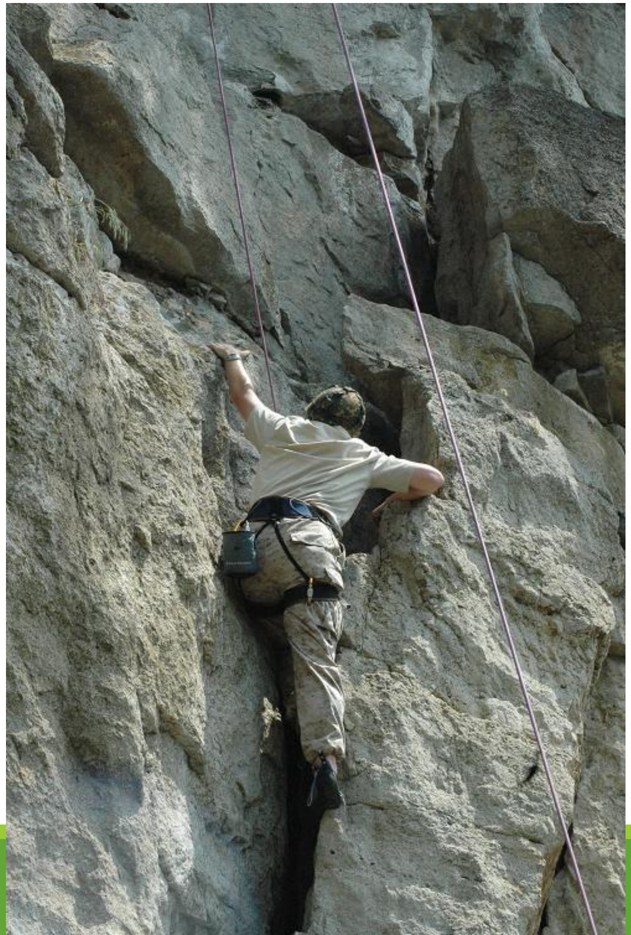
Within a photograph, the same object may be separated by an obstruction; e.g., a road may be split by

- a telephone pole in an image
- an overpass on an aerial photograph

Application: Image Processing

Consider the following image of a man climbing up the Niagara Escarpment at Rattlesnake Point

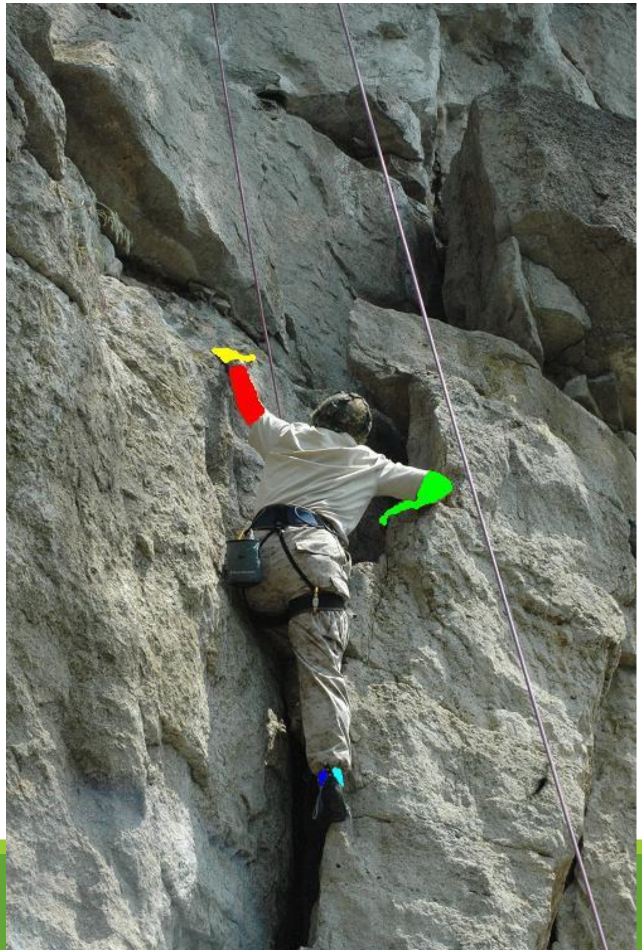
Suppose we have a program which recognizes skin tones



Application: Image Processing

A first algorithm may make an initial pass and recognize five different regions which are recognized as exposed skin

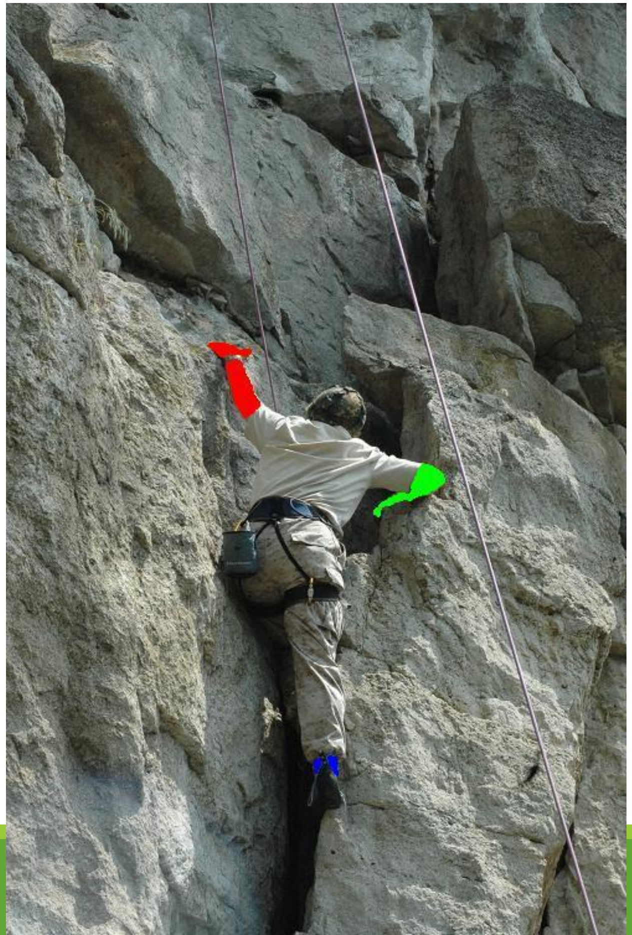
- the left arm and hand are separated by a watch



Each region would be represented by a separate disjoint set

Application: Image Processing

Next, a second algorithm may take sets which are close in proximity and attempt to determine if they are from the same person



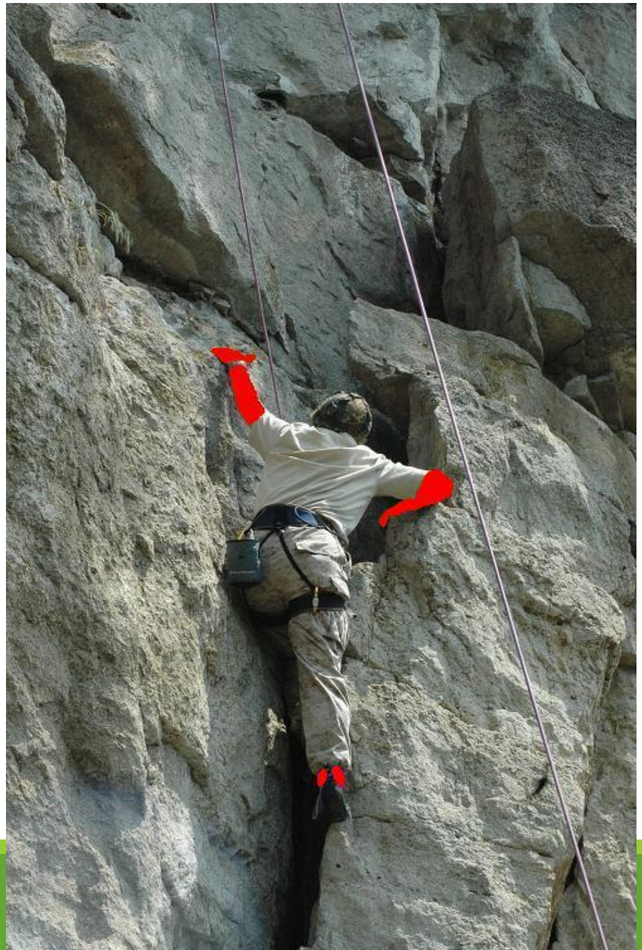
In this case, the algorithm takes the union of:

- the red and yellow regions, and
- the dark and light blue regions

Application: Image Processing

Finally, a third algorithm may take more distant sets and, depending on skin tone and other properties, may determine that they come from the same individual

In this example, the third pass may, if successful, take the union of the red, blue, and green regions



Application: Maze Generation

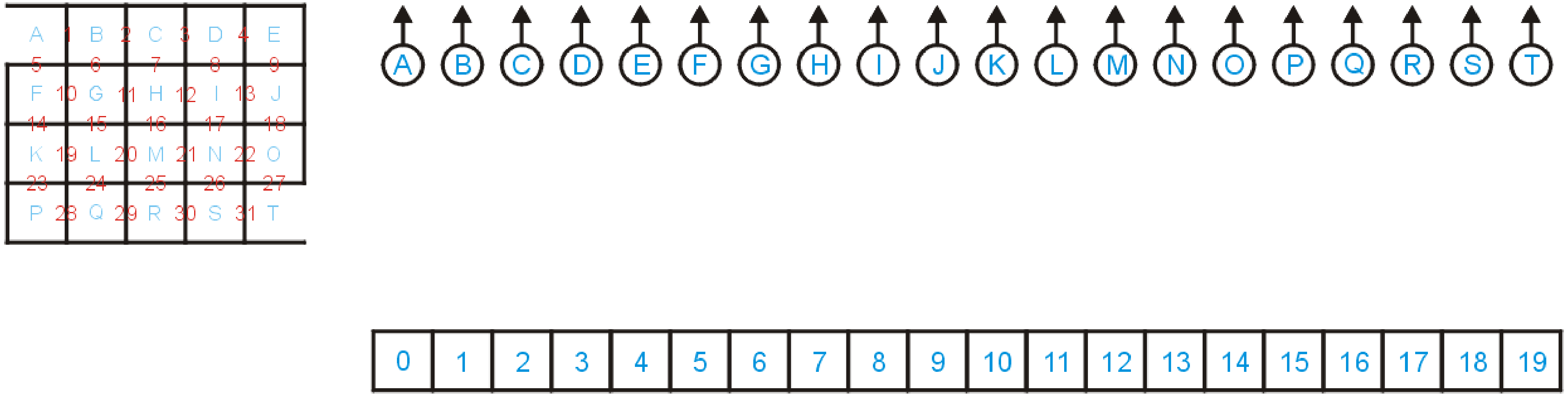
What we will do is the following:

- 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

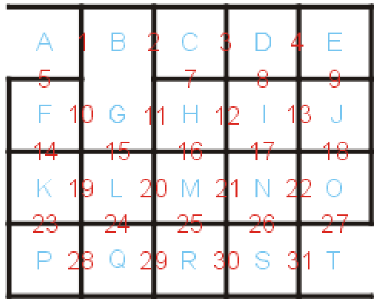
Let us begin with an entrance, an exit, and a disjoint set of 20 squares and 31 interior walls

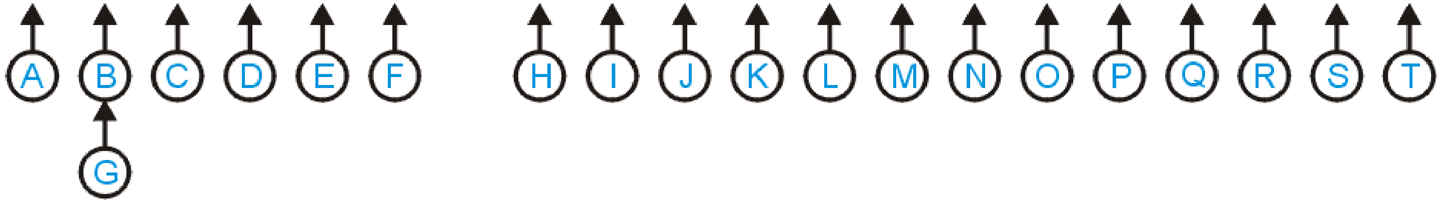


Application: Maze Generation

First, we select 6 which joins cells B and G

- Both have height 0

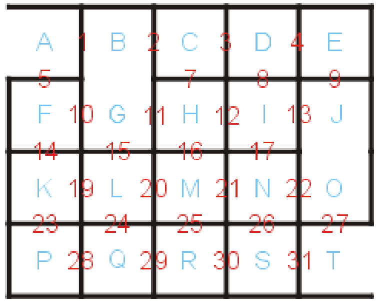


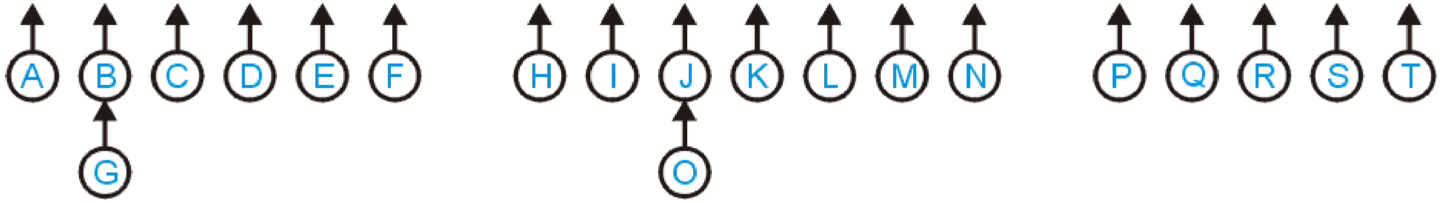


0	1	2	3	4	5	1	7	8	9	10	11	12	13	14	15	16	17	18	19

Application: Maze Generation

Next we select wall 18 which joins regions J and O



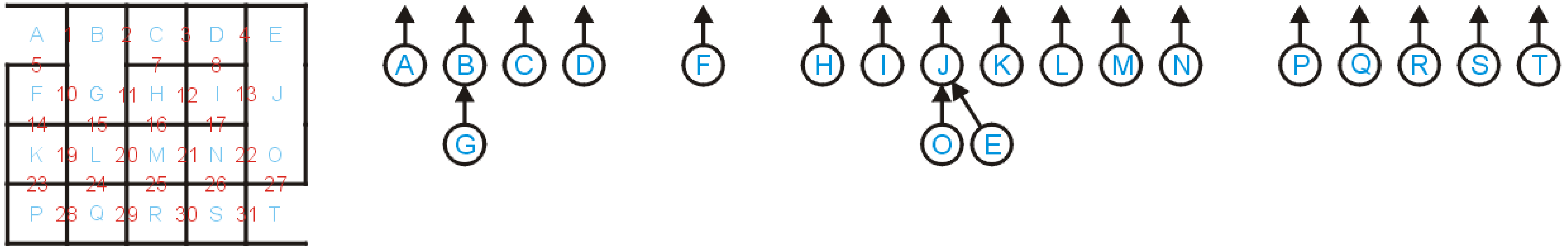


0	1	2	3	4	5	1	7	8	9	10	11	12	13	9	15	16	17	18	19

Application: Maze Generation

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

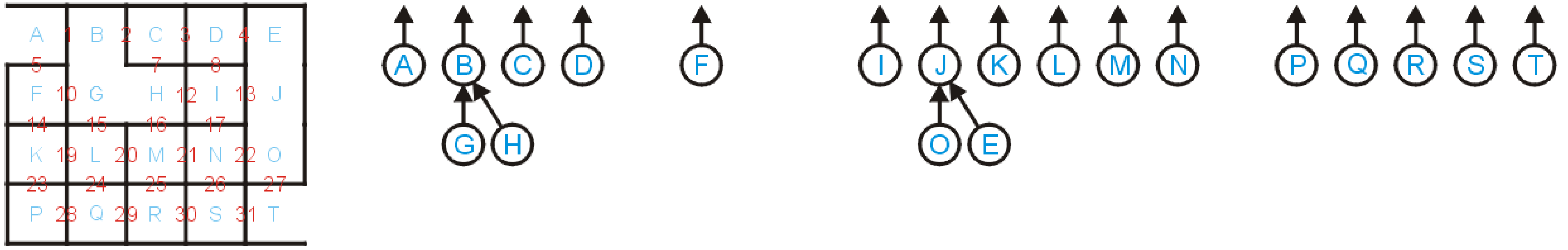


0	1	2	3	9	5	1	7	8	9	10	11	12	13	9	15	16	17	18	19

Application: Maze Generation

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

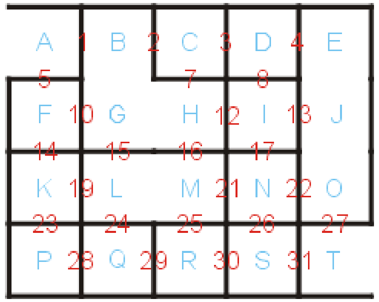


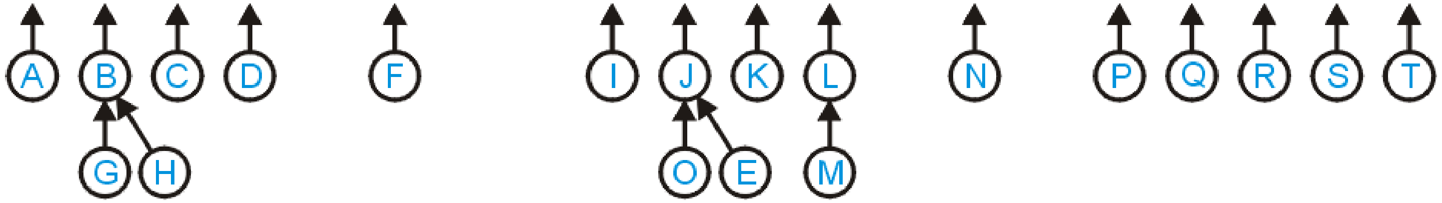
0	1	2	3	9	5	1	1	8	9	10	11	12	13	9	15	16	17	18	19

Application: Maze Generation

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

- Both are height 0



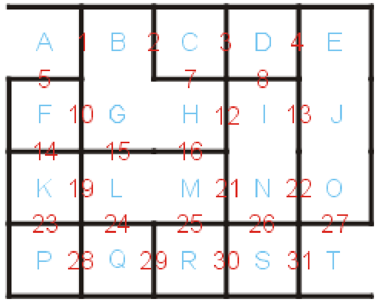


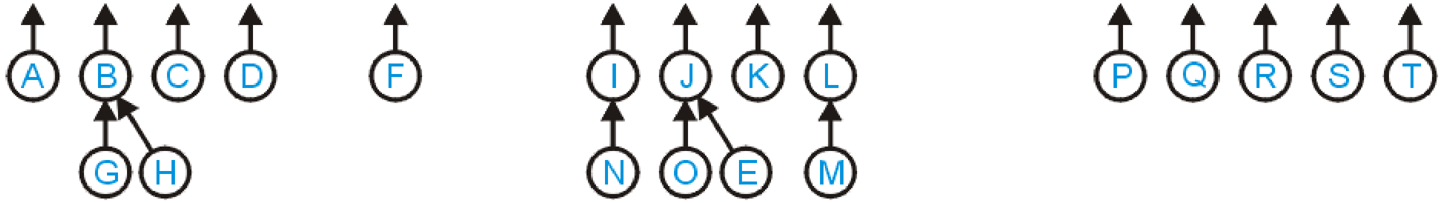
0	1	2	3	9	5	1	1	8	9	10	11	**11**	13	9	15	16	17	18	19

Application: Maze Generation

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

- Both are height 0



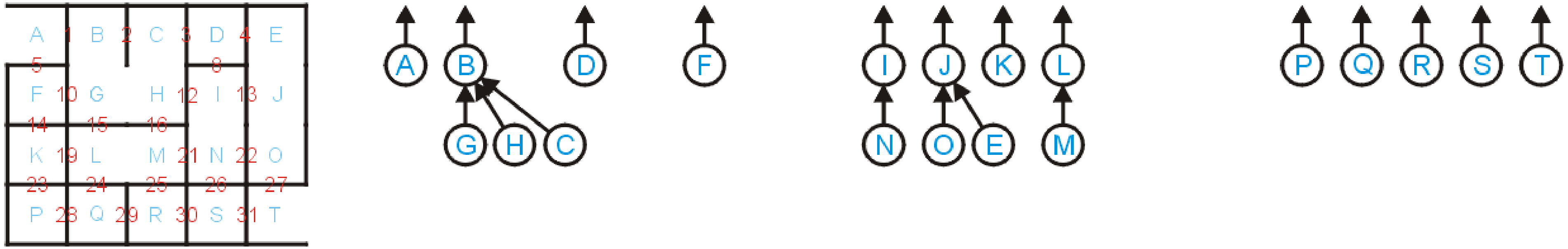


0	1	2	3	9	5	1	1	8	9	10	11	11	8	9	15	16	17	18	19

Application: Maze Generation

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

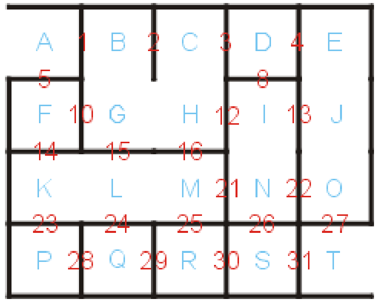


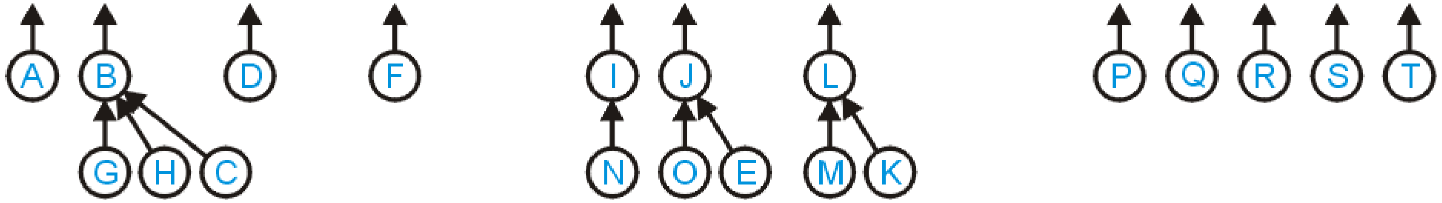
0	1	**1**	3	9	5	1	1	8	9	10	11	11	8	9	15	16	17	18	19

Application: Maze Generation

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



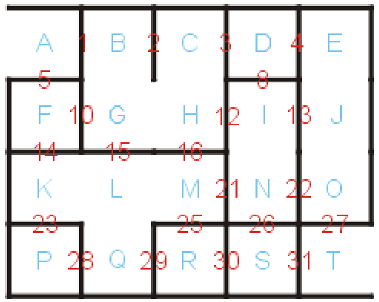


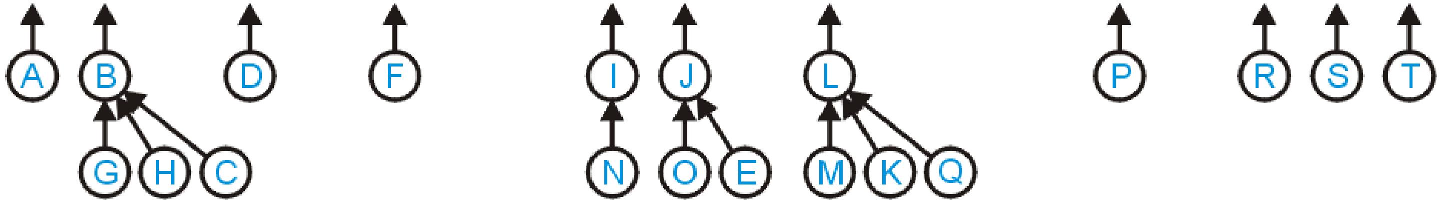
0	1	1	3	9	5	1	1	8	9	11	11	11	8	9	15	16	17	18	19

Application: Maze Generation

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



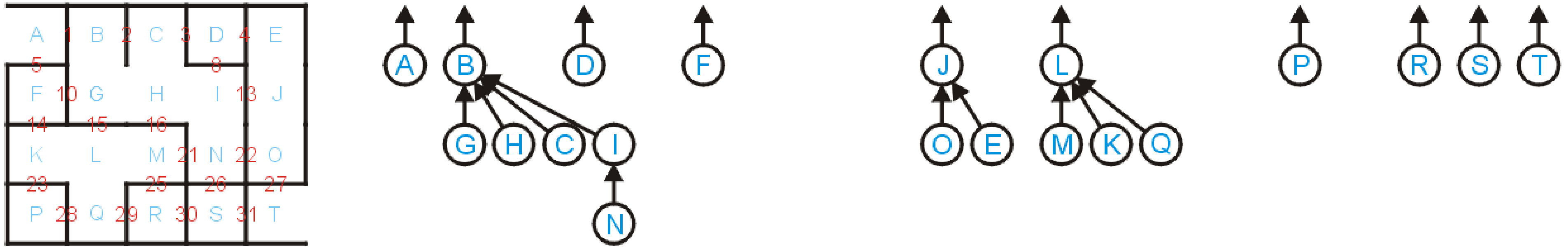


0	1	1	3	9	5	1	1	8	9	11	11	11	8	9	15	**11**	17	18	19

Application: Maze Generation

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

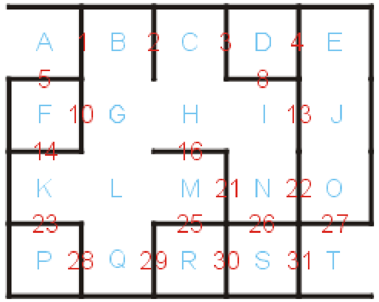


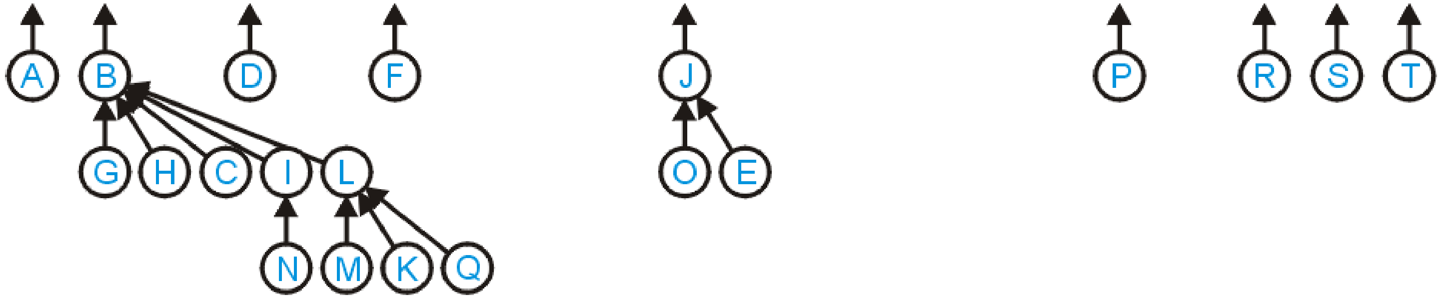
0	1	1	3	9	5	1	1	1	9	11	11	11	8	9	15	11	17	18	19

Application: Maze Generation

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



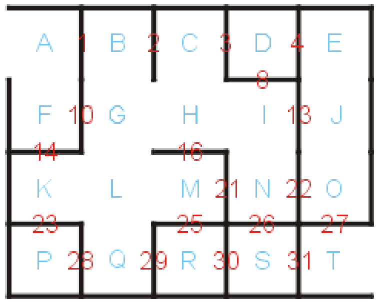


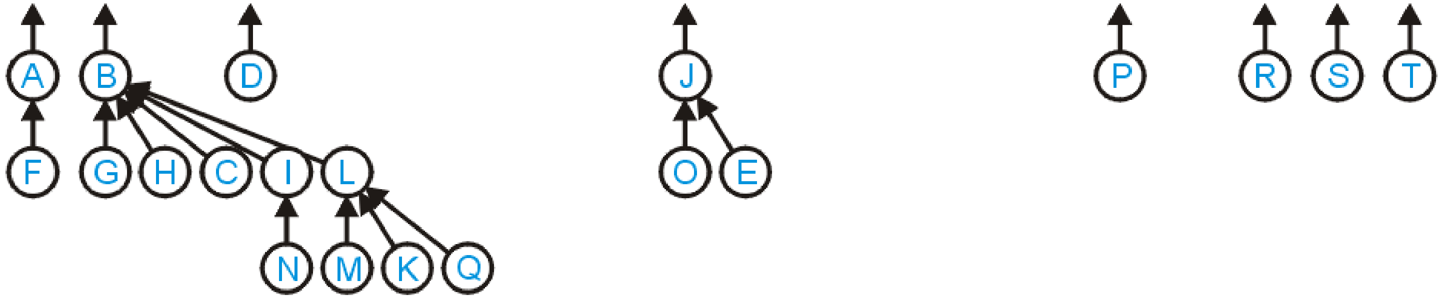
0	1	1	3	9	5	1	1	1	9	11	1	11	8	9	15	11	17	18	19

Application: Maze Generation

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

- Both are height 0

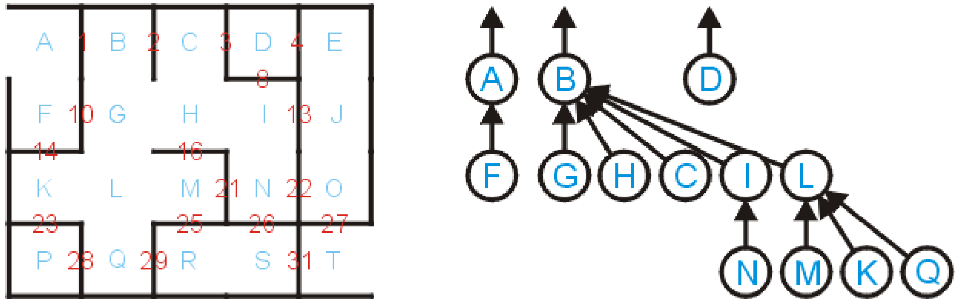




0	1	1	3	9	**0**	1	1	1	9	11	1	11	8	9	15	11	17	18	19

Application: Maze Generation

Selecting wall 30 also joins two disjoint sets R and S

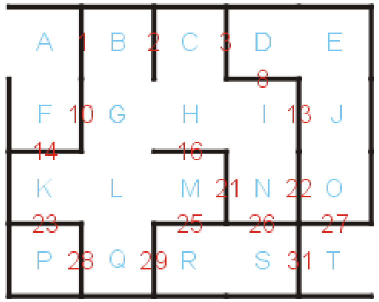


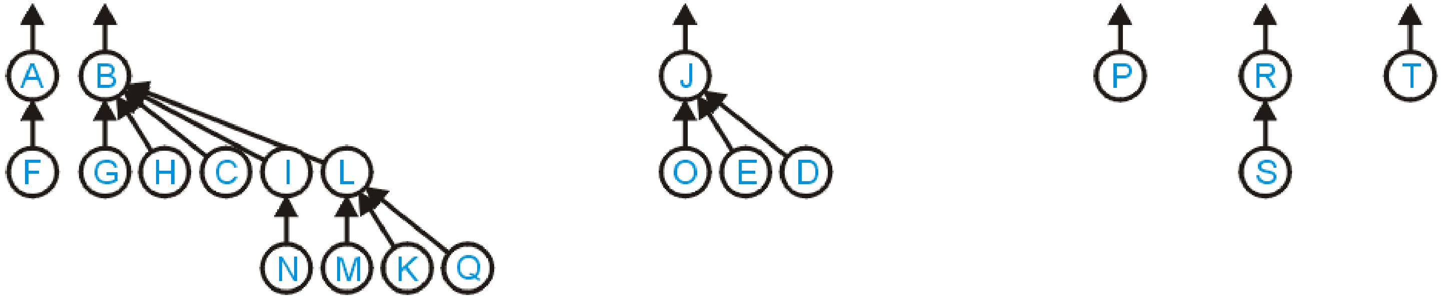
0	1	1	3	9	0	1	1	1	9	11	1	11	8	9	15	11	17	**17**	19

Application: Maze Generation

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



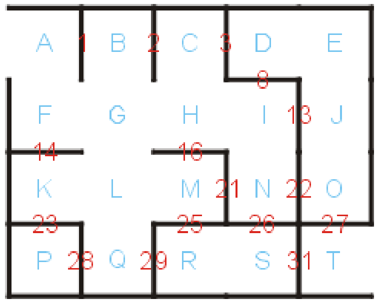


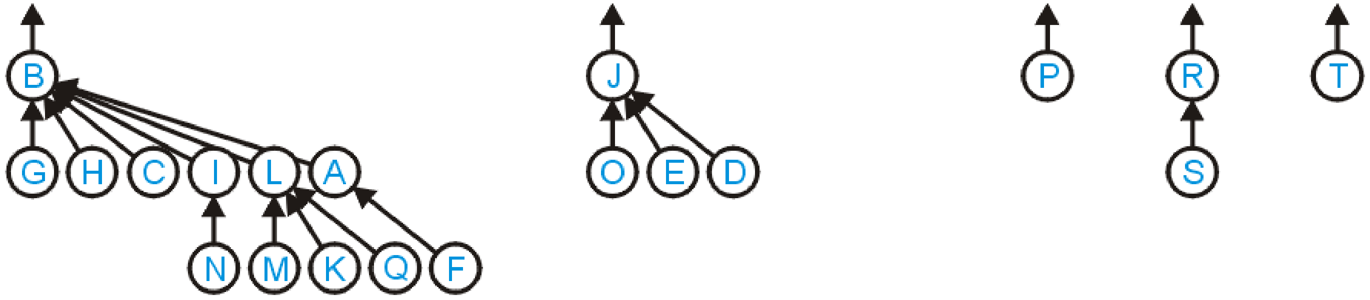
0	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	19

Application: Maze Generation

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



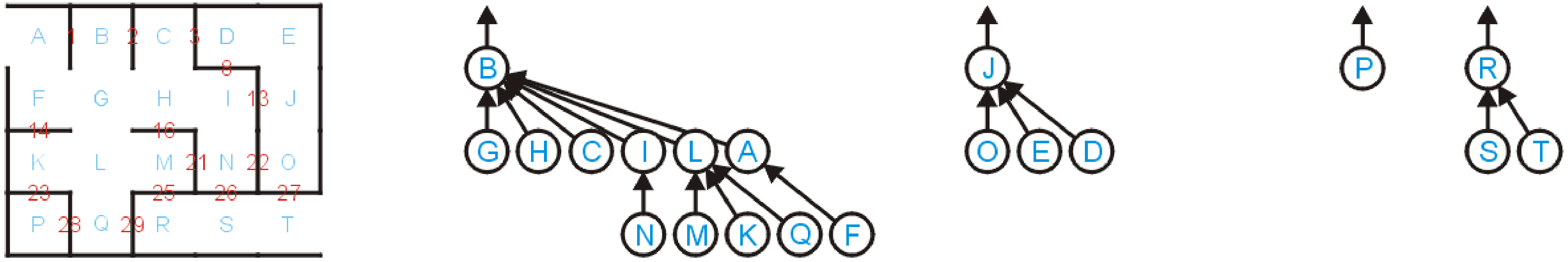


1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	19

Application: Maze Generation

Selecting wall 31, we union the sets identified by R and T

- T has height 0 so we attach it to I

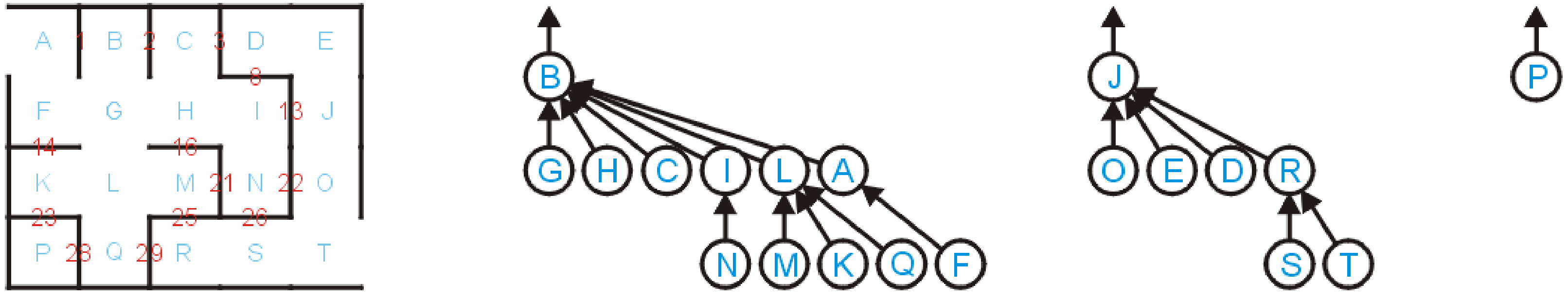


1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	17	17	**17**

Application: Maze Generation

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

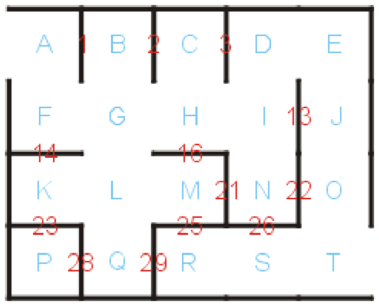


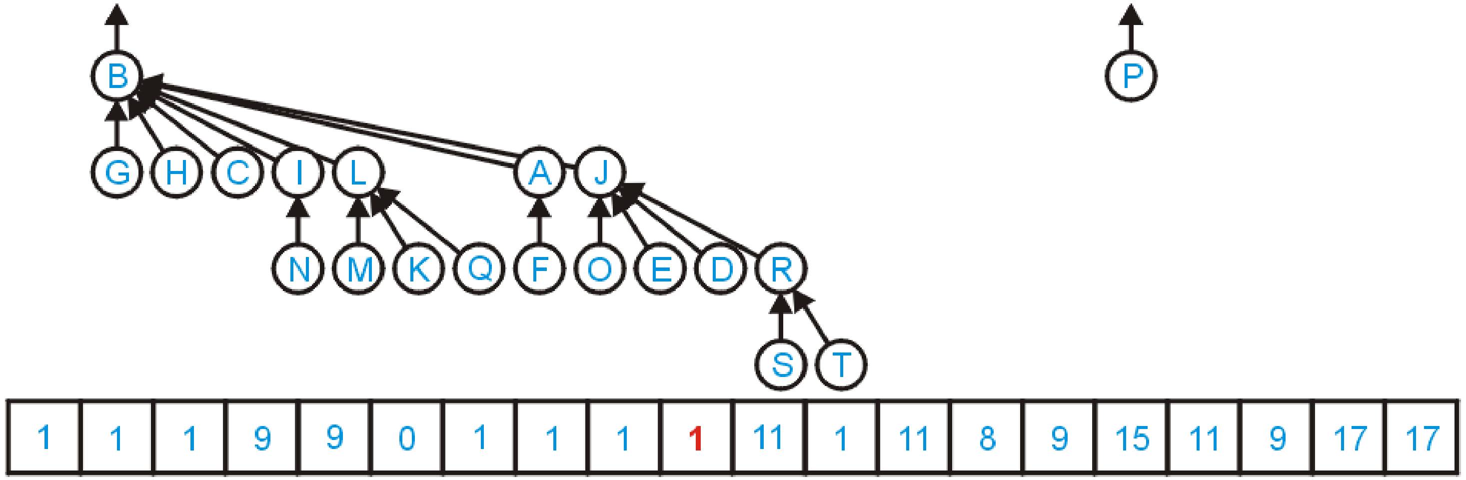
1	1	1	9	9	0	1	1	1	9	11	1	11	8	9	15	11	**9**	17	17

Application: Maze Generation

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

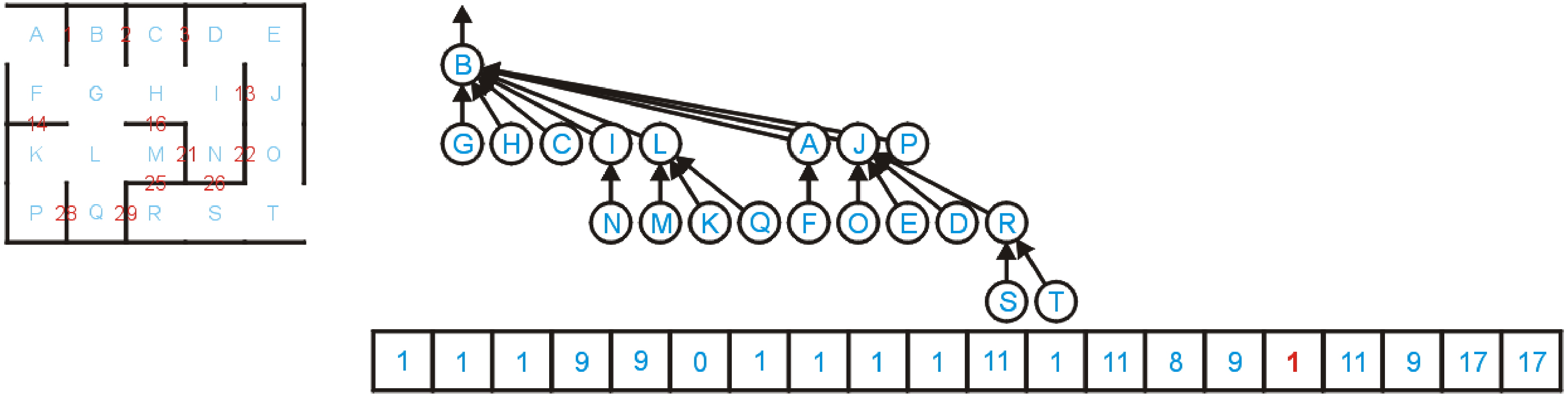




Application: Maze Generation

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



Application: Maze Generation

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