CS101 Algorithms and Data Structures

Disjoint Sets

Textbook Ch 21

Disjoint Sets

- Definition: a set of elements partitioned into a number of disjoint subsets

  For example, a partition of the 10 numerals

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

  into three disjoint subsets

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

- Also called:
  - union–find data structure
  - merge–find set

Operations on Disjoint Sets

There are two operations we would like to perform on disjoint sets:

- Determine if two elements are in the same disjoint set, and
- Take the union of two disjoint sets creating a single set

We will determine if two objects are in the same disjoint set by defining a **`find`** function

- **`find(a)`**: find the representative object of the disjoint set that **`a`** belongs to
- Given two elements **`a`** and **`b`**, they are in the same set if

```
find( a ) == find( b )
```

Implementation

What **`find`** returns is irrelevant so long as:

- If **`a`** and **`b`** are in the same set, **`find( a ) == find( b )`**
- If **`a`** and **`b`** are not in the same set, **`find( a ) != find( b )`**

Here we assume **`find`** returns an integer

Implementation

Here is a poor implementation:

- Have two arrays and the second array stores the representative objects

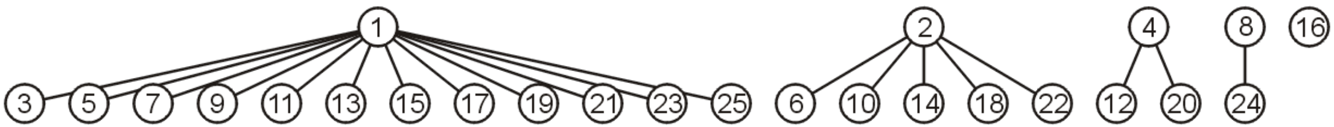
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16	1	2	1	4	1	2	1	8	1

- Given the index of an element, finding the representative object is $\Theta(1)$
- However, taking the union of two sets is $\Theta(n)$
  - It would be necessary to check each array entry

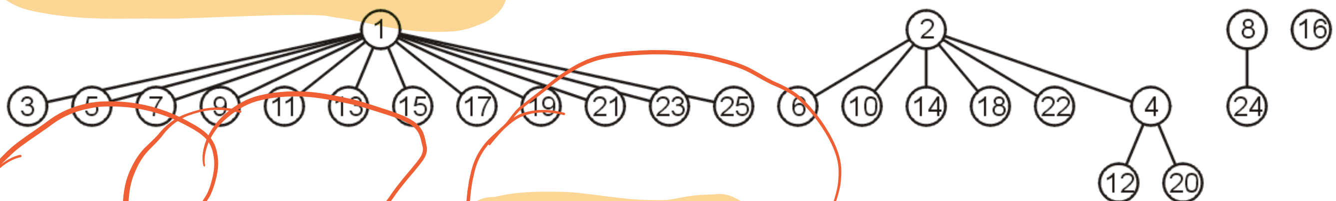
Implementation

As an alternate implementation, let each disjoint set be represented by a general tree

– The root of the tree is the representative object



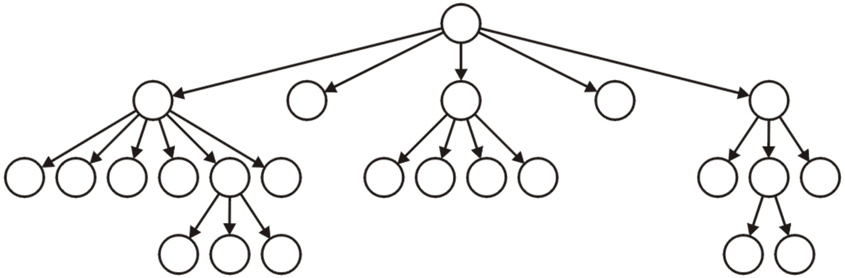
To take the union of two such sets, we will simply attach one tree to the root of the other



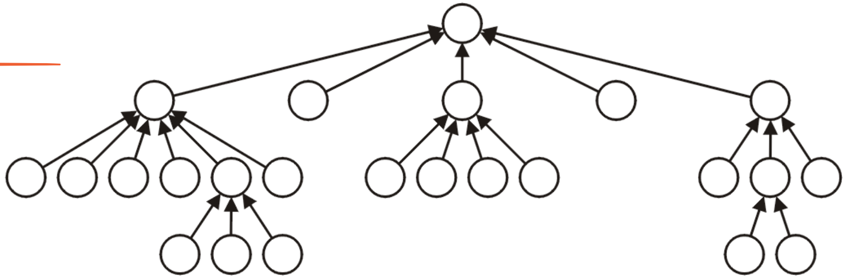
Find and union are now both $\mathbf{O}(h)$

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, assume we are creating disjoint sets for the n integers

$$\underbrace{0, 1, 2, \ldots, n-1}$$

We will define an array

```
parent = new int[n];
```

n

rood 判

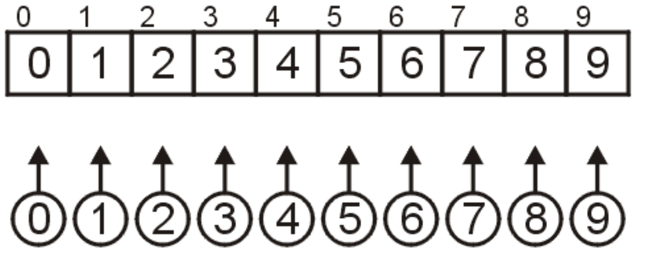
If **`parent[i] == i`**, then **`i`** is a root node

Initially, each integer is in its own set

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

Example

Consider the following disjoint set on the ten decimal digits:



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

Implementation

- `find( int i )`
  - Find the root element of the tree that contains `i`
- `set_union( int i, int j )`
  - Find the root elements of `i` and `j`
  - Update the parent of one root element to be the other root element

Implementation

We will define the function

```
size_t Disjoint_set::find( size_t i ) const {
    while( parent[i] != i ) {
        i = parent[i];
    }

    return i;
}
```

return parent[i]

$$T_{find}(n) = \mathbf{O}(h)$$

Implementation

Initially, you will note that

```
find( i ) != find( j )
```

for `i != j`, and therefore, we begin with each integer being in its own set

We must next look at the *union* operation

- how to join two disjoint sets into a single 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;
    }
}
```

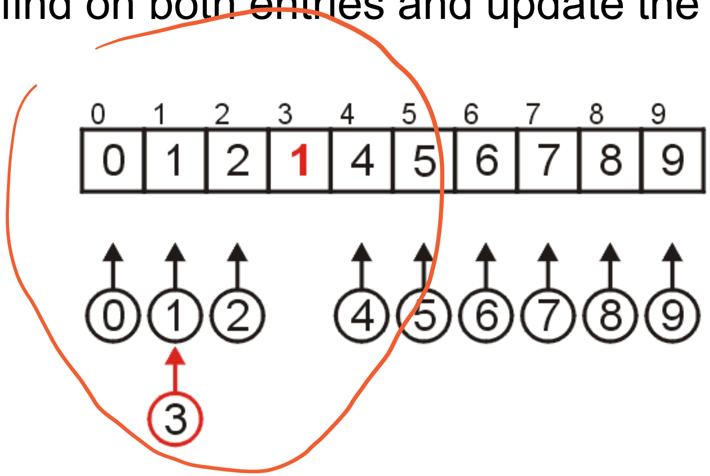
$$T_{set\_union}(n) = 2T_{find}(n) + \mathbf{\Theta}(1)$$
$$= \mathbf{O}(h)$$

Example

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

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

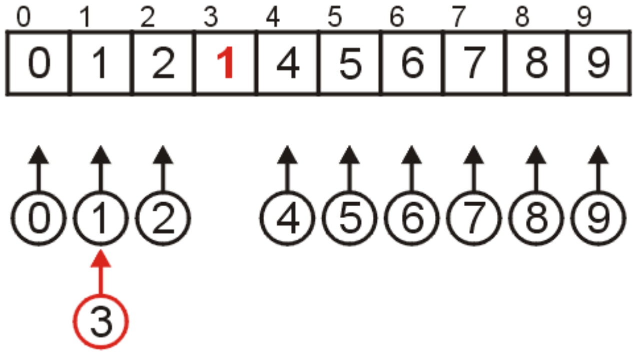
we perform a find on both entries and update the second



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

Example

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



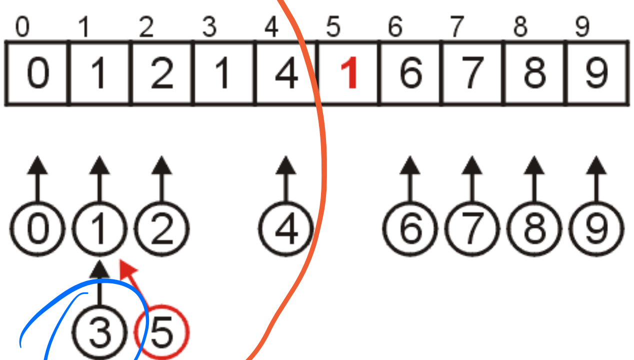
{0}, {1, 3}, {2}, {4}, {5}, {6}, {7}, {8}, {9}

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

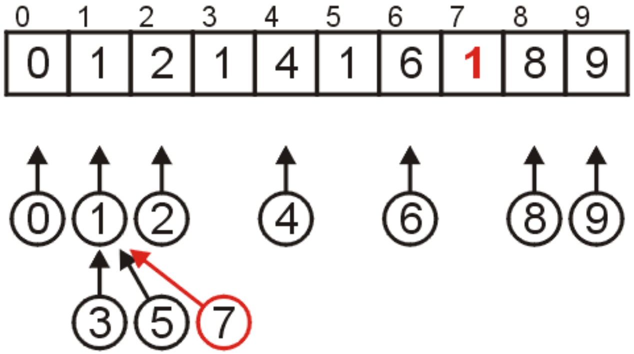


{0}, {1, 3, 5}, {2}, {4}, {6}, {7}, {8}, {9}

Example

Now, if we take the union of the sets containing 5 and 7

```
set_union(5, 7);
```

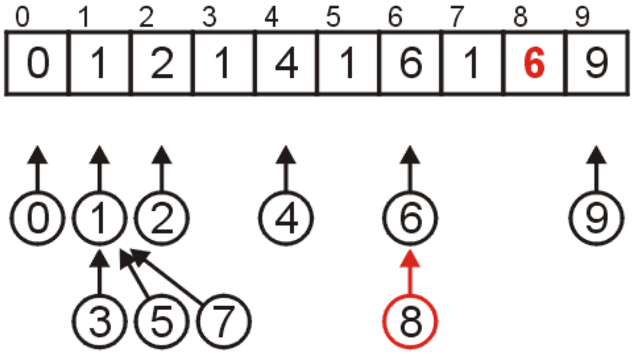


$\{0\}, \{1, 3, 5, 7\}, \{2\}, \{4\}, \{6\}, \{8\}, \{9\}$

Example

Taking the union of the sets containing 6 and 8

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

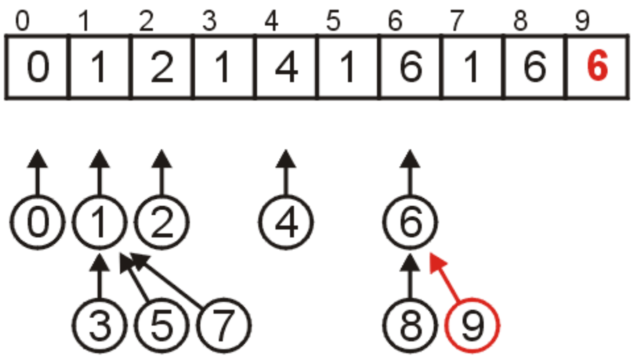


{0}, {1, 3, 5, 7}, {2}, {4}, {6, 8}, {9}

Example

Taking the union of the sets containing 8 and 9

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



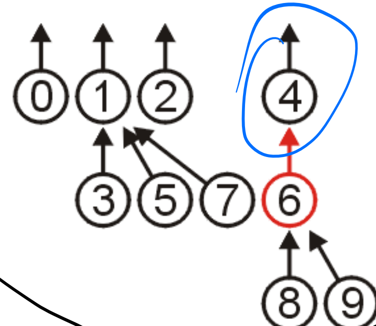
{0}, {1, 3, 5, 7}, {2}, {4}, {6, 8, 9}

Example

Taking the union of the sets containing 4 and 8

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

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



老板

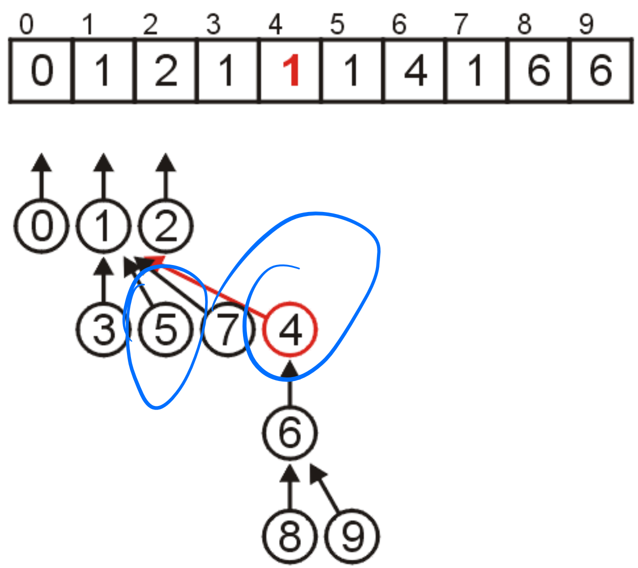
级数+1

{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);
```



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

Optimization 1

Problem:

- The height of the tree may grow very large

To optimize both `find` and `set_union`, we must minimize the height of the tree

- Therefore, point the root of the shorter tree to the root of the taller tree
- The height of the taller will increase if and only if the trees are equal in height

Worst-Case Scenario

Let us consider creating the worst-case disjoint set

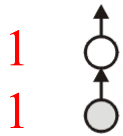
– The tallest tree with the least number of nodes

The worst case tree of height h must result from taking union of two worst case trees of height $h-1$

Worst-Case Scenario

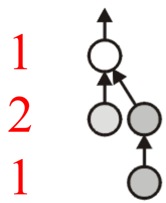
Thus, building on this, we take the union of two sets with one element

- We will keep track of the number of nodes at each depth



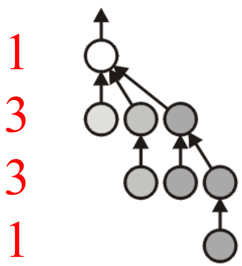
Worst-Case Scenario

Next, we take the union of two sets, that is, we join two worst-case sets of height 1:



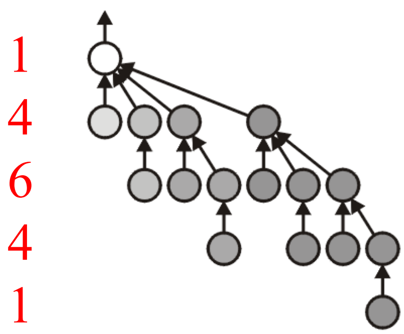
Worst-Case Scenario

And continue, taking the union of two worst-case trees of height 2:



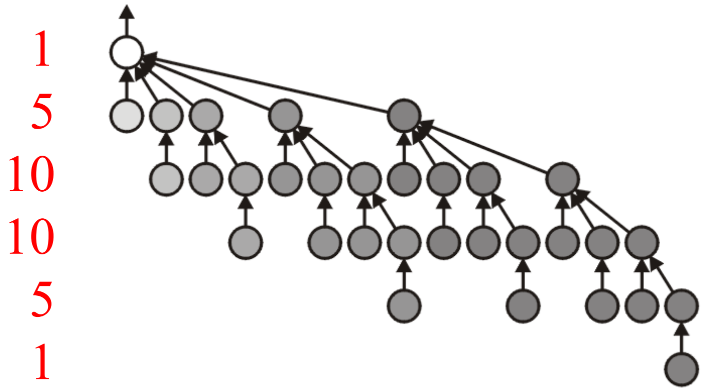
Worst-Case Scenario

Taking the union of two worst-case trees of height 3:



Worst-Case Scenario

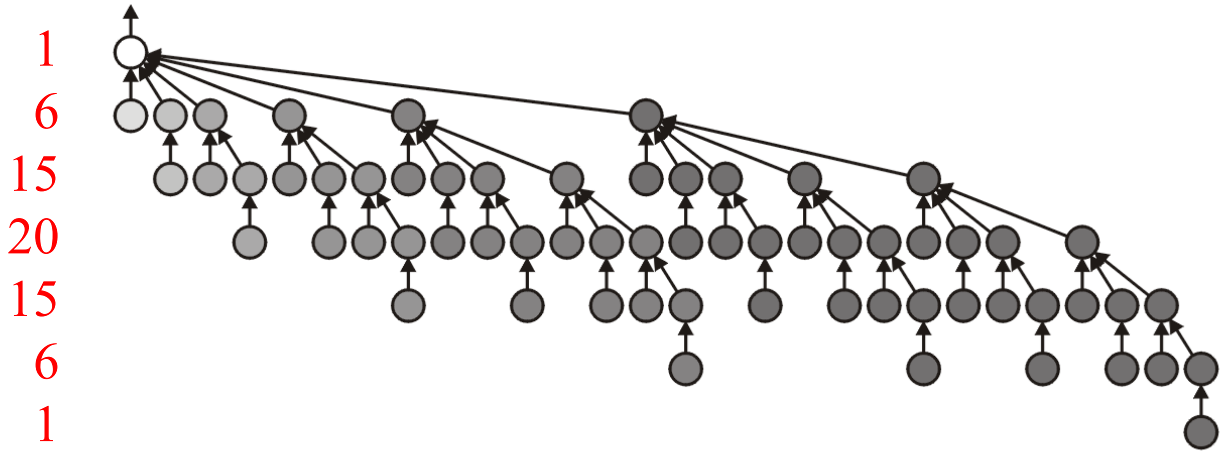
And of four:



Worst-Case Scenario

And finally, take the union of two worst-case trees of height 5:

- These are *binomial trees*



Worst-Case Scenario

From the construction, it should be clear that this would define Pascal's triangle

– The *binomial* coefficients

$$\binom{n}{m} = \begin{cases} 1 & m = 0 \text{ or } m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1} & 0 < m < n \end{cases}$$

$$= \frac{n!}{m!(n-m)!}$$

						1
					1	
				1		6
			1		5	
		1		4		15
	1		3		10	
1		2		6		20
	1		3		10	
		1		4		15
			1		5	
				1		6
					1	
						1

Worst-Case Scenario

Thus, suppose we have a worst-case tree of height h

– The number of nodes is $$\sum_{k=0}^{h} \binom{h}{k} = 2^h = n$$

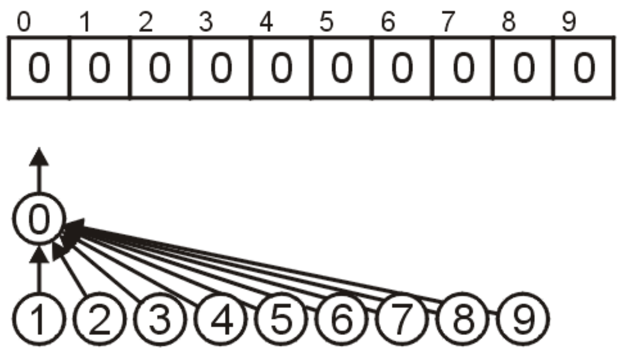
– The sum of node depth is $$\sum_{k=0}^{h} k\binom{h}{k} = h2^{h-1}$$

– Therefore, the average depth is $$\frac{h2^{h-1}}{2^h} = \frac{h}{2} = \frac{\lg(n)}{2}$$

– The height and average depth of the worst case are $\mathbf{O}(\ln(n))$

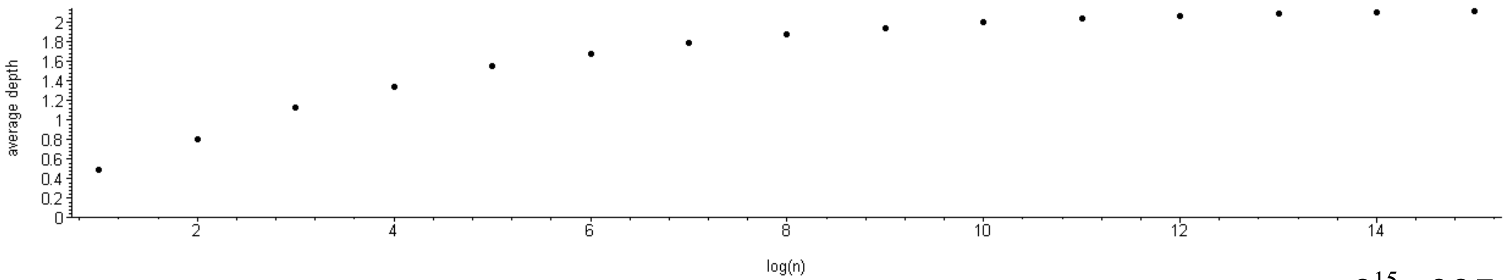
Best-Case Scenario

In the best case, all elements point to the same entry with a resulting height of $\Theta(1)$:



Average-Case Scenario

The resulting graph shows the average height of a randomly generated disjoint set data structure with 2^m elements



$2^{15} = 32768$

This suggests that the average height of such a tree is $\mathbf{o}(\ln(n))$

Optimization 2: Path Compression

Another optimization is that, whenever find is called, update the object to point to the root

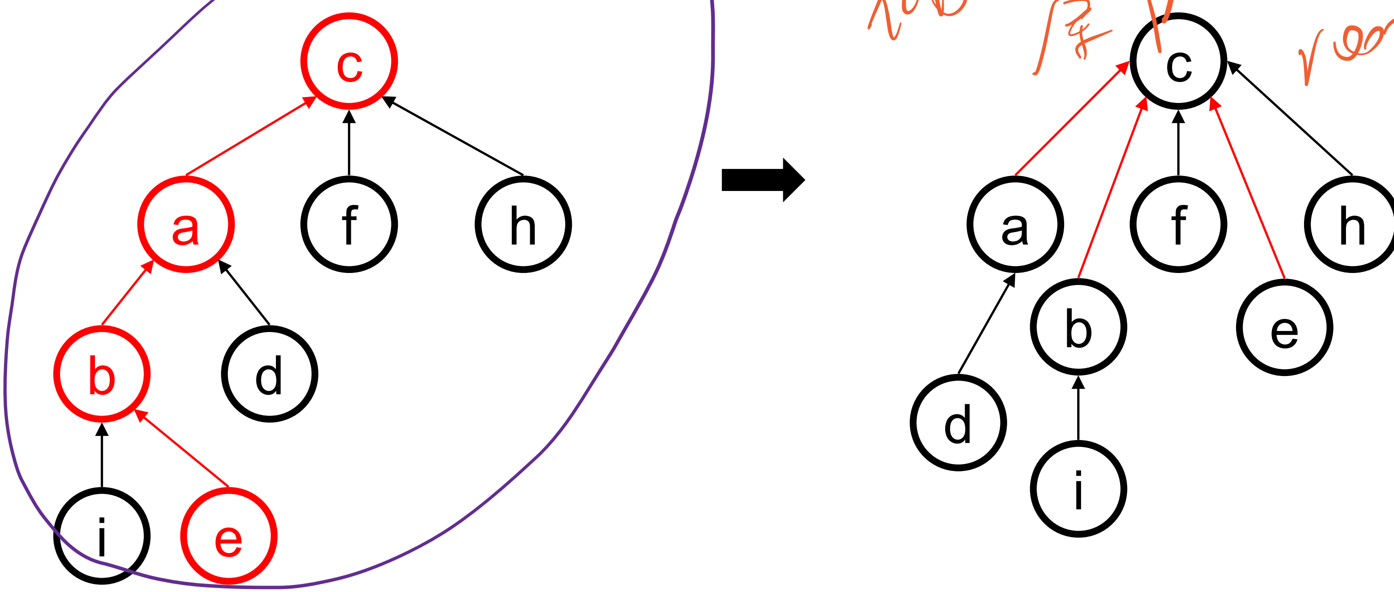
```
size_t Disjoint_set::find( size_t n ) {
    if ( parent[n] == n ) {
        return n;
    } else {
        parent[n] = find( parent[n] );
        return parent[n];
    }
}
```

The next call to `find(n)` is $\Theta(1)$

The cost is $\mathbf{O}(h)$ memory

Optimization 2: Path Compression

find(e)



被 find 的全 point to root 了

Time complexity

With both optimization methods, could it be any better than $\mathbf{O}(\ln(n))$?

– is there something better?

The amortized time complexity is $\mathbf{O}(\alpha(n))$ where $\alpha(n)$ is the inverse of the function $\mathrm{A}(n, n)$ where $\mathrm{A}(m, n)$ is the Ackermann function:

$$\mathrm{A}(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ \mathrm{A}(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ \mathrm{A}(m-1,\mathrm{A}(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

The first values are:

$$\mathrm{A}(0, 0) = 1, \quad \mathrm{A}(1, 1) = 3, \quad \mathrm{A}(2, 2) = 7, \quad \mathrm{A}(3, 3) = 61$$

Time complexity

$A(4, 4) = 2^{A(3, 4)} - 3$ where $A(3,4)$ is the 19729-decimal-digit number

$A(3, 4) = 2003529930406846464979072351$

Thus, $A(4, 4) - 3$, in binary, is 1 followed by this many zeros....

Time complexity

Therefore, we (as engineers) can, in clear conscience, state that the time complexity is $\Theta(1)$

- There are no physical circumstances where $\alpha(n)$ could by anything more than 4

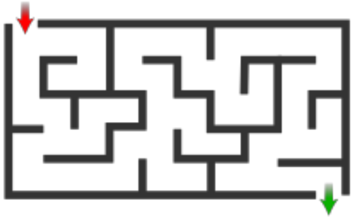
$\Theta(1)$

Application: Maze Generation

A fun application is in the generation of mazes

Impress your (non-engineering) friends

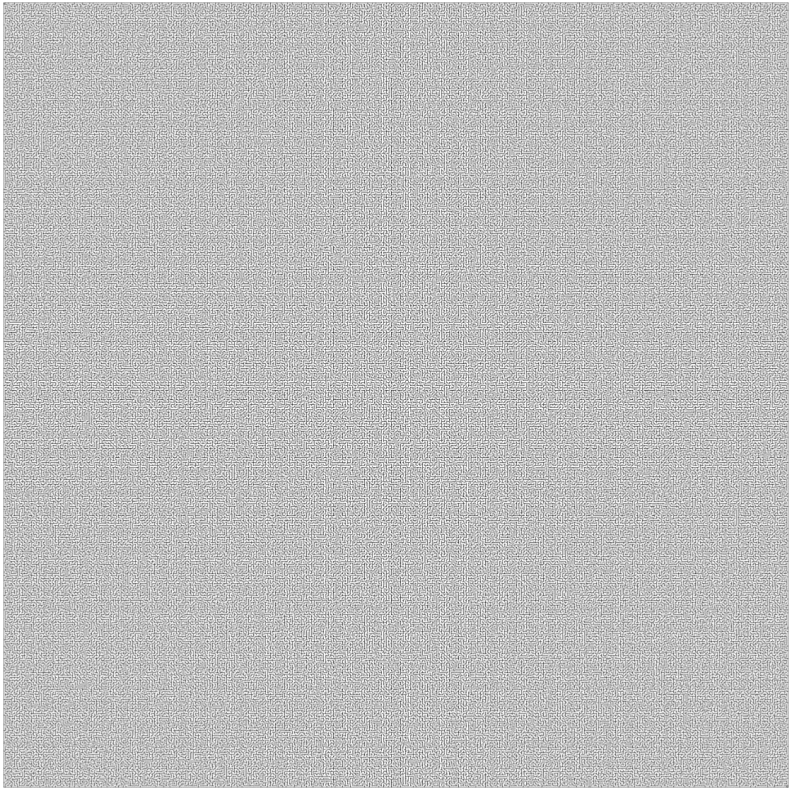
- They'll never guess how easy this is...



Application: Maze Generation

Here we have a maze which spans a 500 × 500 grid of squares where:

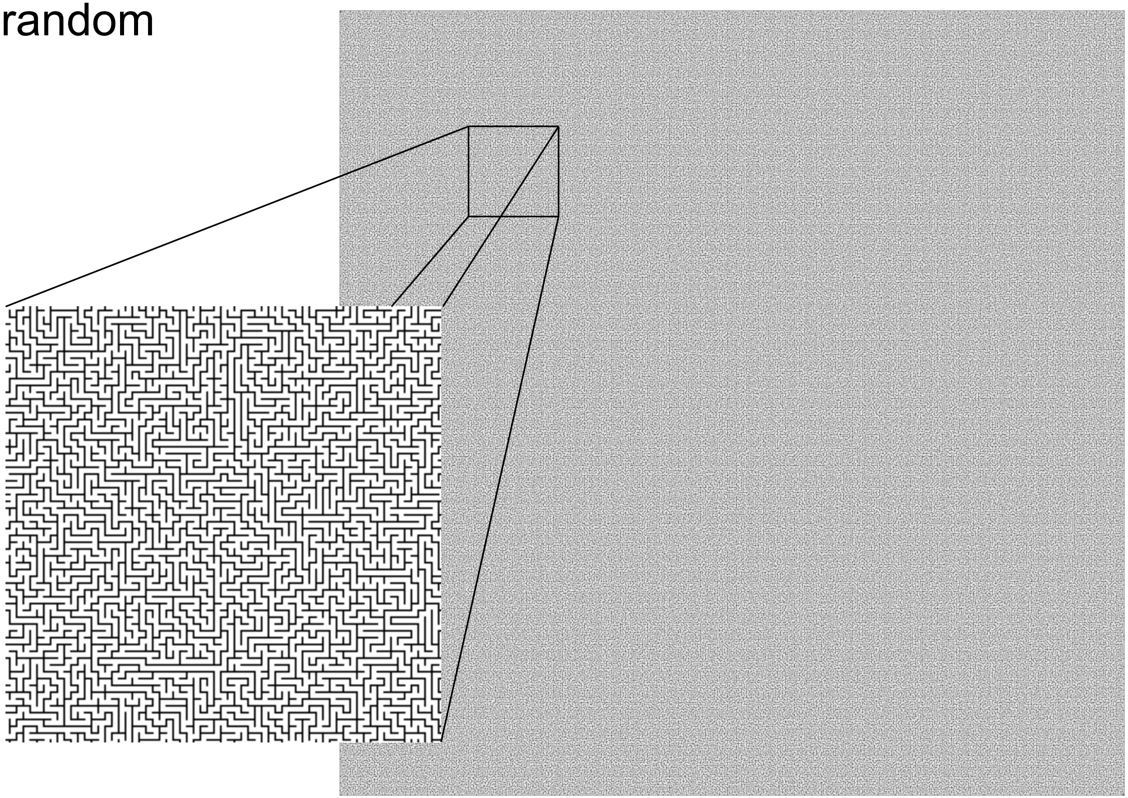
- There is one unique solution
- Each point can be reached by one unique path from the start



Ref: Lance Hampton http://littlebadwolf.com/mazes/

Application: Maze Generation

Zooming in on the maze, you will note that it is rather complex and seemingly random



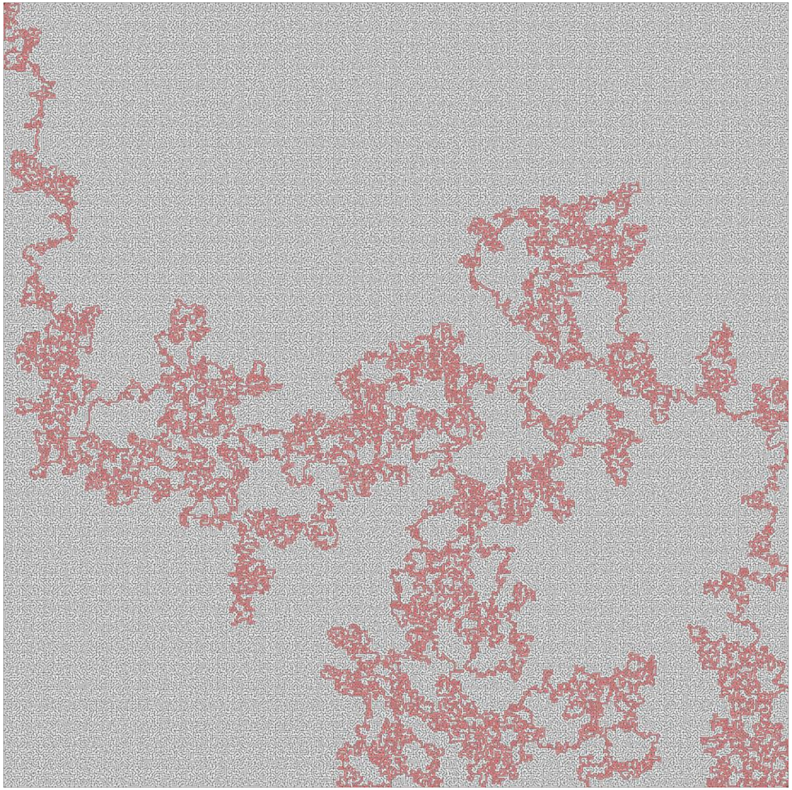
Ref: Lance Hampton http://littlebadwolf.com/mazes/

Application: Maze Generation

Finding the solution is a problem for a different lecture

- Backtracking algorithms

We will look at creating the maze using disjoint sets



Ref: Lance Hampton http://littlebadwolf.com/mazes/

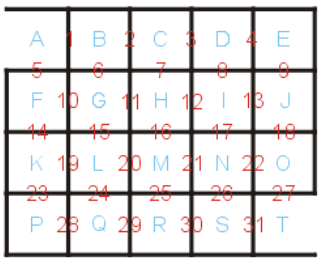
Application: Maze Generation

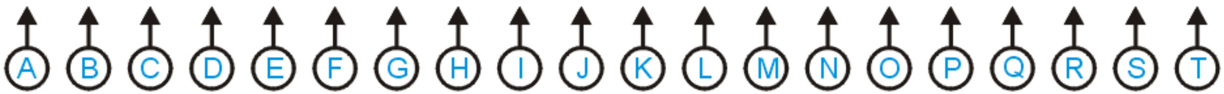
What we will do is the following:

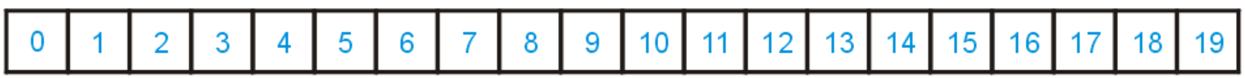
- Start with the entire grid subdivided into squares
- Represent each square as a separate disjoint set
- Repeat the following algorithm:
  - Randomly choose a wall
  - If that wall connects two disjoint sets of cells, then remove the wall and union the two sets
- To ensure that you do not randomly remove 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



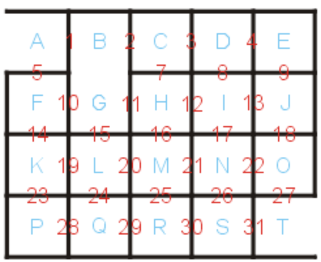


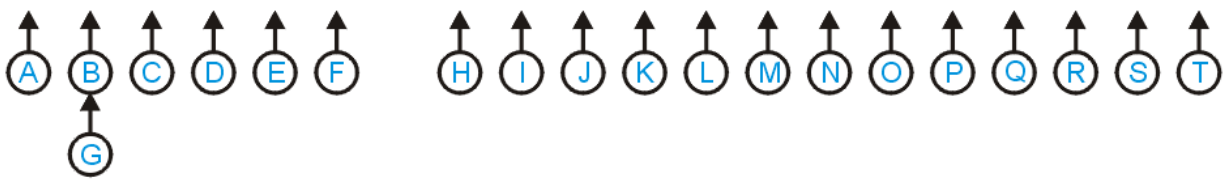


Application: Maze Generation

First, we select 6 which joins cells B and G

– Both have height 0

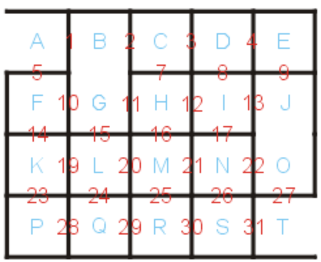


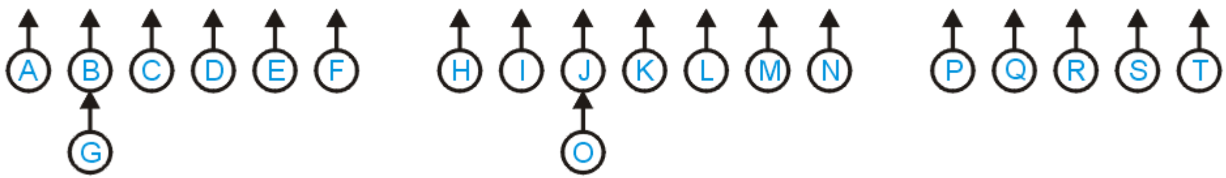


0	1	2	3	4	5	**1**	7	8	9	10	11	12	13	14	15	16	17	18	19

Application: Maze Generation

Next we select wall 18 which joins regions J and O



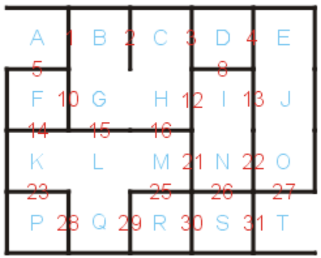


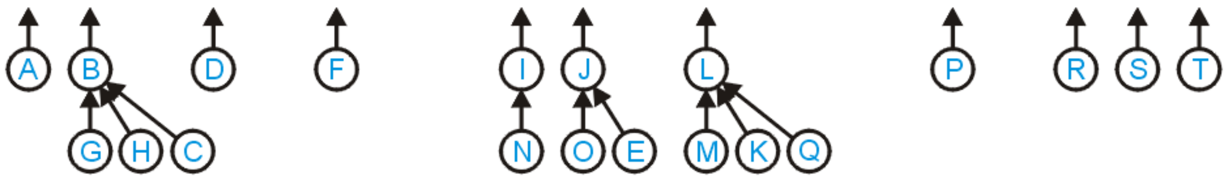
0	1	2	3	4	5	1	7	8	9	10	11	12	13	**9**	15	16	17	18	19

Application: Maze Generation

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

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



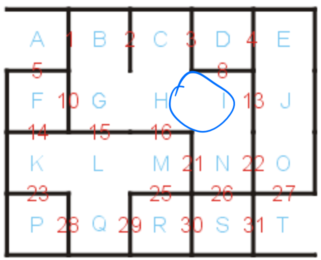


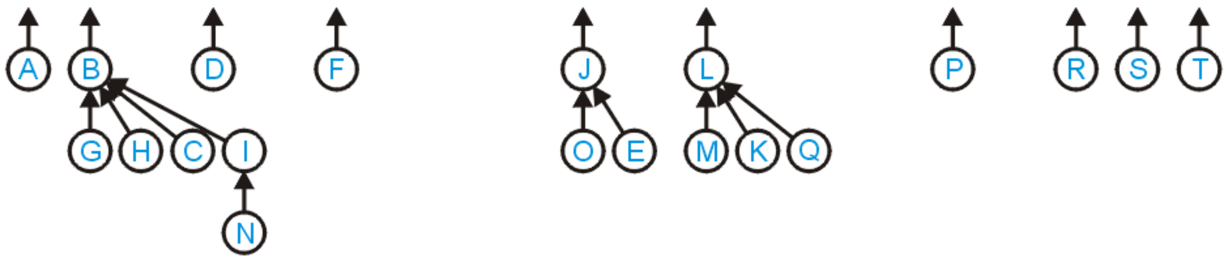
0	1	1	3	9	5	1	1	8	9	11	11	11	8	9	15	**11**	17	18	19

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



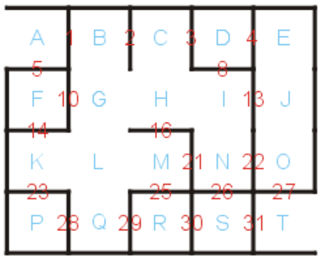


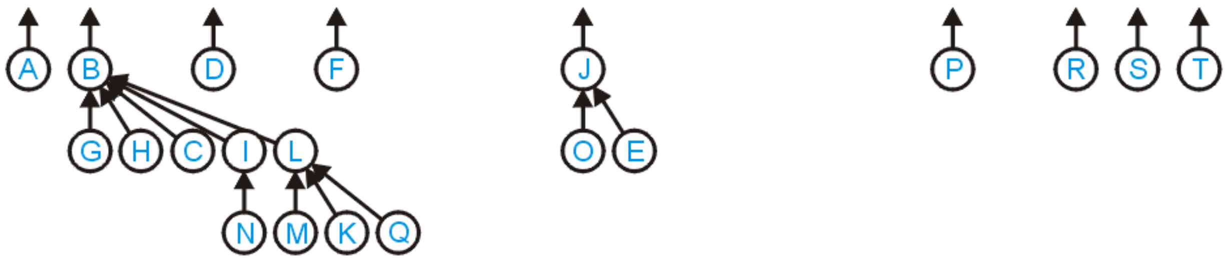
0	1	1	3	9	5	1	1	1	9	11	11	11	8	9	15	11	17	18	19

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



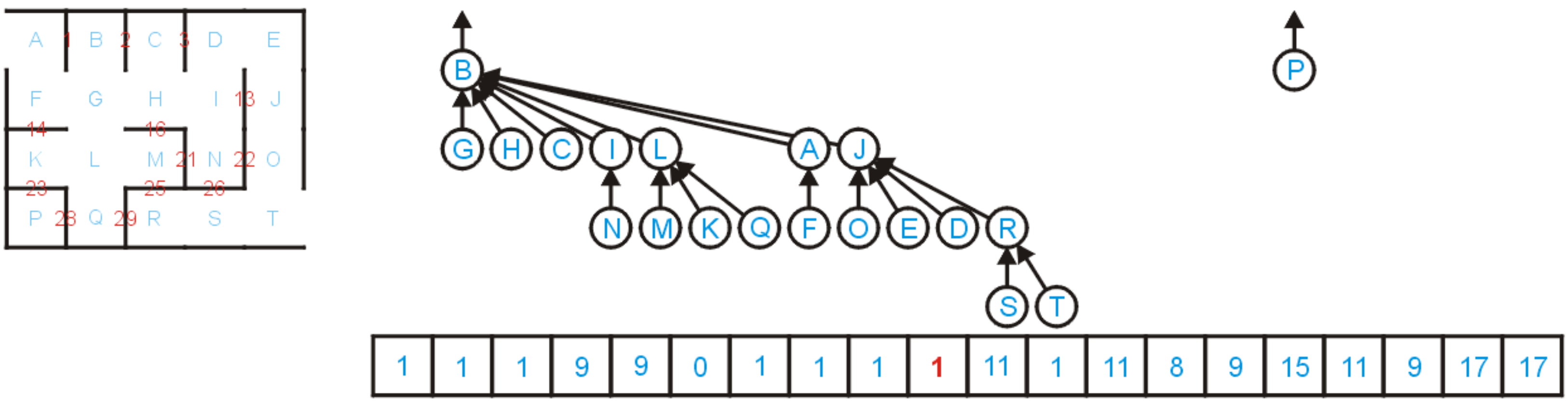


0	1	1	3	9	5	1	1	1	9	11	1	11	8	9	15	11	17	18	19

Application: Maze Generation

Selecting wall 8 joins sets identified by B and J

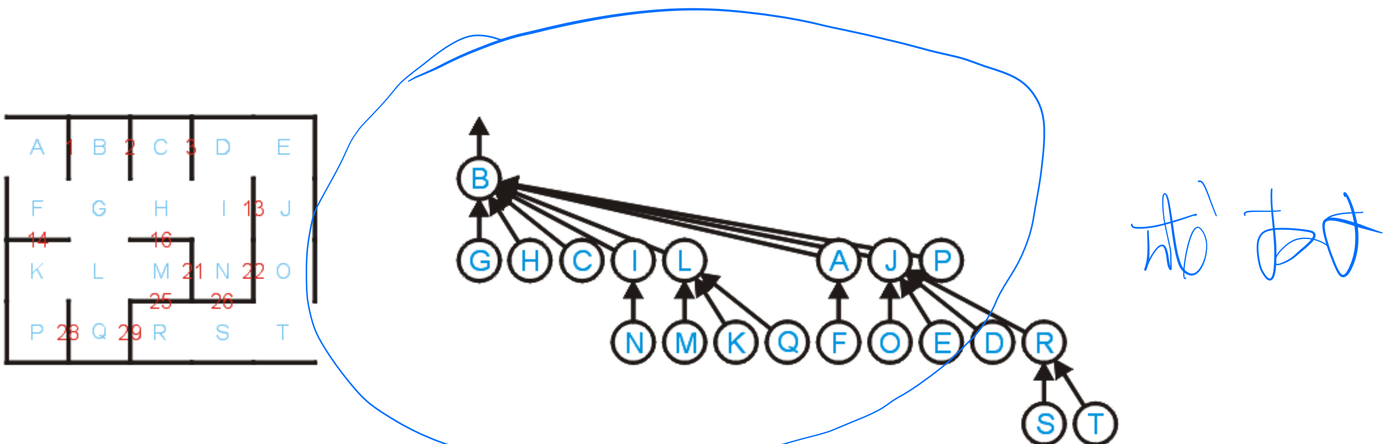
- They both have height 2 so we note that J has fewer nodes than B, so we add J to B



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

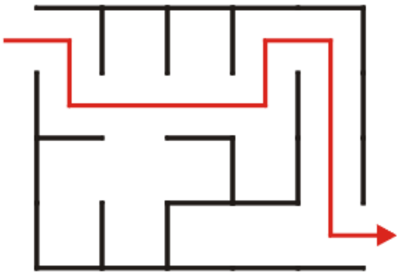


1	1	1	9	9	0	1	1	1	1	11	1	11	8	9	1	11	9	17	17

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



How can we prove these two properties?

Application: Maze Generation