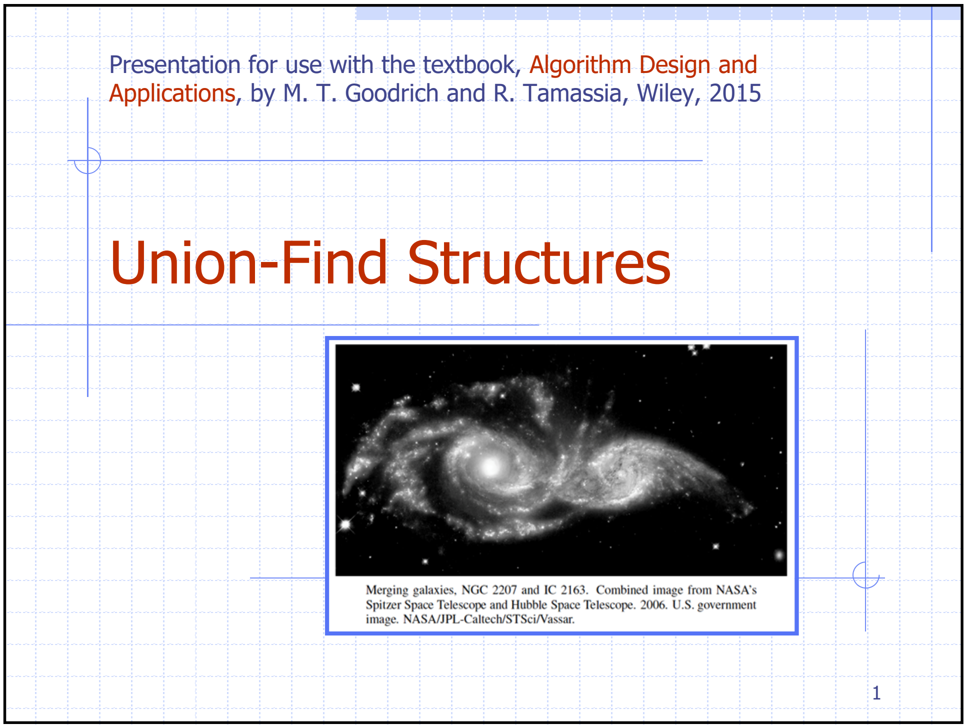
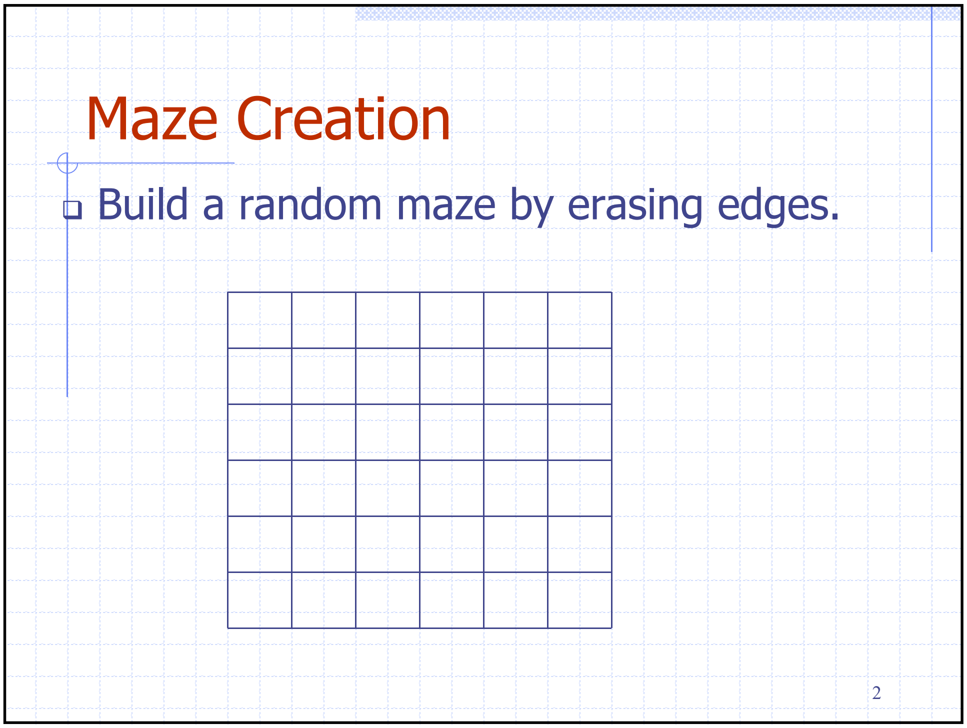


Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

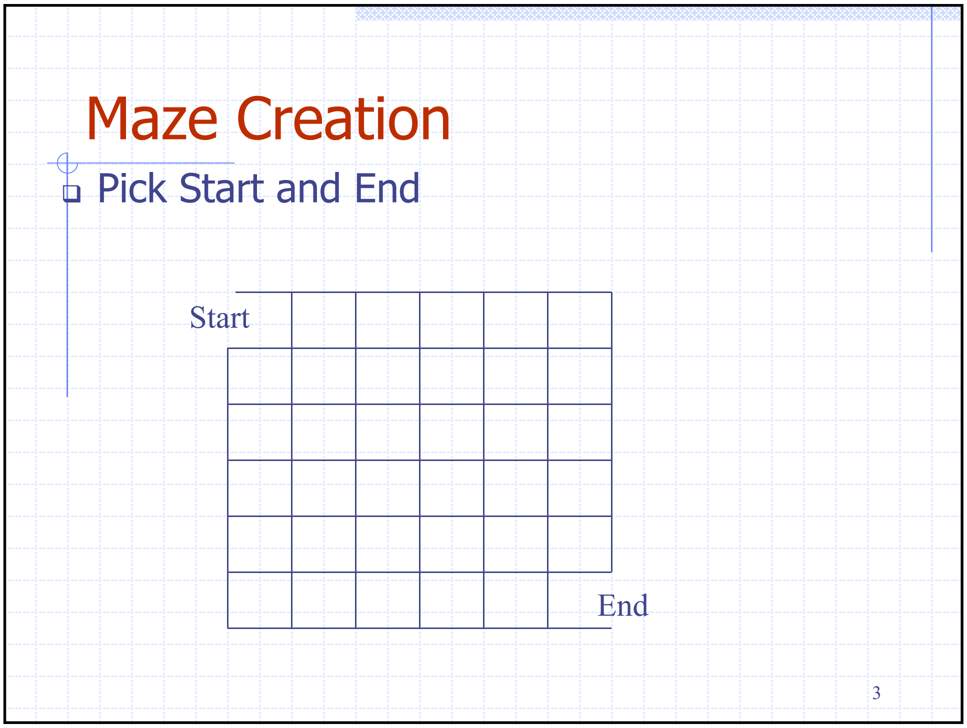
Union-Find Structures

Merging galaxies, NGC 2207 and IC 2163. Combined image from NASA's Spitzer Space Telescope and Hubble Space Telescope. 2006. U.S. government image. NASA/JPL-Caltech/STSci/Vassar.



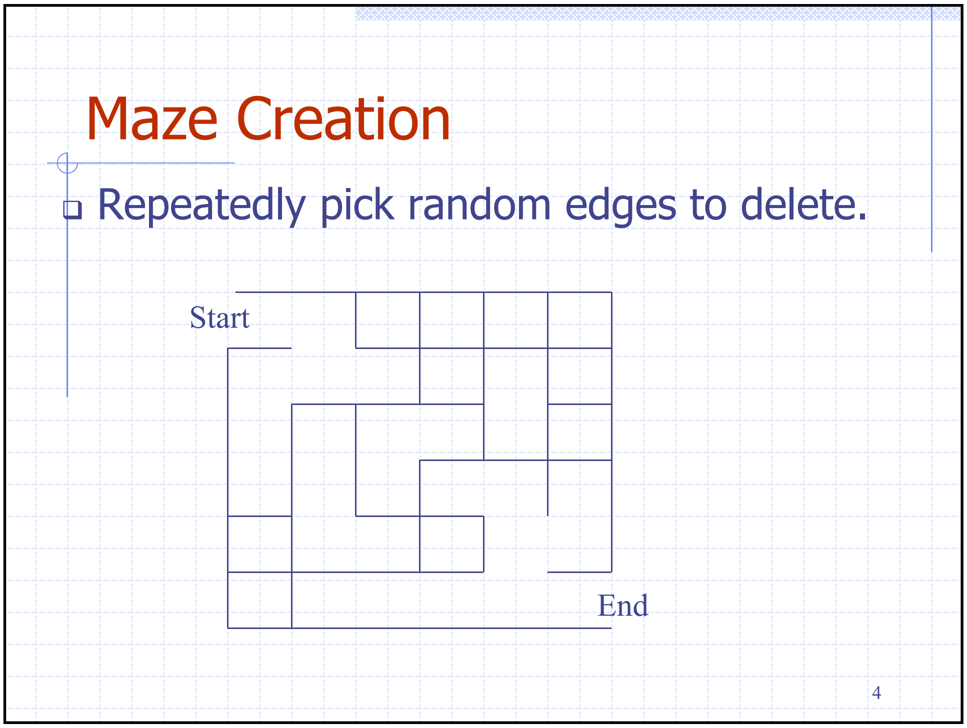
Maze Creation

- Build a random maze by erasing edges.



Maze Creation

- Pick Start and End



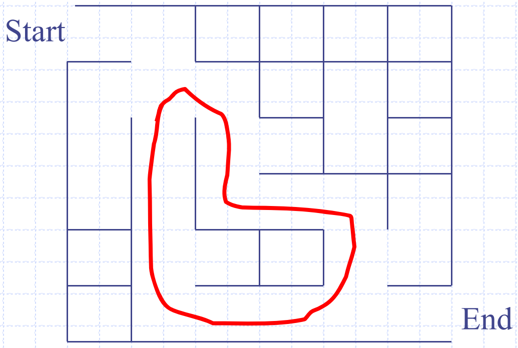
Maze Creation

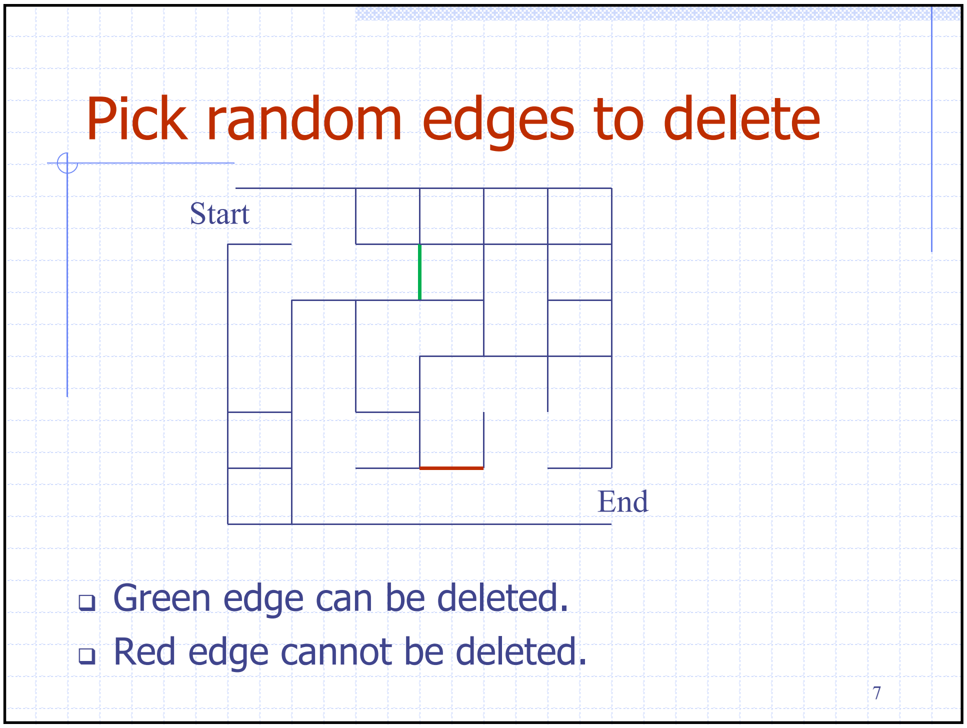
- Repeatedly pick random edges to delete.

Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.

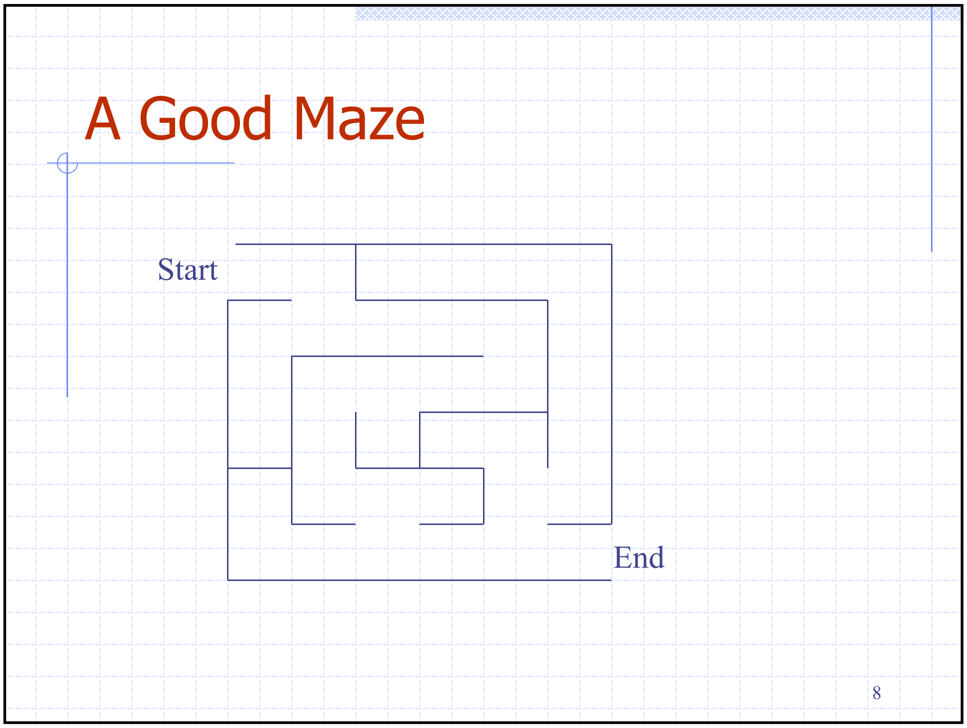
A Cycle, not allowed





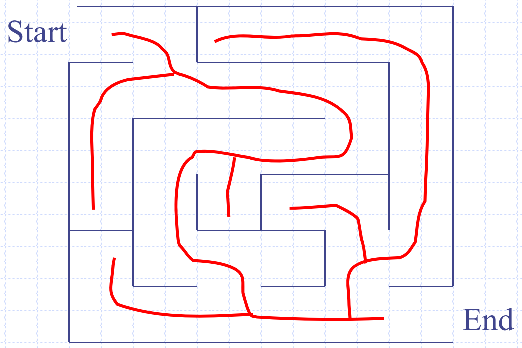
Pick random edges to delete

- Green edge can be deleted.
- Red edge cannot be deleted.



A Good Maze

A Hidden Tree



Maze Creation: Algorithm

	Cost:
1. Given the dimension s of the maze, create an s by s matrix, give a name to each cell of the matrix, identify the beginning and ending cells.	O(s^2)
2. Collect all the possible edges between the cells, excluding the boundary edges, into E.	O(s^2)
3. If not all the cells are reachable from each other, randomly pick and remove an edge e from E; otherwise go to 5.	O(s^2)
4. If the two ends of edge e are already connected by a path, add e into M; otherwise, throw away e, and go to 3.	O(s^2)
5. Return the union of E and M as the edges of the maze.	O(1)

Number the Cells

We have disjoint sets S ={ {1}, {2}, {3}, {4},… {36} } each cell is a singleton set.
We have all possible edges E ={ (1,2), (1,7), (2,8), (2,3), … } 60 edges total, representing the neighborhood relation. Boundary edges are excluded.

Start	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	26	27	28	29	30	
	31	32	33	34	35	36	End

For an s by s matrix, there are s^2 cells and $2s(s - 1)$ edges.
We need to delete $s^2 - 1$ edges.
There are $(s - 1)^2$ edges in the maze.

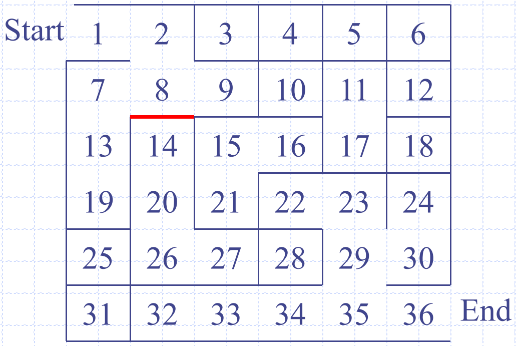
Group Cells into Disjoint Sets

- At any moment, two cells are in the same set if and only if they are connected by a path in the maze.
- An edge (x, y) is safe to delete, if x and y are not in the same set.
- After (x, y) is deleted, the two sets containing x and y are joined together.

Disjoint sets are good data structure for implementing equivalence relations.

Example of Deletion

Pick edge (8,14)



Start	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	26	27	28	29	30	
	31	32	33	34	35	36	End

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,30,32
33,34,35,36}

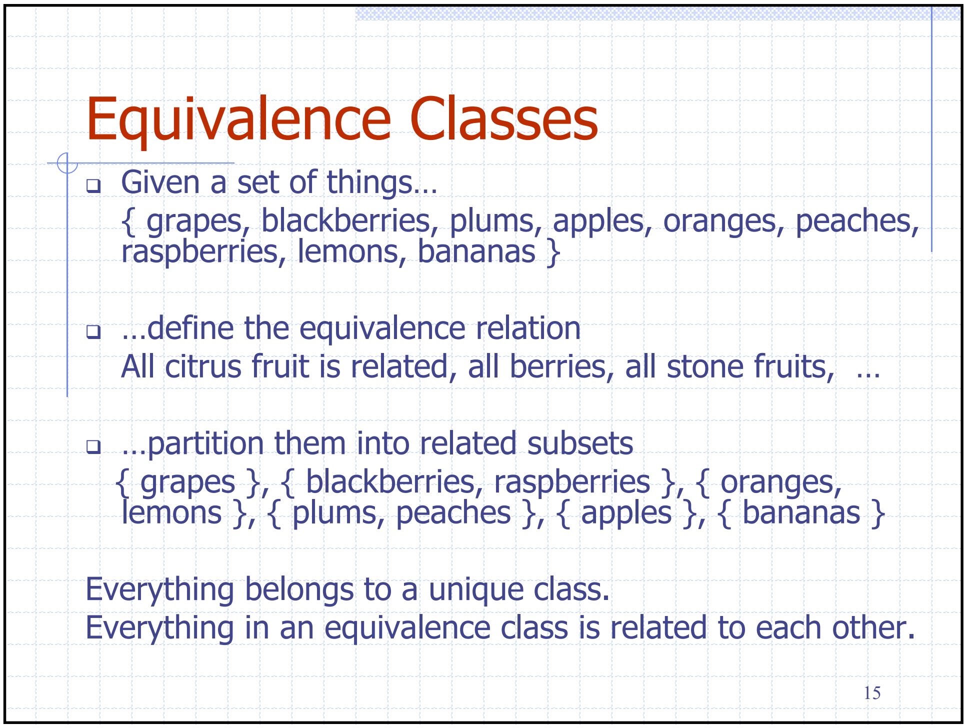
Equivalence Relation

Relation *R* on *S* is a subset of *SxS*.

- For every pair of elements *a, b* from a set *S*, *a R b* is either true or false.
- *a R b* is true iff (*a, b*) is in *R*. In this case, we say *a is related* to *b*.

An equivalence relation satisfies:

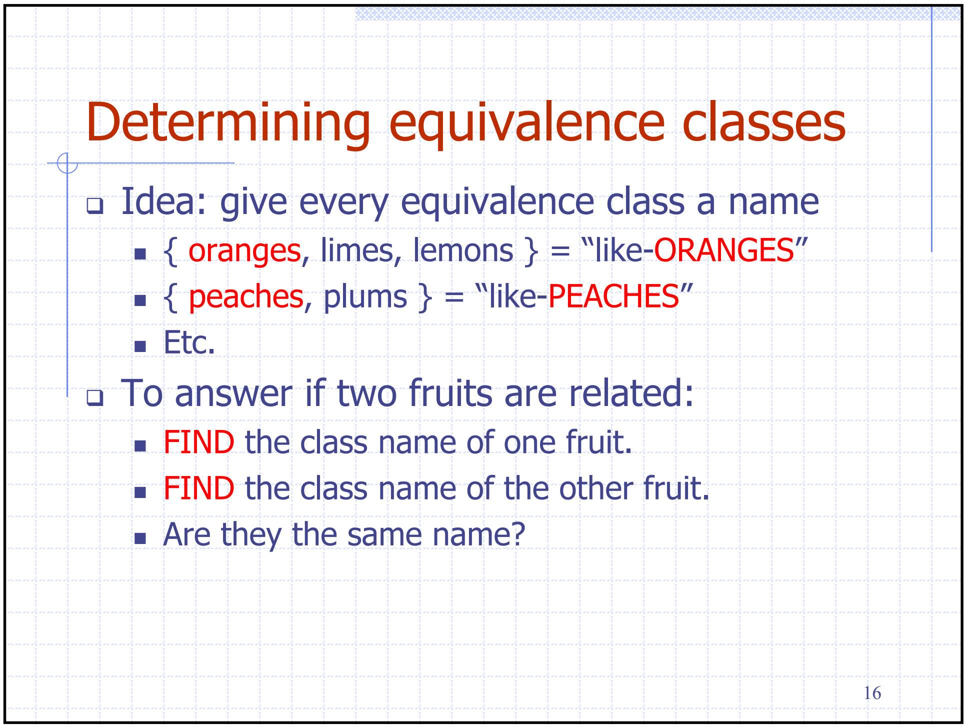
1. (Reflexive) a *R* a
2. (Symmetric) a *R* b iff b *R* a
3. (Transitive) a *R* b and b *R* c implies a *R* c



Equivalence Classes

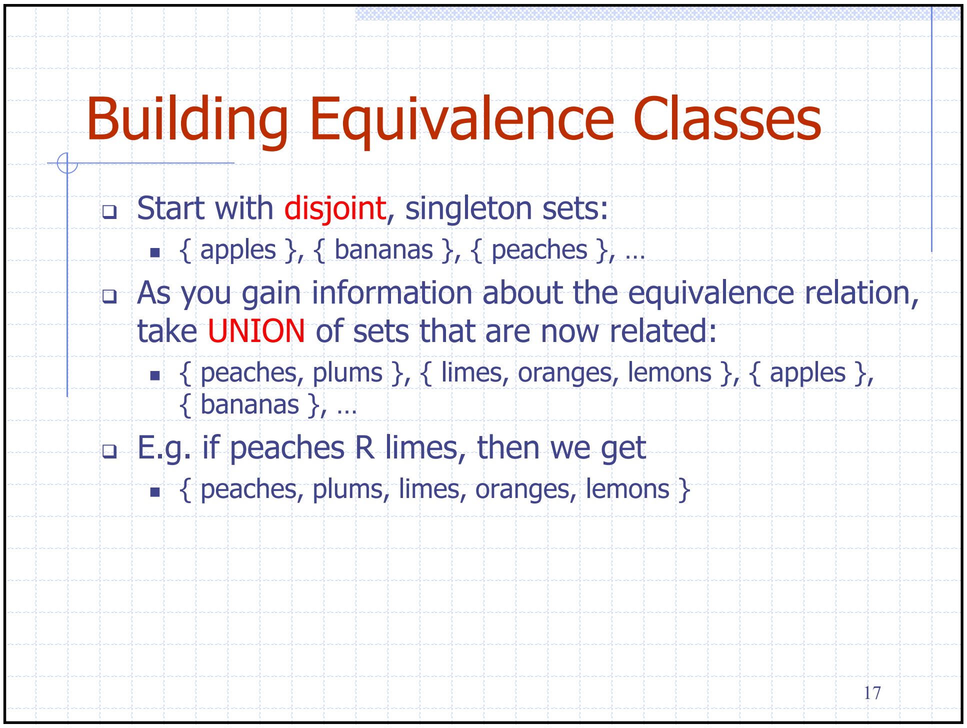
- Given a set of things…
  { grapes, blackberries, plums, apples, oranges, peaches, raspberries, lemons, bananas }
- …define the equivalence relation
  All citrus fruit is related, all berries, all stone fruits, …
- …partition them into related subsets
  { grapes }, { blackberries, raspberries }, { oranges, lemons }, { plums, peaches }, { apples }, { bananas }

Everything belongs to a unique class.
Everything in an equivalence class is related to each other.



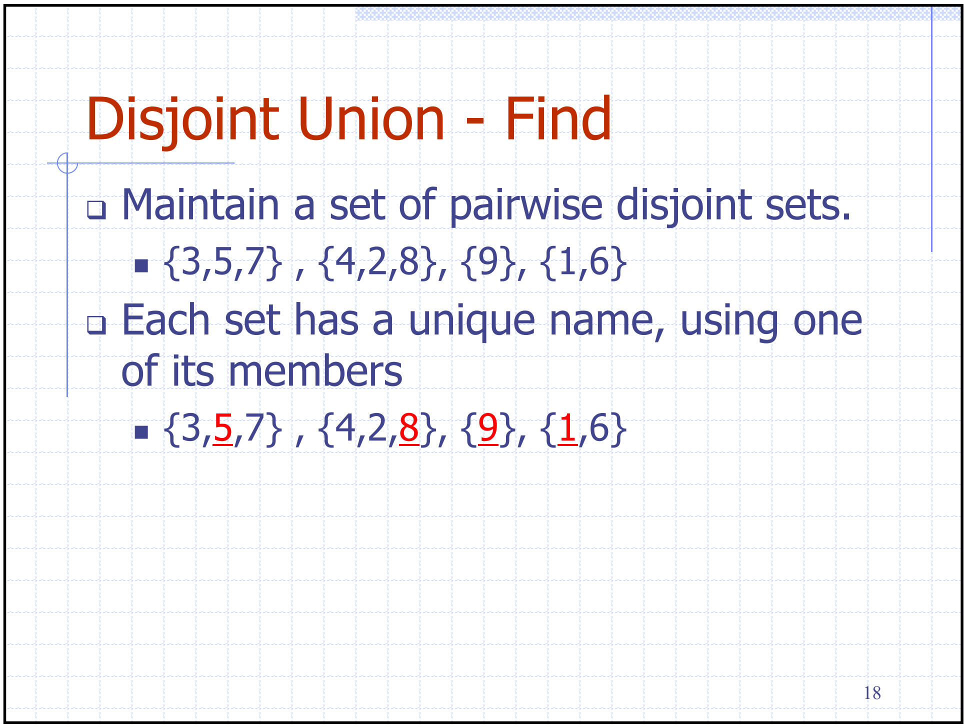
Determining equivalence classes

- Idea: give every equivalence class a name
  - { oranges, limes, lemons } = “like-ORANGES”
  - { peaches, plums } = “like-PEACHES”
  - Etc.
- To answer if two fruits are related:
  - FIND the class name of one fruit.
  - FIND the class name of the other fruit.
  - Are they the same name?



Building Equivalence Classes

- Start with disjoint, singleton sets:
  - { apples }, { bananas }, { peaches }, …
- As you gain information about the equivalence relation, take UNION of sets that are now related:
  - { peaches, plums }, { limes, oranges, lemons }, { apples }, { bananas }, …
- E.g. if peaches R limes, then we get
  - { peaches, plums, limes, oranges, lemons }



Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, using one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union

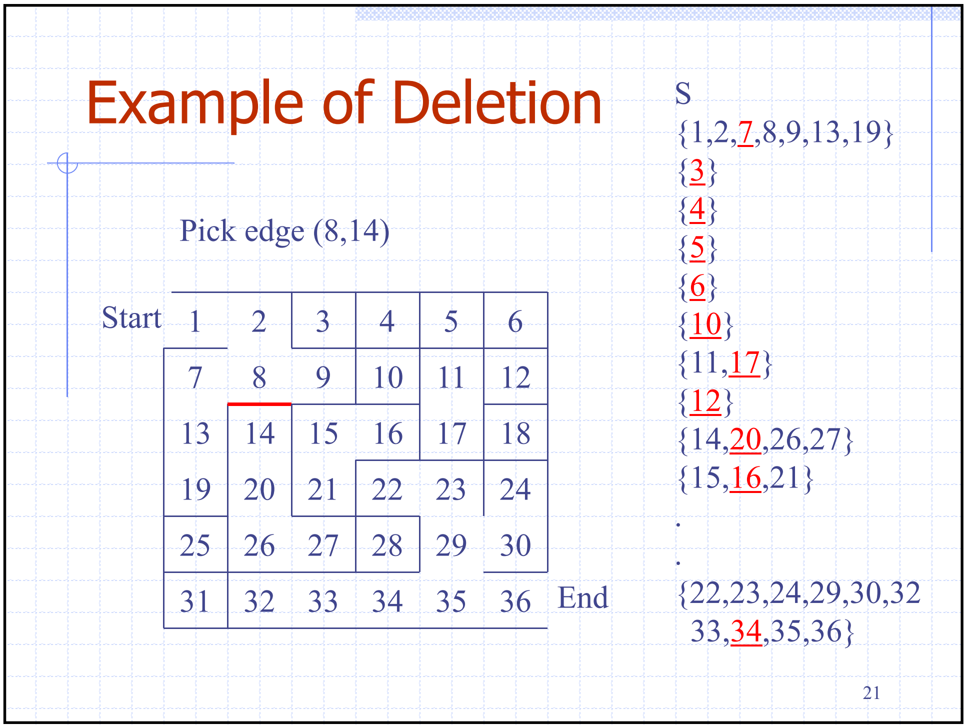
- Union(x,y) – return the union of two sets named by x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union(5,1)

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

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

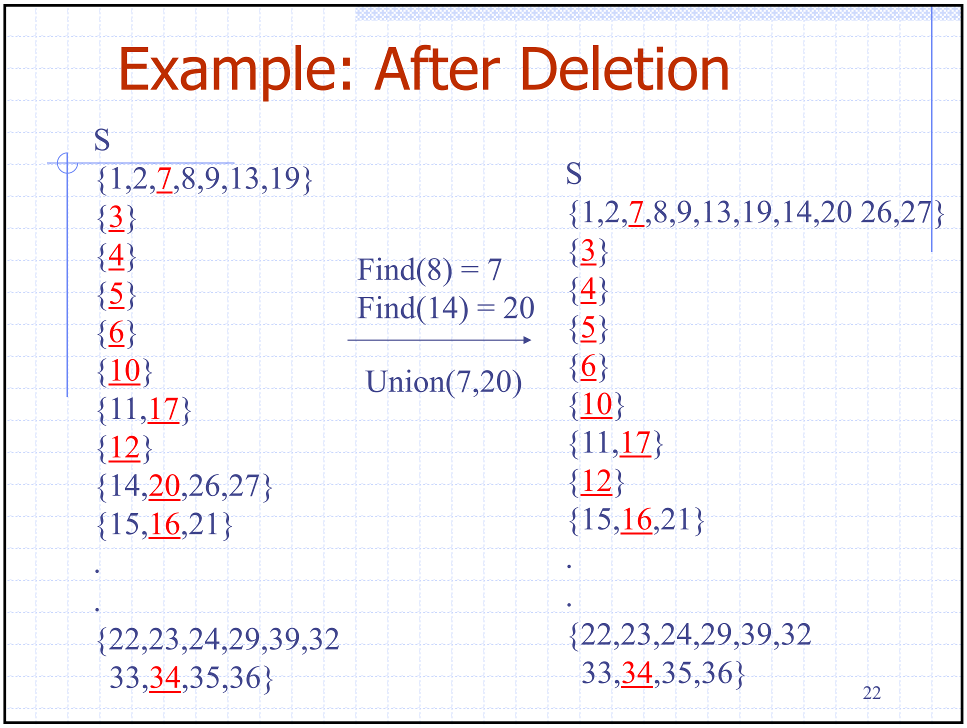


Example of Deletion

Pick edge (8,14)

Start	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	26	27	28	29	30	
	31	32	33	34	35	36	End

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,30,32
33,34,35,36}



Example: After Deletion

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Find(8) = 7
Find(14) = 20
$\longrightarrow$
Union(7,20)

S
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

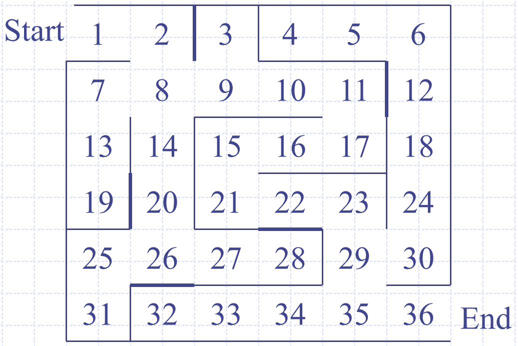
Example

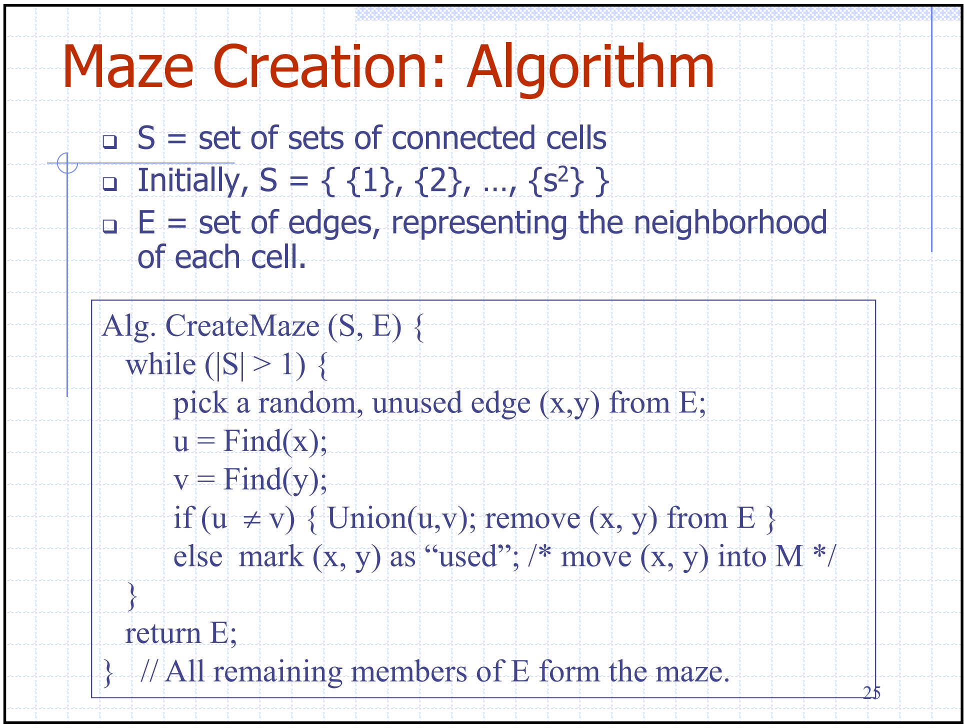
Pick (19,20)

S
{1,2,7,8,9,13,19
14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example at the End

S
{1,2,3,4,5,6,7,... 36}

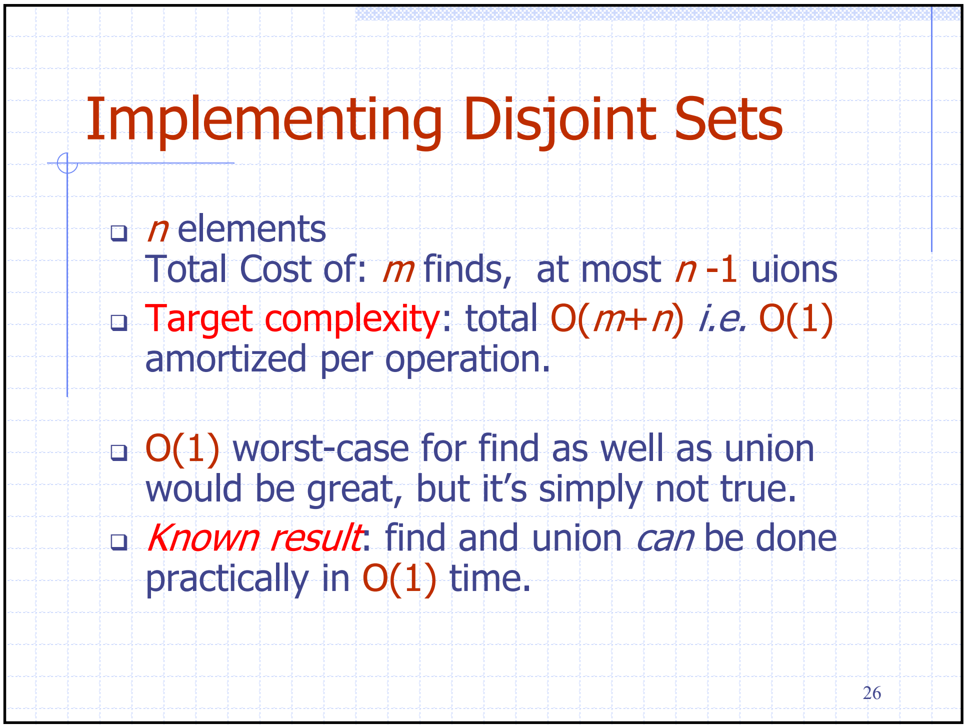




Maze Creation: Algorithm

- S = set of sets of connected cells
- Initially, S = { {1}, {2}, …, {s^2} }
- E = set of edges, representing the neighborhood of each cell.

```
Alg. CreateMaze (S, E) {
  while ($|S| > 1$) {
      pick a random, unused edge (x,y) from E;
      u = Find(x);
      v = Find(y);
      if ($u \neq v$) { Union(u,v); remove (x, y) from E }
      else  mark (x, y) as “used”; /* move (x, y) into M */
  }
  return E;
}  // All remaining members of E form the maze.
```

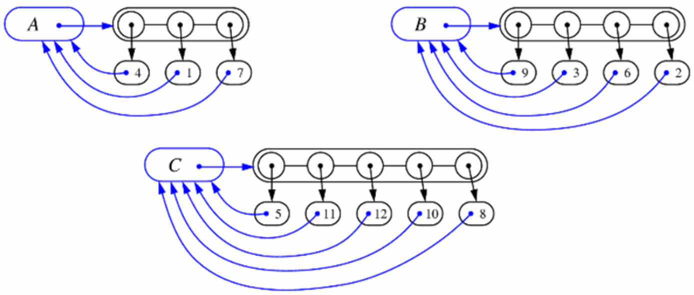


Implementing Disjoint Sets

- *n* elements
  Total Cost of: *m* finds, at most *n* -1 uions
- Target complexity: total O(*m*+*n*) *i.e.* O(1) amortized per operation.
- O(1) worst-case for find as well as union would be great, but it’s simply not true.
- *Known result*: find and union *can* be done practically in O(1) time.

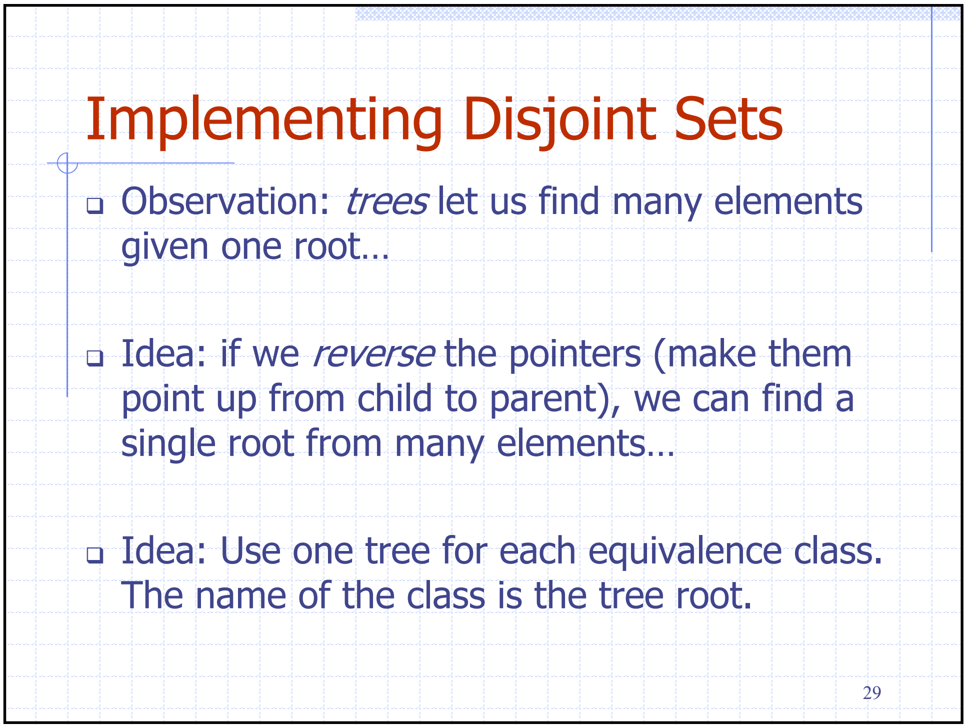
List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name



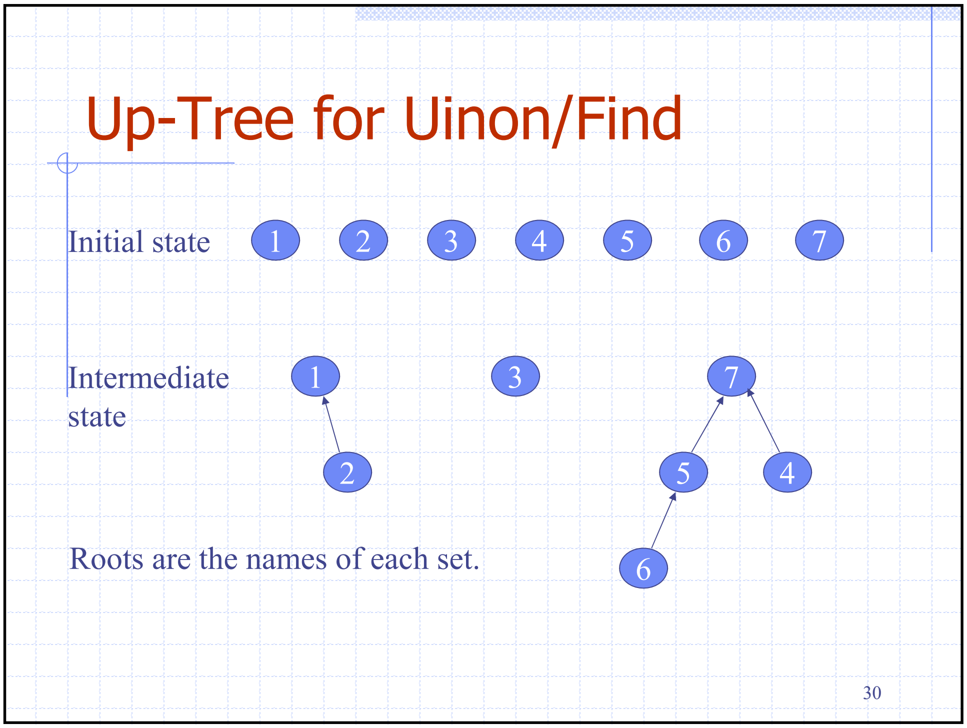
Analysis of List-based Representation

- Worst case time for find is O(1).
- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times
- Total time needed to do n – 1 unions and m finds is O(n log n + m).



Implementing Disjoint Sets

- Observation: *trees* let us find many elements given one root…
- Idea: if we *reverse* the pointers (make them point up from child to parent), we can find a single root from many elements…
- Idea: Use one tree for each equivalence class. The name of the class is the tree root.

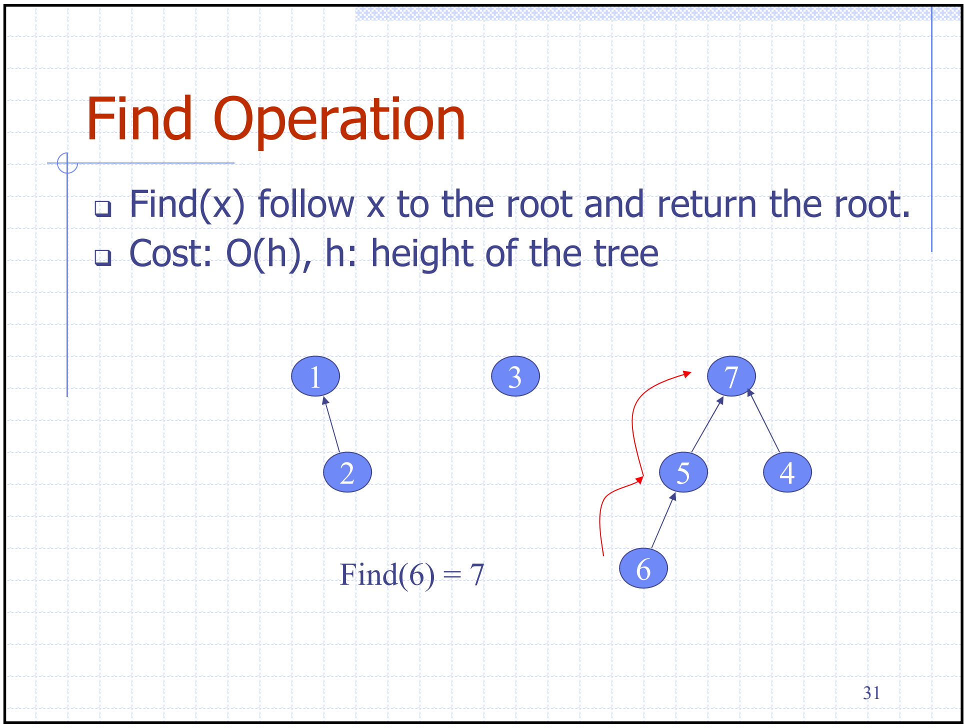


Up-Tree for Uinon/Find

Initial state

Intermediate state

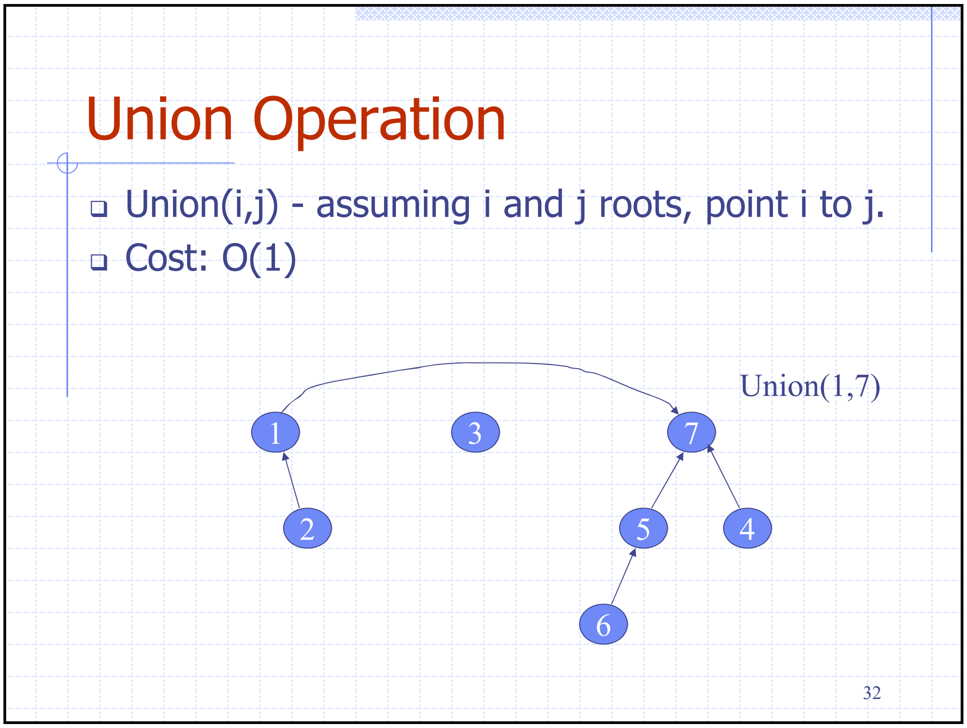
Roots are the names of each set.



Find Operation

- Find(x) follow x to the root and return the root.
- Cost: O(h), h: height of the tree

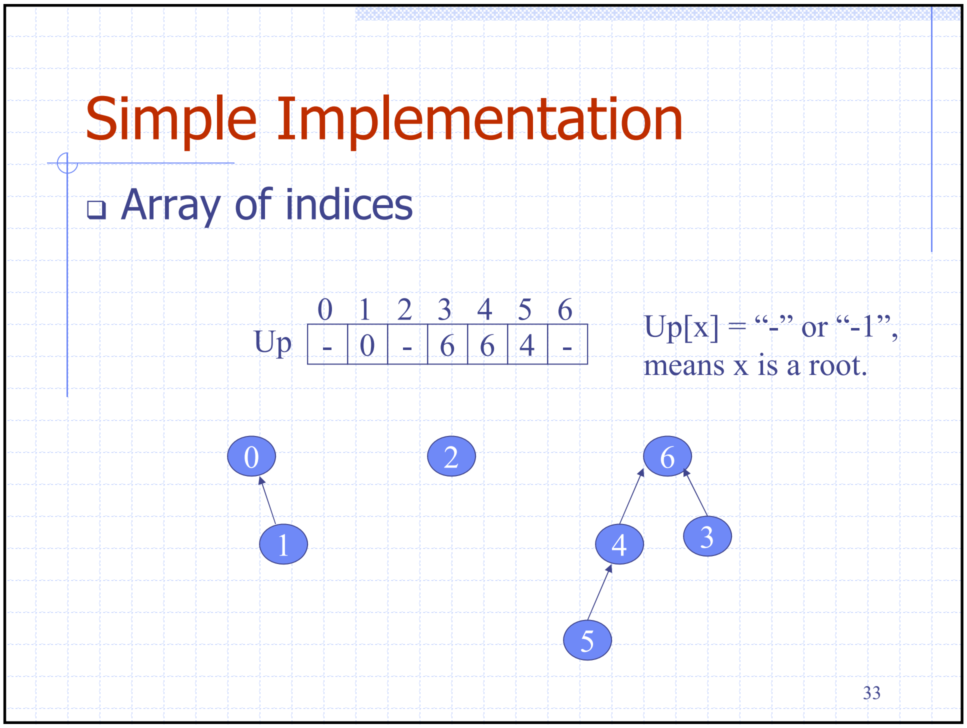
Find(6) = 7



Union Operation

- Union(i,j) - assuming i and j roots, point i to j.
- Cost: O(1)

Union(1,7)

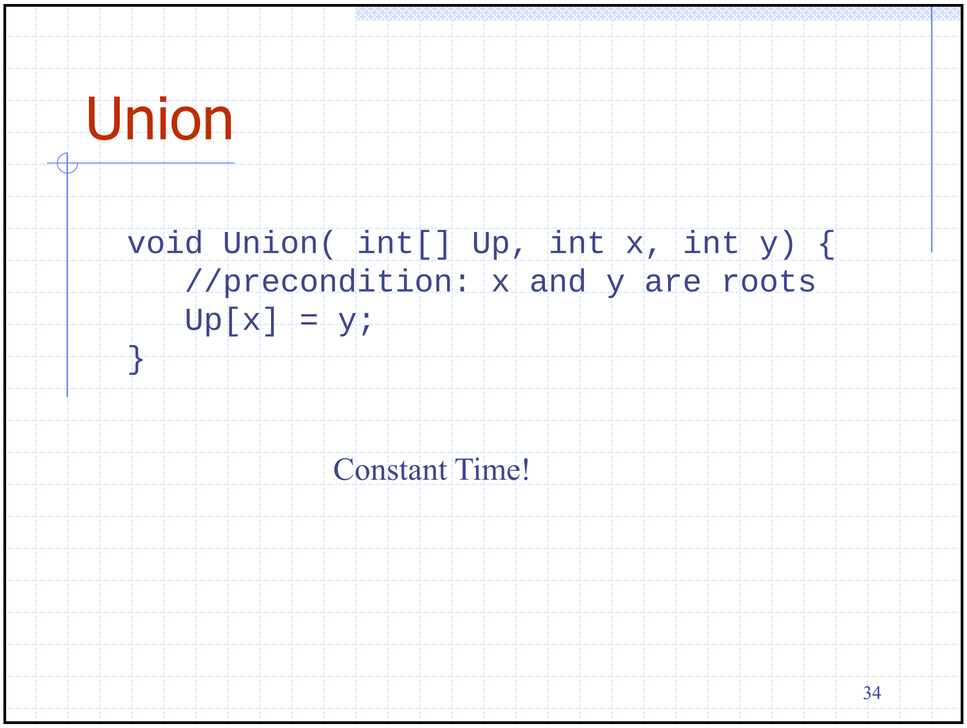


Simple Implementation

- Array of indices

	0	1	2	3	4	5	6
Up	-	0	-	6	6	4	-

Up[x] = “-” or “-1”, means x is a root.



Union

```
void Union( int[] Up, int x, int y) {
   //precondition: x and y are roots
   Up[x] = y;
}
```

Constant Time!

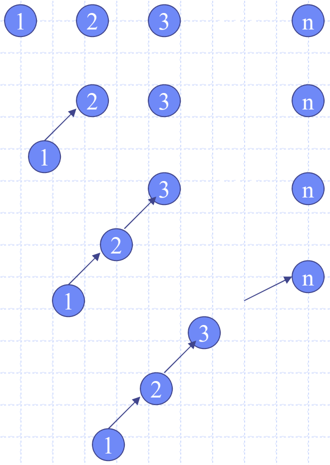
FIND

- Design Find operator
  - Recursive version
  - Iterative version

```
static int Find(int[] Up, int x) {
  //Pre: Up[0..(siz-1)] is the parent info;
  // x is in the range 0 to size-1
  if (Up[x] == “-1”) return x;
  return Find(Up[x]);
}
```

Complexity: depth of x in the tree.

A Bad Case



Union(1,2)

Union(2,3)

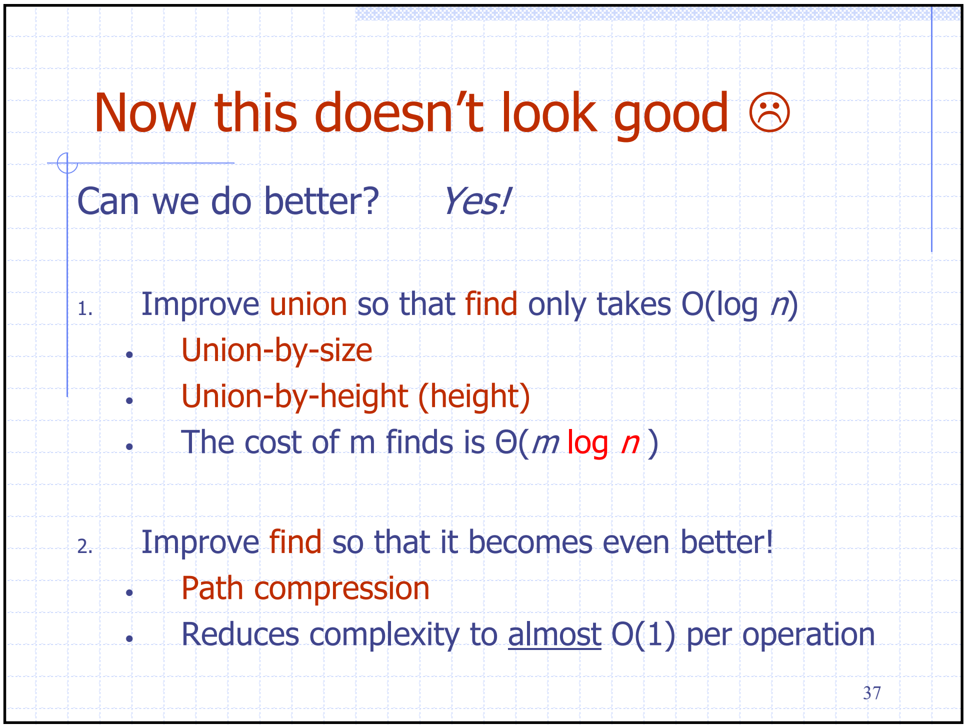
:

:

Union(n-1,n)

Find(1) n steps!!

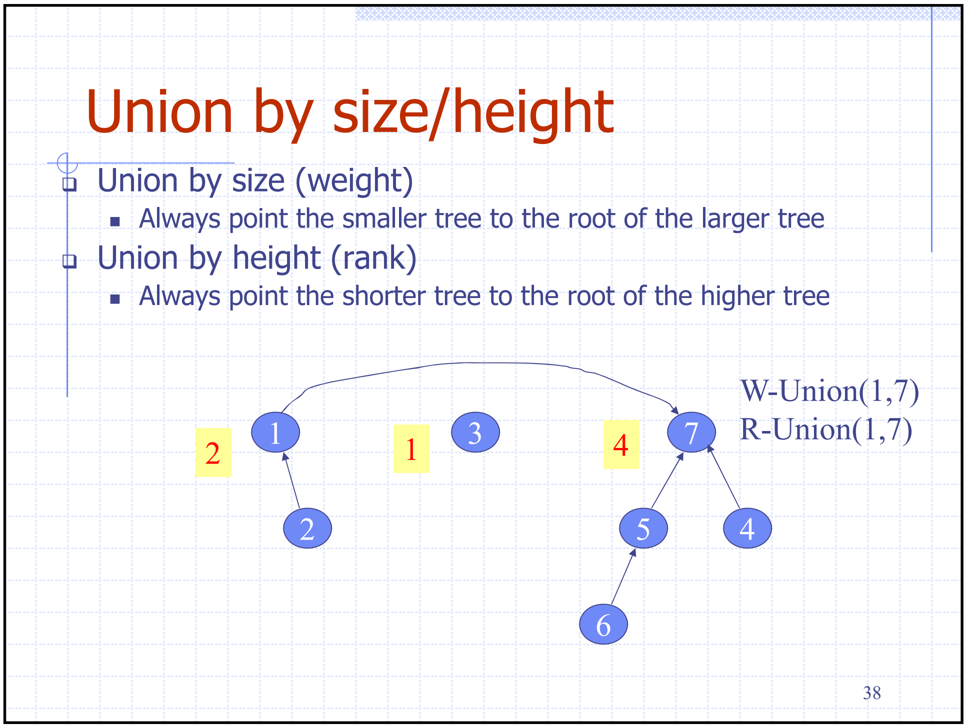
m finds: O(mn)



Now this doesn't look good ☹

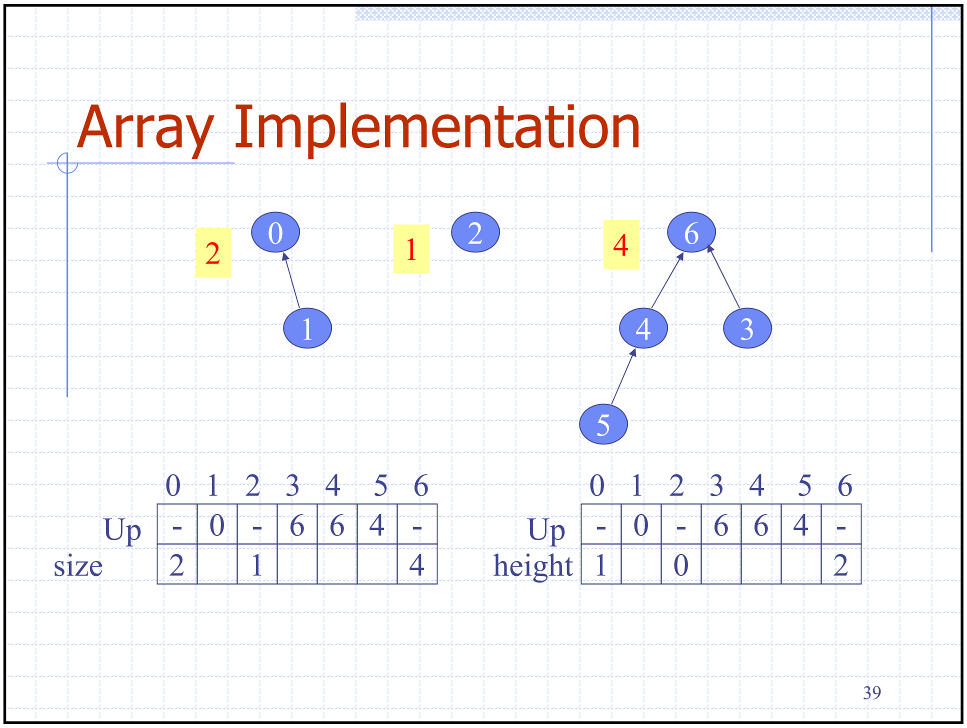
Can we do better? *Yes!*

1. Improve union so that find only takes O(log *n*)
   - Union-by-size
   - Union-by-height (height)
   - The cost of m finds is Θ(*m* log *n*)
2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost O(1) per operation



Union by size/height

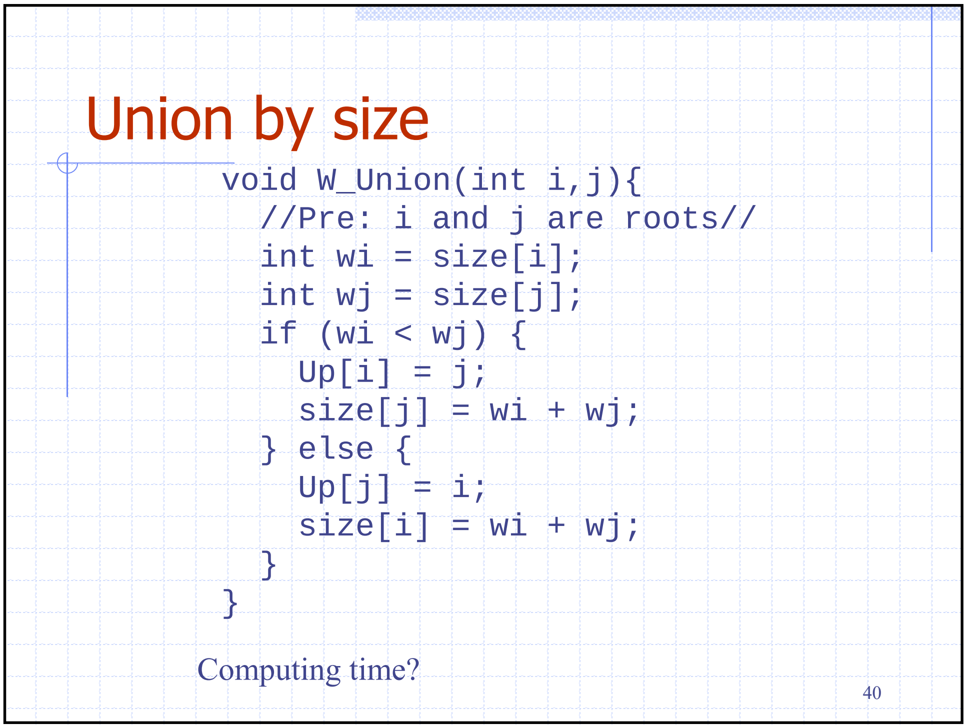
- Union by size (weight)
  - Always point the smaller tree to the root of the larger tree
- Union by height (rank)
  - Always point the shorter tree to the root of the higher tree



Array Implementation

	0	1	2	3	4	5	6
Up	-	0	-	6	6	4	-
size	2		1				4

	0	1	2	3	4	5	6
Up	-	0	-	6	6	4	-
height	1		0				2



Union by size

```
void W_Union(int i,j){
  //Pre: i and j are roots//
  int wi = size[i];
  int wj = size[j];
  if (wi < wj) {
    Up[i] = j;
    size[j] = wi + wj;
  } else {
    Up[j] = i;
    size[i] = wi + wj;
  }
}
```

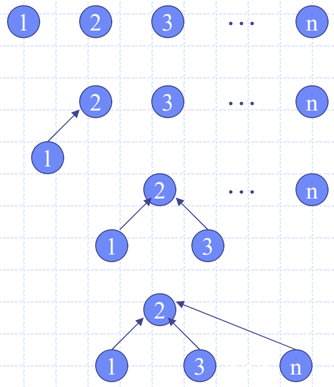
Computing time?

Union by height

```
void R_Union(int i,j){
  //Pre: i and j are roots//
  int ri = height[i];
  int rj = height[j];
  if (ri < rj) {
    Up[i] = j;
  } if (ri > rj) {
    Up[j] = i;
  } else { // ri == rj
    height[j]++; Up[j] = i;
  }
}
```

Computing time?

Example Again



Union(1,2)

Union(2,3)

:

:

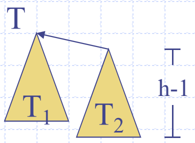
Union(n-1,n)

Find(1) constant time

Analysis of Union by size/height

- **Theorem**: With union by size/height an up-tree of height h has size at least 2^h.
- Proof by induction on height
  - Basis: h = 0. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all h′ < h.

A tree T of height h must have a child T_2 of height h-1



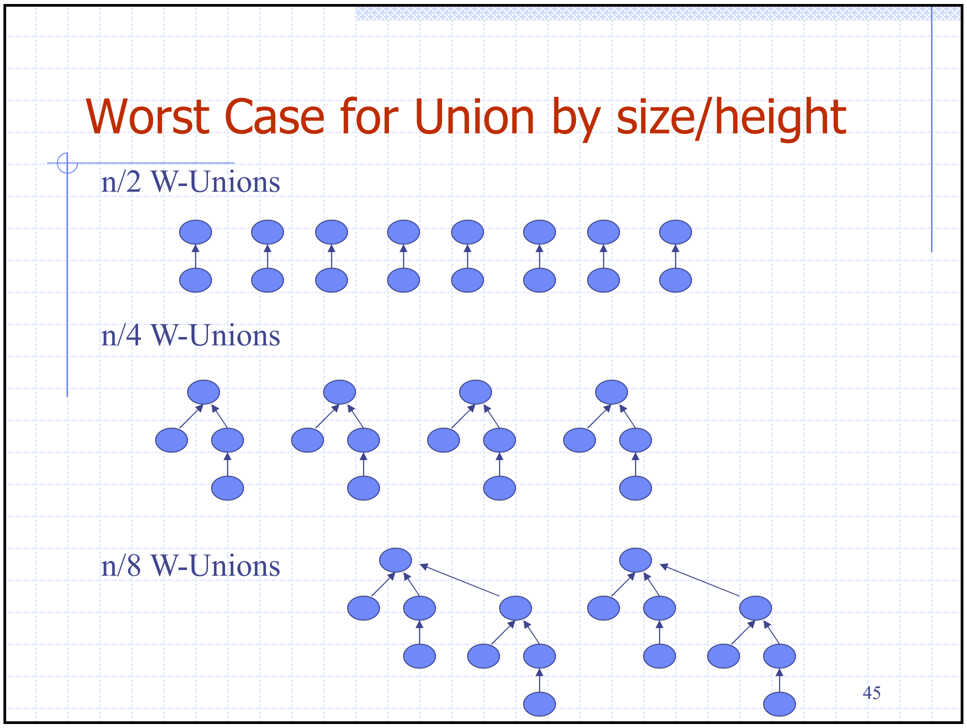
$$W(T_1) \underset{\text{Union by size/height}}{\geq} W(T_2) \underset{\text{Induction hypothesis}}{\geq} 2^{h-1}$$

$$W(T) = W(T_1) + W(T_2) \geq 2^{h-1} + 2^{h-1} = 2^h$$

Analysis of Union by size/height

- Let T be an up-tree of size n formed by union by size/height. Let h be its height.
- $n \geq 2^h$ (just proved)
- $\log n \geq h$

- Find(x) in tree T takes O(log n) time.
- Can we do better?

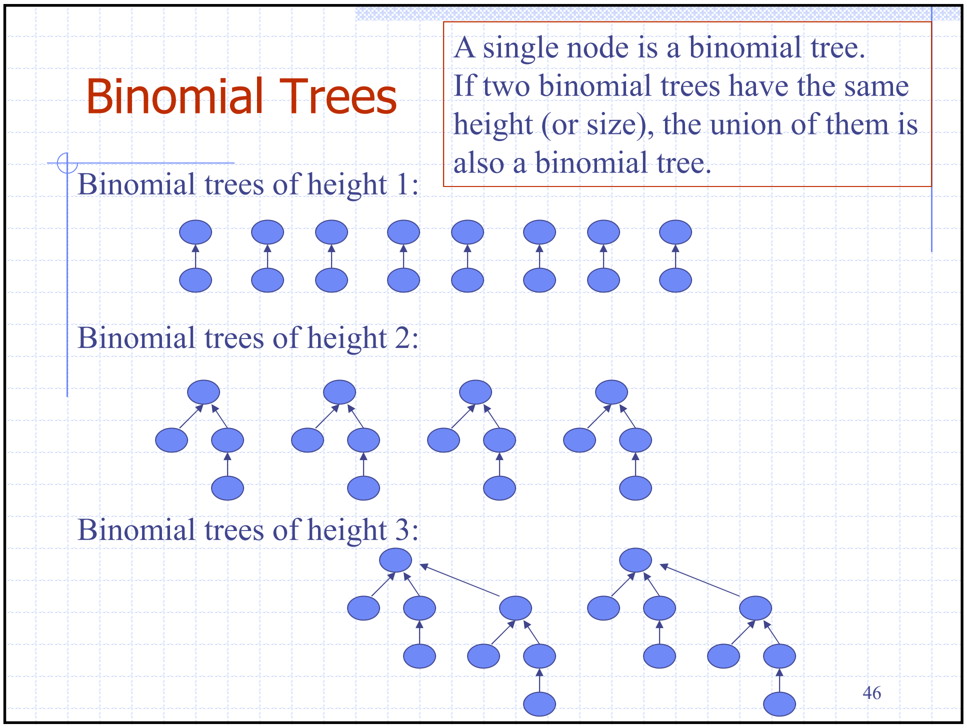


Worst Case for Union by size/height

n/2 W-Unions

n/4 W-Unions

n/8 W-Unions



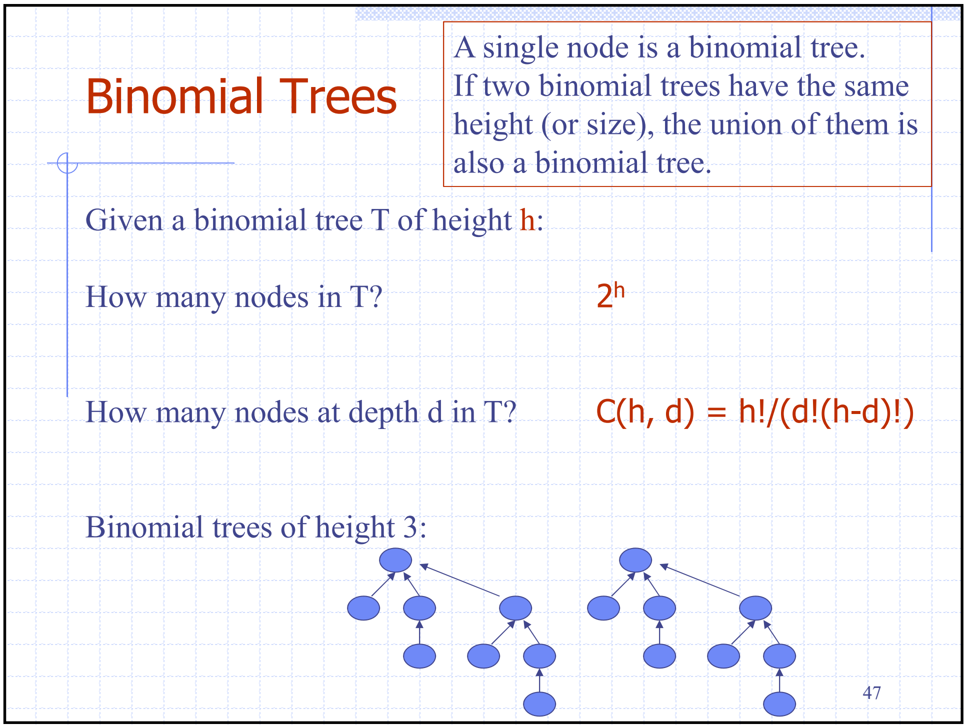
Binomial Trees

A single node is a binomial tree.
If two binomial trees have the same height (or size), the union of them is also a binomial tree.

Binomial trees of height 1:

Binomial trees of height 2:

Binomial trees of height 3:



Binomial Trees

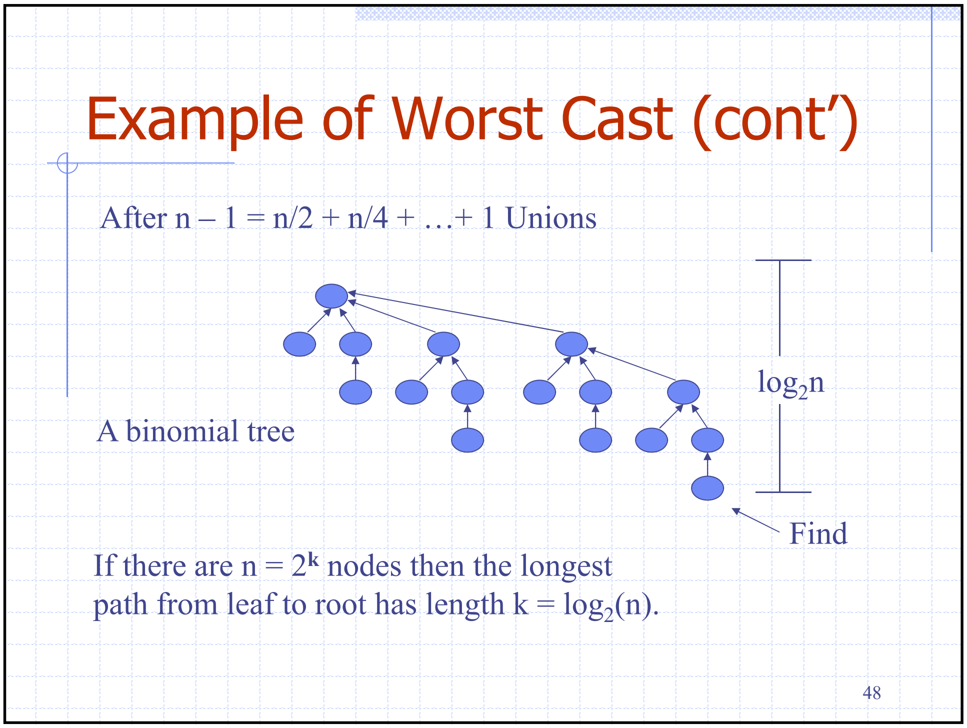
A single node is a binomial tree.
If two binomial trees have the same height (or size), the union of them is also a binomial tree.

Given a binomial tree T of height h:

How many nodes in T? 2^h

How many nodes at depth d in T? C(h, d) = h!/(d!(h-d)!)

Binomial trees of height 3:

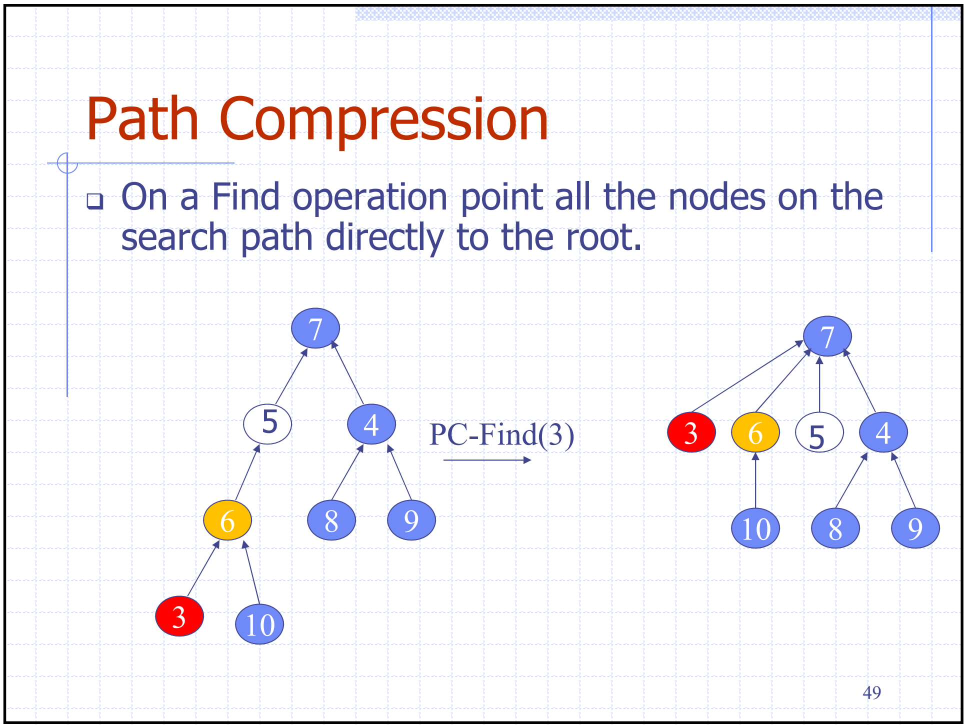


Example of Worst Cast (cont')

After n – 1 = n/2 + n/4 + …+ 1 Unions

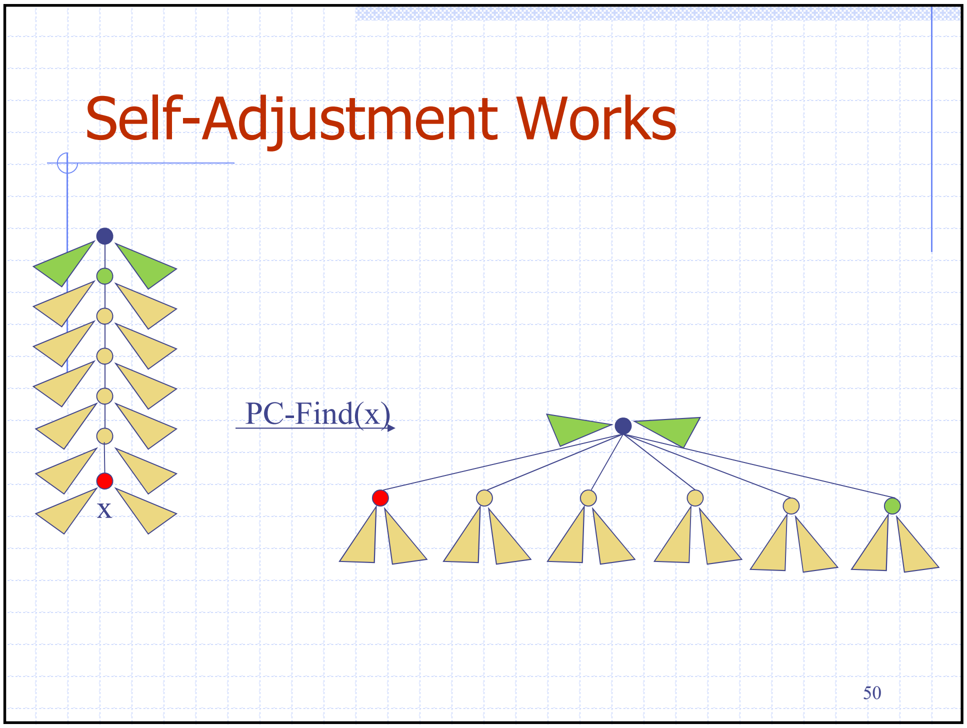
A binomial tree

If there are $n = 2^k$ nodes then the longest path from leaf to root has length $k = \log_2(n)$.



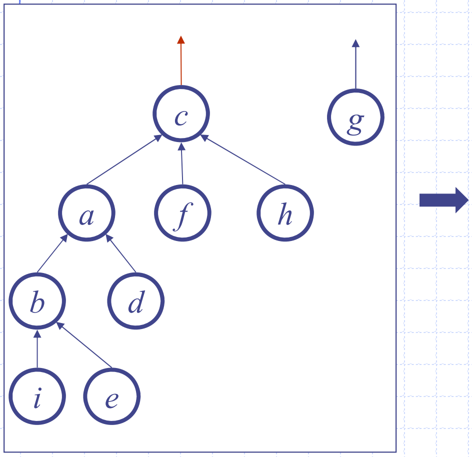
Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Self-Adjustment Works

Exercise: Draw the result of Find(e)



Path Compression Find

```
int PC_Find(int i) {
  int r = i;
  while (Up[r] != -1) //find root
    r = Up[r];
  if (i != r) { //compress path//
    int k = Up[i];
    while (k != r) {
      Up[i] = r;
      i = k;
      k = Up[k];
    }
  }
  return r;
}
```

Function Definition

Ackermann's function was defined in 1920s by German mathematician and logician Wilhelm Ackermann (1896-1962).

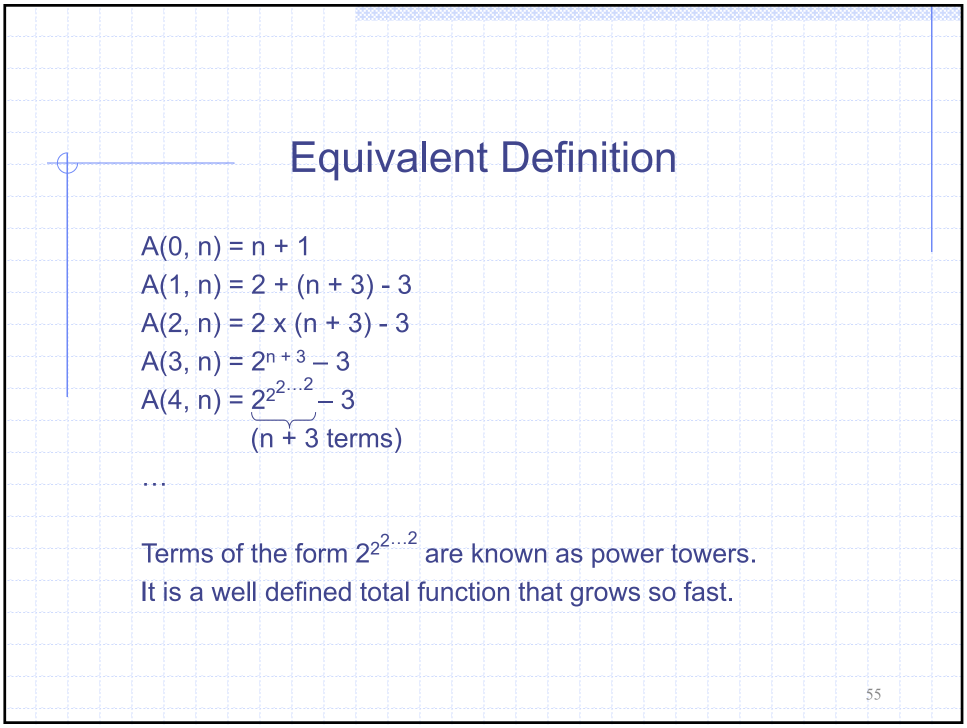
$A(m,n)$, $m,n \in \mathbf{N}$ such that,

$$A(0, n) = n + 1, \quad n \geq 0;$$
$$A(m,0) = A(m-1, 1), \quad m > 0;$$
$$A(m,n) = A(m-1, A(m, n-1)), \quad m, n > 0;$$

Example

$$\begin{aligned} A(1, 2) &= A(0, A(1, 1)) \\ &= A(0, A(0, A(1, 0))) \\ &= A(0, A(0, A(0, 1))) \\ &= A(0, A(0, 2)) \\ &= A(0, 3) \\ &= 4 \end{aligned}$$

Simple addition and subtraction!!



Equivalent Definition

$A(0, n) = n + 1$

$A(1, n) = 2 + (n + 3) - 3$

$A(2, n) = 2 \times (n + 3) - 3$

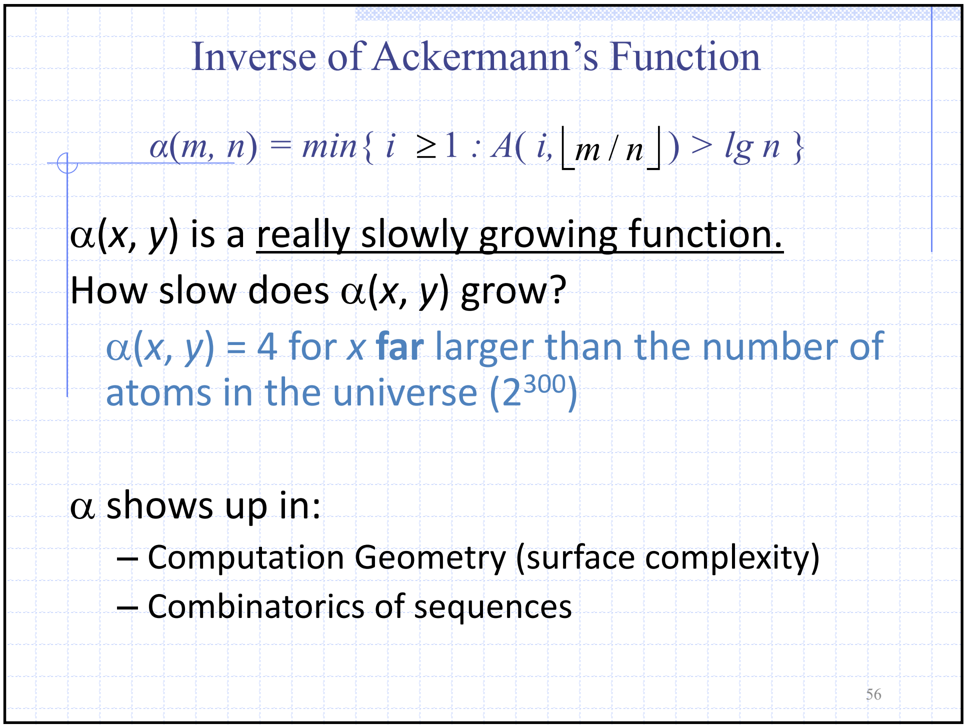
$A(3, n) = 2^{n + 3} - 3$

$A(4, n) = \underbrace{2^{2^{2\ldots 2}}}_{(n + 3 \text{ terms})} - 3$

…

Terms of the form $2^{2^{2\ldots 2}}$ are known as power towers.

It is a well defined total function that grows so fast.



Inverse of Ackermann's Function

$$\alpha(m, n) = \min\{\, i \geq 1 : A(i, \lfloor m / n \rfloor) > \lg n \,\}$$

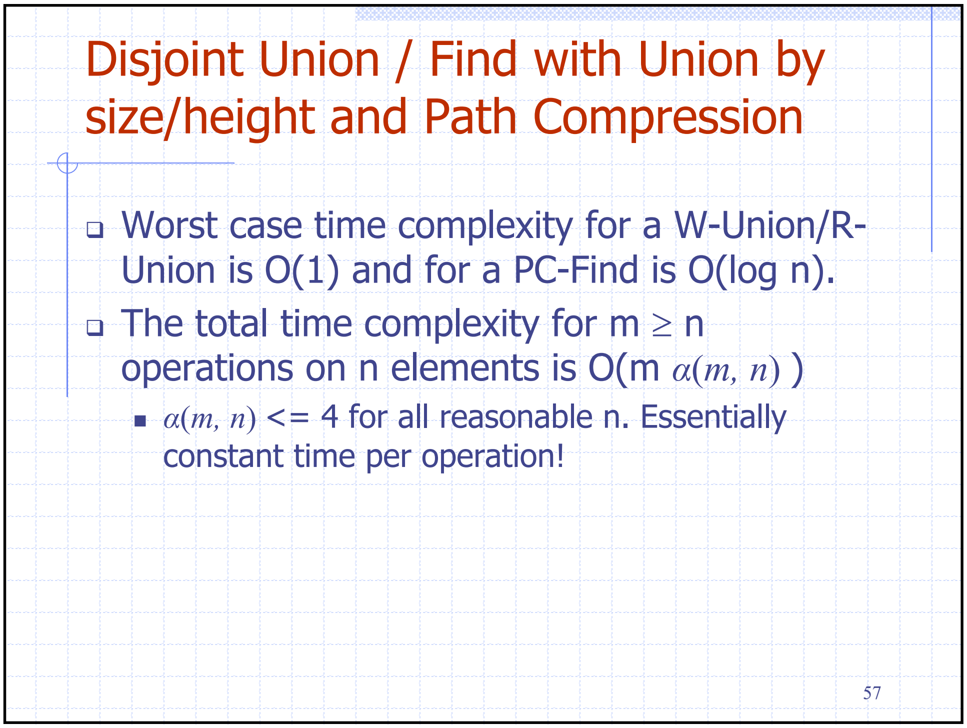
$\alpha(x, y)$ is a really slowly growing function.

How slow does $\alpha(x, y)$ grow?

- $\alpha(x, y) = 4$ for x **far** larger than the number of atoms in the universe (2^{300})

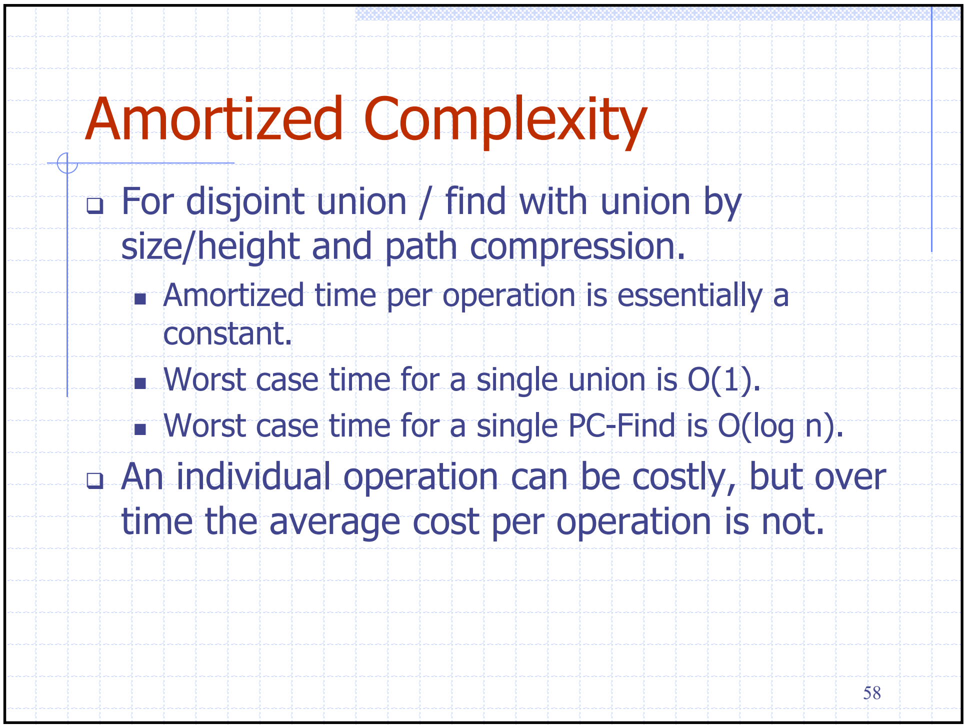
α shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences



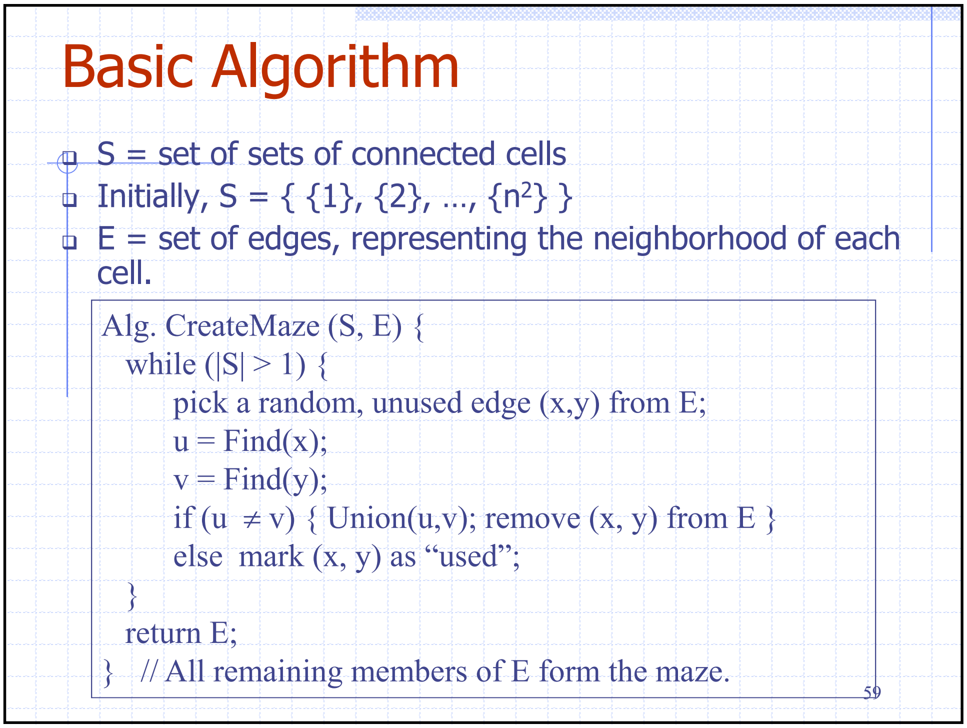
Disjoint Union / Find with Union by size/height and Path Compression

- Worst case time complexity for a W-Union/R-Union is O(1) and for a PC-Find is O(log n).
- The total time complexity for $m \geq n$ operations on n elements is O(m $\alpha(m, n)$)
    - $\alpha(m, n)$ <= 4 for all reasonable n. Essentially constant time per operation!



Amortized Complexity

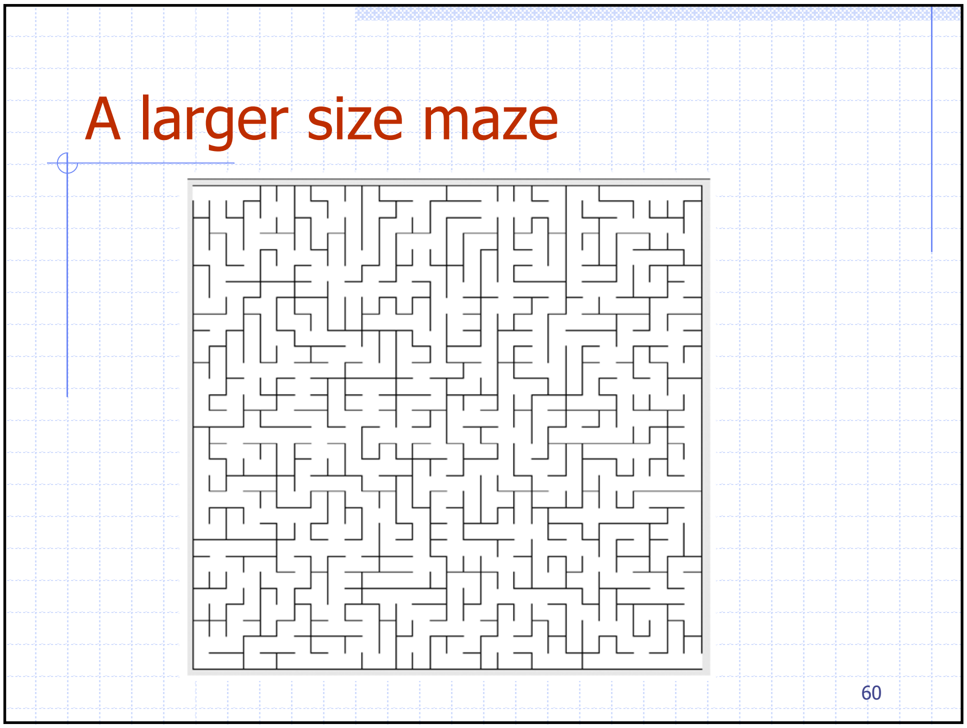
- For disjoint union / find with union by size/height and path compression.
    - Amortized time per operation is essentially a constant.
    - Worst case time for a single union is O(1).
    - Worst case time for a single PC-Find is O(log n).
- An individual operation can be costly, but over time the average cost per operation is not.



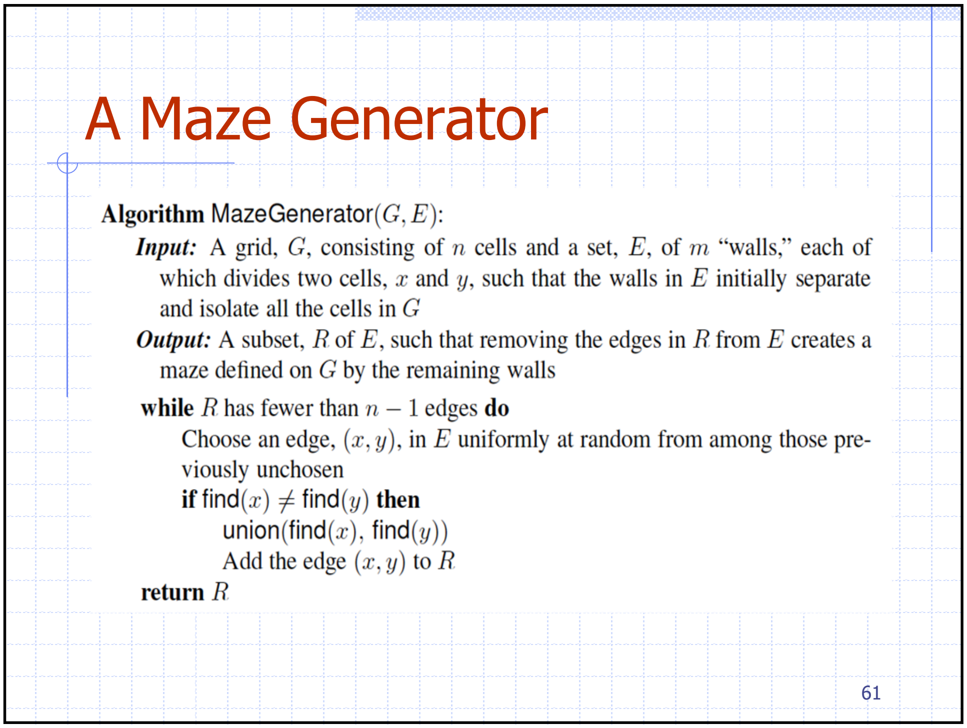
Basic Algorithm

- S = set of sets of connected cells
- Initially, S = { {1}, {2}, …, {n^2} }
- E = set of edges, representing the neighborhood of each cell.

```
Alg. CreateMaze (S, E) {
  while (|S| > 1) {
      pick a random, unused edge (x,y) from E;
      u = Find(x);
      v = Find(y);
      if (u ≠ v) { Union(u,v); remove (x, y) from E }
      else  mark (x, y) as "used";
  }
  return E;
}  // All remaining members of E form the maze.
```



A larger size maze



A Maze Generator

```
Algorithm MazeGenerator($G, E$):
   Input: A grid, $G$, consisting of $n$ cells and a set, $E$, of $m$ "walls," each of
      which divides two cells, $x$ and $y$, such that the walls in $E$ initially separate
      and isolate all the cells in $G$
   Output: A subset, $R$ of $E$, such that removing the edges in $R$ from $E$ creates a
      maze defined on $G$ by the remaining walls
   while $R$ has fewer than $n - 1$ edges do
      Choose an edge, $(x, y)$, in $E$ uniformly at random from among those previously unchosen
      if find($x$) $\neq$ find($y$) then
         union(find($x$), find($y$))
         Add the edge $(x, y)$ to $R$
   return $R$
```