CHAPTER 8

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

THE DISJOINT SET

- In computer science, a **disjoint-set** data structure, also called a union–find data structure or merge–find **set**, is a data structure that keeps track of a **set** of elements partitioned into a number of **disjoint**(non-overlapping) subsets.
- Simple Data Structure to maintain disjoint sets.
- The data structure is interesting from a theoretical point of view, because its analysis is extremely difficult.

EQUIVALENCE RELATION

- A relation R is defined on a Set S if for every pair of element (a,b)

  a, b ϵ S and a R b is either true or false.
- An equivalence relation is a relation R that satisfies three properties:

1. Reflexive: a R a for all a ϵ S
2. Symmetric: a R b if and only if b R a
3. Transitive a R b and b R c implies a R c

THE EQUIVALENCE CLASS

- The equivalence class of an element a ϵ S is the subset that contains all the elements that are related to a. Notice that equivalence classes form a partition of S
- The equivalence class forms a partition of S, every element is in exactly one equivalence class.
  - To decide a ~ b we need only to check if a and b are in the same subset.

THE DYNAMIC EQUIVALENCE PROBLEM

- Given an equivalence relation ~ the problem is decide for any a and b if a ~ b.
- The input is initially a collection of N sets, each with one element. so $S_i \cap S_j = \emptyset$ this makes the set disjoint.
- Two operations
  - find : which returns the name of the set containing the element
  - Union : merges two equivalence class into a new equivalence class.
- This algorithm is called the disjoint set union / find algorithm.

Key Points

- We do not perform any operation comparing the values of elements but merely require knowledge of their location.
- All elements are numbered sequentially from 0 to N-1, initially we have $S_i = \{i\}$ for i = 0 through N-1
- The name of the set returned by find is actually fairly arbitrary. All that matters is find(a) == find(b) is true if and only if a and b are in the same set.

BASIC DATA STRUCTURE

- Represent each set by a tree.
- Name of the set is given by the root.
- This tree is stored implicitly in an array. Each entry s[i] in the array represents the parent of element i. if i the root then s[i] = -1

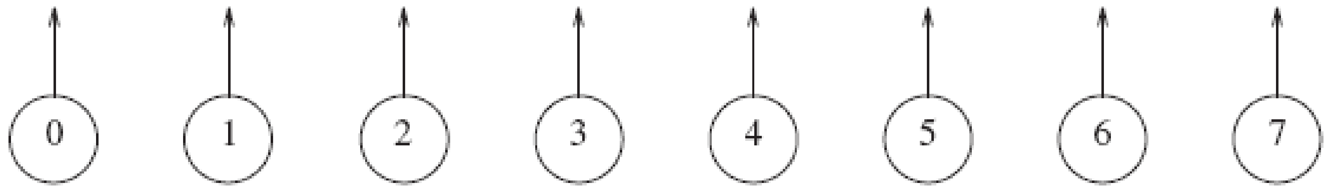


Figure 8.1 Eight elements, initially in different sets

Example

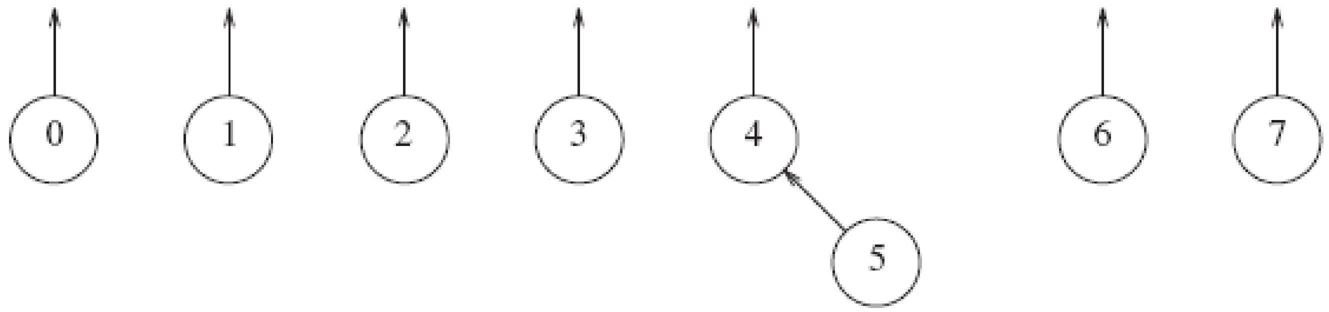


Figure 8.2 After `union(4,5)`

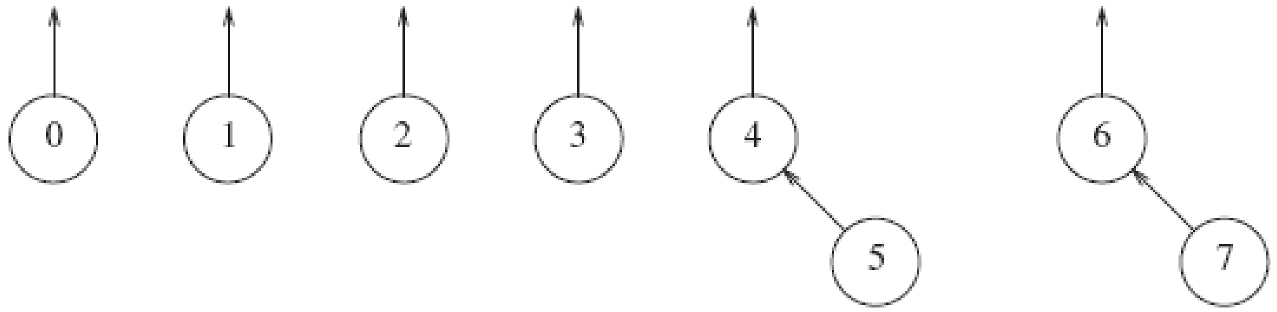


Figure 8.3 After `union(6,7)`

Example

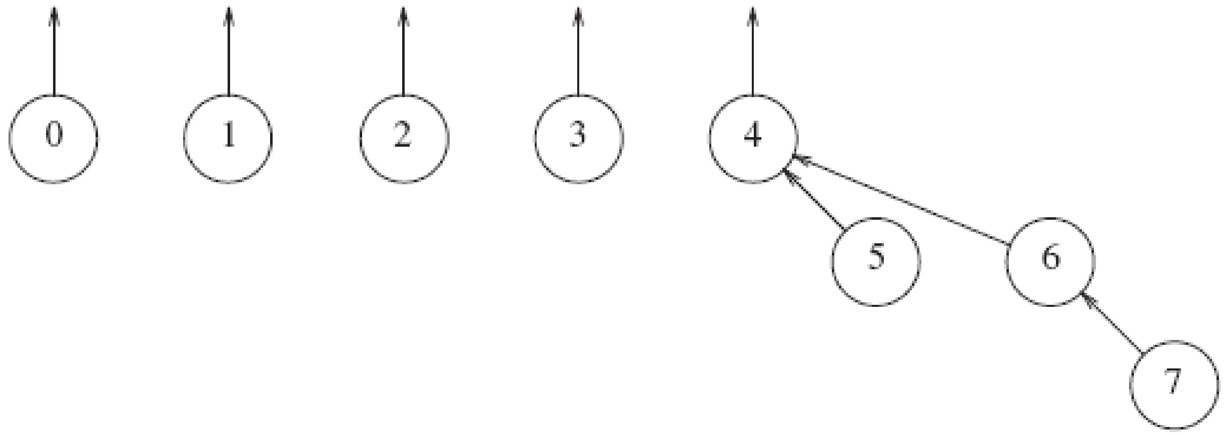
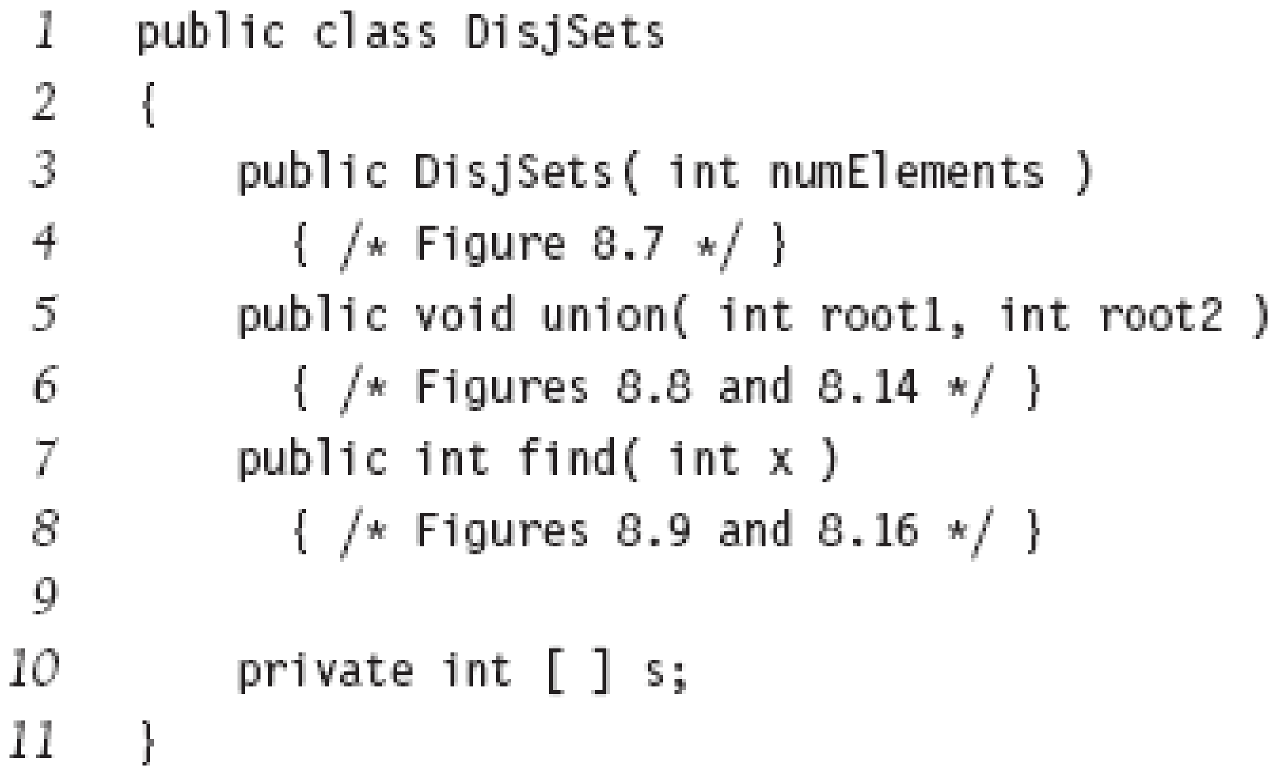


Figure 8.4 After `union(4,6)`

–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

Figure 8.5 Implicit representation of previous tree

BASIC CLASS

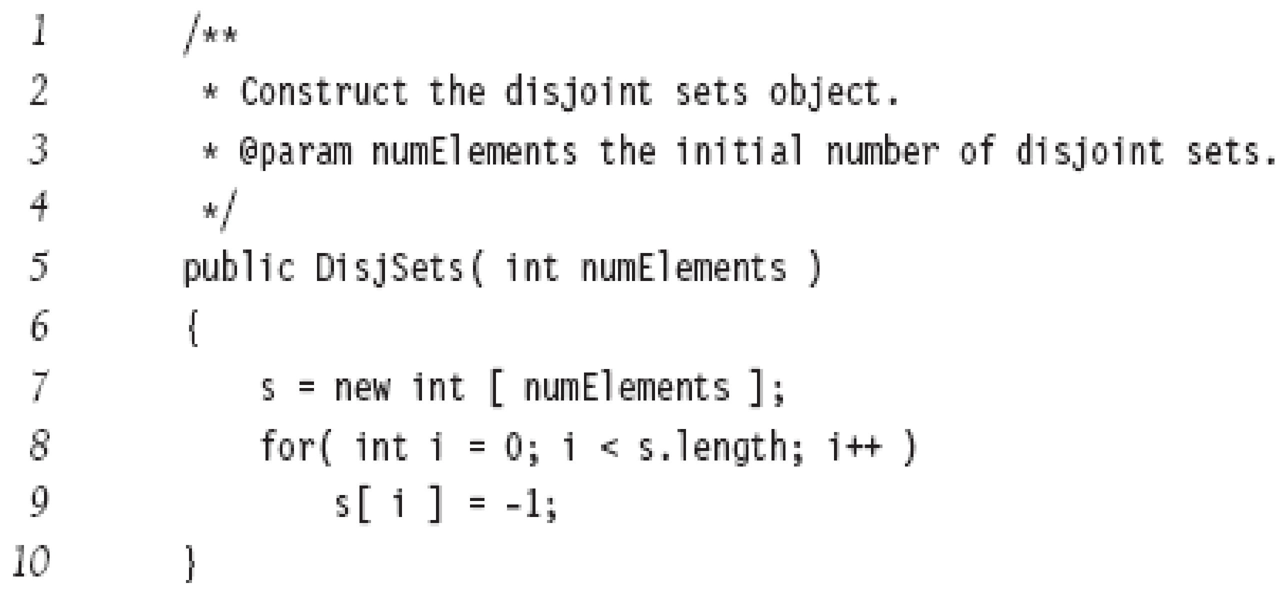


```
public class DisjSets
{
    public DisjSets( int numElements )
      { /* Figure 8.7 */ }
    public void union( int root1, int root2 )
      { /* Figures 8.8 and 8.14 */ }
    public int find( int x )
      { /* Figures 8.9 and 8.16 */ }

    private int [ ] s;
}
```

Figure 8.6 Disjoint set class skeleton

BASIC CLASS



```
/**
 * Construct the disjoint sets object.
 * @param numElements the initial number of disjoint sets.
 */
public DisjSets( int numElements )
{
    s = new int [ numElements ];
    for( int i = 0; i < s.length; i++ )
        s[ i ] = -1;
}
```

Figure 8.7 Disjoint set initialization routine

union Method

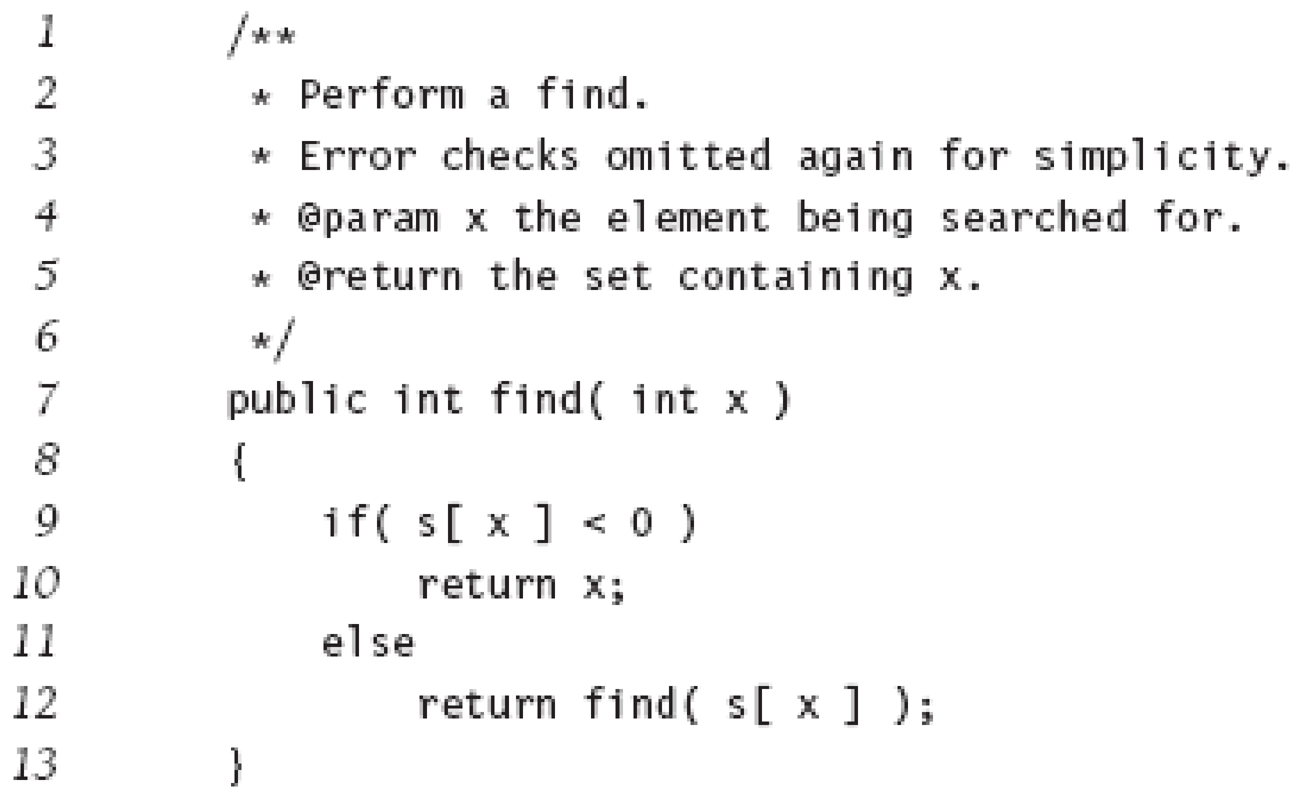
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 */
public void union( int root1, int root2 )
{
    s[ root2 ] = root1;
}
```

Figure 8.8 union (not the best way)

find Method

- A find(x) is performed by returning the root of the tree containing x.
- The time to perform this operation is proportional to depth of the node representing x.
- So the worst case running time of find is $\theta(N)$

find Method



```
/**
 * Perform a find.
 * Error checks omitted again for simplicity.
 * @param x the element being searched for.
 * @return the set containing x.
 */
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}
```

Figure 8.9 A simple disjoint set find algorithm

SMART UNION ALGORITHM

- A simple improvement is always to make the smaller tree a subtree of the larger, this approach is called union by size.
- We can prove that if unions are done by size, the depth of any node is never more than log N. This implies that the running time of find operation is O(log N)

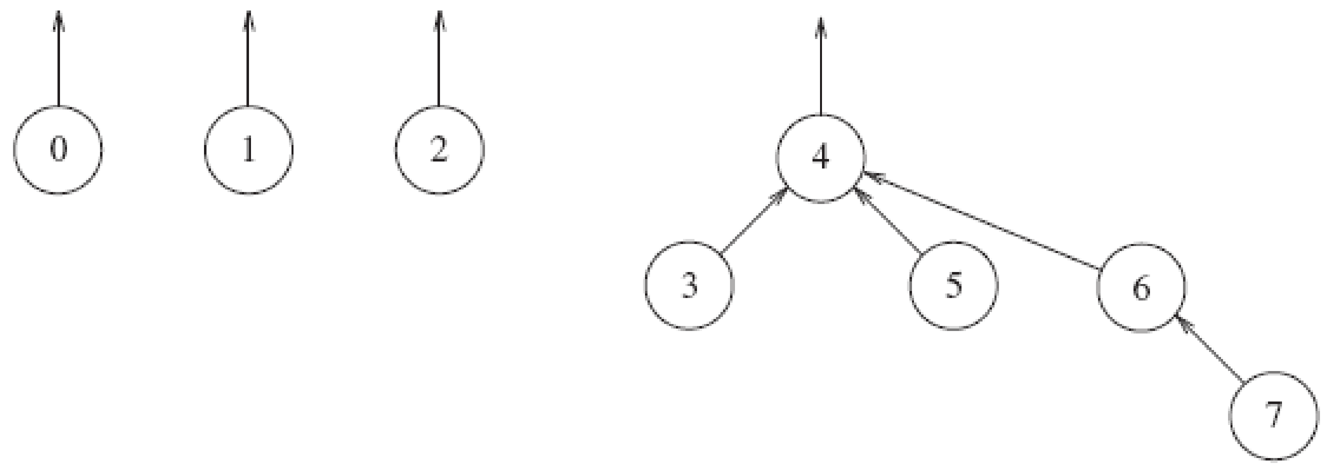


Figure 8.10 Result of union-by-size

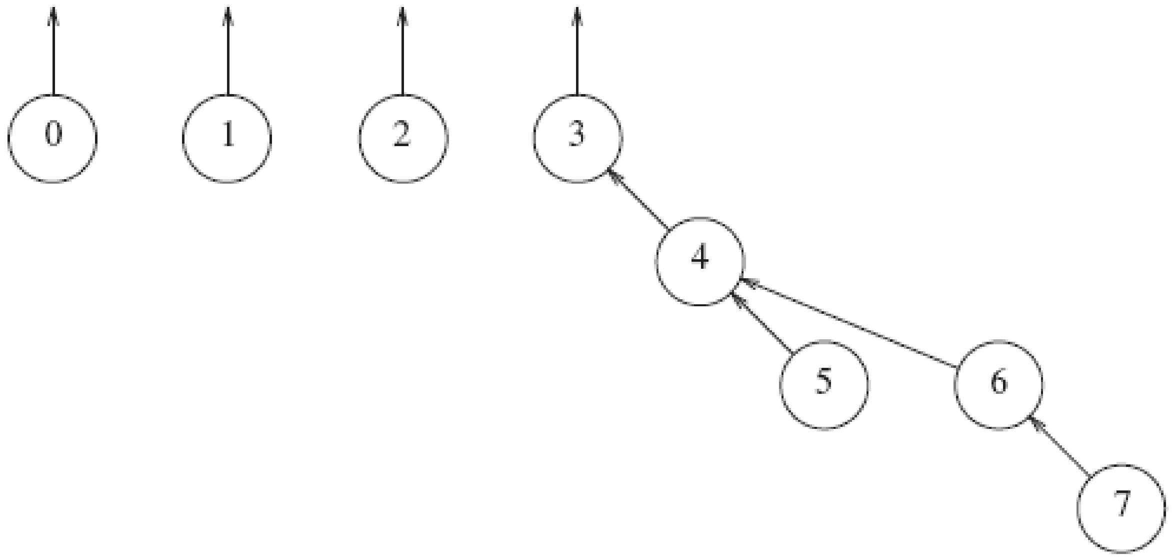


Figure 8.11 Result of an arbitrary `union`

- The worst case tree possible after 16 unions and is obtained if all the between equal-sized trees

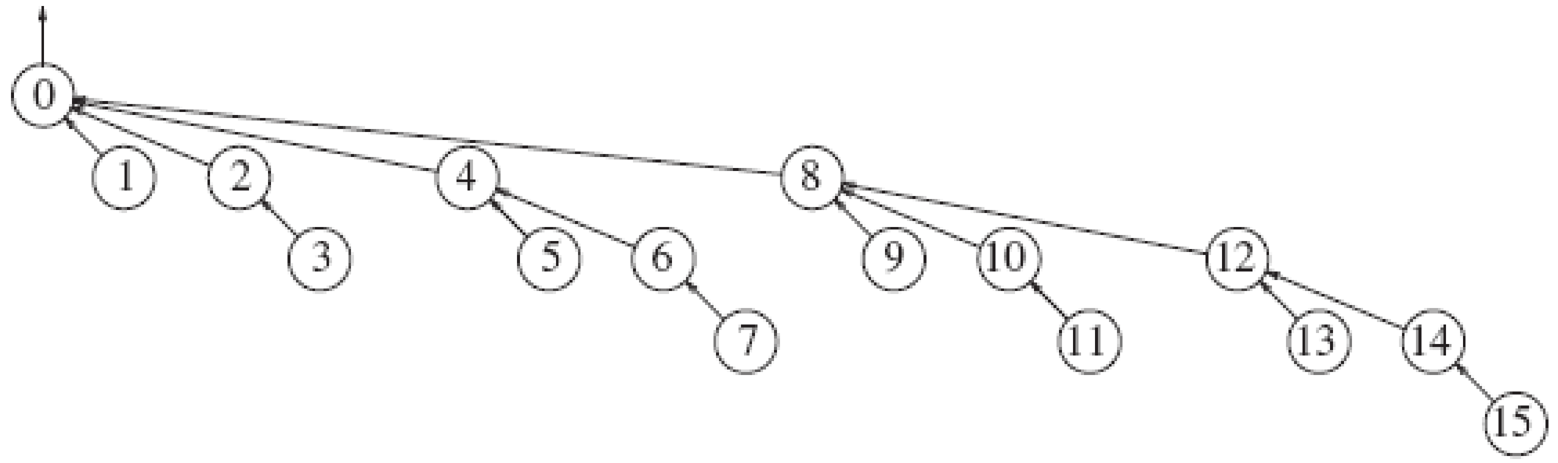
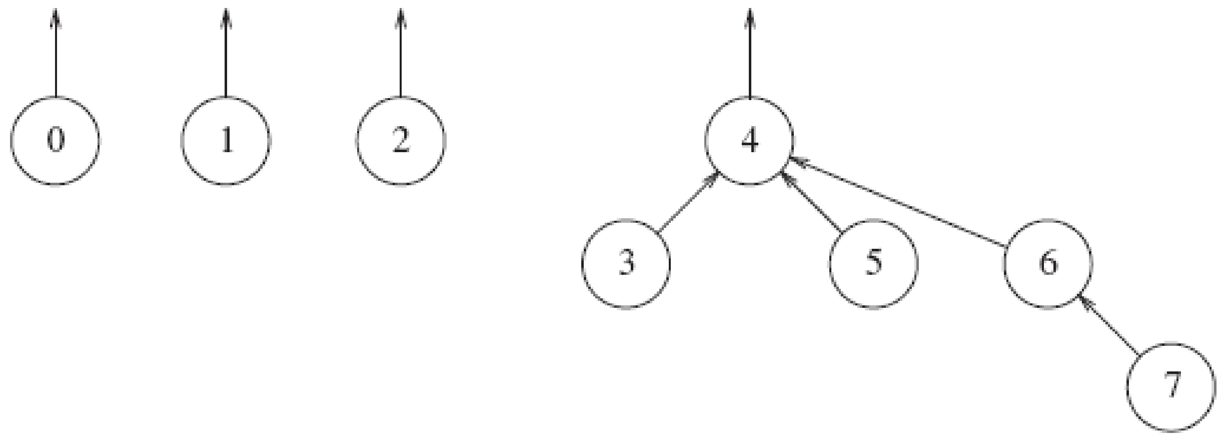


Figure 8.12 Worst-case tree for $N = 16$

Union by Size or by Height

- To implement this strategy, we need to keep track of the size of each tree.
- Thus initially the array representation of the tree is all -1. When the union is performed the new size of sum of the old.
- Alternate implementation is union by height. Stores negative of the height and minus an additional 1.



–1	–1	–1	4	–5	4	4	6
0	1	2	3	4	5	6	7

–1	–1	–1	4	–3	4	4	6
0	1	2	3	4	5	6	7

Figure 8.13 Forest with implicit representation for union-by-size and union-by-height

Union by Height

```
/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 */
public void union( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
        s[ root1 ] = root2;        // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;        // Make root1 new root
    }
}
```

Figure 8.14 Code for union-by-height (rank)

PATH COMPRESSION

- Path compression is performed during find operation.
- Effect of the operation is that every node after find(x) on the path from x to the root has its parent changed to the root.
- When union is done arbitrarily, path compression is a good idea, because there is an abundance of deep nodes and these are brought near to the root.

Path compression after find(14)

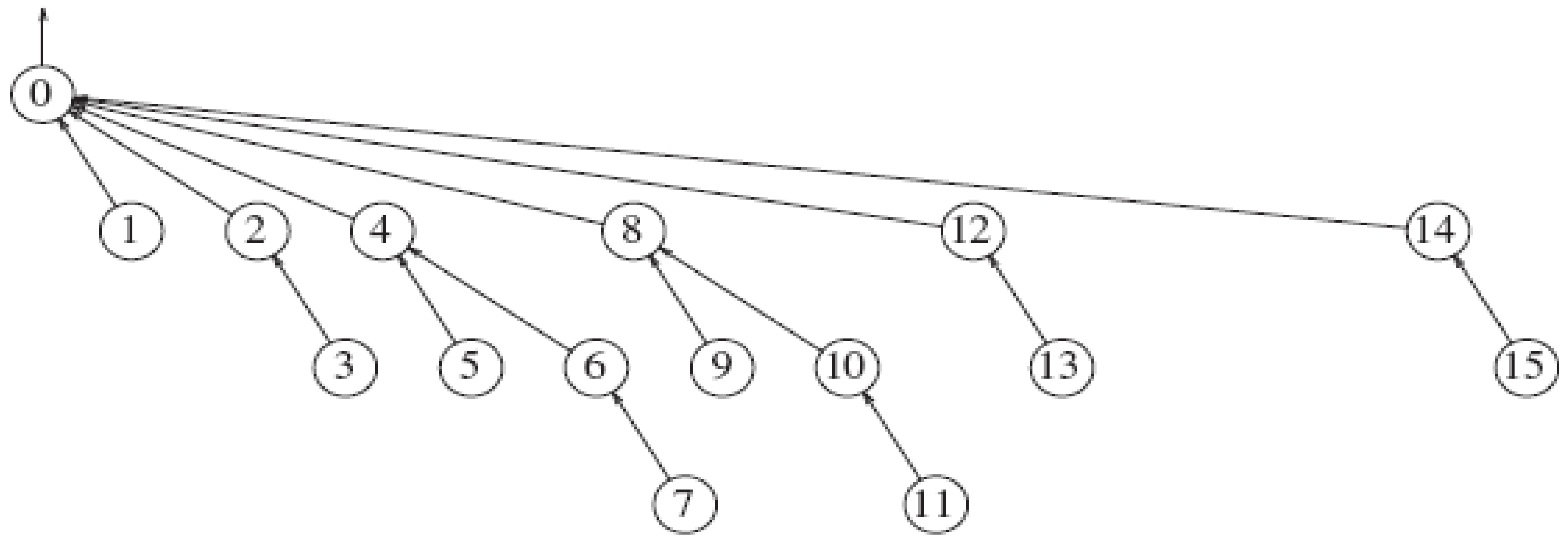
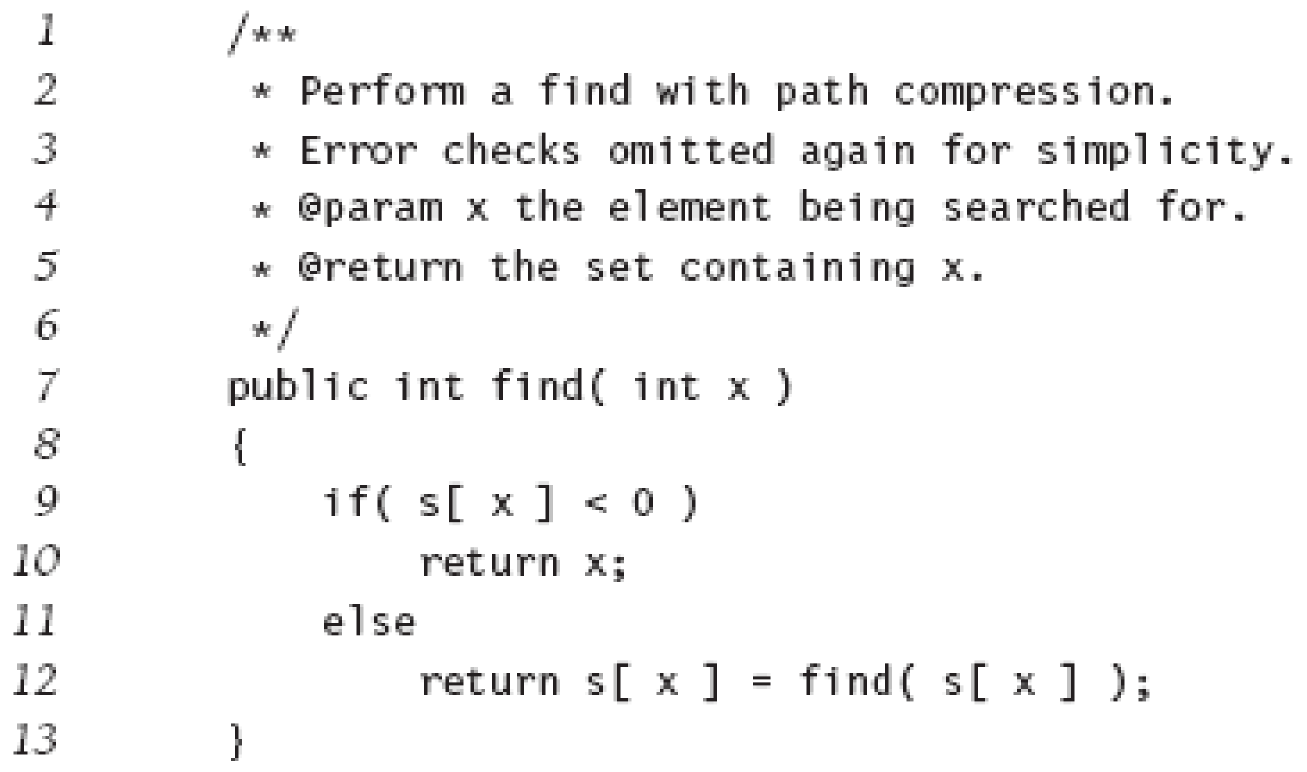


Figure 8.15 An example of path compression

PATH COMPRESSION



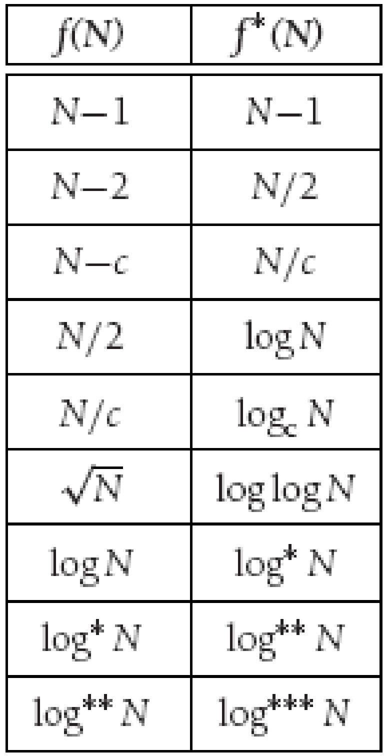
```
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * @param x the element being searched for.
 * @return the set containing x.
 */
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

Figure 8.16 Code for the disjoint set `find` with path compression

WORST CASE FOR UNION BY RANK AND PATH COMPRESSION

- The time required in worst case is θ(Mα(M,N)) provided M> =N and α(M,N) is an incredibly slowly growing function.
- Slowly growing function is a recurrence function that iteratively reaches 1.
- $T(N) = \begin{cases} 0 & N <= 1 \\ T(\lfloor f(N) \rfloor & N > 1 \end{cases}$
- T(N) represents the number of times starting at N, that we must iteratively apply *f(N)* until we reach 1. *f(N)* is a function that reduces N. Solution to the equation T(N) is called *f*(N)*

SLOWLY GROWING FUNCTIONS



$f(N)$	$f^*(N)$
$N-1$	$N-1$
$N-2$	$N/2$
$N-c$	N/c
$N/2$	$\log N$
N/c	$\log_c N$
$\sqrt{N}$	$\log \log N$
$\log N$	$\log^* N$
$\log^* N$	$\log^{**} N$
$\log^{**} N$	$\log^{***} N$

Figure 8.17 Different values of the iterated function

O(M log*N) Bound

- Text book has an elaborate proof, wherein they reduce the problem to a different partial find problem.

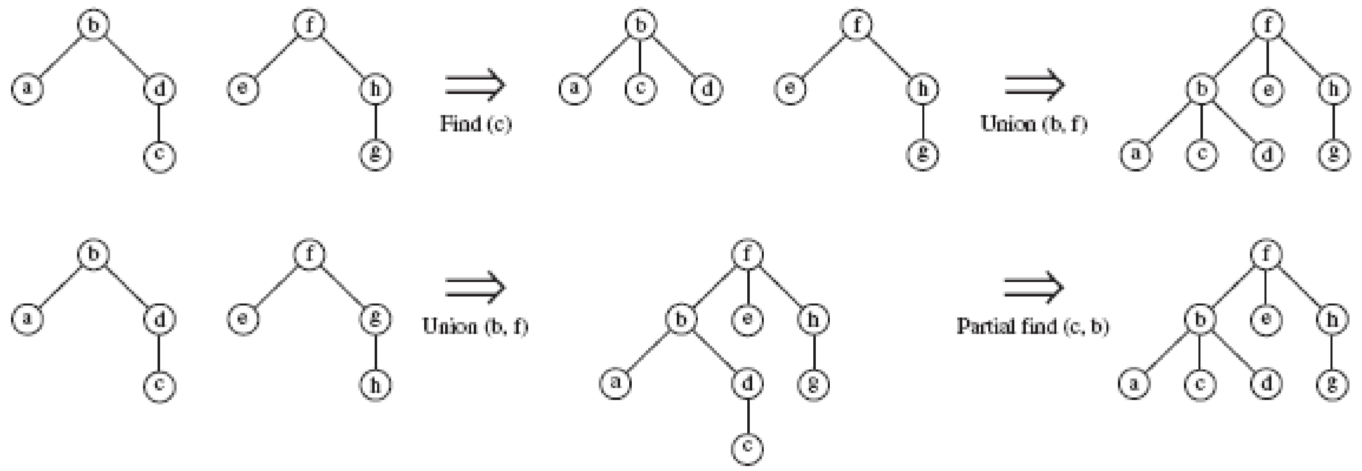


Figure 8.19 Sequences of union and find operations replaced with equivalent cost of union and partial find operations

AN APLICATION

- An example of union find data structure is the generation of maze.

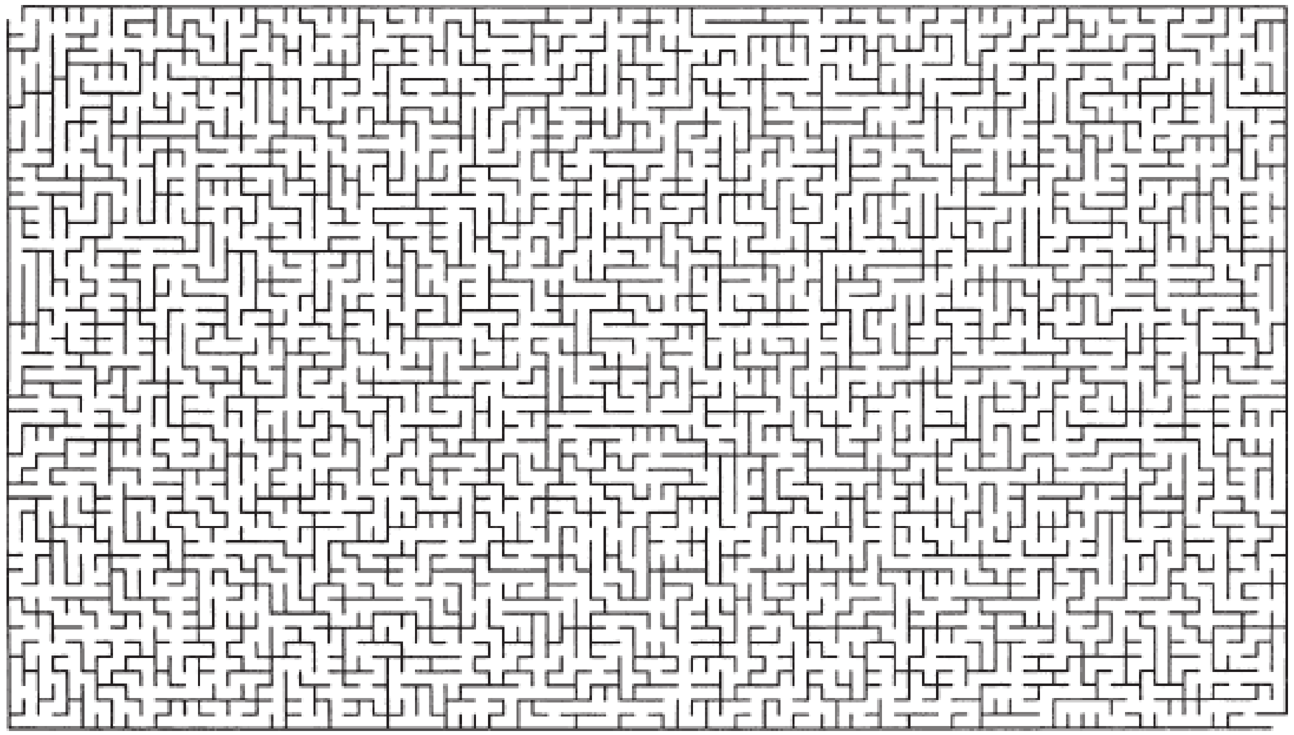


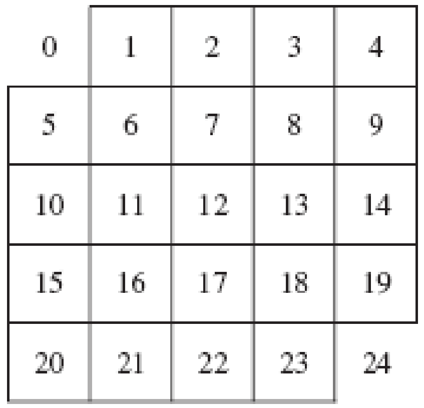
Figure 8.25 A 50-by-88 maze

ALGORITHM

- Start with walls everywhere expect for the entrance and exit.
- Choose a wall randomly and knock it down, if the cells that the wall separates are not already connected to each other.
- Repeat until starting and ending cells are connected.
- To generate false leads, knock down walls until every cell is reachable from every other cell

ILLUSTRATION

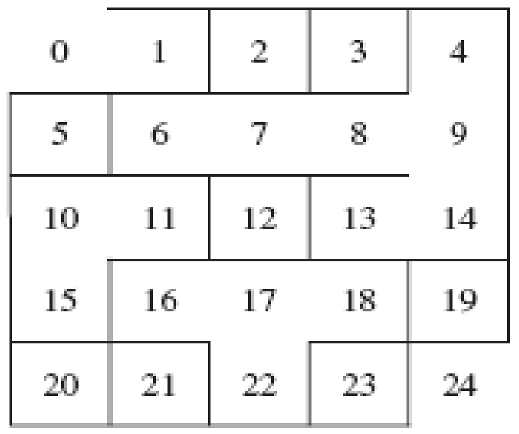
- Example: 5 by 5 maze. Union/find data structure is used to represent sets of cells that are connected to each other. Initially walls are everywhere, and each cell is in its own equivalence class.



{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

Figure 8.26 Initial state: all walls up, all cells in their own set

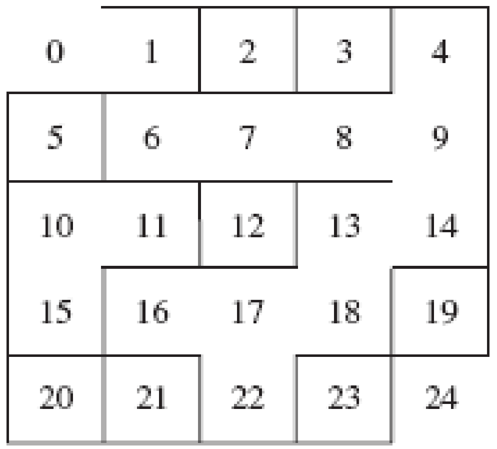
ILLUSTRATION



{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

Figure 8.27 At some point in the algorithm: Several walls down, sets have merged; if at this point the wall between 8 and 13 is randomly selected, this wall is not knocked down, because 8 and 13 are already connected

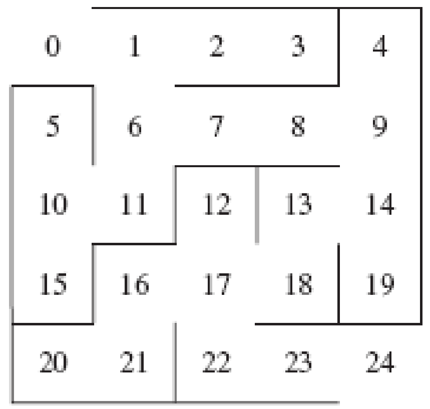
Suppose that cells 18 and 13 are randomly targeted next. By performing two find operation we see that these are in different sets. so they can be joined by an union



{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}

Figure 8.28 Wall between squares 18 and 13 is randomly selected in Figure 8.22; this wall is knocked down, because 18 and 13 are not already connected; their sets are merged

Eventually all elements are in the same set.



{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

Figure 8.29 Eventually, 24 walls are knocked down; all elements are in the same set

RUNNING TIME

- The running time of the algorithm is dominated by the union find cost.
- The number of find operation is propositional to the number of cells.
- If N is the number of cells, there can be only twice the number of walls to be knocked down. Each wall required 2 find operation so we have roughly between 2N to 4N finds.
- Overall running time of the algorithm is O(N log*N)