Priority Queues

- It is a variant of queues
- Each item has an associated priority value.
- When inserting an item in the queue, the priority value is also provided for it.
- The data structure provides a method to delete the item with the highest priority.

Priority Queues

```
package weiss.nonstandard;

// PriorityQueue interface
//
// ******************PUBLIC OPERATIONS*********************
// Position insert( x )   --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// int size( )            --> Return size
// void decreaseKey( p, v)--> Decrease value in p to v
// ******************ERRORS********************************
// Throws UnderflowException for findMin and deleteMin when empty
```

Applications of Priority Queues

- Implementing job queues in computer systems, or queues in an emergency room in a hospital.
- Implementing Dijkstra's shortest path algorithm

Binary Search Trees

```
// BinarySearchTree class
//
// ***************PUBLIC OPERATIONS******************
// void insert( x )       --> Insert x       O(h)
// void remove( x )       --> Remove x    O(h)
// void removeMin( )      --> Remove minimum item       O(h)
// Comparable find( x )   --> Return item that matches x O(h)
// Comparable findMin( )  --> Return smallest item       O(h)
// Comparable findMax( )  --> Return largest item        O(h)
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
```

The height of the tree = ?

O(log n) on the average, and O(n) on the worst case.

Balanced Binary Search Trees

- There can be many binary search trees for the same data.
- Not all of them have the same characteristics. Some are better than the others.
- Worst case height of the binary search tree is O(log n)
- More work when you insert or delete, because you try to fix any imbalances in the tree caused by the change.
- No change when you search, because the tree is still a binary search tree.

Figure 19.19

(a) The balanced tree has a depth of □log *N*; (b) the unbalanced tree has a depth of *N* – 1.

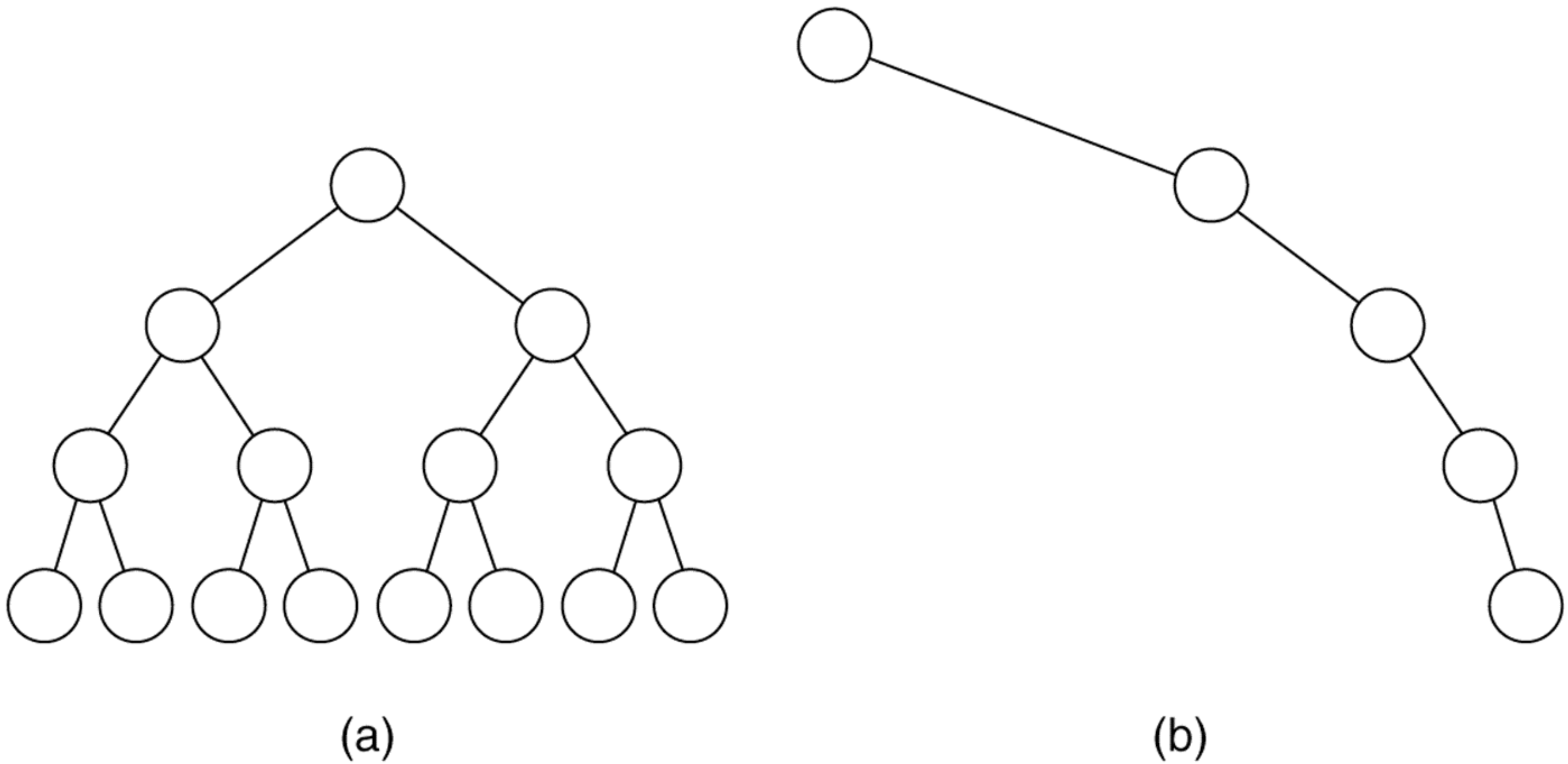


Figure 19.20

Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree shown in part (c) is twice as likely to result as any of the others.

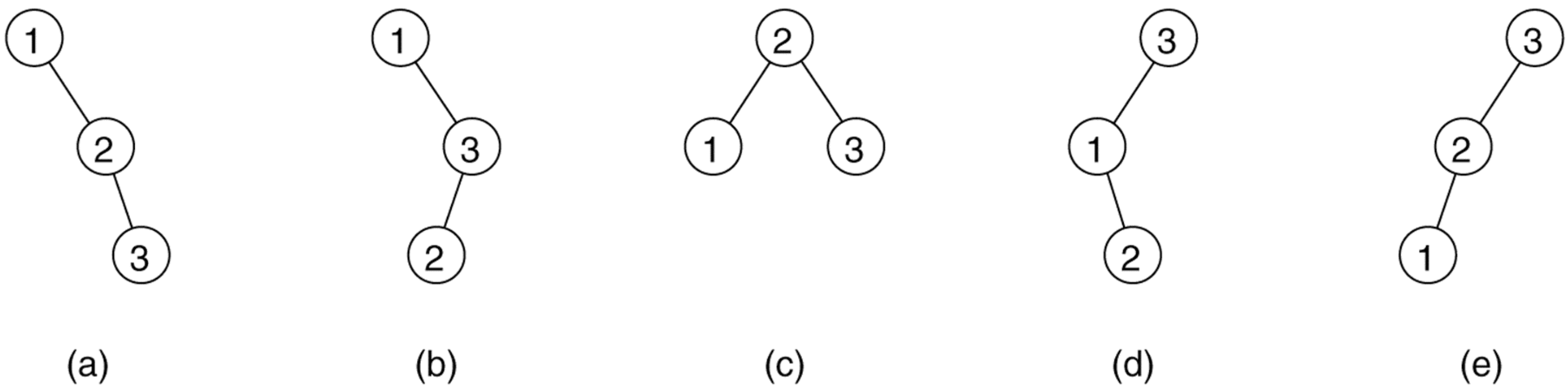


Figure 19.22
Minimum tree of height *H*

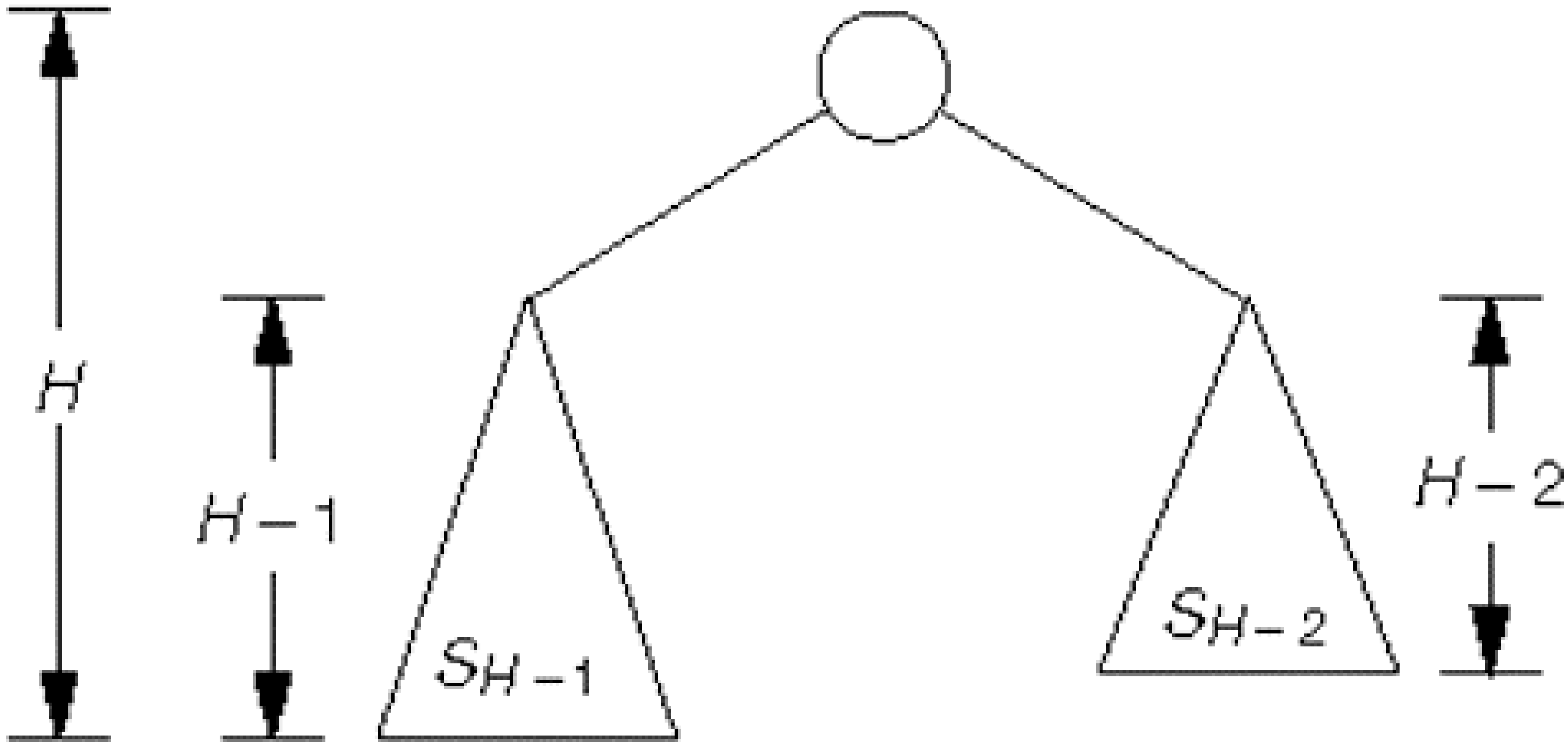
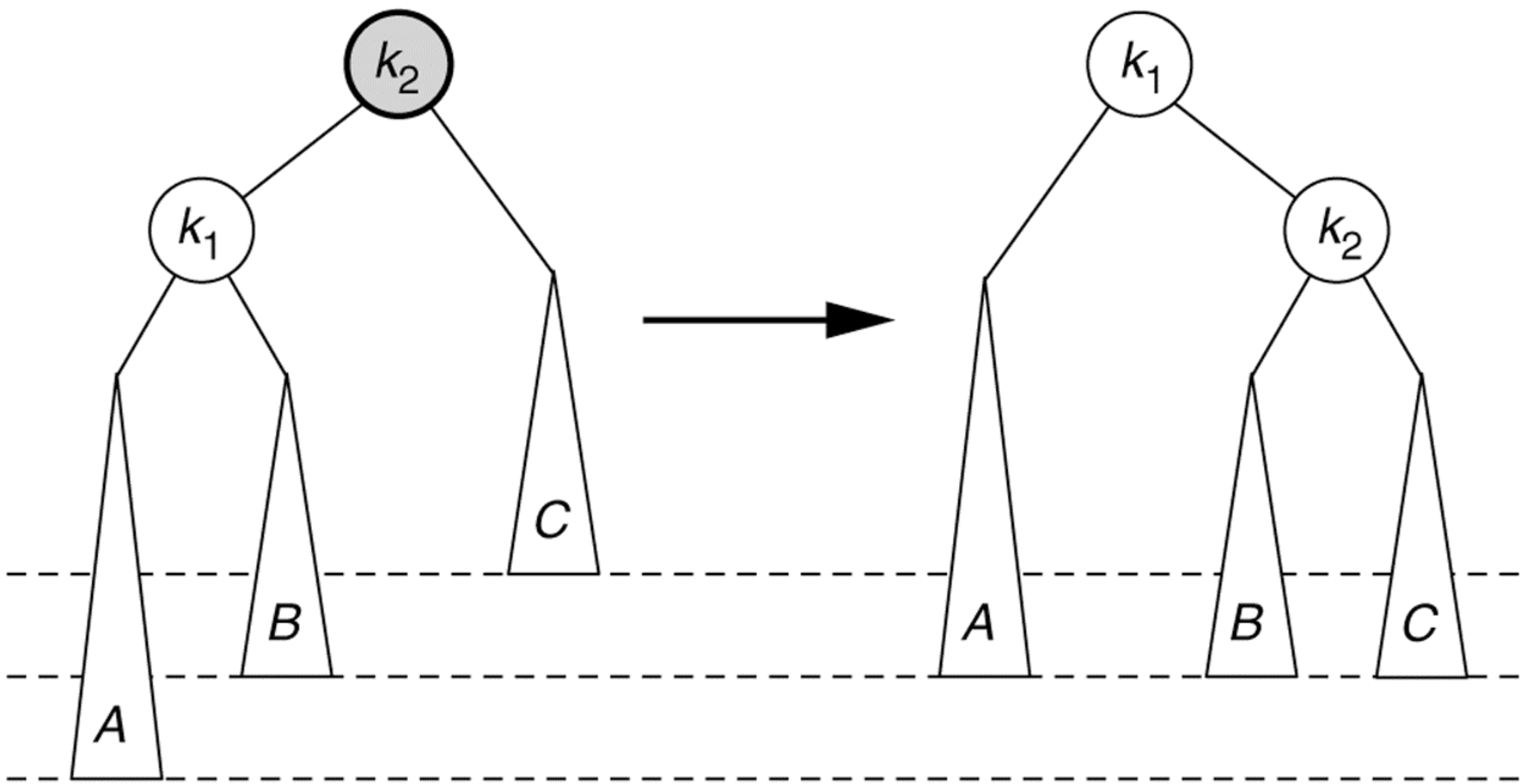


Figure 19.23
Single rotation to fix case 1

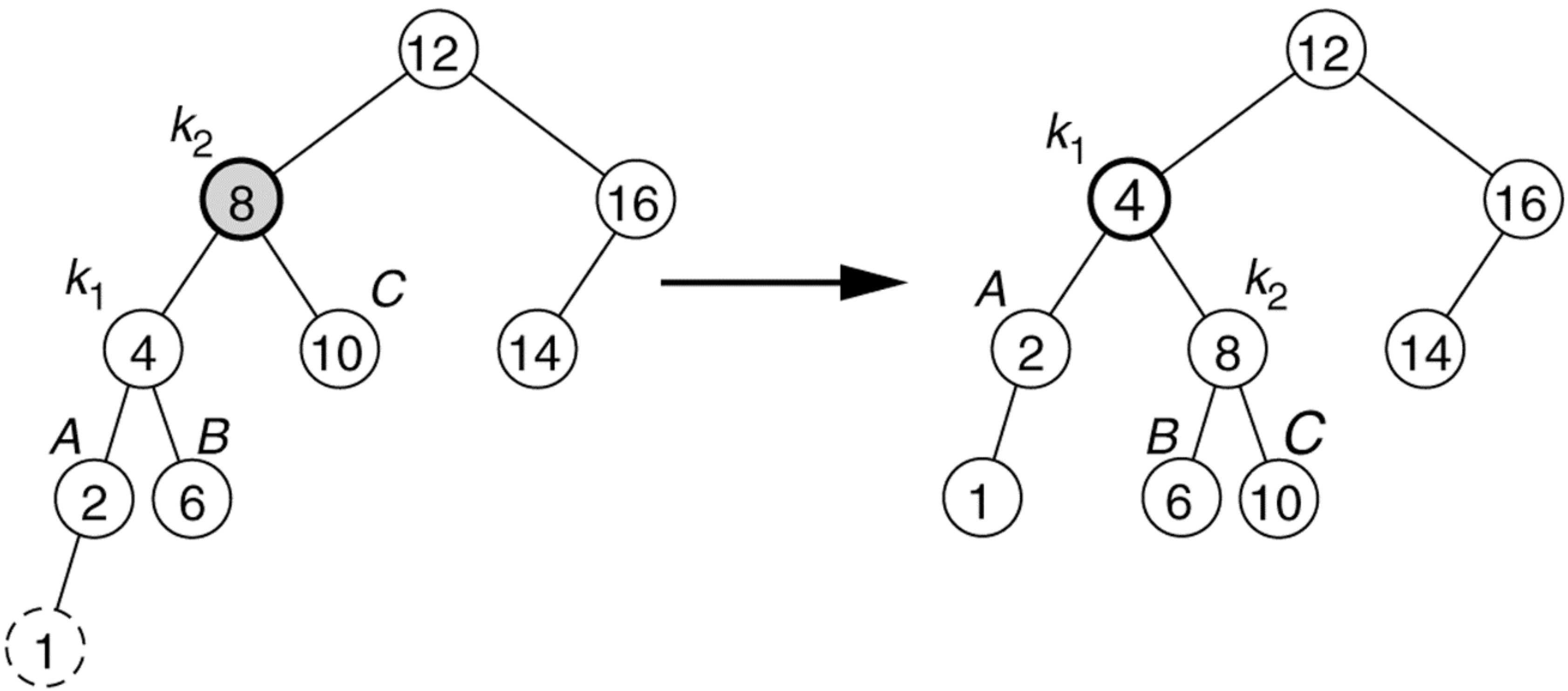


(a) Before rotation

(b) After rotation

Figure 19.25

Single rotation fixes an AVL tree after insertion of 1.



(a) Before rotation

(b) After rotation

Applications of Balanced BSTs

- Anywhere where you want to store a dynamic set of items.
- Static means that the items in the set are fixed at the start and do not change. To implement static sets, a sorted array would do well.
- Dynamic means that the items in the set are changing (inserts and deletes).
- Balanced BSTs guarantee the worst-case performance of the data structure.

Disjoint Set Union-Find Data Structure

- Maintains disjoint sets.
- Main operations:
  - Union: unions two given sets
  - Find: finds set containing given item

Disjoint Set Union-Find Data Structure

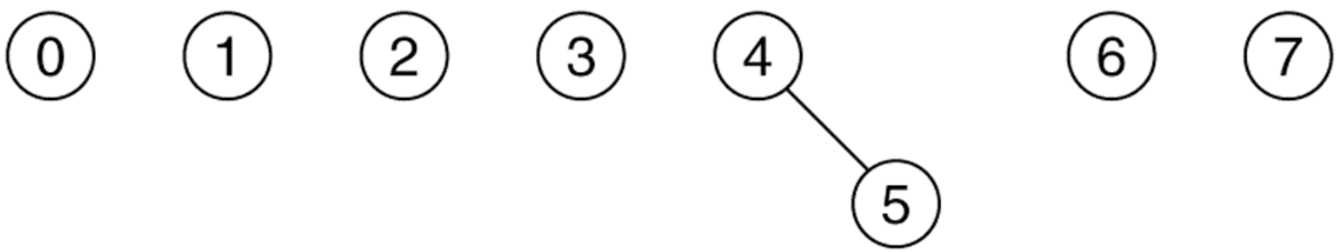
```
public class DisjointSets
{
   public DisjointSets( int numElements )
   public void union( int root1, int root2 )
   public int find( int x )
   private int [ ] s;
}
```

Figure 24.12

A forest and its eight elements, initially in different sets

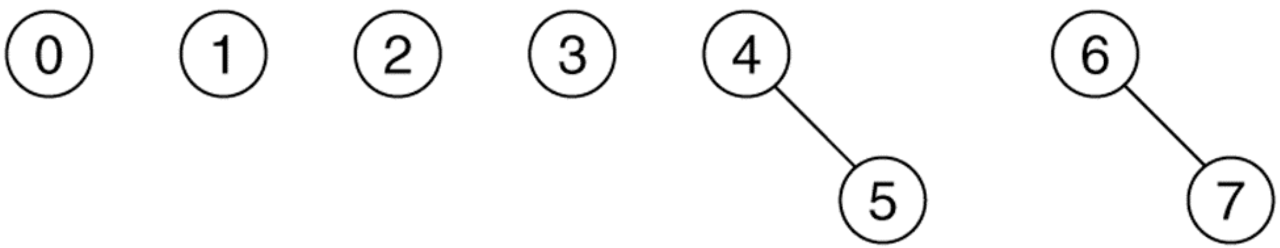
Figure 24.13

The forest after the union of trees with roots 4 and 5



0	-1
1	-1
2	-1
3	-1
4	-1
5	4
6	-1
7	-1

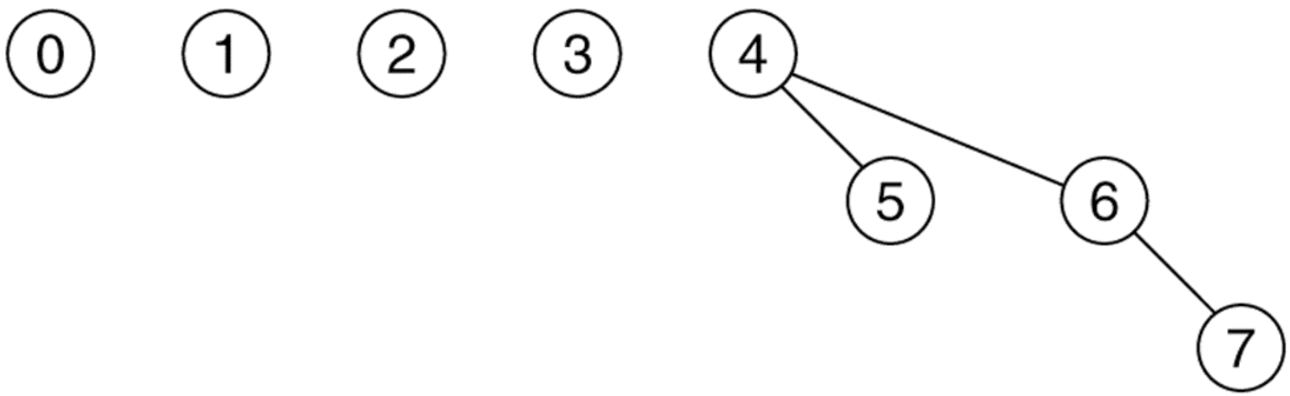
Figure 24.14
The forest after the union of trees with roots 6 and 7



0	-1
1	-1
2	-1
3	-1
4	-1
5	4
6	-1
7	6

Figure 24.15

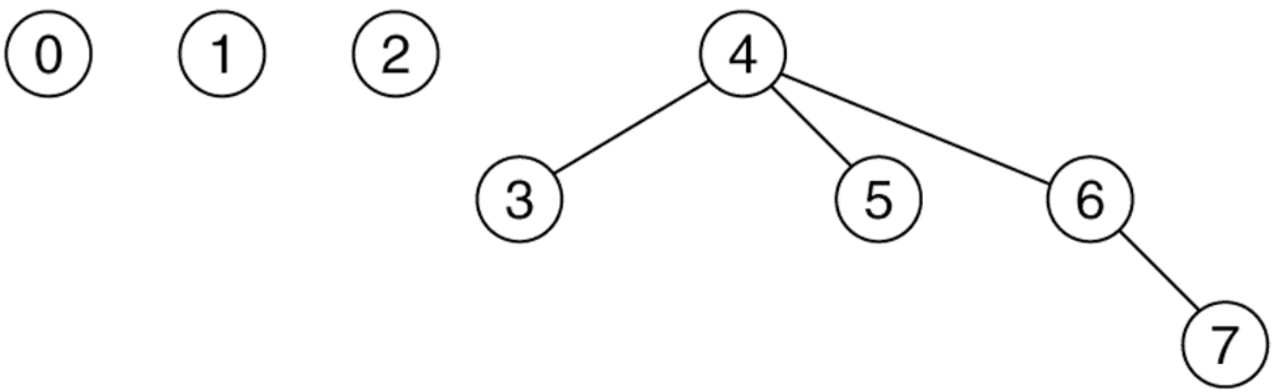
The forest after the union of trees with roots 4 and 6



0	-1
1	-1
2	-1
3	-1
4	-1
5	4
6	4
7	6

Figure 24.16

The forest formed by union-by-size, with the sizes encoded as negative numbers



0	-1
1	-1
2	-1
3	4
4	-5
5	4
6	4
7	6

Applications: Disjoint Set Union-Find Data Structure

- Minimum Spanning Tree problem

Figure 24.6

(a) A graph *G* and (b) its minimum spanning tree

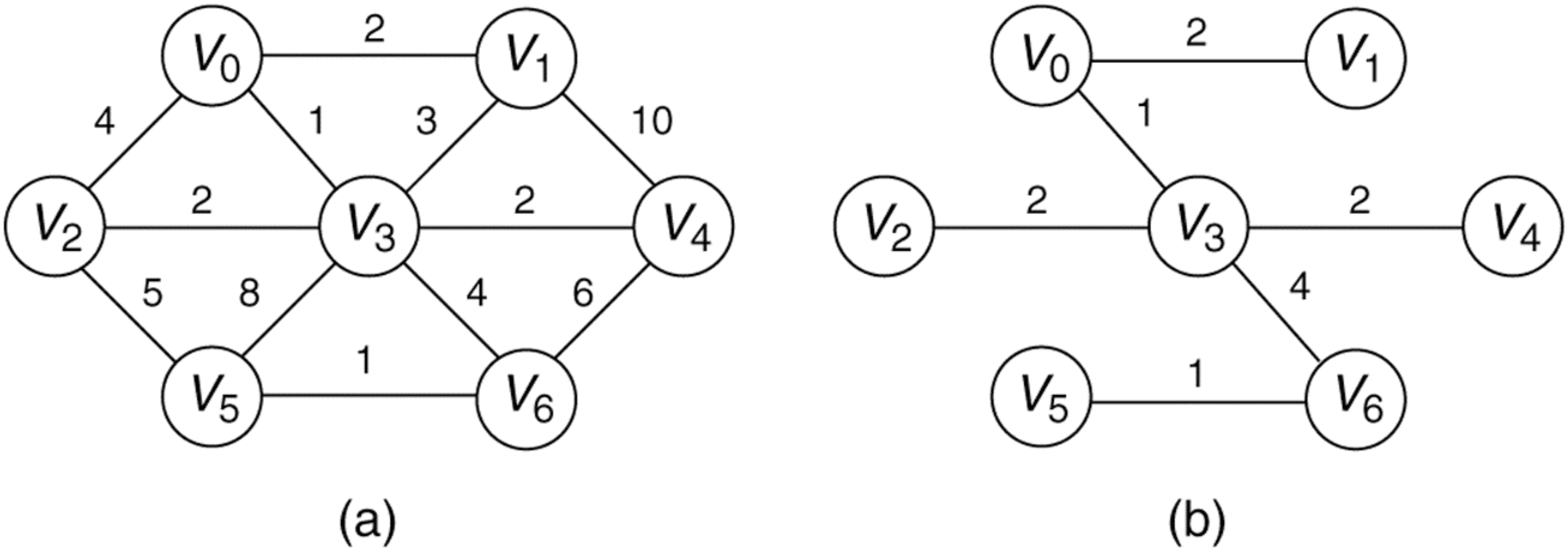


Figure 24.7 (A)

Kruskal's algorithm after each edge has been considered. The stages proceed left-to-right, top-to-bottom, as numbered. (*continued*)

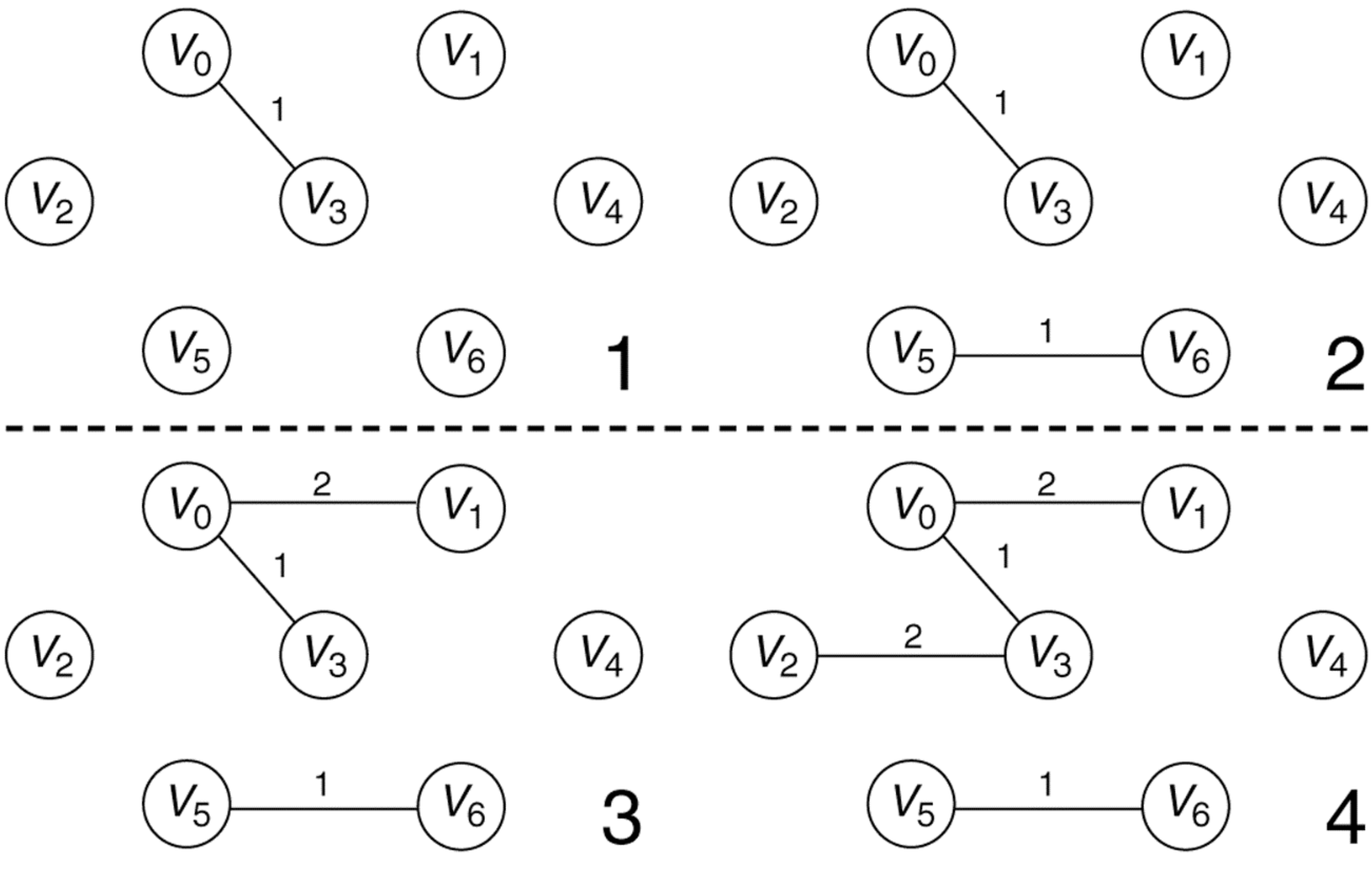


Figure 24.7 (B)

Kruskal's algorithm after each edge has been considered. The stages proceed left-to-right, top-to-bottom, as numbered.

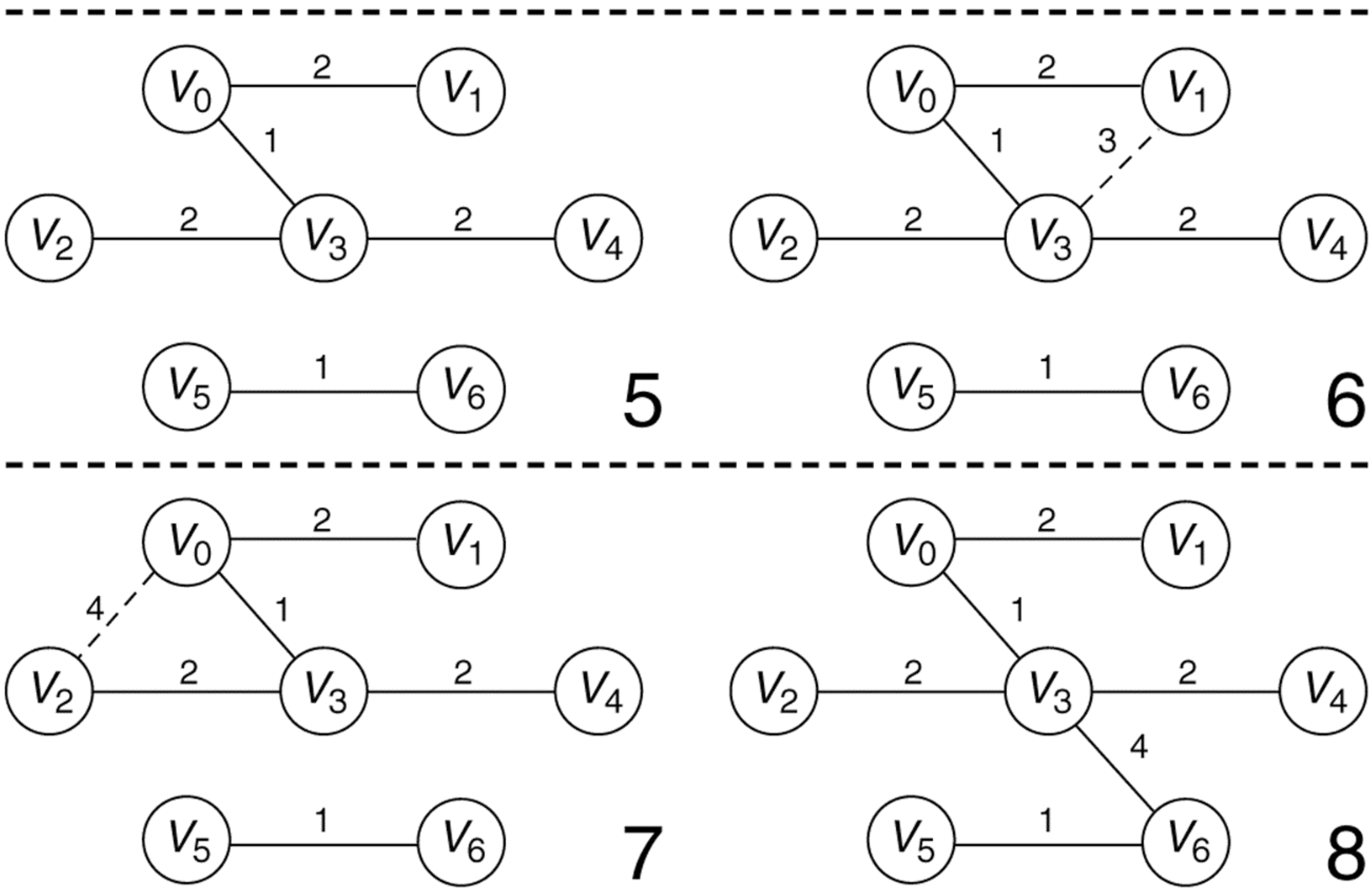


Figure 24.1

A 50 x 88 maze

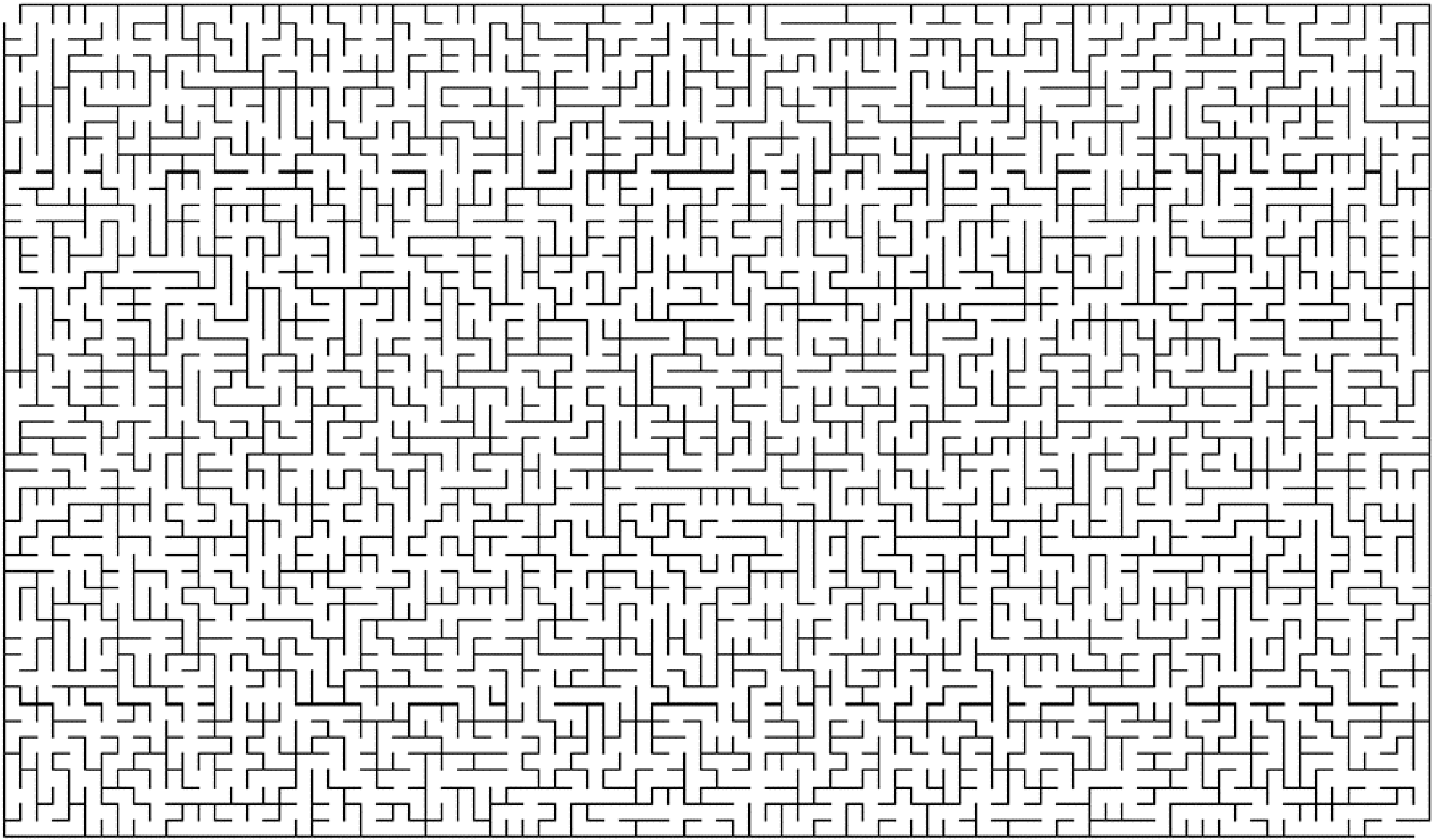
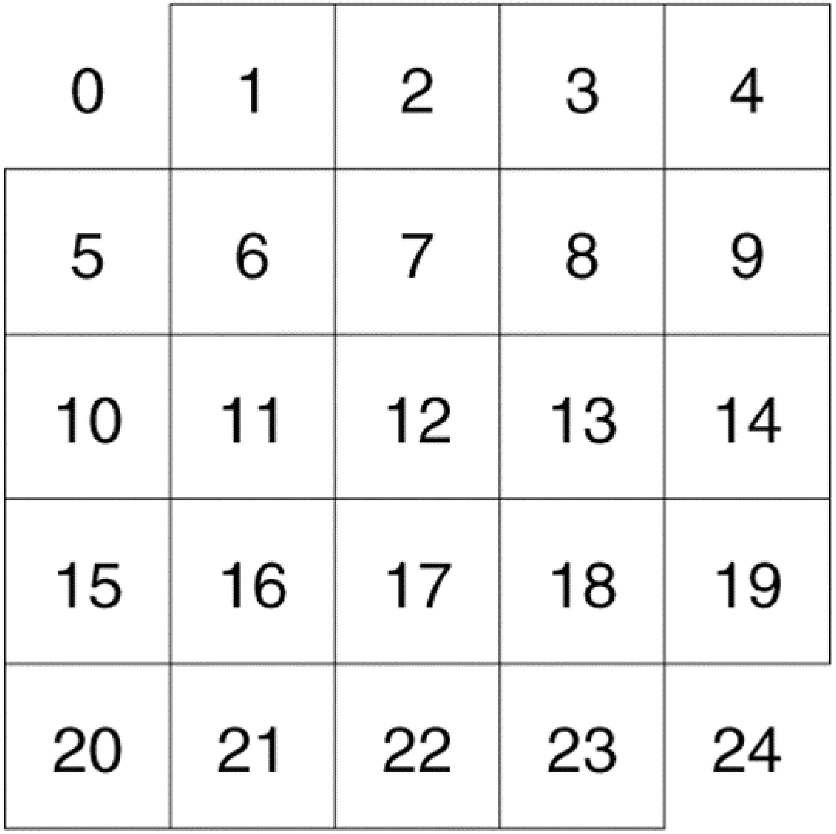


Figure 24.2

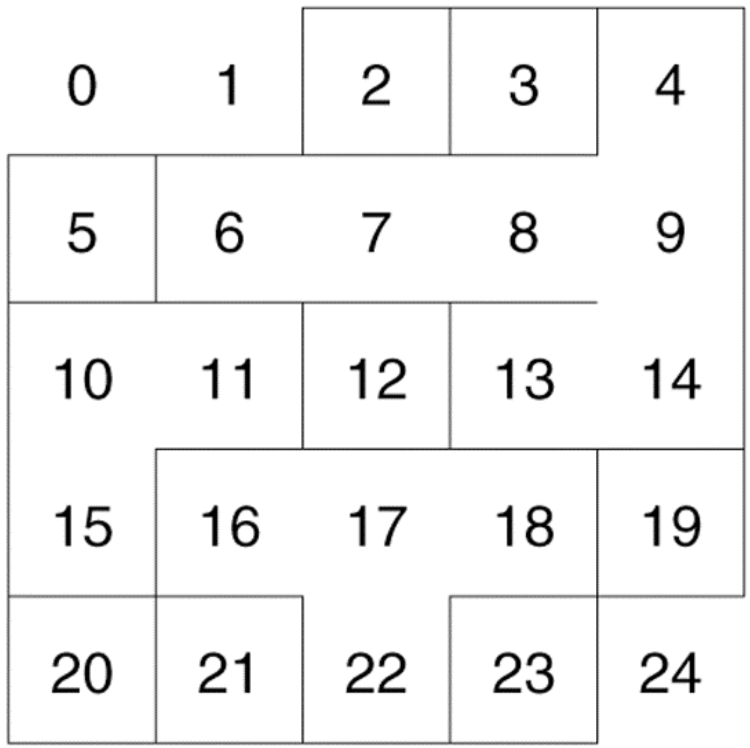
Initial state: All walls are up, and all cells are in their own sets.



(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 24.3

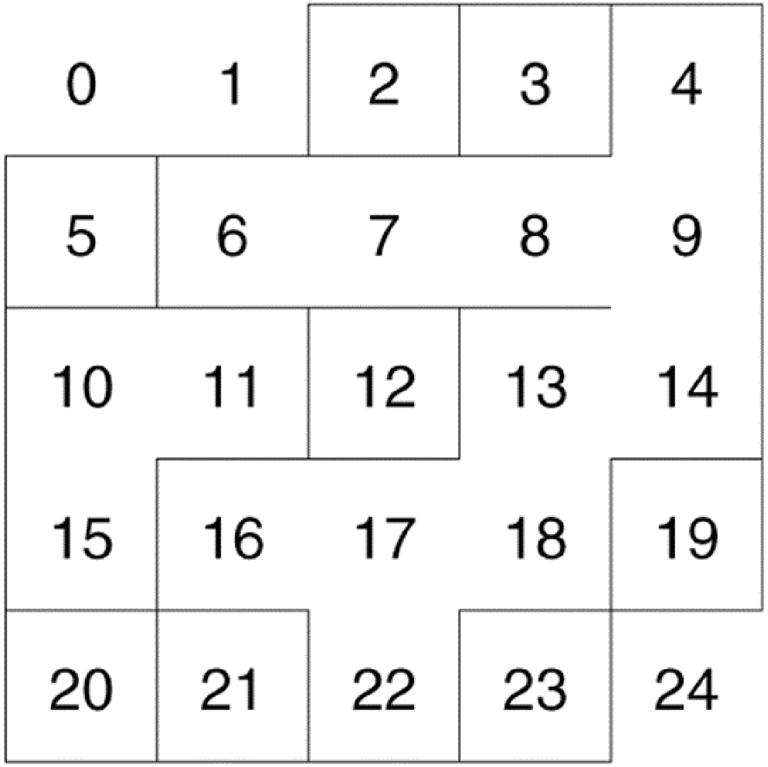
At some point in the algorithm, several walls have been knocked down and sets have been merged. At this point, if we randomly select the wall between 8 and 13, this wall is not knocked down because 8 and 13 are already connected.



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

Figure 24.4

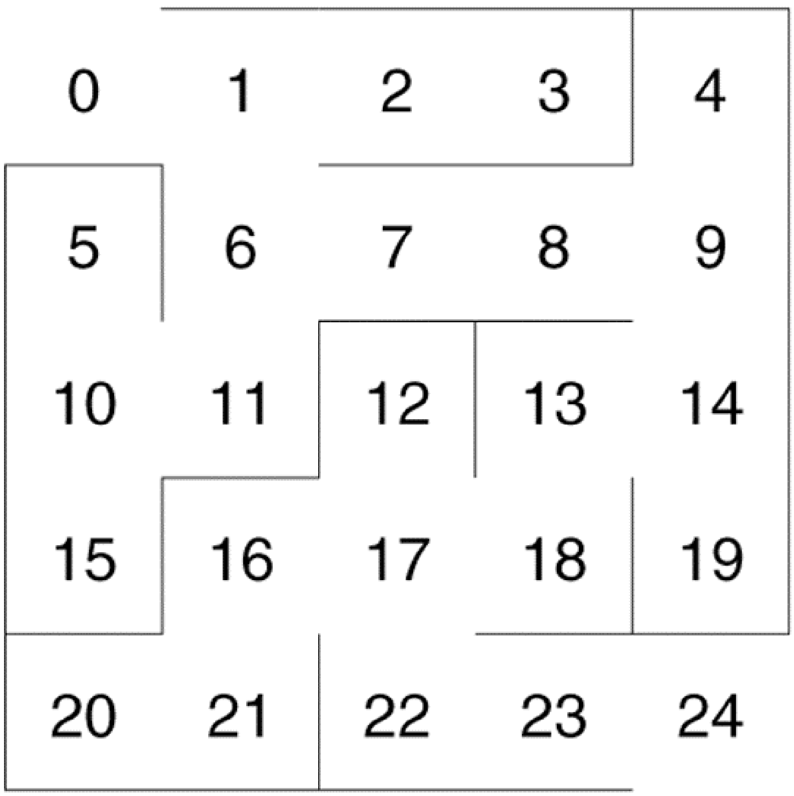
We randomly select the wall between squares 18 and 13 in Figure 24.3; this wall has been knocked down because 18 and 13 were not already connected, and their sets have been merged.



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

Figure 24.5

Eventually, 24 walls have been knocked down, and all the 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)