3.6 Disjoint Sets

In this section, we discuss an abstract data type that contains nonempty pairwise disjoint sets (i.e., for all sets X and Y in the container, $X \neq \emptyset$, $Y \neq \emptyset$, and either $X = Y$ or $X \cap Y = \emptyset$). Furthermore, each set X in the container has one of its members *marked* as a representative of that set. The operations supported are:

- *makeset*(i): Construct the set $\{i\}$.
- *findset*(i): Return the marked member of the set to which i belongs.
- *union*(i, j): Replace the sets containing i and j with their union. (It is assumed that i and j do not belong to the same set.)

We assume that the elements of the sets are positive integers.

Example 3.6.1. After

makeset(1)
makeset(2)
makeset(3)
makeset(4)
makeset(5)

we have

$$\{1\}, \quad \{2\}, \quad \{3\}, \quad \{4\}, \quad \{5\}.$$

Each element is marked; so, the value of

findset(i)

is i, for all i.

After

union(1, 4)
union(3, 5)

we have

$$\{1,4\}, \quad \{2\}, \quad \{3,5\}.$$

If we assume that 4 is marked, the value of

findset(1)

or

findset(4)

is 4.

Notice that *findset* can be used to check whether elements i and j belong to the same set. For example,

findset(1) == *findset*(4)

is true; but

findset(2) == *findset*(3)

is false. (The value of *findset*(2) is 2, and the value of *findset*(3) is either 3 or 5, whichever is marked.)

After

union(4, 2)

we have

$$\{1,2,4\}, \quad \{3,5\};$$

and after

union(1, 5)

we have

$$\{1,2,3,4,5\}.$$

Now the value of *findset*(*i*) is the same for all *i*. (The common value is whichever element is marked in the set $\{1,2,3,4,5\}$.) □

To implement the disjoint-set abstract data type, we represent a set as a tree with the marked element as the root. The elements in a set are not arranged in any special way, and the tree is not necessarily a binary tree.

Example 3.6.2. Three of the many ways to represent the set $\{2,4,5,8\}$, with 5 as the marked element, are shown in Figure 3.6.1.

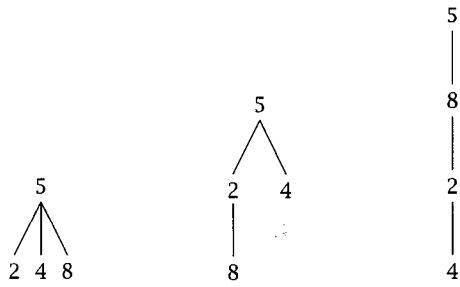


Figure 3.6.1 Representations of the set $\{2,4,5,8\}$, with 5 as the marked element. □

We represent disjoint sets using an array, *parent*, in which the value, *parent*[*i*], is the parent of *i*, unless *i* is the root. In the latter case, the value of *parent*[*i*] is *i*.

Example 3.6.3. If the disjoint sets

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

are represented by the trees shown in Figure 3.6.2, the *parent* array is

1	5	7	5	5	7	7	2
1	2	3	4	5	6	7	8

Figure 3.6.2 Disjoint sets represented as trees. □

We turn now to algorithms to manipulate disjoint sets represented as trees. We begin with the first version of *makeset*, which is straightforward.

Algorithm 3.6.4 Makeset, Version 1. This algorithm represents the set $\{i\}$ as a one-node tree.

Input Parameter: i
Output Parameters: None

```
makeset1(i) {
  parent[i] = i
}
```

The first version of *findset* simply follows a path from the input element to the root.

Algorithm 3.6.5 Findset, Version 1. This algorithm returns the root of the tree to which i belongs.

Input Parameter: i
Output Parameters: None

```
findset1(i) {
  while (i != parent[i])
    i = parent[i]
  return i
}
```

Algorithm 3.6.5 returns the root since, by convention, i is equal to the root precisely when i and $parent[i]$ are equal.

To compute the union of two sets, we must merge the trees that represent them. To merge the trees, we make one root a child of the other root. The following algorithm gives the details. The algorithm assumes that the input is the two roots.

Algorithm 3.6.6 Mergetrees, Version 1. This algorithm receives as input the roots of two distinct trees and combines them by making one root a child of the other root.

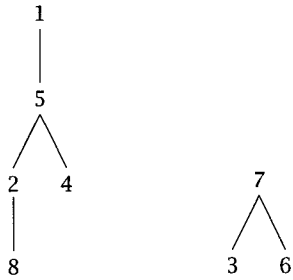
Input Parameters: i, j
Output Parameters: None

mergetrees1(i, j) {
  parent$[i] = j$
}

Example 3.6.7. Given the disjoint sets as represented in Figure 3.6.2, after

mergetrees1$(5, 1)$

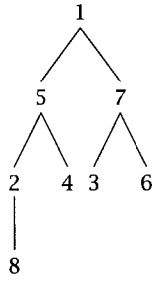
we obtain



After

mergetrees1$(7, 1)$

we obtain



□

The union algorithm receives as input two *arbitrary* values (i.e., the values are not necessarily roots). It first invokes *findset1* (Algorithm 3.6.5) to find the roots of the trees and then invokes *mergetrees1* (Algorithm 3.6.6).

Algorithm 3.6.8 Union, Version 1. This algorithm receives as input two arbitrary values i and j and constructs the tree that represents the union of the sets to which i and j belong. The algorithm assumes that i and j belong to different sets.

Input Parameters: i, j
Output Parameters: None

```
union1(i, j) {
  mergetrees1(findset1(i), findset1(j))
}
```

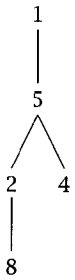
We next consider the time required by our disjoint-set algorithms. When these algorithms are used, there are typically *many* calls to the various algorithms; thus, we are interested in the total time required when these algorithms are called repeatedly. Therefore, we assume throughout the remainder of this section that there are n makeset operations and a total of m union and findset operations. We also assume that the makeset operations are performed first.

Since the makeset algorithm (Algorithm 3.6.4) runs in constant time, the n makeset operations take time $\Theta(n)$. The worst-case time of findset (Algorithm 3.6.5) for any tree occurs when the argument is a node at the lowest level. In this case, the time is proportional to the height of the tree. The maximum height of a tree with n nodes is $n-1$ (which occurs when each parent has exactly one child). Thus, the time of findset is $O(n)$. The union algorithm (Algorithm 3.6.8) calls the findset algorithm twice and the mergetrees algorithm (Algorithm 3.6.6) once. Since the mergetrees algorithm runs in time $\Theta(1)$ and the findset algorithm runs in time $O(n)$, the union algorithm runs in time $O(n)$. The findset and union algorithms are called a total of m times. Therefore, the time required by the findset and union algorithms is $O(mn)$, and the time required by all of the algorithms is $O(n+mn)$, where there are n makeset operations and a total of m union and findset operations.

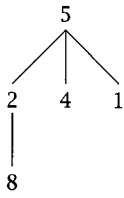
If $m < n$, we can derive a sharper estimate. In this case, the height of any tree is at most m (see Exercise 7); so, the time required by the findset and union algorithms is $O(m^2)$. Thus, the time required by all of the algorithms is $O(n+m^2)$. It follows that for any value of m, the time required by all of the algorithms is $O(n+m\cdot\min\{m,n\})$. We leave as an exercise (see Exercise 8) to show that this estimate is sharp, that is, that the worst-case time required by all of the algorithms is $\Omega(n+m\cdot\min\{m,n\})$. It follows that the worst-case time required by all of the algorithms is $\Theta(n+m\cdot\min\{m,n\})$, where there are n makeset operations and a total of m union and findset operations.

The time of the findset algorithm, and by extension the time of the union algorithm, is bounded by the height of the tree. It follows that we can improve the performance of our algorithms if we can constrain the tree heights.

Notice that when we execute *union1*$(5,1)$ for the trees in Figure 3.6.2, we obtain



We have increased the maximum height among the trees to 3 because we made the tree with the greater height a subtree of the tree with the smaller height. Had we made the tree with the smaller height a subtree of the tree with the greater height



we would not have increased the maximum height among all trees. In order to make the tree with the smaller height a subtree of the tree with the greater height, we maintain an array *height*, in which *height*[*i*] is the height of the tree rooted at *i*. The revised algorithms follow.

Algorithm 3.6.9 Makeset, Version 2. This algorithm represents the set $\{i\}$ as a one-node tree and initializes its height to 0.

Input Parameter: i
Output Parameters: None

```
makeset2(i) {
  parent[i] = i
  height[i] = 0
}
```

The findset algorithm is unchanged.

Algorithm 3.6.10 Findset, Version 2. This algorithm returns the root of the tree to which i belongs.

Input Parameter: i
Output Parameters: None

```
findset2(i) {
  while (i != parent[i])
    i = parent[i]
  return i
}
```

Algorithm 3.6.11 Mergetrees, Version 2. This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller height a child of the other root. If the trees have the same height, we arbitrarily make the root of the first tree a child of the other root.

Input Parameters: i, j
Output Parameters: None

```
mergetrees2(i, j) {
  if (height[i] < height[j])
    parent[i] = j
  else if (height[i] > height[j])
    parent[j] = i
  else {
    parent[i] = j
    height[j] = height[j] + 1
  }
}
```

The union algorithm now calls *mergetrees2* and *findset2*.

Algorithm 3.6.12 Union, Version 2. This algorithm receives as input two arbitrary values i and j and constructs the tree that represents the union of the sets to which i and j belong. The algorithm assumes that i and j belong to different sets.

Input Parameters: i, j
Output Parameters: None

```
union2(i, j) {
  mergetrees2(findset2(i), findset2(j))
}
```

Example 3.6.13. Suppose that we begin by calling *makeset2*[i] for $i = 1$ to 8. The resulting trees are the singleton nodes and the arrays are

parent

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

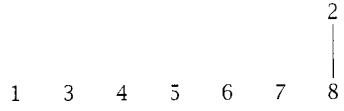
height

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

When *union2*(8, 2) is called, *mergetrees2* is called as *mergetrees2*(8, 2). Since *height*[8] and *height*[2] are equal, *mergetrees2* executes

parent[8] = 2
height[2] = *height*[2] + 1

The trees become



and the arrays become

parent

1	2	3	4	5	6	7	2
1	2	3	4	5	6	7	8

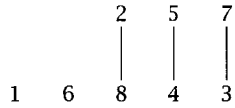
height

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

Similarly, after

union2(4, 5)
union2(3, 7)

the trees become



and the arrays become

parent

1	2	7	5	5	6	7	2
1	2	3	4	5	6	7	8

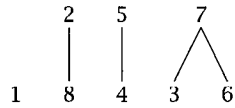
height

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

When *union2*(3, 6) is called, *mergetrees2* is called as *mergetrees2*(7, 6). Since *height*[7] > *height*[6], *mergetrees2* executes

parent[6] = 7

The trees become



and the arrays become

parent

1	2	7	5	5	7	7	2
1	2	3	4	5	6	7	8

height

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

□

In version 1 of our disjoint-set algorithms, the height of a k-node tree was at most $k - 1$. We show that, in version 2, the height of a k-node tree is at most $\lfloor \lg k \rfloor$.

Theorem 3.6.14. *Using version 2 of the disjoint-set algorithms (Algorithms 3.6.9–3.6.12), the height of a k-node tree is at most* $\lfloor \lg k \rfloor$.

Proof. The proof is by induction on k, and the basis step is $k = 1$. Since a one-node tree has height zero, the statement is true in this case.

Now suppose that $k > 1$, and for all $p < k$, the height of a p-node tree is at most $\lfloor \lg p \rfloor$. Let T be a k-node tree. Since $k > 1$, T is the union of two trees: T_1, of height h_1 with $k_1 < k$ nodes; and T_2, of height h_2 with $k_2 < k$ nodes. By the inductive assumption, $h_1 \leq \lfloor \lg k_1 \rfloor$ and $h_2 \leq \lfloor \lg k_2 \rfloor$. If $h_1 \neq h_2$, the height of T is

$$\max\{h_1, h_2\} \leq \max\{\lfloor \lg k_1 \rfloor, \lfloor \lg k_2 \rfloor\} \leq \lfloor \lg k \rfloor.$$

Now suppose that $h_1 = h_2$. We may assume that $k_1 \geq k_2$. Notice that $k_2 \leq k/2$. (If this last inequality is false, we have $k_2 > k/2$ and $k_1 \geq k_2 > k/2$, which implies that $k = k_1 + k_2 > k$.) The height of T is

$$1 + h_2 \leq 1 + \lfloor \lg k_2 \rfloor \leq 1 + \lfloor \lg k/2 \rfloor = 1 + \lfloor (\lg k) - 1 \rfloor = \lfloor \lg k \rfloor.$$

The inductive step is complete. ■

Consider the time required by version 2 of the disjoint-set algorithms. The makeset algorithm (Algorithm 3.6.9) still runs in constant time, and the n makeset operations take time $\Theta(n)$. By Theorem 3.6.14, the height of any tree is bounded by $\lg n$. Thus, the time of findset (Algorithm 3.6.10) is $O(\lg n)$. The union algorithm (Algorithm 3.6.12) calls the findset algorithm twice and the mergetrees algorithm (Algorithm 3.6.11) once. Since the mergetrees algorithm runs in time $\Theta(1)$ and the findset algorithm runs in time $O(\lg n)$, the union algorithm runs in time $O(\lg n)$. The findset and union algorithms are called a total of m times. Therefore, the time required by the findset and union algorithms is $O(m \lg n)$, and the time required by all of the algorithms is $O(n + m \lg n)$, where there are n makeset operations and a total of m union and findset operations.

As for version 1 of our disjoint-set algorithms, if $m < n$ we can derive a sharper estimate. We first note that after makeset initializes the parent array, each cell in the parent array is modified at most one time—by a call to union. In a tree containing k nodes, each node, except the root, has had its cell in the parent array modified (since it is no longer a root). Therefore, a tree with p nodes was constructed by exactly $p - 1$ calls of the union algorithm. Let k be the number of times the union algorithm was called. From our preceding comments, each tree has at most $k + 1 \leq m + 1$ nodes. By Theorem 3.6.14, the height of any tree is bounded by $\lg(m + 1)$. Thus, the time required by the findset and union algorithms is $O(m \lg m)$, and the time required by all of the algorithms is $O(n + m \lg m)$. It follows that for any value of m, the time required by all of the algorithms is $O(n + m \cdot \min\{\lg m, \lg n\})$. We leave as an exercise (see Exercise 9) to show that this estimate is sharp, that is, that the worst-case time required by all of the algorithms is $\Omega(n + m \cdot \min\{\lg m, \lg n\})$. It follows that the worst-case time required by all of the algorithms is $\Theta(n + m \cdot \min\{\lg m, \lg n\})$, where there are n makeset operations and a total of m union and findset operations.

The final enhancement to the disjoint-set algorithms involves the findset algorithm. Again, our goal is to decrease the heights of the trees. In the call *findset*(i), after locating the root, we make every node on the path from i to the root, except the root itself, a child of the root, thereby potentially decreasing the height of the tree. This process is called **path compression.**

Example 3.6.15. After the call

findset(11)

the tree in Figure 3.6.3(a) becomes the tree shown in Figure 3.6.3(b).

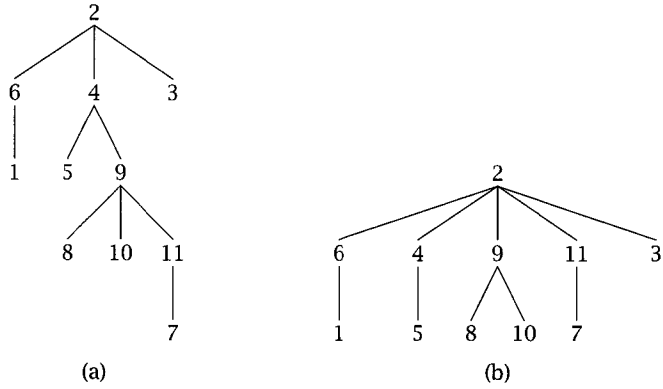


Figure 3.6.3 Path compression. *findset*(11) is called on tree (a). Every node on the path from 11 to the root, except the root itself, becomes a child of the root, yielding tree (b). □

The fastest known implementation of the disjoint-set algorithms combines path compression and an analog of the *height* array. The *height* array no longer gives the *exact* heights of the trees because path compression can reduce the height of the tree. For this reason, we rename the *height* array the *rank* array; *rank*[i] is an *upper bound* on the height of the tree rooted at i. The version of the union algorithm that uses the *rank* array is called **union by rank**. The algorithms follow.

Algorithm 3.6.16 Makeset, Version 3. This algorithm represents the set $\{i\}$ as a one-node tree and initializes its rank to 0.

```
   Input Parameter:  i
Output Parameters:   None

makeset3(i) {
  parent[i] = i
  rank[i] = 0
}
```

Algorithm 3.6.17 Findset, Version 3. This algorithm returns the root of the tree to which i belongs and makes every node on the path from i to the root, except the root itself, a child of the root.

```
   Input Parameter:  i
Output Parameters:   None

findset3(i) {
  root = i
  while (root != parent[root])
    root = parent[root]
```

```
    j = parent[i]
    while (j != root) {
      parent[i] = root
      i = j
      j = parent[i]
    }
    return root
}
```

Algorithm 3.6.18 Mergetrees, Version 3. This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller rank a child of the other root. If the trees have the same rank, we arbitrarily make the root of the first tree a child of the other root.

```
 Input Parameters:  i, j
Output Parameters:  None

mergetrees3(i, j) {
  if (rank[i] < rank[j])
    parent[i] = j
  else if (rank[i] > rank[j])
    parent[j] = i
  else {
    parent[i] = j
    rank[j] = rank[j] + 1
  }
}
```

Algorithm 3.6.19 Union, Version 3. This algorithm receives as input two arbitrary values i and j and constructs the tree that represents the union of the sets to which i and j belong. The algorithm assumes that i and j belong to different sets.

```
 Input Parameters:  i, j
Output Parameters:  None

union3(i, j) {
  mergetrees3(findset3(i), findset3(j))
}
```

The analysis of the time required by the disjoint-set algorithms using both path compression and union by rank (Algorithms 3.6.16–3.6.19) is complicated and, through the years, increasingly sharper upper bounds on the time were obtained. Finally, in 1975, R. E. Tarjan (see Tarjan, 1975) proved that the worst-case running time is

$$\Theta(t\alpha(t,n)),$$

where n is the number of makeset operations, $t = m + n$ is the total number of operations, and α is a function with an *extremely* slow rate of growth. For the α defined by Tarjan, $\alpha(t, n) \leq 4$, for all $n \leq 10^{19728}$. (Other definitions of α in the literature differ from Tarjan's definition by an additive constant.) Whichever essentially equivalent way α is defined, for all practical values of n, $\alpha(t, n)$ is bounded by a constant. So, from a practical standpoint (but *not* from a theoretical standpoint!), the worst-case time of the disjoint-set algorithms using both path compression and union by rank is linear in t.

Exercises

In Exercises 1–3, show the trees that result after the following statements are executed:

```
for i = 1 to 8
    makeset(i)
union(1, 2)
union(3, 4)
union(5, 6)
union(5, 7)
union(5, 8)
union(4, 8)
union(3, 2)
```

1S. Substitute *makeset1* for *makeset* and *union1* for *union*.

2. Substitute *makeset2* for *makeset* and *union2* for *union*.

3. Substitute *makeset3* for *makeset* and *union3* for *union*.

4S. What happens if union, version 1, (Algorithm 3.6.8) is erroneously called with i and j in the same tree?

5. What happens if union, version 2, (Algorithm 3.6.12) is erroneously called with i and j in the same tree?

6. What happens if union, version 3, (Algorithm 3.6.19) is erroneously called with i and j in the same tree?

7S. Show that if $m < n$, after executing Algorithms 3.6.4, 3.6.5, and 3.6.8, the maximum height of a tree is m.

8. Show that the worst-case time required by Algorithms 3.6.4, 3.6.5, and 3.6.8 is

$$\Omega(n + m \cdot \min\{m, n\}).$$

9. Show that the worst-case time required by Algorithms 3.6.9, 3.6.10, and 3.6.12 is

$$\Omega(n + m \cdot \min\{\lg m, \lg n\}).$$

10S. Write a recursive version of the findset algorithm without path compression (Algorithm 3.6.5).

11. Write a recursive version of the findset algorithm with path compression (Algorithm 3.6.17).

12. An alternative to path compression is *path halving*. In path halving, when the path is traversed from the node to the root, we make the grandparent of every other node i on the path the new parent of i. Write the path-halving algorithm. Path compression requires two passes from the node to the root (one to find the root and one to reset the parents), but path halving requires only one pass. Tarjan and van Leeuwen (see Tarjan, 1984) showed that path halving, together with union by rank, also gives worst-case time $\Theta(t\alpha(t,n))$.

Notes

Classic books on data structures are Aho, 1983; Knuth, 1997; and Tarjan, 1983. Recent books on data structures and their implementation in programming languages are Standish, 1998, and Weiss, 2001.

The heapsort algorithm and the term "heap" were invented by J. W. J. Williams (see Williams, 1964). Fibonacci heaps (see Fredman, 1987) and relaxed heaps (see Driscoll, 1988) improve the asymptotic times of binary heaps.

Chapter Exercises

3.1. Write a version of *push* for a stack that throws an exception if the array is full. If the array is not full, the behavior is the same as the original *push*. Assume that *SIZE* specifies the size of the array.

3.2. Write a version of *pop* for a stack that throws an exception if the stack is empty. If the stack is not empty, the behavior is the same as the original *pop*.

3.3. Write a version of *top* for a stack that throws an exception if the stack is empty. If the stack is not empty, the behavior is the same as the original *top*.

3.4. Write a version of *enqueue* for a queue that throws an exception if the array is full. If the array is not full, the behavior is the same as the original *enqueue*.

3.5. Write a version of *dequeue* for a queue that throws an exception if the queue is empty. If the queue is not empty, the behavior is the same as the original *dequeue*.

3.6. Write a version of *front* for a queue that throws an exception if the queue is empty. If the queue is not empty, the behavior is the same as the original *front*.

A deque *(pronounced "deck") is like a queue, except that items may be added and deleted at the rear or the front.*

3.7. Implement a deque using an array. Do not incorporate error checking into your functions.

3.8. Implement a deque using an array. Incorporate error checking into your functions.

3.9. Implement a deque using a linked list. Do not incorporate error checking into your functions.

3.10. Implement a deque using a linked list. Incorporate error checking into your functions.

3.11. Let T_1 be a binary tree with root r_1 and let T_2 be a binary tree with root r_2. The binary trees are *isomorphic* if there is a one-to-one, onto function f from the vertex set of T_1 to the vertex set of T_2 satisfying

- Vertices v_i and v_j are adjacent in T_1 if and only if the vertices $f(v_i)$ and $f(v_j)$ are adjacent in T_2.
- $f(r_1) = r_2$.
- v is a left child of w in T_1 if and only if $f(v)$ is a left child of $f(w)$ in T_2.
- v is a right child of w in T_1 if and only if $f(v)$ is a right child of $f(w)$ in T_2.

Write an algorithm, which runs in linear time in the worst case, to determine whether two binary trees are isomorphic. Prove that your algorithm does run in linear time.

3.12. Find input that produces worst-case time for heapsort (Algorithm 3.5.16).

3.13. A *d-heap* is like a binary heap except that the nodes have d children or less rather than two children or less. Write d-heap versions of siftdown, delete, insert, and heapify. Also write an algorithm that returns the largest value in a d-heap. The asymptotic times should be the same as the binary heap algorithms. Show that your algorithms do have the same asymptotic times as those for a binary heap. In practice, the 3- or 4-heap algorithms tend to run faster than the binary heap algorithms.

3.14. Implement a d-heap (see Exercise 3.13) as an indirect heap. Write versions of siftdown, delete, insert, and heapify. Also write an algorithm that returns the largest value in an indirect d-heap. The asymptotic times should be the same as the binary heap algorithms. Show that your algorithms do have the same asymptotic times as those for a binary heap.

3.15. Implement the disjoint-set abstract data type by using linked lists to represent the sets. Make your algorithms as efficient as you can, and provide sharp asymptotic time bounds.

3.16. Prove a version of Theorem 3.6.14 in which *height*[i] is replaced by *count*[i], where *count*[i] is the number of nodes in the tree rooted at i.