(a) Write a C++ program to delete the largest element in a binary search tree. Your procedure should have complexity $O(h)$, where h is the height of the search tree. Since h is $O(\log n)$ on average, you can perform each of the priority queue operations in average time $O(\log n)$.

(b) Compare the performances of max heaps and binary search trees as data structures for priority queues. For this comparison, generate random sequences of insert and delete max operations and measure the total time taken for each sequence by each of these data structures.

6. Input is a sequence X of n keys with many duplications such that the number of distinct keys is d ($< n$). Present an $O(n \log d)$-time sorting algorithm for this input. (For example, if $X = 5, 6, 1, 18,\ 6, 4, 4, 1,\ 5, 17$, the number of distinct keys in X is six.)

2.5 SETS AND DISJOINT SET UNION

2.5.1 Introduction

In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers $1, 2, 3, \ldots, n$. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (i.e., if S_i and S_j, $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). For example, when $n = 10$, the elements can be partitioned into three disjoint sets, $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, and $S_3 = \{3, 4, 6\}$. Figure 2.17 shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.

The operations we wish to perform on these sets are:

1. **Disjoint set union.** If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j, the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.

2. **Find**(i). Given the element i, find the set containing i. Thus, 4 is in set S_3, and 9 is in set S_1.

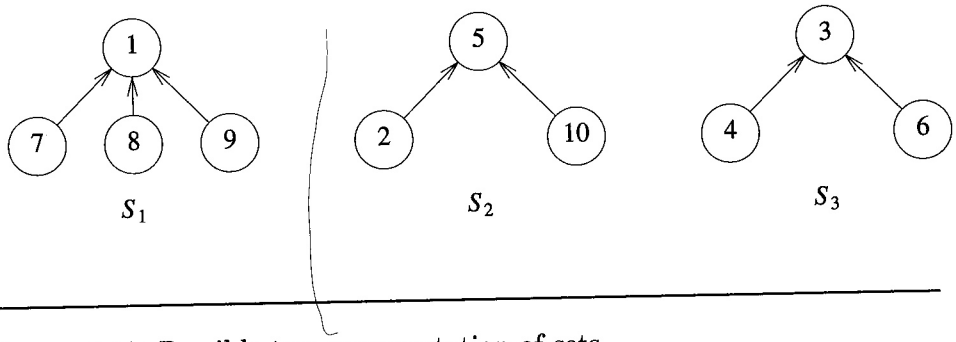


Figure 2.17 Possible tree representation of sets

2.5.2 Union and Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 (see Figure 2.17). Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 2.18.

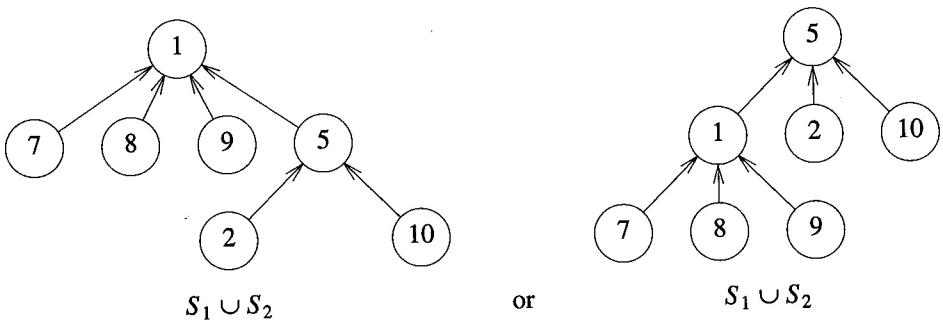


Figure 2.18 Possible representations of $S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the

root of its tree and use the pointer to the set name. The data representation for S_1, S_2, and S_3 may then take the form shown in Figure 2.19.

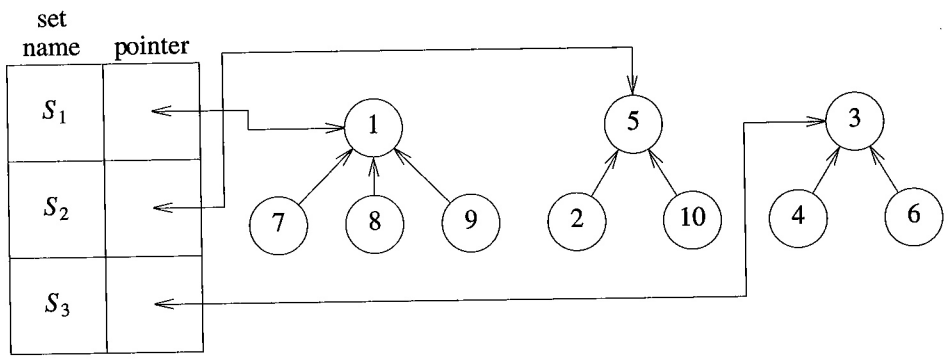


Figure 2.19 Data representation for S_1, S_2, and S_3

In presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them. This simplifies the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j, and j has a pointer to entry k in the set name table, then the set name is just $name[k]$. If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots $FindPointer(S_i)$ and $FindPointer(S_j)$. Here $FindPointer$ is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root. The operation of $Find(i)$ now becomes: Determine the root of the tree containing element i. The function $Union(i, j)$ requires two trees with roots i and j be joined. Another simplifying assumption we make is that the set elements are the numbers 1 through n.

Since the set elements are numbered 1 through n, we represent the tree nodes using an array `p[1:n]`, where `n` is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. Figure 2.20 shows this representation of the sets S_1, S_2, and S_3 of Figure 2.17. Notice that root nodes have a parent of -1.

We can now implement $Find(i)$ by following the indices, starting at i until we reach a node with parent value -1. For example, $Find(6)$ starts at 6 and then moves to 6's parent, 3. Since `p[3]` is negative, we have reached the root. The operation $Union(i, j)$ is equally simple. We pass in two trees with roots i and j. Adopting the convention that the first tree becomes a

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	−1	5	−1	3	−1	3	1	1	1	5

Figure 2.20 Array representation of S_1, S_2, and S_3 of Figure 2.17

subtree of the second, the statement `p[i]=j;` accomplishes the union.

Program 2.16 gives the algorithms for the union and find operations just discussed. It also defines the class `Sets`. Although these two algorithms are very easy to state, their performance characteristics are not very good. For instance, if we start with q elements each in a set of its own (i.e., $S_i = \{i\}$, $1 \leq i \leq q$), then the initial configuration consists of a forest with q nodes, and $p[i] = 0$, $1 \leq i \leq q$. Now let us process the following sequence of *union-find* operations:

$$Union(1,2),\ Union(2,3),\ Union(3,4),\ Union(4,5),\ldots,\ Union(n-1,n)$$

$$Find(1),\ Find(2),\ldots,\ Find(n)$$

This sequence results in the degenerate tree of Figure 2.21.

Since the time taken for a union is constant, the $n-1$ unions can be processed in time $O(n)$. However, each find requires following a sequence of *parent* pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is $O(i)$, the total time needed to process the n finds is $O(\sum_{i=1}^{n} i) = O(n^2)$. □

We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. To accomplish this, we make use of a weighting rule for $Union(i,j)$.

Definition 2.5 [Weighting rule for *Union(i,j)*]: If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j. □

When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of Figure 2.22. In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a `count` field in the root of every tree. If i is a root node, then $count[i]$ equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number in the `p` field, we can maintain the count in the `p` field of the roots as a negative number.

```
class Sets
{
    private:
       int *p, n;
    public:
       Sets(int Size): n(Size)
       {   p = new int[n+1];
           for (int i=0;i<=n;i++)
              p[i]=-1;
       }
       ~Sets() {delete []p;}
       void SimpleUnion(int i, int j);
       int SimpleFind(int i);
};

void Sets::SimpleUnion(int i, int j)
{   p[i] = j;   }

int Sets::SimpleFind(int i)
{   while (p[i]>=0) i = p[i];
    return i;
}
```

Program 2.16 Simple algorithms for union and find

Figure 2.21 Degenerate tree

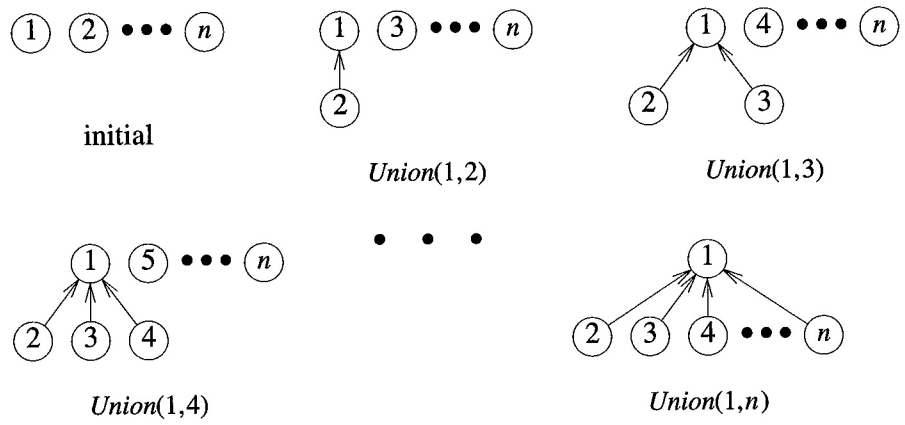


Figure 2.22 Trees obtained using the weighting rule

```
void Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i != j,
// using the weighting rule. p[i] ==
// -count[i] and p[j] == -count[j].
{
    int temp = p[i] + p[j];
    if (p[i] > p[j]) { // i has fewer nodes.
       p[i] = j; p[j] = temp;
    }
    else { // j has fewer or equal nodes.
       p[j] = i; p[i] = temp;
    }
}
```

Program 2.17 Union algorithm with weighting rule

Using this convention, we obtain the union algorithm of Program 2.17. In this algorithm the time required to perform a union has increased somewhat but is still bounded by a constant (i.e., it is $O(1)$). The find algorithm remains unchanged. The maximum time to perform a find is determined by Lemma 2.3.

Lemma 2.3 Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using `WeightedUnion`. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

Proof: The lemma is clearly true for $m = 1$. Assume it is true for all trees with i nodes, $i \leq m - 1$. We show that it is also true for $i = m$. Let T be a tree with m nodes created by `WeightedUnion`. Consider the last union operation performed, $Union(k, j)$. Let a be the number of nodes in tree j, and $m - a$ the number in k. Without loss of generality we can assume $1 \leq a \leq \frac{m}{2}$. Then the height of T is either the same as that of k or is one more than that of j. If the former is the case, the height of T is $\leq \lfloor \log_2(m - a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$. However, if the latter is the case, the height of T is $\leq \lfloor \log_2 a \rfloor + 2 \leq \lfloor \log_2 \frac{m}{2} \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$. □

Example 2.4 shows that the bound of Lemma 2.3 is achievable for some sequence of unions.

Example 2.4 Consider the behavior of `WeightedUnion` on the following sequence of unions starting from the initial configuration $p[i] = -count[i] = -1$, $1 \leq i \leq 8 = n$:

$$Union(1,2),\ Union(3,4),\ Union(5,6),\ Union(7,8),$$

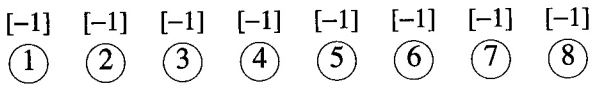
$$Union(1,3),\ Union(5,7),\ Union(1,5)$$

The trees of Figure 2.23 are obtained. As is evident, the height of each tree with m nodes is $\lfloor \log_2 m \rfloor + 1$. □

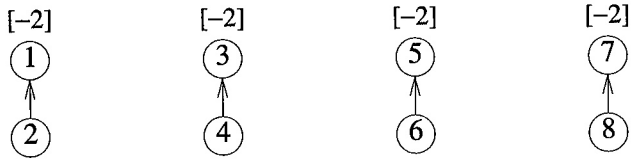
From Lemma 2.3, it follows that the time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n-tree forest.

Surprisingly, further improvement is possible. This time the modification is made in the find algorithm using the *collapsing rule.*

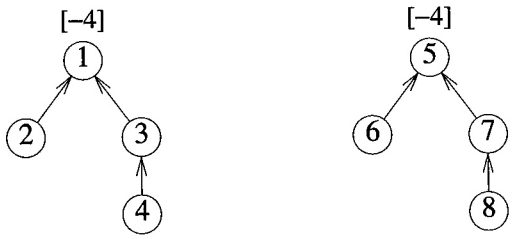
Definition 2.6 [Collapsing rule]: If j is a node on the path from i to its root and $p[i] \neq root[i]$, then set $p[j]$ to $root[i]$. □



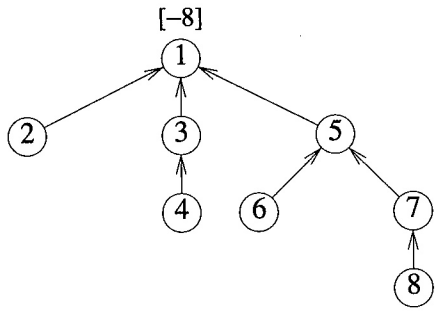
(a) Initial height-1 trees



(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)



(c) Height-3 trees following *Union*(1,3) and (5,7)



(d) Height-4 tree following *Union*(1,5)

Figure 2.23 Trees achieving worst-case bound

```
int Sets::CollapsingFind(int i)
// Find the root of the tree containing
// element i. Use the collapsing rule to
// collapse all nodes from i to the root.
{
    int r = i;
    while (p[r] > 0) r = p[r]; // Find root.
    while (i != r) { // Collapse nodes from i to root r.
       int s = p[i]; p[i] = r; i = s;
    }
    return(r);
}
```

Program 2.18 Find algorithm with collapsing rule

Function `CollapsingFind` (Program 2.18) incorporates the collapsing rule.

Example 2.5 Consider the tree created by `WeightedUnion` on the sequence of unions of Example 2.4. Now process the following eight finds:

$$Find(8),\ Find(8), \ldots,\ Find(8)$$

If `SimpleFind` is used, each $Find(8)$ requires going up three parent link fields for a total of 24 moves to process all eight finds. When `CollapsingFind` is used, the first $Find(8)$ requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, `CollapsingFind` will reset three (the parent of 5 is reset to 1). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. □

In the algorithms `WeightedUnion` and `CollapsingFind`, use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-case time over a sequence of finds. The worst-case complexity of processing a sequence of unions and finds using `WeightedUnion` and `CollapsingFind` is stated in Lemma 2.4. This lemma makes use of a function $\alpha(p, q)$ that is related to a functional inverse of Ackermann's function $A(i, j)$. These functions are defined as follows:

$$\begin{array}{ll} A(1,j) = 2^j & \text{for } j \geq 1 \\ A(i,1) = A(i-1,2) & \text{for } i \geq 2 \\ A(i,j) = A(i-1, A(i,j-1)) & \text{for } i,j \geq 2 \end{array}$$

$$\alpha(p,q) = \min\{z \geq 1 | A(z, \lfloor \frac{p}{q} \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

The function $A(i,j)$ is a very rapidly growing function. Consequently, α grows very slowly as p and q are increased. In fact, since $A(3,1) = 16$, $\alpha(p,q) \leq 3$ for $q < 2^{16} = 65,536$ and $p \geq q$. Since $A(4,1)$ is a very large number and in our application q is the number n of set elements and p is $n+f$ (f is the number of finds), $\alpha(p,q) \leq 4$ for all practical purposes.

Lemma 2.4 [Tarjan and Van Leeuwen] Assume that we start with a forest of trees, each having one node. Let $T(f,u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq \frac{n}{2}$. Then

$$k_1[n + f\alpha(f+n,n)] \leq T(f,u) \leq k_2[n + f\alpha(f+n,n)]$$

for some positive constants k_1 and k_2. □

The requirement that $u \geq \frac{n}{2}$ in Lemma 2.4 is really not significant, as when $u < \frac{n}{2}$, some elements are involved in no union operation. These elements remain in singleton sets throughout the sequence of union and find operations and can be eliminated from consideration, as find operations that involve these can be done in $O(1)$ time each. Even though the function $\alpha(f,u)$ is a very slowly growing function, the complexity of our solution to the set representation problem is not linear in the number of unions and finds. The space requirements are one node for each element.

In the exercises, we explore alternatives to the weight rule and the collapsing rule that preserve the time bounds of Lemma 2.4.

EXERCISES

1. Suppose we start with n sets, each containing a distinct element.

   (a) Show that if u unions are performed, then no set contains more than $u+1$ elements.

   (b) Show that at most $n-1$ unions can be performed before the number of sets becomes 1.

   (c) Show that if fewer than $\lceil \frac{n}{2} \rceil$ unions are performed, then at least one set with a single element in it remains.

   (d) Show that if u unions are performed, then at least $\max\{n-2u, 0\}$ singleton sets remain.

```
1   float Kruskal(int E[][SIZE], float cost[][SIZE],
2                 int n, int t[][2])
3   // E is the set of edges in G. G has n vertices.
4   // cost[u][v] is the cost of edge (u,v). t is
5   // the set of edges in the minimum-cost spanning
6   // tree. The final cost is returned.
7   {
8      int parent[SIZE];
9      construct a heap out of the edge costs
10     using Heapify;
11     for (int i=1; i<=n; i++) parent[i] = -1;
12     // Each vertex is in a different set.
13     i = 0; float mincost = 0.0;
14     while ((i < n-1) && (heap not empty)) {
15        delete a minimum cost edge (u,v) from the
16        heap and reheapify using Adjust;
17        int j = Find(u); int k = Find(v);
18        if (j != k) {
19           i++;
20           t[i][1] = u; t[i][2] = v;
21           mincost += cost[u][v];
22           Union(j, k);
23        }
24     }
25     if (i != n-1) cout << "No spanning tree" << endl;
26     else return(mincost);
27  }
```

Program 4.10 Kruskal's algorithm