Disjoint Set ADT (Chapter 8)

We will describe an efficient data structure to solve the equivalence problem. We start by defining the notion of an equivalence relation.

An *equivalence relation* is a relation R that satisfies three properties:

1. (*Reflexive*) $a\ R\ a$, for all $a \in S$.
2. (*Symmetric*) $a\ R\ b$ if and only if $b\ R\ a$.
3. (*Transitive*) $a\ R\ b$ and $b\ R\ c$ implies that $a\ R\ c$.

Are the following relations equivalence relations? $\leq$, same-country, friend, married-to, $=$.

We denote the equivalence relation by $\sim$.

Equivalence problem: for any a and b, decide whether $a \sim b$ is true.

Applications of the Equivalence Problem

Routing Problem:

Is it possible to send a file from any computer on the network to any other? (The same holds for airplane flights, mailing packages, etc.)

The equivalence problem also arises in graph theory algorithms (Chpt. 9) and in compilers where we need to decide whether two variables have an equivalent type given a variety of type declarations and assignment operations that change the type of a variable.

Disjoint Set ADT

The equivalence *class* of an element $a \in S$ is the subset of S that contains all the elements that are equivalent to a.

Operations:

- FIND(x): returns the name of the set equivalence class containing x.
- UNION(a, b): merges the two equivalence classes containing a and b into a new equivalence class (detroying the original classes).

The algorithm to support these operations is known as the Disjoint Set UNION-FIND Algorithm. Every element of S appears in exactly one equivalence class.

To decide if $a \sim b$, we need only check whether a and b are in the same equivalence class.

The Dynamic Equivalence Problem

Input: a collection of n sets, each with exactly one element.

Each set has a different element, so that $S(i) \cap S(j) = \emptyset$; i.e., the sets are disjoint.

The problem is called dynamic because the set can change via the UNION operation.

We will now consider two cases:

1. where the FIND instruction can be performed in constant worst-case time; and
2. the UNION instruction can be perform in constant worst-case time.

It has been shown that it's impossible to have both UNION and FIND take constant worst-case time.

Fast Find Data Structure

Assume that all the elements are numbered sequentially from 1 to n.

How could we perform a FIND operation in constant worst-case time?

How do we perform UNION(a, b)?

How long does this operation take in the worst case?

Starting with n disjoint sets, what is the maximal number of UNION operations we can execute?

So, what is the worst-case running time for an arbitrary sequence of UNION-FIND instructions?

Suppose we kept all the elements that are in the same equivalence class in a linked list. The linked list is maintained in addition to the array described earlier.

Would this slow down the FIND operation?

Would this speed up the UNION operation in the worst case?

Suppose we keep track of the size of each equivalence class, and then always merge the smaller one into the larger one.

This trick means that the total time spent for $n-1$ unions is only $O(n \log n)$.

The reason is that each element can only change its equivalence class at most $\log n$ times.

Since every time its class is changed, the new equivalence class is at least twice as large as the old one.

We will now consider a data structure that makes UNIONs easy but FINDs harder.

Up Trees

We represent each set with a special kind of tree, where each element of tree points to its parent. The root of the tree contains the name of the set.

To perform a UNION of two sets, we merge the two trees by making the root of one tree point up to the root of the other tree. This operation takes constant time. (See Figs. 8.1-8.4 for an example.)

FIND(x) is performed by returning the root of the tree containing x.

How long does this operation take?
How are Up Trees represented?

We can represent the tree in an array, where each entry $P(i)$ represents the parent of element i. (See Fig. 8.5).

Smart Union Algorithms

Why don't we keep the set name information in every node in the tree, thereby making FIND constant time?

Instead, we always make the smaller tree a subtree of the larger tree.
(This is called UNION-BY-SIZE.)

We can prove that this method guarantees that the depth of any node is never more than $\log n$.

Basically, each time the depth of the tree is increased by 1, the number of nodes in the tree at least doubles.

Thus, the depth can be increased at most $\log n$ times if we have n nodes.

Path Compression

In path compression, we perform extra work when we do a FIND in order to speed up future FIND operations.

Suppose the operation is FIND(x). The effect of path compression is that *every* node on the path from x to the root has its parent changed to the root.

Figs. 8.12 and 8.14 show us the before and after behavior for FIND(15).

How much extra work is done each time?

Ackerman's Function

The time required to do m UNION-FIND operations is $O(m\ \alpha(m,n))$, where $\alpha(m,n)$ is the functional inverse of Ackerman's function, which is defined as follows:

$A(1,j) = 2^j$ for $j \geq 1$

$A(i,1) = A(i-1,2)$ for $i \geq 2$

$A(i,j) = A(i-1, A(i,j-1))$ for $i,j \geq 2$

$A(i) = A(i,i)$

$A(2) = A(2,2) = A(1,4) = 2^4$

$A(4) = 2^2, \quad 65536$ Times.