Data Structures for Disjoint Sets

S = a set of n elements

R = relation defined over S. For every pair a, b ∈ S, aRb (a is related to b by R) is true or false.
R is reflexive: aRa is true for all elements in S

R is symmetric: aRb is true if and only if bRa is true for every pair a, b ∈ S

R is transitive: if aRb and bRc are true then it implies that aRc is true for every triple a, b, c ∈ S
R is an Equivalence Relation (ER): R is reflexive, symmetric and transitive.

Two sets S1 and S2 are disjoint if S1 ∩ S2 = Φ (the empty set). An Equivalence Class (EC) of an element a in S is a subset S' of S containing all elements in S that are related to a by R. ECs form a partition of S, i.e., divides S into subsets $S_1,..S_n$ that are disjoint (have no element in common).
We need a data structure that allows us to represent ER's and EC's so that:

1. The Equivalence Problem (EP) can be solved: given an ER R, and a pair of elements a, b ∈ S, is aRb true? aRb is true if a and b belong to the same EC.
2. We also want to be able to maintain a database of elements and their EC's corresponding to some ER, so that new assertions relating two elements can be efficiently added to the database.

Disjoint set is a data structure that allows us to do both of the above efficiently.

Basic idea

For the set S and relation R maintain all ECs using the disjoint set data structure.
Then aRb can be determined by checking whether the EC of a = EC of b.
To assert aRb, combine the ECs of a and b into one EC, if they originally were in different ECs.
So we need two operations:

Find-Set(e) returns the EC or disjoint set to which data element e belongs.
Union(set1,set2) merges the two sets unless these are the same.

At the beginning, if S has n elements, we allow each data element to be in a disjoint set by itself since we don't know which elements are related. As we assert relations between pairs of elements we will do the corresponding unions. We assume that data elements are numbered 1..n. We also name sets with numbers 1,2,3…n. The name of a set is the number of a *representative* data element in that set.

Implementation of the Basic Idea: Approach 1

Use an array A with A[i] containing the name of the set that contains element i.

Find is Θ(1) best, worst and average case since Find-Set(i) = A[i].
Union(i,j) where i and j are names of sets - if i and j are different, change all j's to i in A (or vice versa). Union is Θ(n) best, worst and average case.
If initially each element is in its own set and we assert m relations between element pairs, doing the necessary finds and unions, this requires O(mn) time.

Implementation of the Basic Idea: Approach 2

Keep all elements in the same set in a tree, with the name of the set being the same as the number of the data element in the root of the tree. The tree is implemented as an array P where P[i] = parent of i if i is not a root, and 0 if i is the root.

Find-Set(i) then returns the root of the tree containing i and has a time complexity of Θ(depth of node containing i) =O(n) worst case and Θ(1) best case. Union(i, j) where i and j are the roots of two trees containing two sets: if i and j are different, then set P[j] = i (or vice versa). Union is Θ(1) best, worst and average case. If initially each element is in its own set and we assert m relations between element pairs, doing the necessary finds and unions, this requires O(mn) worst case time.
Note that when Find-Set is efficient, Union is not and vice versa. It has been shown that we can't do both Find-Set and Union in constant time. In both approaches, a sequence of n unions and finds can take $O(n^2)$ worst case time - this quadratic complexity is not very good. Solution to this problem: "Smart" Union algorithms.

Smart Union: Union-by-Size

Union(i,j) - make the smaller tree a subtree of the larger tree.

Implementation: store the negative of tree size in the array cell corresponding to the root node instead of 0; add sizes when doing a union. Since tree size will be at least 1, a root will be identifiable by the negative integer in the corresponding array cell.

This makes Find-Set O(logn) worst case, O(1) average case, and Union Θ(1) always. A sequence of m union/finds has complexity O(mlogn).
Depth of any node in a tree with n nodes will never exceed logn if unions are done by size.
Thinking Assignment: The worst case for Union-by-Size is when the trees being merged are all the same size - Why?

Smart Union: Union-by-Height

Union(i,j) - make the shallower tree a subtree of the deeper one.

Implementation: take the negative of tree height and store this in the array cell corresponding to the root instead of 0; add a –1 to the negative value of height when doing the union of two trees of same height to reflect the fact that when you union trees of the same height, the height of the resulting tree will increase by one. When implementing

this idea in a program, to avoid the need for representing –0 for trees of zero height, the negative value of (tree height + 1) is used instead of the negative of tree height.
Thinking Assignment: When union-by-height is done, the depth of a node in a tree of n nodes will never exceed logn. Show this using an informal proof similar to that for Union-by-Size discussed in class.
Path Compression (PC)

We can't improve the Union algorithm any more. The worst cases in smart unions (when trees are of equal size/height) can easily occur so that the resulting tree depth is logn, so m Find-Set operations can take O(mlogn) time. To improve this situation, we now modify how the Find-Set algorithm works.

Whenever Find-Set(x) is done, restructure the tree so that every node in the path from the root to x is made a direct child of the root. Thus, later find operations on these nodes will be constant time operations!

PC is compatible with Union-by-size. This is because PC does not change the size of a tree. Union-by-size and Find-Set-with-PC make a very efficient combination.

PC is not compatible with Union-by-height since PC changes the height of the tree. It is computationally expensive to recompute tree heights every time PC is done. So if Union-by-height and Find-Set-with-PC are to be implemented together, what is done is to store *estimates* of heights instead of actual heights. Unions using these height estimates (called ranks) are called Union-by-Rank. In practice this works as well as Union-by-height.

Disjoint Set Summary
This data structure supports storing a set S of elements in such a way that asserting that an equivalence relation R holds among pairs of elements, and checking whether any given pair of elements are related by R, can both be done very efficiently.

All elements related to each other by R belong to an equivalence class. These classes partition the set S into disjoint subsets S_i. Asserting e_i R e_j where e_i and e_j are elements in S then amounts to "unioning" the equivalence classes to which e_i and e_j belong. Checking whether e_i and e_j are related amounts to "finding" the equivalence classes of e_i and e_j and checking whether these are one and the same. Hence, the fundamental operations are Union and Find-Set.

Two implementation approaches:
Approach 1: As an array whose indexes stand for elements, and the name of the equivalence class to which an element belongs to is stored in the cell indexed by that element. This implementation supports a $\Theta(1)$ Find-Set and $\Theta(n)$ Union.
Approach 2: As an array that implements a forest of trees, with each tree containing elements belonging to the same equivalence class. Array indexes stand for elements and array cells contain the parent of an element if it is not the root of a tree and 0 (or a negative number) if it is the root. This implementation yields a Θ(node depth) Find-Set and $\Theta(1)$ Union.
Approach 2 is commonly used. In this case, the Union, which is a merge of two trees, can be done in one of 3 ways:
Arbitrary: make either of the two trees a subtree of the root of the other.
Union-by-Size: make the smaller tree a subtree of the root of the larger one.
Union-by-Height: make the shallower tree a subtree of the root of the deeper one.
Independently of the way Union is implemented, Find-Set can be implemented with or without *Path Compression*. This gives six possible combinations of Union and Find-Set implementations with implementation Approach 2.

In summary, we can implement Find-Set with or without Path Compression, and Unions in three ways: arbitrary, by-size, or by-height. When Unions are arbitrary, a sequence of m Union/Find-Set operations will require O(mn) time, but if Find-Sets are done with Path Compression will require only O(mlogn) time. When Unions are done by size or height, a sequence of m Union/Find-Set can be done in O(mlogn) time. The best combination is Unions by size or rank and Find-Sets with Path Compression, in this case the time complexity of a sequence of m Union/Find-Set operations is still O(mlogn) but in practice will execute much faster than Union by size/height plus Find-Set without Path Compression.