DISJOINT SETS AND UNION/FIND ALGORITHM

1.0 EQUIVALENCE RELATIONS

A relation is represented by ψ. Let a and b be the two elements of set S. We say that a ψ b if a and b are related (i.e. if they belong to the same set).

An equivalence relation is one that satisfies the following three properties.

i. **Reflexive:** a ψ a, for all a $\in$ S

ii. **Symmetric:** a ψ b if and only if b ψ a

iii. **Transitive:** a ψ b and b ψ c => a ψ c

An example of equivalence relation is electrical connectivity. Here all the connections with wires are an equivalence relation. This is shown by the fact that the above three properties hold good for an electrical connection.

1.1 THE DISJOINT SET ADT

A disjoint set data structure keeps note of a set of non-overlapping subsets. It is a data structure that helps us solve the dynamic equivalence problem.

1.1.1 DYNAMIC EQUIVALENCE PROBLEM

The dynamic equivalence problem expresses the problem of deciding, given two elements a and b, if a ψ b. The problem is often because the relation is not explicitly, but implicitly defined. The equivalence class of an element a $\in$ S is the subset of S that contains all the elements that are related to a. To decide if a ψ b, we need only to check whether a and b are in the same equivalence class.

We are given an input of n sets each containing one element. Initially all sets are represented in such a way that they are not related. This means that all the sets are disjoint and can be represented as $S_i \cap S_j = \varnothing$

The three permissible operations on disjoint sets are:

i. **MAKE-SET:** It creates a set with only one member in each set.
ii. **FIND-SET:** Returns the name of the set (equivalence class) containing a given element

iii. **UNION:** If we want to add the relation a ψ b, then we first see if a and b are already related. This is done by performing finds on both a and b and checking whether they are in the same equivalence class. If they are not, then we apply union. This operation merges the two equivalence classes containing a and b into a new equivalence class.

This is known as **UNION/FIND** algorithm. This algorithm is dynamic because, as the algorithm proceeds, the sets can change via the union operation. The algorithm operates in two modes:

i. **On-line**: Each operation should give an answer before continuing to the next operation.
ii. **Off-line:** This type of algorithm is allowed to see the entire sequence of operations before starting to implement and use them.

1.2. VARIOUS REPRESENTATIONS OF DISJOINT SET ADT

1.2.1. ARRAY REPRESENTATION

This representation assigns one position for each element. Each position stores the element and an index to the representative. To make the Find-Set operation fast we store the name of each equivalence class in the array. Thus the find takes constant time, O(1). Assume element a belongs to set i and element b belongs to set j. When we perform Union(a,b) all j's have to be changed to i's. Each union operation unfortunately takes Θ(n) time. So for n-1 unions the time taken is $\Theta(n^2)$.

1.2.2. LINKED LIST REPRESENTATION

All the elements of an equivalence class is maintained in a linked list. The first object in each linked list is its set's representative. Each object in the linked list contains a set member, a pointer to the object containing the next member of the set, and a pointer back to the representative. Each list maintains head pointer, to the representative, and tail pointer, to the last object in the list. Within each linked list, the objects may appear in any order.

Consider a sequence of *n* Make-set operations followed by *n* - 1 Union operations, so that *m* = 2*n* - 1. We spend Θ(n) time performing the *n* Make-set operations. Because the *i*th Union operation updates *i* objects, the total number of objects updated by all *n* - 1 UNION operations is $\sum_{i=1}^{n-1} i = \Theta(n^2)$.

In the worst case, the implementation of the Union procedure takes an average of Θ(n) time per call because we may be appending a longer list onto a shorter list, and the pointer to the representative for each member of the longer list should be updated. With this simple weighted-union heuristic, a single Union operation can still take Ω(n) time if both sets have Ω(n) members. A sequence of m Make-set, Union and Find-Set operations, n of which are Make-set operations take O(m+nlogn) time.

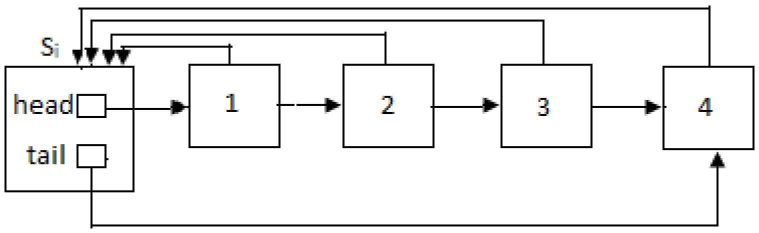


Figure 1.1 Linked list representation of equivalent set S_i

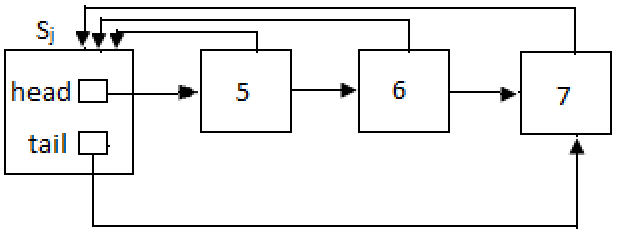


Figure 1.2 Linked list representation of equivalent set S_j

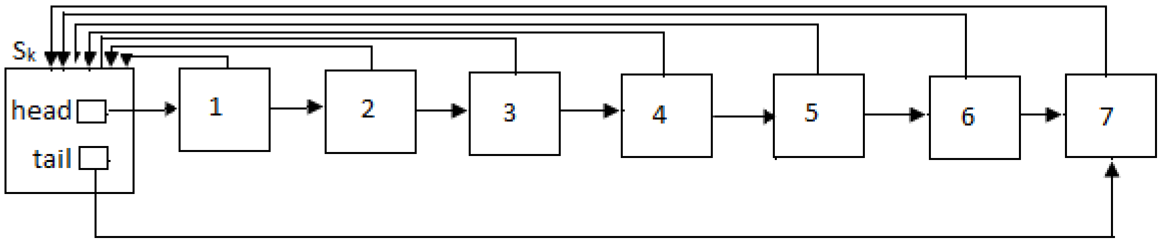


Figure 1.3 Linked list representation after union ($S_{i,}$ S_j)

1.2.3. TREE REPRESENTATION

(Examples and pseudocode for tree representation taken from Data Structures and Algorithms in C – Mark Allen Weiss & Introduction to Algorithms - Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein)

A tree data structure can be used to represent a disjoint set ADT. Each set is represented by a tree. The elements in the tree have the same root and hence the root is used to name the set. The trees do not have to be binary since we only need a parent pointer.

Make-set (DISJ_SET S)

```
 int i;
  for( i = N; i > 0; i-- )
       p[i] = 0;
```

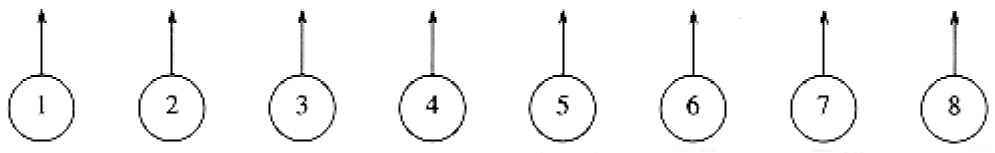


Figure 1.4 Tree representation of disjoint set ADT after Make-set operation

Initially, after the Make-set operation, each set contains one element. The Make-set operation takes O(1) time.

set_type **Find-set**(element_type x, DISJ_SET S)

```
  if( p[x] <= 0 )
   return x;
  else
   return( find( p[x], S ) );
```

The Find-Set operation takes a time proportional to the depth of the tree. This is inefficient for an unbalanced tree

void **Union**(DISJ_SET S, set_type root1, set_type root2)

```
  p[root2] = root1;
```

The union operation takes a constant time of O(1).

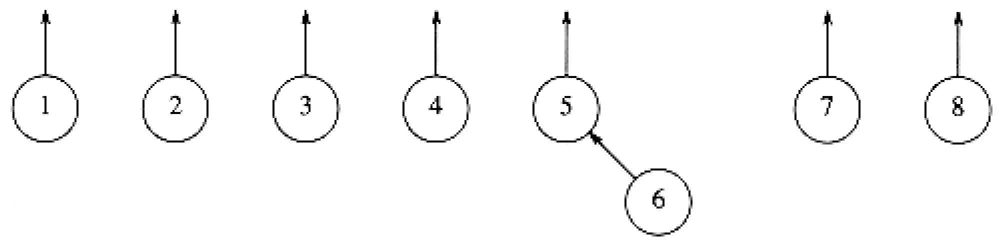


Figure 1.5 Tree representation of disjoint set ADT after union (5,6)

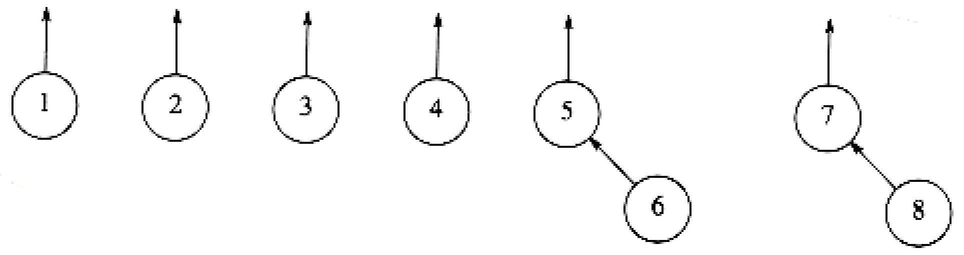


Figure 1.6 Tree representation of disjoint set ADT after union (7,8)

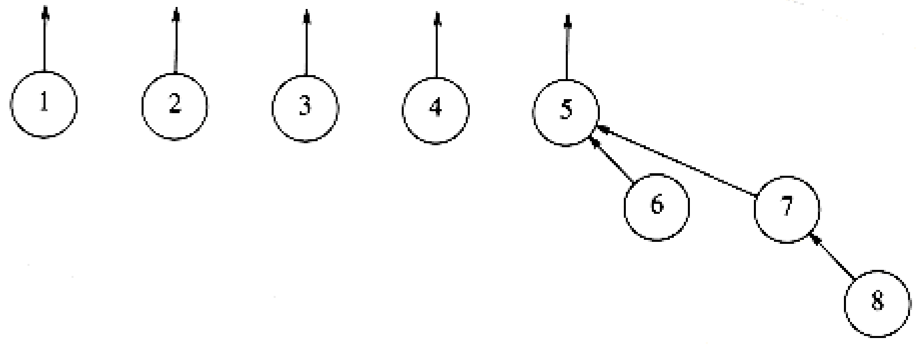


Figure 1.7 Tree representation of disjoint set ADT after union (5,7)

1.3 DISJOINT SET FORESTS

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, each member points only to its parent. The root of each tree contains the representative and is its own parent. Algorithms that use this representation are no faster than the linked-list representation. For this reason, we use two heuristics to achieve asymptotically fastest disjoint set. The two heuristics are:

i. Smart Union Algorithm
ii. Path Compression

1.3.2. SMART UNION ALGORITHM

(Examples and pseudocode for smart union algorithm & path compression taken from Data Structures and Algorithms in C – Mark Allen Weiss & Introduction to Algorithms - Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein)

The *unions* in the basic tree data structure representation were performed arbitrarily, by making the second tree a subtree of the first. A basic improvement is to make the smaller tree a subtree of the larger. We call this approach ***union-by-size***.

If *unions* are done by size, the depth of any node is never more than log n. Note that a node is initially at depth 0. When its depth increases as a result of a *union*, it is placed in a tree that is at least twice as large as before. Thus, its depth can be increased at most log n times. This implies that the running time for a *find* operation is $O(\log n)$, and a sequence of m operations takes $O(m \log n)$.

We need to keep track of the size of each tree. Let us assign a size variable for each node and let it contain the size of the tree (Initially a 0 0r 1 according to the convenience). When a *union* is performed, check the sizes and make the new size as the sum of the old. Thus, union-by-size is not at all difficult to implement and requires no extra space. It is also fast, on average. It has been shown that a sequence of m operations requires $O(m)$ average time if union-by-size is used. This is because when random *unions* are performed, small sets are merged with large sets throughout the algorithm.

An alternative implementation, which also guarantees that all the trees will have depth at most $O(\log n)$, is ***union-by-rank***. We keep track of the height, instead of the size, of each tree and perform *unions* by making the shallow tree a subtree of the deeper tree. This is an easy algorithm, since the height of a tree increases only when two equally deep trees are joined (and then the height goes up by one). Thus, union-by-height is a trivial modification of union-by-size.

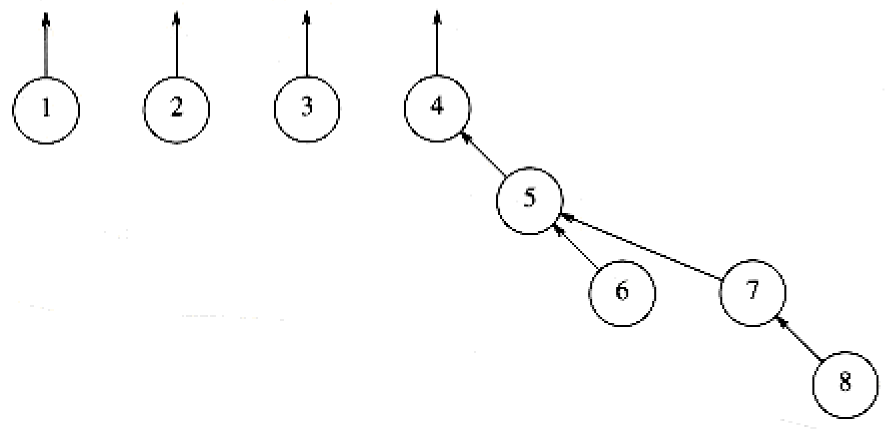


Fig.1.8. Result of arbitrary union

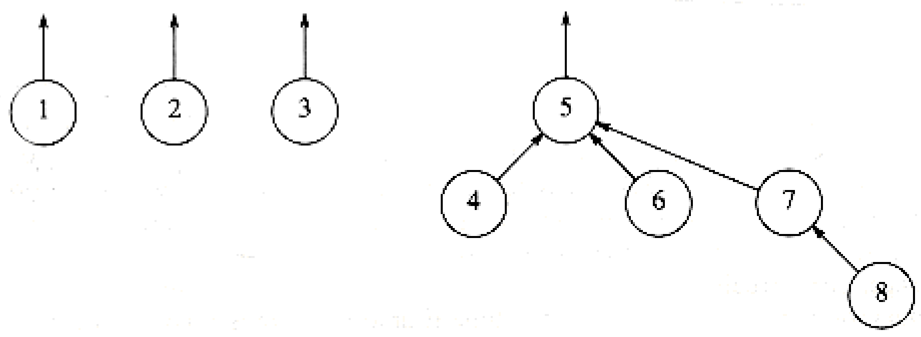


Fig. 1.9 Result of union-by-size/ union-by-rank

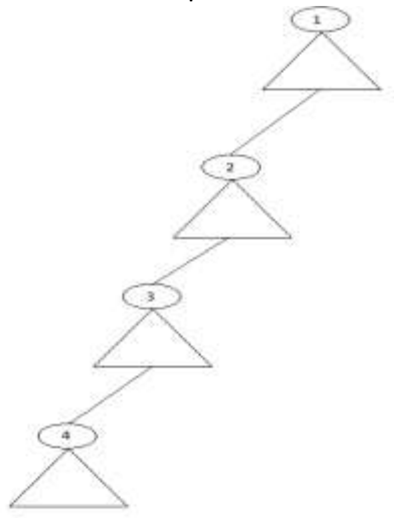
A small snippet for the union-by-rank algorithm is shown below: Assume x and y are two nodes

```
Union(x, y)
  Link(Find-set(x), Find-set(y))

Link(x, y)
 if rank[x] > rank[y]
    then p[y] = x
   else p[x] = y
       if rank[x] == rank[y]
          then rank[y] = rank[y] + 1
```

1.3.3. PATH COMPRESSION

The second heuristic, ***path compression***, is also quite simple and very effective. As shown in Figure 1.10, we use it during Find-set operations to make each node on the find path point directly to the root. Path compression does not change any ranks.



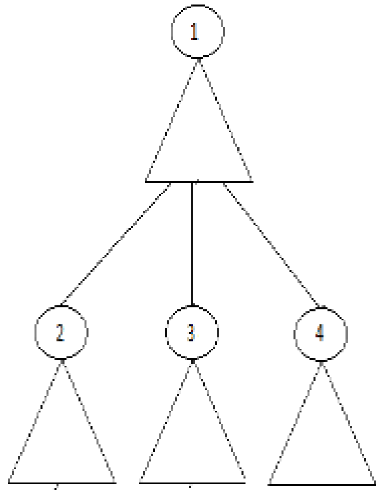


Fig.1.10(a) **Fig.1.10(b)**

Figure 1.10: Path compression during the operation Find-set. *(a)* A tree representing a set prior to executing Find-set(*a*). *(b)* The same set after executing Find-set(*a*). Each node on the find path now points directly to the root.

The Find-set procedure is a ***two-pass method***: it makes one pass up the find path to find the root, and it makes a second pass back down the find path to update each node so that it points directly to the root.

Snippet for Find-set function using path compression:

```
Find-set($x$)
  **if** $x \neq p[x]$
     **then** $p[x]$ = Find-set($p[x]$)
  **return** $p[x]$
```

Union-by-rank or path-compression improves the running time of the operations on disjoint-set forests, and the improvement is even better when the two heuristics are used together.

We do not prove it, but, if there are n Make-set operations and at most n - 1 Union operations and f Find-set operations, the path-compression heuristic alone gives a worst-case running time of $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

When we use union by rank and path compression together, the worst-case running time is $O(m\,\alpha(m,n))$, where $\alpha(m,n)$ is a *very* slowly growing function and the value of it is derived from the inverse of Ackermann function. In any application of a disjoint-set data structure, $\alpha(m,n) \le 4$. Thus, for practical purposes the running time can be viewed as linear in *m (no. of operations)* . The Ackermann function has been explained in detailed in the next section.

1.4. ACKERMANN FUNCTION

Assume m and n are non-negative integers. The Ackermann Function is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

[Source: Wikipedia]

The value of this function grows rapidly, even for small inputs.

For example $A(4,2)$ is an integer of 19,729 decimal digits.

Since the function $f(n) = A(m, n)$ grows very rapidly, its inverse function, f^{-1}, grows very slowly. This **inverse Ackermann function** f^{-1} is usually denoted by $\boldsymbol{\alpha}$. $\alpha(m,n)$ is less than 5 for any practical input size *n.*

This running time that makes use of the inverse Ackermann function can be proved by doing the amortized analysis on the potential method. A high level overview on these topics is given in the coming sections.

1.5. AMORTIZED ANALYSIS

In amortized analysis, we average the time required to perform a sequence of data structure operations over all the operations performed in the algorithm.

Using this analysis we can show that the average cost of an operation is small if we average over a sequence of operations, even though a single operation might be expensive.

Amortized analysis guarantees average performance of each operation in worst case.
To carry out this analysis we assign a charge for different operations. This cost is known **amortized cost.**

When the amortized cost for an operation exceeds the actual cost, the difference is assigned to different objects in the data structure as **"credit"**.

Credit can later pay for operations where the amortized cost ends up being lesser than the actual cost.

We build the potential method based on the above concepts.

1.6. POTENTIAL METHOD

In the potential method, instead of representing the prepaid work as credit, we represent it as something called as "potential" which can be utilized to pay for future operations.

Assume that we are performing n operations.

We start with initial data structure D_0

For i=1,2,…n operations let c_i be the actual cost.

Let D_i represent the data structure after applying the ith operation to the data structure D_{i-1}

Potential function is represented by ϕ – maps each data structure D_i to real number

The amortized cost $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$\phi(D_i) - \phi(D_{i-1})$ – change in potential

1.7 SOME IMPORTANT PROPERTIES

(Lemmas taken from Data Structures and Algorithms in C – Mark Allen Weiss & Introduction to Algorithms - Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein)

LEMMA 1.1.

When executing a sequence of union instructions, a node of rank r must have 2^r descendants (including itself).

PROOF:

By induction:

The base, $r = 0$, is clearly true. Let *T* be the tree of rank *r* with the fewest number of descendants and let *x* be *T*'s root. Suppose the last *union* *x* was involved in was between T_1 and T_2. SupposeT_1's root was *x*. If T_1 had rank *r*, then T_1 would be a tree of height *r* with fewer descendants than *T*, which contradicts the assumption that *T* is the tree with the smallest number of descendants. Hence the rank of $T_1 \leq r - 1$. The rank of $T_2 \leq$ rank of T_1. Since *T* has rank *r* and the rank could only increase because of T_2, it follows that the rank of $T_2 = r - 1$. Then the rank of $T_1 = r - 1$. By the induction hypothesis, each tree has at least 2^{r-1} descendants, giving a total of 2^r and establishing the lemma.

LEMMA 1.2.

The number of nodes of rank r is at most $n/2^{r}$.

PROOF:

Without path compression, each node of rank *r* is the root of a subtree of at least 2^r nodes. No node in the subtree can have rank *r*. Thus all subtrees of nodes of rank *r* are disjoint. Therefore, there are at most $n/2^r$ disjoint subtrees and hence $n/2^r$ nodes of rank *r*.

LEMMA 1.3.

For all nodes x, we have rank[x] ≤ rank[p[x]], with strict inequality if x ≠ p[x].

PROOF:

The value of rank[x] is initially 0 and increases through time until x ≠ p[x]; from then on, rank[x] does not change. The value of rank[p[x]] monotonically increases over time.

LEMMA 1.4.

Every node has rank at most n - 1.

PROOF:

Each node's rank starts at 0, and it increases only upon Link operations. Because there are at most *n* - 1 Union operations, there are also at most *n* - 1 Link operations. Because each Link operation either leaves all ranks alone or increases some node's rank by 1, all ranks are at most *n* - 1.

1.8. APPLICATIONS

Problem:

We need to form a network of computers and bidirectional connections between them. Each of these connections allow a file transfer from one computer to another.

Solution:

- Initially put each computer in it’s own set.
- Computers can transfer files to other computers if and only if they are in the same set.
- If they are not in the same set perform union operation.
- At the end of the algorithm, the graph is connected if and only if there is exactly one set.
- If there are M connections and N computers, the space requirement become O(N)
- Using path compression and union-by-rank we can achieve the worst case O(M α(N))

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

Another application of the disjoint set data structure is that it is used in Kruskal’s algorithm to check for safe edges and to eliminate the unsafe ones. Only the safe edges get added to the graph in order to generate a minimum spanning tree.

BIBLIOGRAPHY

- Data Structures and Algorithms in C – Mark Allen Weiss
- Introduction to Algorithms - Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein