INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

# COURSE HANDOUT

| | |
|---|---|
| Course Code | ACSC13 |
| Course Name | Design and Analysis of Algorithms |
| Class / Semester | IV SEM |
| Section | A-SECTION |
| Name of the Department | CSE-CYBER SECURITY |
| Employee ID | IARE11023 |
| Employee Name | Dr K RAJENDRA PRASAD |
| Topic Covered | Disjoint set operations, union and find algorithms |
| Course Outcome/s | Make use of union and find algorithms |
| Handout Number | 20 |
| Date | |

## Content about topic covered : Disjoint set operations, union and find algorithms

Set is a collection of elements.

Let U is finite universe of n elements. Assume sets will be constructed from U. These sets may be empty or contain any subset of the elements of U.

A common way to represent such sets is to allocate a bit vector of length n, SET(1:n), such that SET(i) = 1, if the $i^{th}$ element of U is in this set and 0 otherwise. This array is called Characteristic vector.

The advantage of this representation is:

1. One can quickly determine whether any particular element is present or not.
2. Operations such as union and intersection of sets can be carried out using logical-and and logical-or.

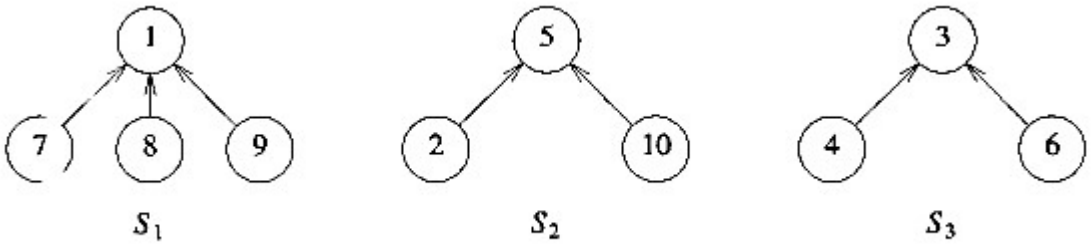
This is efficient particularly when n is small.

Another alternative is to represent a list of its elements.

Disjoint sets: if $S_i$ and $S_j$ are two sets, where $i \neq j$. if there is no element which is in both $S_i$ and $S_j$, then those two sets are called pairwise disjoint sets.

Eg: $S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$ $S_3 = \{3, 4, 6\}$

Sets are represented using trees as shown below:




$S_1$ $S_2$ $S_3$

**Disjoint set operations:**

Two operations can be performed on disjoint sets.

1. Disjoint set Union
2. Find

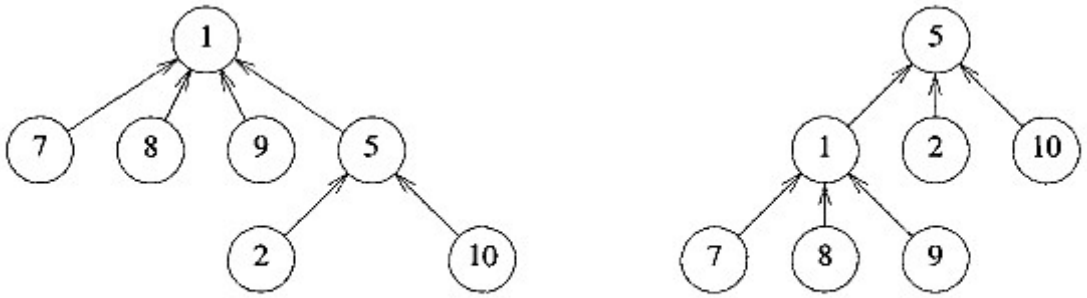
1. **Disjoint set Union**: If $S_i$ and $S_j$ are two disjoint sets, then their union
   $S_i \cup S_j$ = { all elements x such that x is in $S_i$ or $S_j$ }.

$$S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$$

2. **Find(i):** Find the set containing the element i.
   Eg: 4 is in set $S_3$
   9 is in set $S_1$

**Tree representation of Union:**

Make one of the trees as a subtree of the other.




$S_1 \cup S_2$ or $S_1 \cup S_2$

Since the set elements are numbered1throughn, we represent the tree nodes using an array p{1 :n], where n is the maximum number of elements.

The $i^{th}$ element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node.

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

Note that root nodes have a parent of -1.

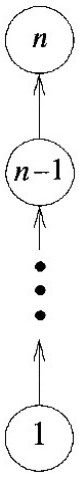
We can now implement Find(i) by following the indices, starting at I until we reach a node with parent value -1.For example, Find(6) starts at 6 and then moves to 6's parent, 3.Since p[3] is negative, we have reached the root. The operation Union(i,j) is equally simple. We pass in two trees with roots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

**Simple Union and Find Algorithms:**

```
Algorithm SimpleUnion(i, j)
{
    p[i] :=j;
 }
Algorithm SimpleFind(i)
{
   while (p[i] ≥0) do
        i :=p[i];
   return i;
}
```

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 (that is, $S_i = \{ i \}$ , $1 < i < q$), then the initial configuration consists of a forest with q nodes, and p[i] = 0, 1<i <q.

Now let us process the following sequence of union-find operations:
Union(l,2), Union(2,3), Union(3,4), Union(4,5), …Union(n-l, n), Find(l), Find{2), … Find{n)
This results in the degenerate tree




Time required for Union is constant. So all n-1 unions can be processed in O(n).

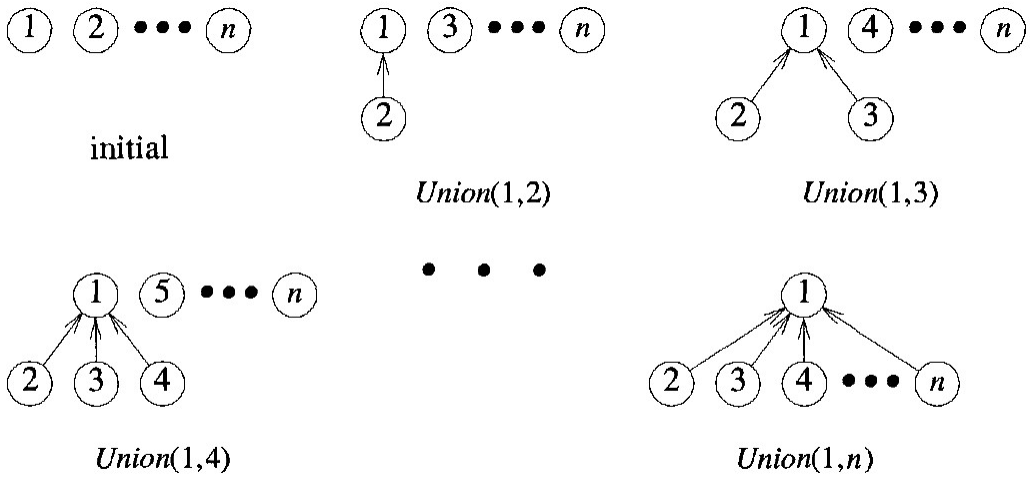
The time required to process a FIND for an element at level i of a tree is O(i).
Hence total time needed to process the n-2 finds is

$$1+2+3+\ldots+(n-2)=(n-2)(n-1)/2 = O(n^2).$$

We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees.

Weighting Rule for Union(i,j):

If the number of nodes in the tree with root i is less than the number of nodes in the tree with root j, then make j as the parent of i, otherwise make i as the parent of j.



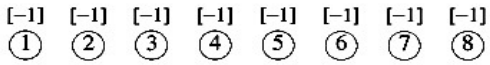

To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a count field in the root of every tree. If i is a root node, then count[i] equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number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

**Algorithm WeightedUnion(i,j)**

```
// Union sets with roots i and j, i ≠ j, using the weighting rule.
//p[i] = -count[i] and p[j] = -count[j].
{
        temp:=p[i] +p[j];
        if (p[i] >p[j]) then
        {  // i has fewer nodes.
                P[i] := j; P[j]:= temp;
        }
        Else
        {  // j has fewer or equal nodes.
                P[j]:=i;   p[i] := temp;
        }
}
```

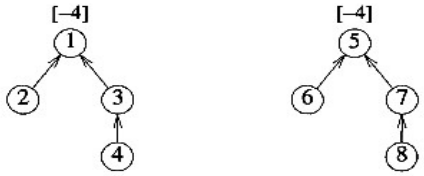
Consider the behavior of WeightedUnion on the following sequence of unions starting from the initial configuration p[i] = -count[i] = -1, 1≤i ≤8=n:

Union(l,2) , Union(3,4), Union(5,6), Union(7,8), Union( 1,3), Union(5,7), Union(1,5).

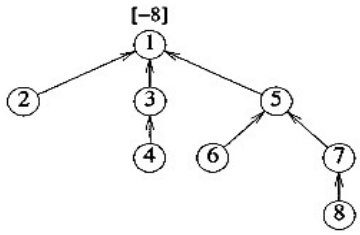



(a) Initial height-1 trees


(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)

The trees are obtained. As is evident, the height of each tree with m nodes is $[\log_2 m] + 1$.

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.

Collapsing Rule: If j is a node on the path from i to its root and p[i] ≠ root[i], then set p[j] to root[i].

Algorithm Collapsing Find(i)
//Find the root of the tree containing element i. Use the collapsing rule to //collapse all nodes from i to the root.

```
{
   r :=i;
   while (p[r]>0) do r :=p[r];  // Find the root.

   while (i ≠ r) do // Collapse nodes from i to root r.
   {
      s :=p[i];   p[i] :=r; i :=s;
   }
   return r;
}
```

Consider the tree created by WeightedUnion on the sequence of unions previously. Now process the following eight finds:

Find(8) , Find(8),…, 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.