Sets

It is assumed that the elements of the sets are the numbers 1, 2, 3, ..., 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* (that is, if Si and Sj, $i \neq j$, are two sets, then there is no element that is in both Si and Sj).

For example, when n = 10, the elements can be partitioned into three disjoint sets:

- S1 = {1, 7, 8, 9}
- S2 = {2, 5, 10}
- S3 = {3, 4, 6}

Figure shows one possible representation for these:

S1:

- 1 (representative)
  - 7
  - 8
  - 9

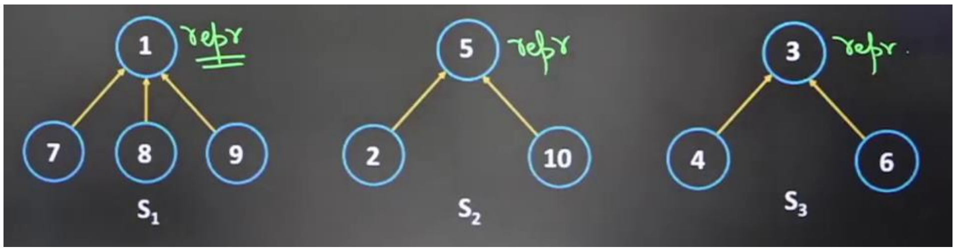
S2:

- 5 (repr)
  - 2
  - 10

S3:

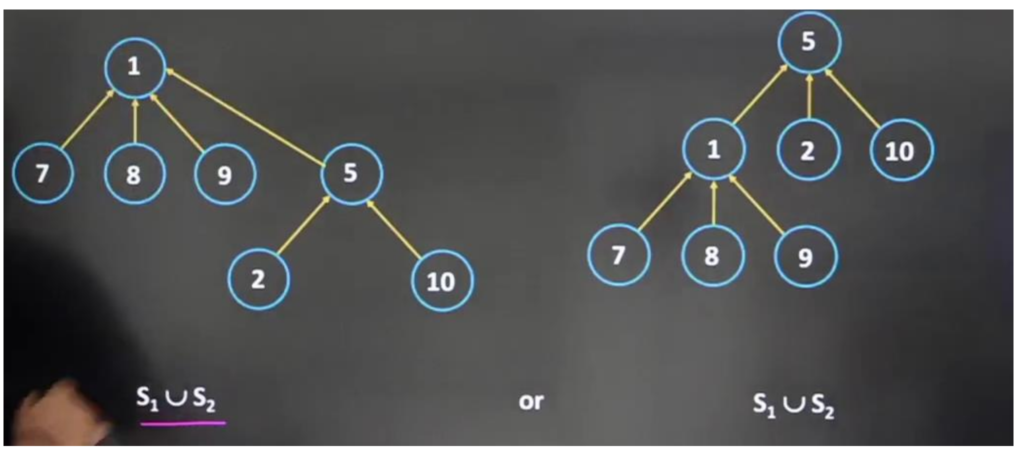
- 3 (repr)
    - 4
    - 6

Possible tree representation of sets



The operations we wish to perform on these sets are:

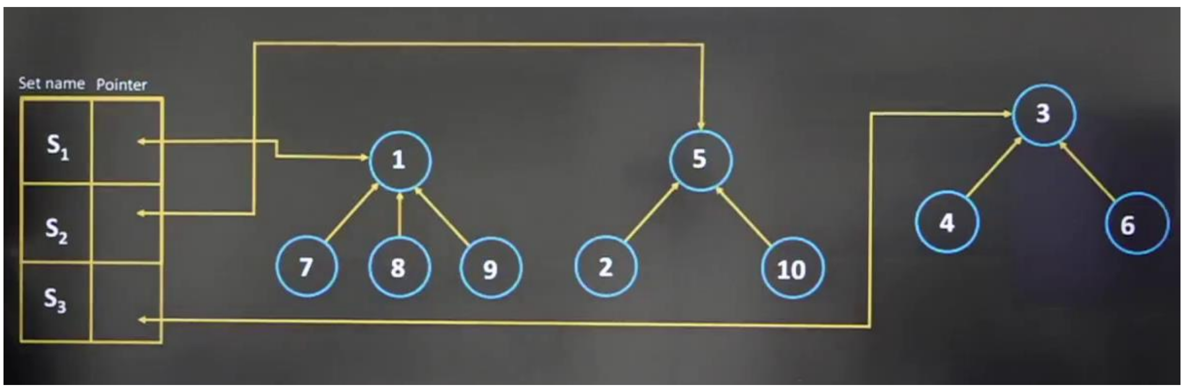
1. **Disjoint set union.** If Si and Sj are two disjoint sets, then their union Si ∪ Sj = all elements x such that x is in Si or Sj. Thus, S1 ∪ S2 = {1, 7, 8, 9, 2, 5, 10}. Since we have assumed that all sets are disjoint, we can assume that following the union of Si and Sj, the sets Si and Sj do not exist independently; that is, they are replaced by Si ∪ Sj in the collection of sets.



2. **Find(i).** Given the element i, find the set containing i. Thus, 4 is in set S3, and 9 is in set S1.

Representation of Sets:

Data Representation:



Array-based Representation:

For n = 10, consider the sets:

- S1 = {1, 7, 8, 9}
- S2 = {2, 5, 10}
- S3 = {3, 4, 6}

This can be represented using an array P such that:

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
P[i]	-1	5	-1	3	-1	3	1	1	1	5

Here, a negative value indicates that the element is the representative (root) of the set, and its magnitude shows the size (optional, not visible here). A positive value points to the index of its parent (i.e., the representative of the set it belongs to).

This representation allows efficient implementation of **find** and **union** operations.

Algorithms:

Algorithm: Union(i, j){

```
        p[i] := j
}
```

Algorithm: Find(i){

```
        while (p[i] ≥ 0) do i := p[i];
        return i;
}
```

Optimizations

Path Compression (Optimized Find):

```
function Find(i):
   if p[i] < 0 then
      return i;
   else
      p[i] := Find(p[i]);
      return p[i];
```

This helps flatten the structure for faster future queries.

Union by Size (or Rank):

```
function Union(i, j):
   iRoot := Find(i);
   jRoot := Find(j);
   if iRoot == jRoot then return;

   if size[iRoot] > size[jRoot] then
      p[jRoot] := iRoot;
      size[iRoot] += size[jRoot];
   else
      p[iRoot] := jRoot;
      size[jRoot] += size[iRoot];
```

(*Note: The size can be encoded by negative values in p[]*)

Time Complexity:

With both **Path Compression** and **Union by Rank/Size**:

- Each operation runs in **O(α(n))**, where α is the **inverse Ackermann function**, which grows extremely slowly.

This makes the disjoint set data structure nearly constant time for practical purposes.