UNIT-II

DISJOINT SETS

DISJOINT SETS : Sets are represented by pair wise disjoint sets. If S_i and S_j are two sets and i # j, then there is no common elements for S_i and S_j .

Example : When N = 10 elements. That can be partitioned into three disjoint sets i.e, S1, S2 and S3.

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

Sets can be represented in 3 ways

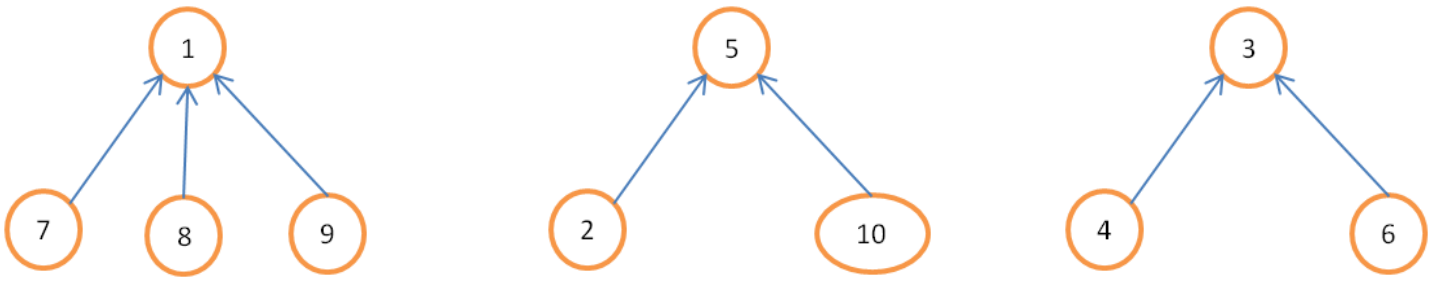
1 : Tree Representation

2 : Data Representation

3 : Array Representation

1 : Tree Representation : The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

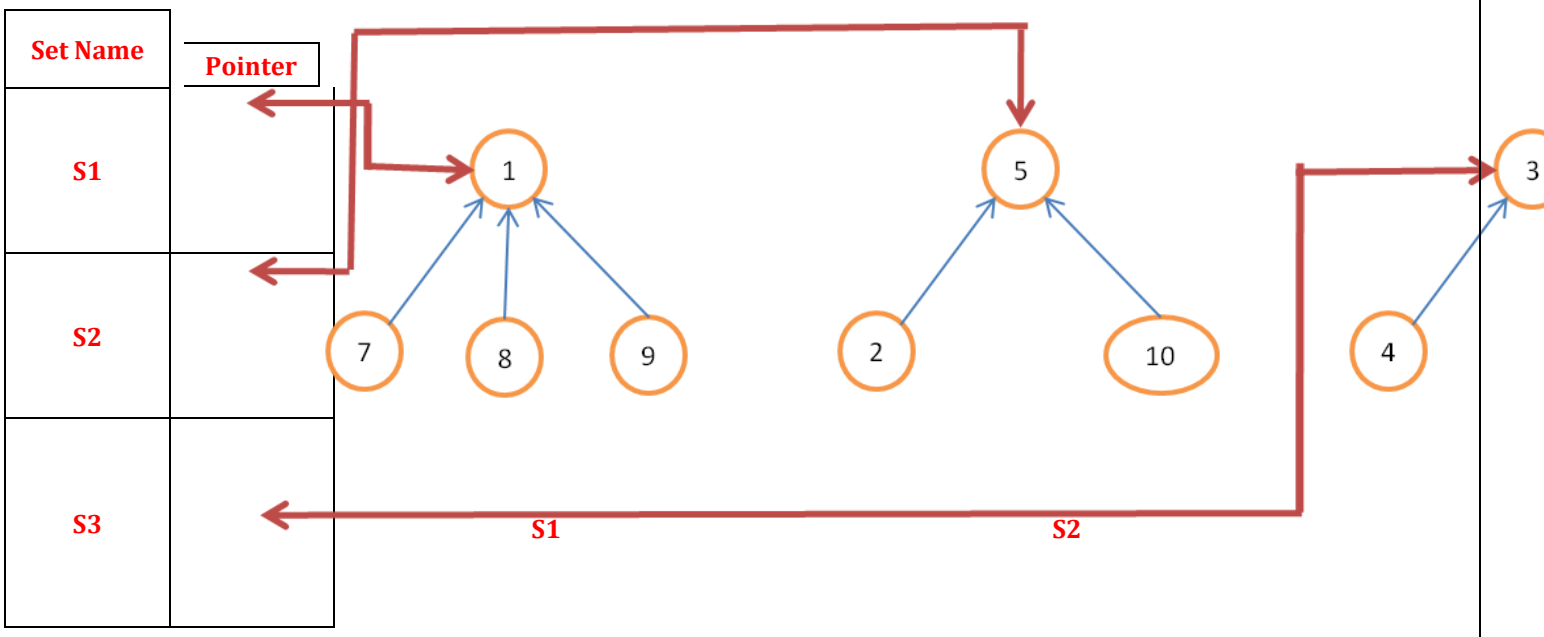
Example: S1 = { 1, 7, 8, 9 } S2 = { 2, 5, 10 } S3 = { 3, 4, 6 } , Then these sets can be represented as



S1 **S2** **S3**

2: Data Representation : We need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root. The data representation for S1, S2 and S3.

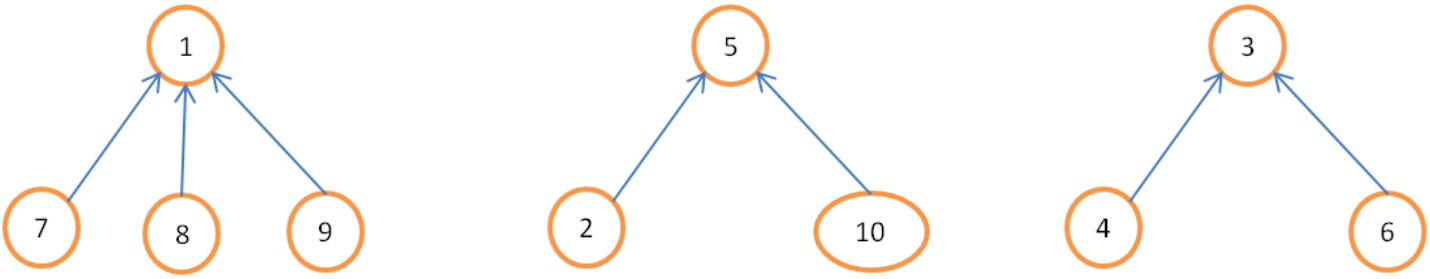
Example: S1 = { 1, 7, 8, 9 } S2 = { 2, 5, 10 } S3 = { 3, 4, 6 }



3 : Array Representation : To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be'-1'.

Array Representation of the sets S1, S2 and S3

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



S1 **S2** **S3**

i	1	2	3	4	5	6	7	8	9	10
p	-1	5	-1	3	-1	3	1	1	1	5

Disjoint Set Operations : Disjoint set supports two operations

1. Union
2. Find

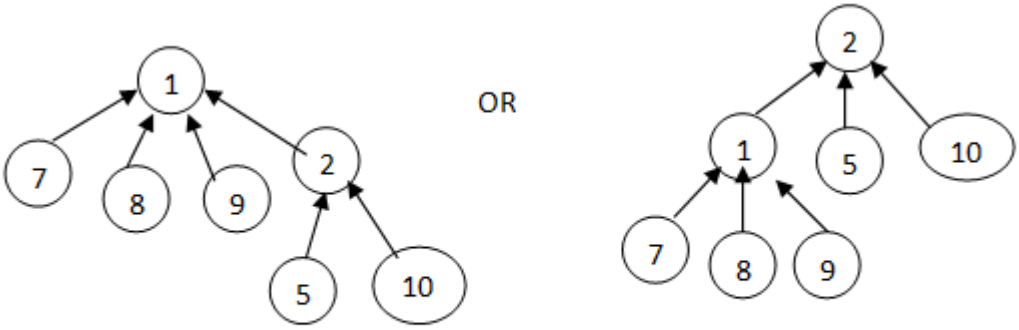
1 : Disjoint set Union : If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = { all elements x such that s is in S_i or S_j }

Example : S1 = { 1, 7, 8, 9 } S2 = { 2, 5, 10 }

S1 U S2 = { 1, 7, 8, 9 , 2, 5, 10 }

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other.

Example : S1 = { 1, 7, 8, 9 } S2 = { 2, 5, 10 } ,Then S1 U S2 can be represented as any one of the following.



S1 U S2

2 : Find (i) : Given the element i, find the set containing i.

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

Find (4) = 4 is in set S3

Find (9) = 9 is in set S1

Find (10) = 10 is in set S2

UNION AND FIND ALGORITHMS :

1 : Simple Union

2 : Weighted Union

3 : Find

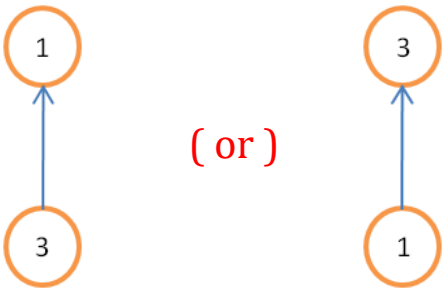
4 : Collapsing Find

1 : Simple Union : To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

Algorithm :

```
Algorithm SimpleUnion (i,j)
{
        P[i]:=j;
}
```

Example : union (1, 3)

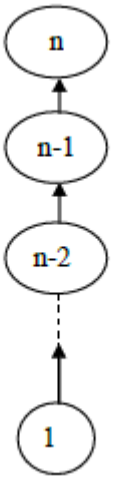


The SimpleUnion(i,j), algorithm is very easy to state, their performance characteristics are not very good.

If we want to perform following sequence of operations

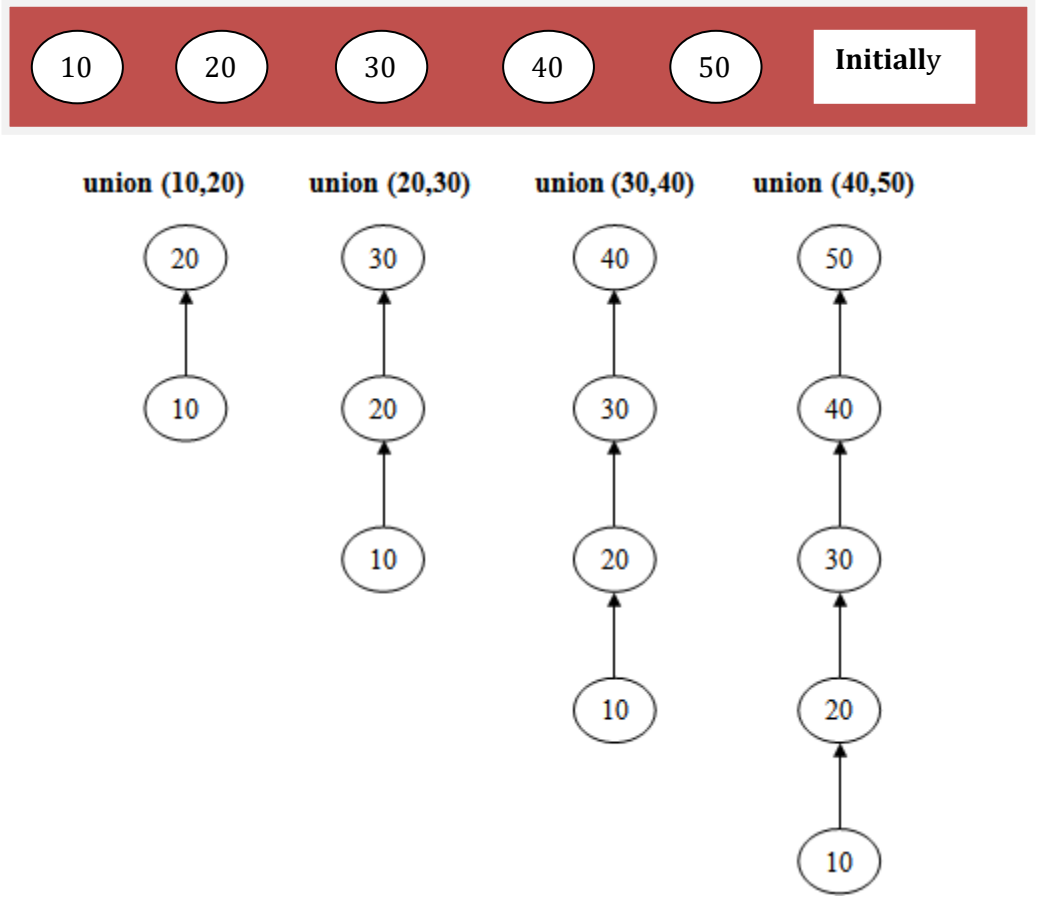
Union(1,2) ,Union(2,3), Union(3,4), Union(4,5)……. Union(n-1,n)

The sequence of Union operations is called **degenerate tree** as below



Example : **The union operation can be performed as follows.**

union(10,20), union(20,30), union(30,40), union(40,50).



Union algorithm Time complexity : O (n)

We can improve the performance of our union algorithm by avoiding the creation of degenerate trees.

We can use Weighted Union rule.

2.Weighted Union Algorithm

Definition : if the number of nodes in the tree with root **i** is less than the number in the tree with root **j**, then make **j** the parent of **i** ; otherwise make **i** the parent of **j**.

Algorithm :

Algorithm : Weighted_Union(i, j)

```
{
      p[i] ← - count[i];   //Initially
       p[j] ← - count[j];
       temp ← p[i] + p[j];
       if(p[i] > p[j]) then    // i has few nodes than j
       {
          p[i] = j;        // j becomes parent of i
          p[j] = temp;
       }
       else
       {
          p[j] = i;        // i becomes parent of j
          p[i] = temp;
       }
}
```

Example : Weighting rule to perform the sequence of set unions given below

Solution : Array Representation

i	1	2	3	4	----------	n
p	-1	-1	-1	-1		-1

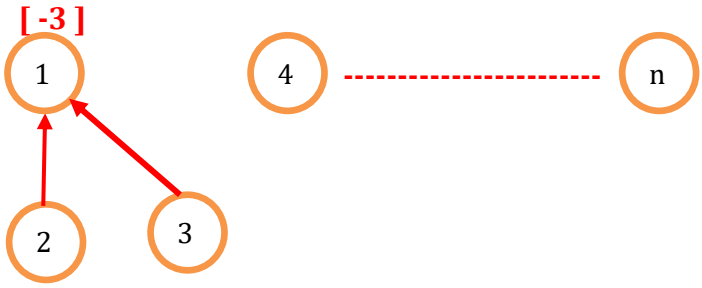
Step1 : **Initially**

Step2 : Union(1,2)

i	1	2	3	4	----------	N
p	-2	1	-1	-1		-1

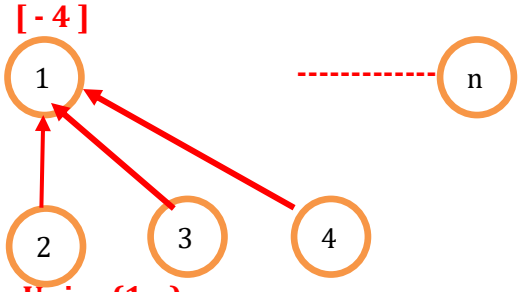
Step3 : Union(1,3)

i	1	2	3	4	----------	n
p	-3	1	1	-1		-1

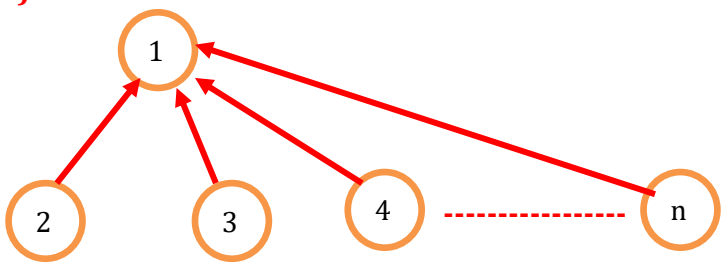


Step4 : Union(1,4)

i	1	2	3	4	----------	n
p	-3	1	1	1		-1



Step4 : Union(1,n)

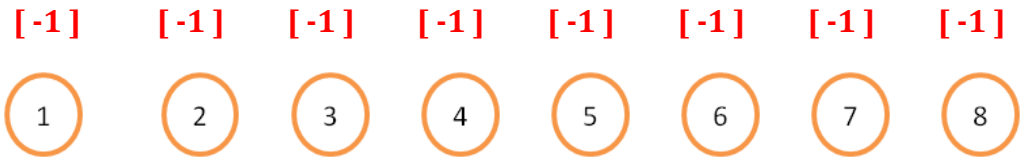


The above trees obtained using the weighting rule

Example : Consider the behavior of Weighted union on the following sequences of unions.

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

Solution: The sequence of unions starting from the initial configuration p[i] = -count[i]=-1



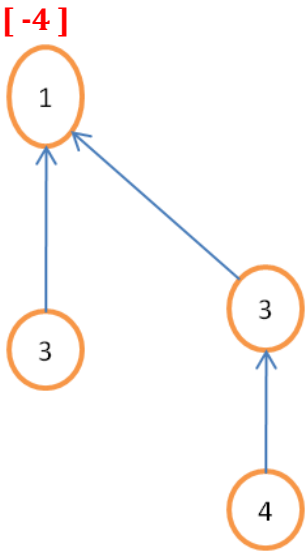
union(1,2)

union(3,4)

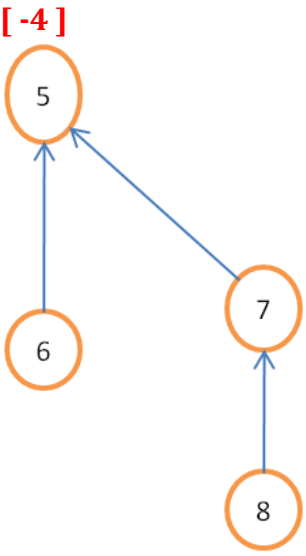
union(5,6)

union(7,8)

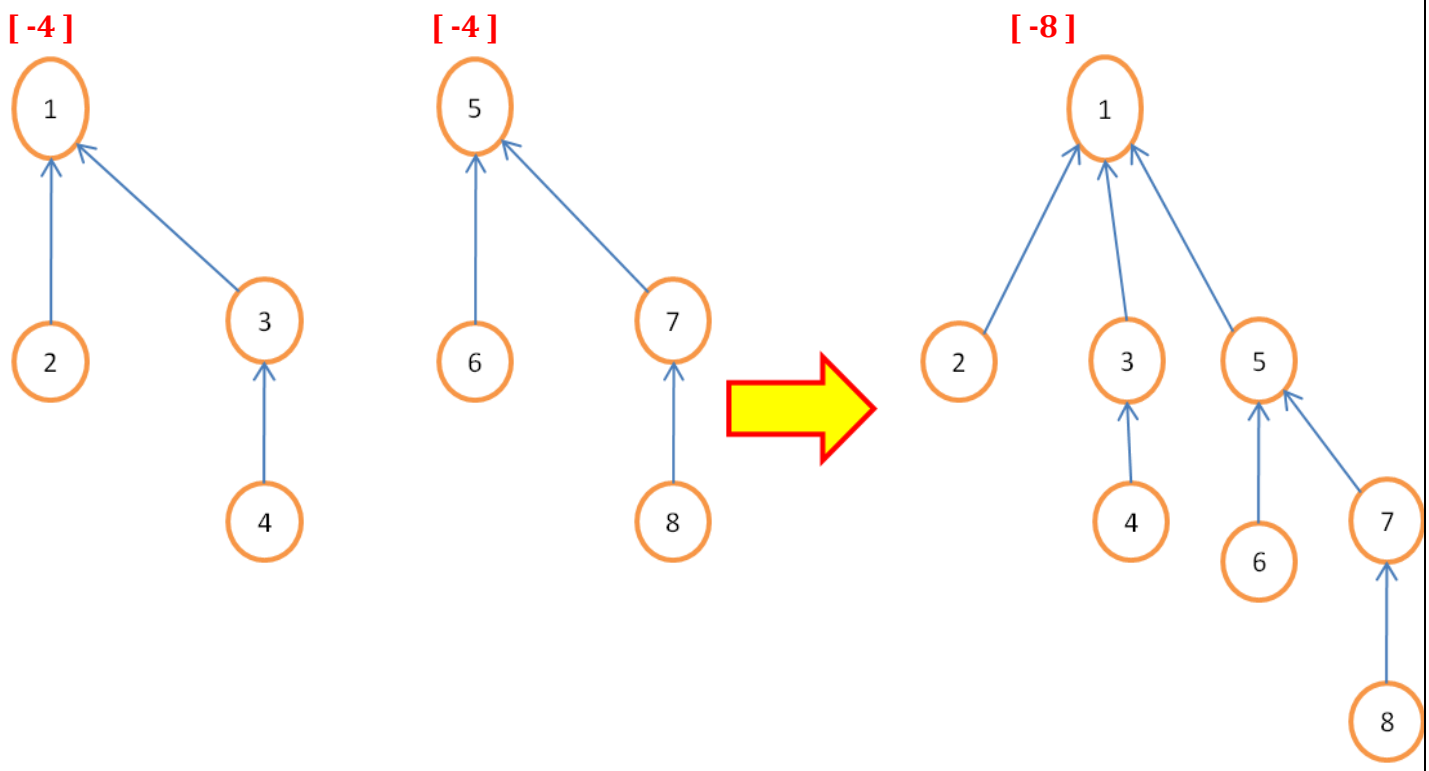
union (1,3)



union (5,7)



union (1,5)



Explanation in Weighted Union Algorithm

Union(1,5)

i	1	2	3	4	5	6	7	8
p	~~-4~~ , -8	1	1	3	~~-4~~, -1	5	5	7

i=1 and J=5

p[i]=-count[i]

p[1]=-count[1]

p[1] = -4

p[j]=-count[j]

p[5]=-count[5]

p[5] = -4

temp = p[i] + p [j]

temp = -4 – 4 = -8

if (p[i] > p[j]) then

-4 > -4 – false

p[j] = i, p[5] = 1 // **for 5th node 1 is the root node**

p[i] = temp

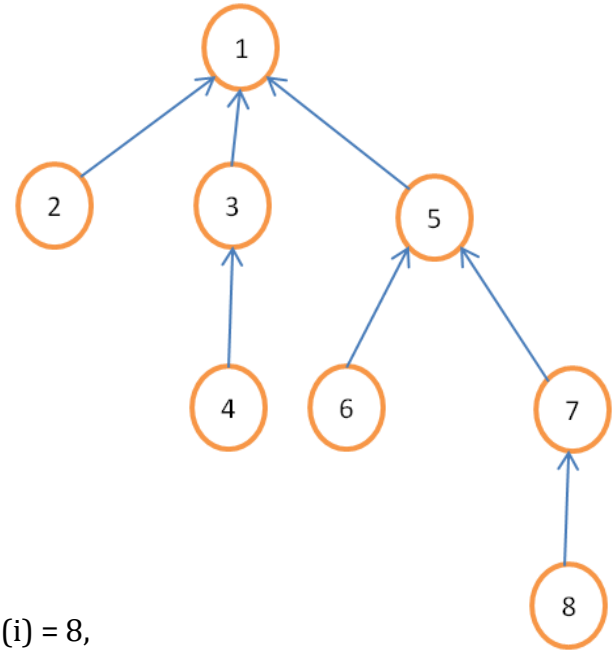
p[1] = -8

Find Algorithm : FIND(i), means it finds the root node of ith node, in other words, it returns the name of set i.

Algorithm SimpleFind(i)

```
{
   while ( p[i] >= 0 ) do
         i = p[i];
  return i;
}
```

Example : Consider a tree



Solution : If we want to Find(i) = 8,

Array Representation of tree

i	1	2	3	4	5	6	7	8
p	-8	1	1	3	1	5	5	7

i = 8

while (p[i]>=0) – true

i=p[8], i.e 7

while (p[i]>=0) – true

i=p[7], i.e 5

while (p[i]>=0) – true

i=p[5], i.e 1

while (p[i]>=0) – false

return i, i.e i = 1

So 1 is the root node of node 8

Find algorithm Time complexity : O (N^2)

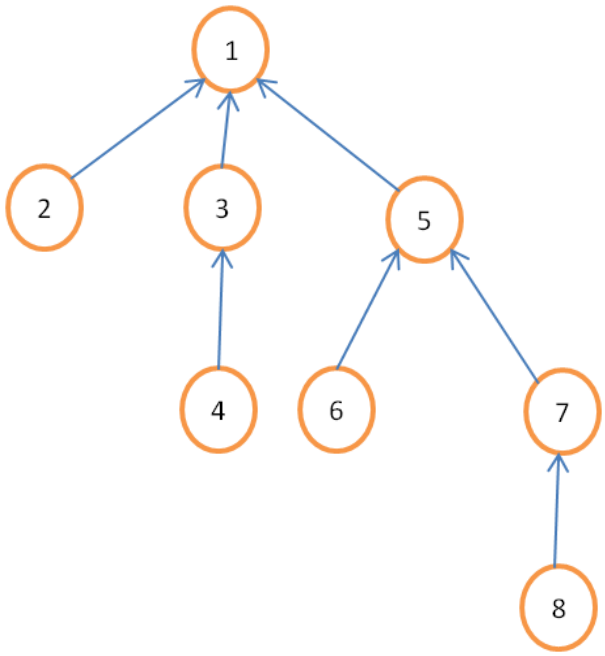
Collapsing Find Algorithm :

Definition : if **j** is a node on the path from **i** to its root and **a[i] # root[i],** then set **a[j]** to **root[i].** using collapsing rule, the find operation can be performed.

Algorithm : Collapse_find(i)

```
{
    // Problem Description : This algorithm Collapse all nodes from i to root node.
     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;
}
```

Example : Consider a tree



Solution : If we want to Find(i) = 8,

Array Representation of tree

i	1	2	3	4	5	6	7	8
p	-8	1	1	3	1	5	5	7

Step1 : i = 8, r = 8

while (p[r]>0) – true

r=p[8], i.e 7

while (p[r]>0) – true

r=p[7], i.e 5

while (p[r]>0) – true

r=p[5], i.e 1

while (p[r]>0) – false

Step2 : Collapse nodes from I to root r.

while (i # r)

8 # 1 – true

s=p[i],
s=p[8]
s=7
p[i]=r
p[8]=1 // means for 8th node 1 is the root node
i=7

again while (i # r)

7 # 1 – true

s=p[i],
s=p[7]
s=5
p[i]=r
p[7]=1 // means for 7th node 1 is the root node
i=5

again while (i # r)

5 # 1 – true

s=p[i],
s=p[5]
s=1
p[i]=r
p[5]=1 // means for 5th node 1 is the root node
i=1

again while (i # r)
1 # 1 – false
return r, i.e r=1

Array Representation of collapsing nodes from i to root j

i	1	2	3	4	5	6	7	8
p	-8	1	1	3	1	5	1	1

After collapsing nodes from i to root j , the given tree is