Sets, and their representation

Consider the universe of symbols, U = {1, 2, …, N} or U = {red, green, blue, … }

And now consider one or more sets, S1, S2, etc. the Union of which is the universe U
For example S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Equivalently, the universe U is portioned into multiple sets, S1, S2, etc.

Question how do we represent them, and carry out operations efficiently

- Make-Set(v)
- Find-Set(u) ≠ Find-set(v)
- Union(u, v)

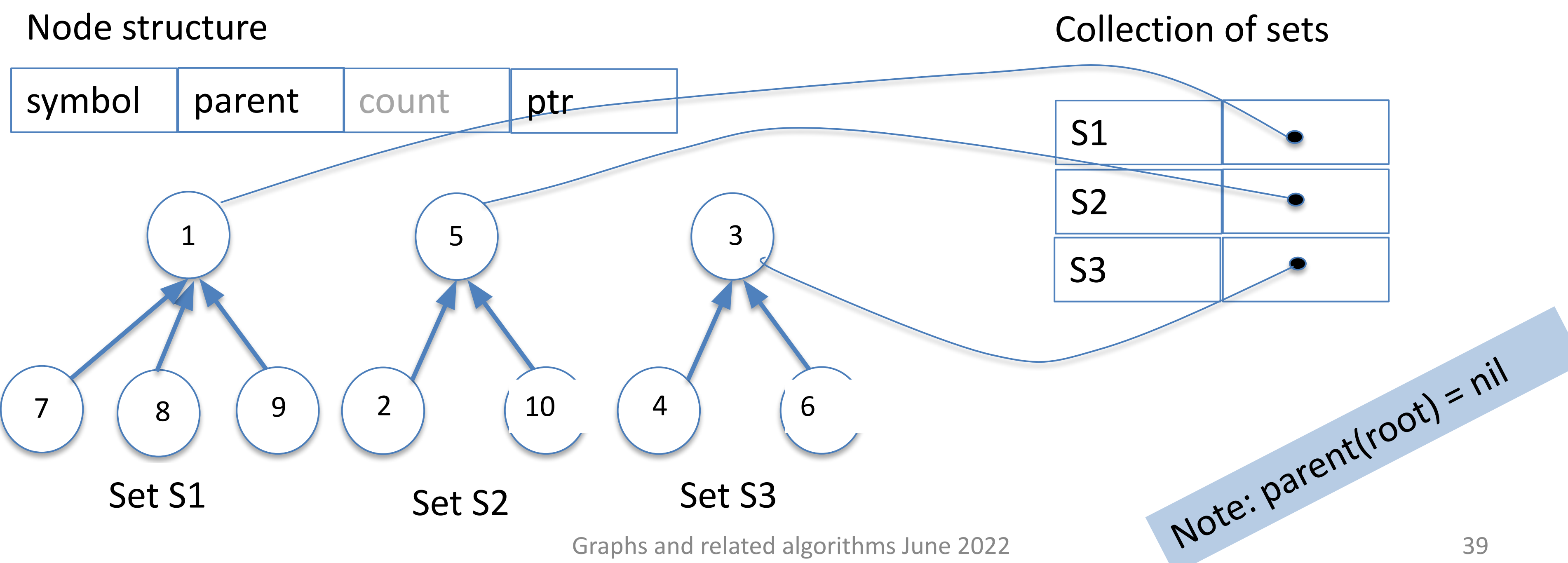
Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Here is one way to represent the disjoint sets that makes it efficient to carry out operations:

- Make-Set(v)
- Find-Set(u) ≠ Find-set(v)
- Union(u, v)

That nodes point to their parents will have significance to "Union" and "Find

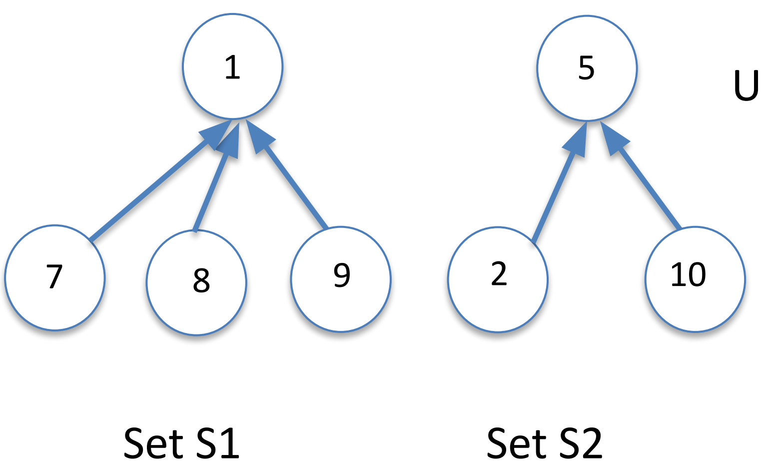


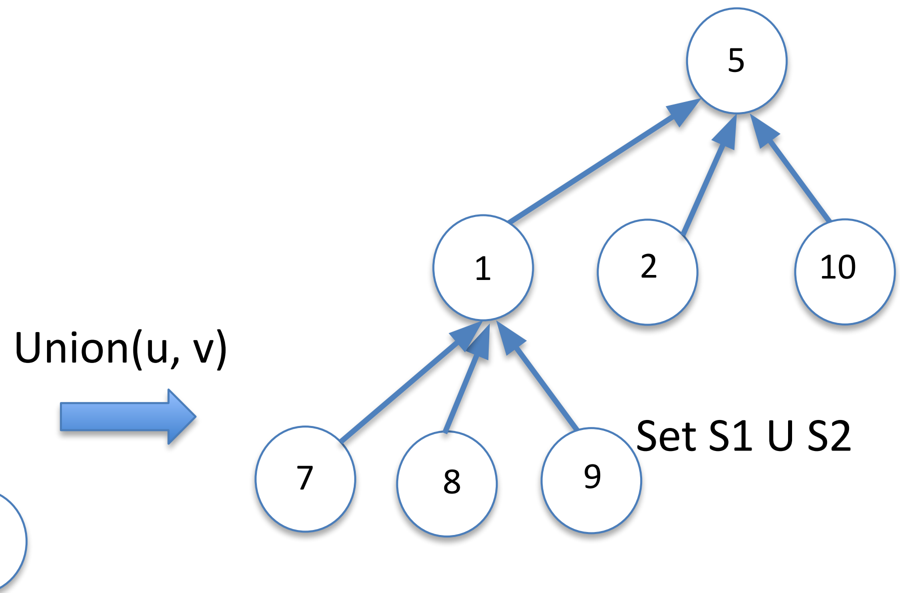
Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Consider operation Union(u, v)

- To be sure consider Union(1, 5)





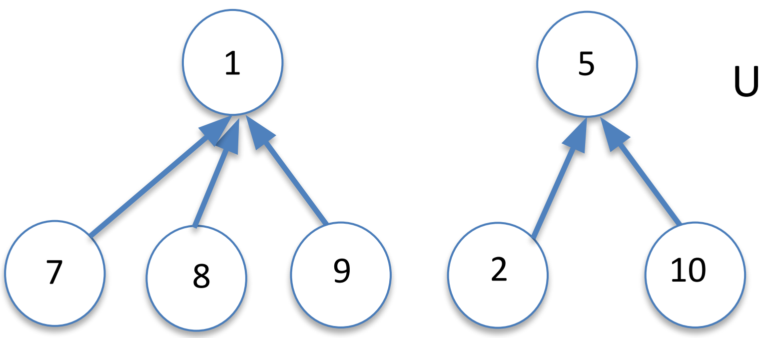

```
function Union (u, v)
    parent(u) = v
return(v)
```

Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

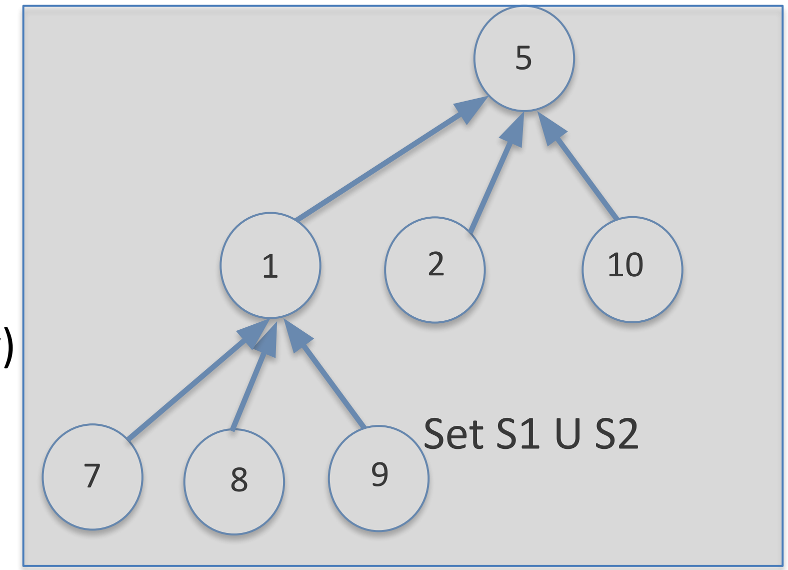
Consider operation Union(u, v)

- To be sure consider Union(1, 5)

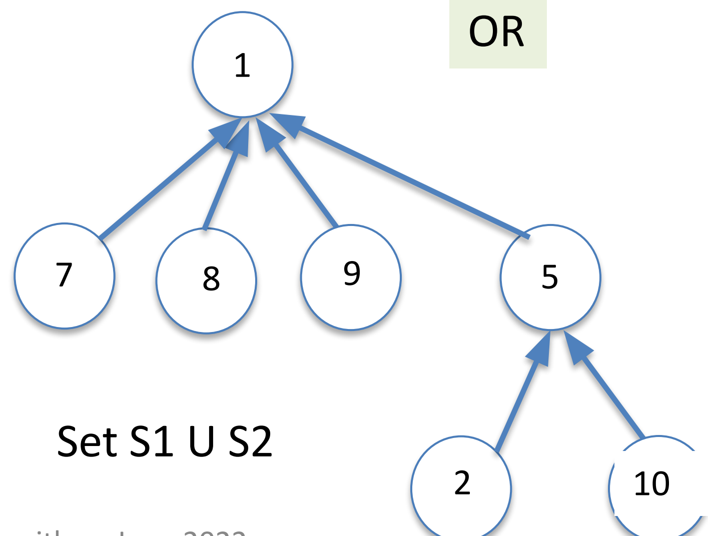


Set S1 Set S2

Union(u, v)



OR



Set S1 U S2

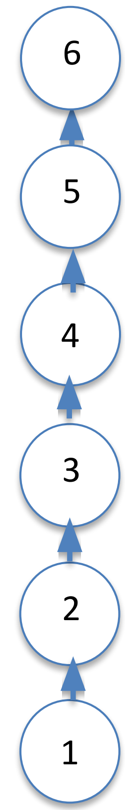
function Union (u, v)
~~parent(u) = v~~ parent(v) = u
return(v)

Sets, and their representation

In the worst case the height of tree will be O(n), where n is the number of symbols

For example, U = {1, 2, …, 6}, S1={1}, S2={2}, S3={3}, S4={4}, S5={5}, S6={6}, and consider

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



While Union(u,v) is efficient, or O(1), a Find(u) operation will be complex, or O(n) in the worst case, where n = |U|

□ Form the union so as to minimize the height of resulting tree

```
function Union (u, v)
        parent(u) = v
return(v)
```

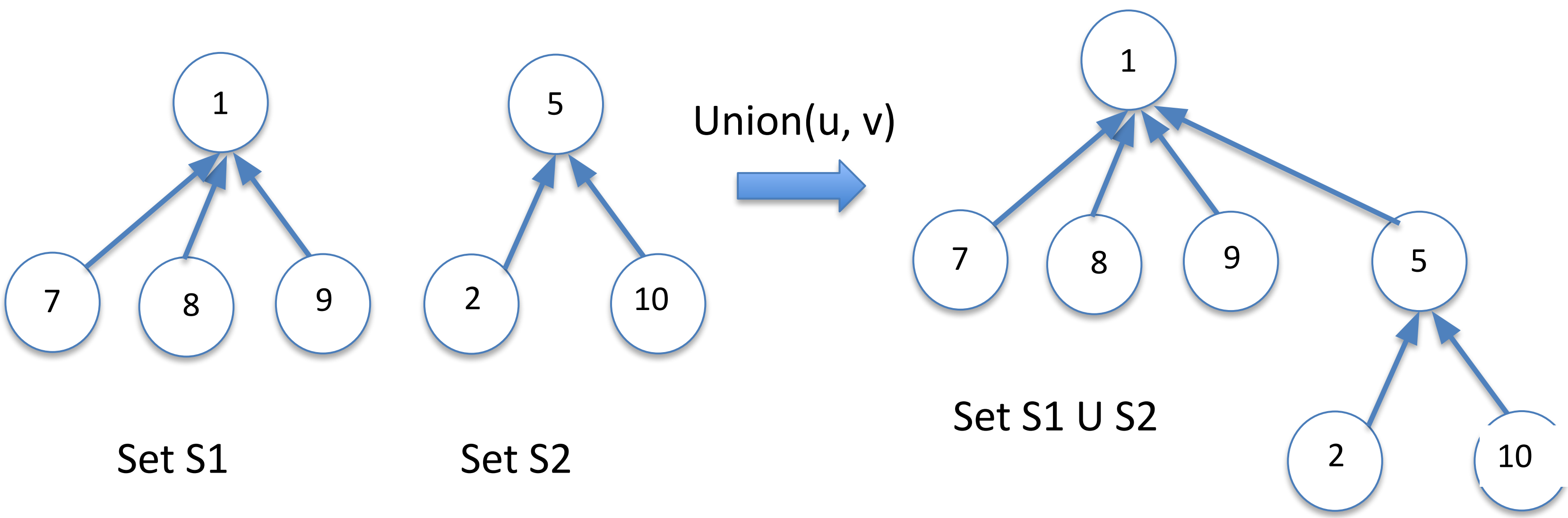
Sets, and their representation

Another approach where we maintain count of symbols in set (or sub-tree):

Node structure

symbol	parent	count

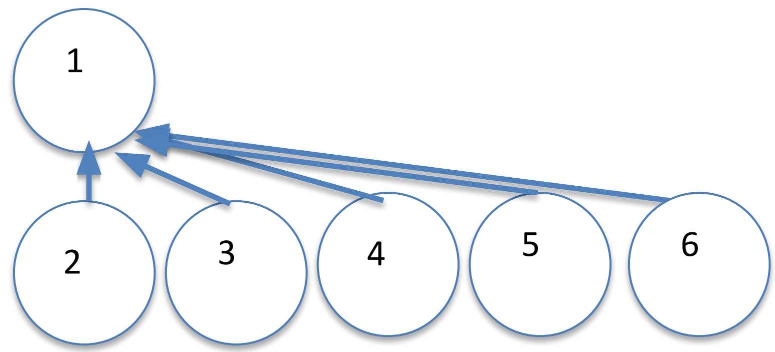
```
function Union (u, v)
    if count(u) < count(v)
    then parent(u) = v
        count(v) = count(v) + count(u)
    else parent(v) = u
        count(u) = count(u) + count(v)
return(v)
```



Sets, and their representation

For example, U = {1, 2, …, 6}, S1={1}, S2={2}, S3={3}, S4={4}, S5={5}, S6={6}, and consider

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




```
function Union (u, v)
        if count(u) < count(v)
        then parent(u) = v
                count(v) = count(v) + count(u)
        else parent(v) = u
                count(u) = count(u) + count(v)
return(v)
```

Sets, and their representation

Another approach where we maintain count of symbols in set (or sub-tree):

Node structure

symbol	parent	count

```
function Union (u, v)
    if count(u) < count(v)
    then parent(u) = v
        count(v) = count(v) + count(u)
    else parent(v) = u
        count(u) = count(u) + count(v)
return(v)
```

- Every node in resulting tree has level $\leq$ floor($\log_2 n$) + 1
- Find(u) runs in time $O(\log_2 n)$

```
function find(u)
    temp = u
    while parent(temp) ≠ nil do
        temp = parent(temp)
return(temp)
```

Sets, and their representation

Time complexity

Union operation: O(1)

```
function Union (u, v)
        if count(u) < count(v)
        then parent(u) = v
                count(v) = count(v) + count(u)
        else parent(v) = u
                count(u) = count(u) + count(v)
return(v)
```

Find operation: $O(\log_2 N)$

```
function find(u)
        temp = u
        while parent(temp) ≠ nil do
                temp = parent(temp)
return(temp)
```

Kruskal's minimum spanning tree

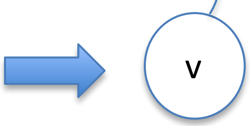
Kruskal's algorithm on G = (V, E), with weights of edges in array W = [w(e)]

```
function MST-Kruskal(G, W)
    T = Φ
    for each vertex v ε V
     Make-Set(v)                   //created |V| sets each with one vertex
                                   //each set is identified by a specific member of the set
    sort edges in E into non-decreasing order by weight w(e)
                                   //instead, partially sort the edges using a (min) binary heap
    for each edge (u, v) in E          //in non-decreasing order of weight w(e)
                                   //Or stop after one has added |V|-1 edges
     if Find-Set(u) ≠ Find-set(v)
         T = T U {(u, v)}          //add edge (u, v) to T
         Union(u, v)               //merge two sets that contain vertices u and v
     delete edge e                 //delete edge e from sorted list or from min heap
return T
```

Kruskal’s minimum spanning tree

Make-Set(v) □

```
function Make-Set(v)
    X = getnode()
    X.symbol = v
    X.parent = nil
    X.count = 1
    X.ptr = k
return X
```



X

symbol	parent	count	ptr
v	nil	1	•

Collection of sets

S1	X1
S2	X2
S_k	…

…	…
S9	X9
S10	X10

Kruskal's minimum spanning tree

Time complexity of Kruskal's algorithm on G = (V, E), with weights of edges in array W = [w(e)]
Let n = |V|, m = |E|

function MST-Kruskal(G, W)
    T = Φ   ← O(1)
    for each vertex v ε V   ← O(n)
     Make-Set(v)   //cremated |V| sets each with one vertex
        //each set is identified by a specific member of the set
    sort edges in E into non-decreasing order by weight w(e)   ← O(m log m) or O(log m)
        //instead, partially sort the edges using a (min) binary heap
    for each edge (u, v) in E   //in non-decreasing order of weight w(e)   ← O(m log m) = O(m log n)
        //Or stop after one has added |V|-1 edges
     if Find-Set(u) ≠ Find-set(v)
        T = T U {(u, v)}   //add edge (u, v) to T
        Union(u, v)   //merge two sets that contain vertices u and v
     delete edge e   //delete edge e from sorted list or from min heap
    return T

☐ Time complexity of Kruskal's algorithm: O(|E| log |E|) = O(|E| log |V|)