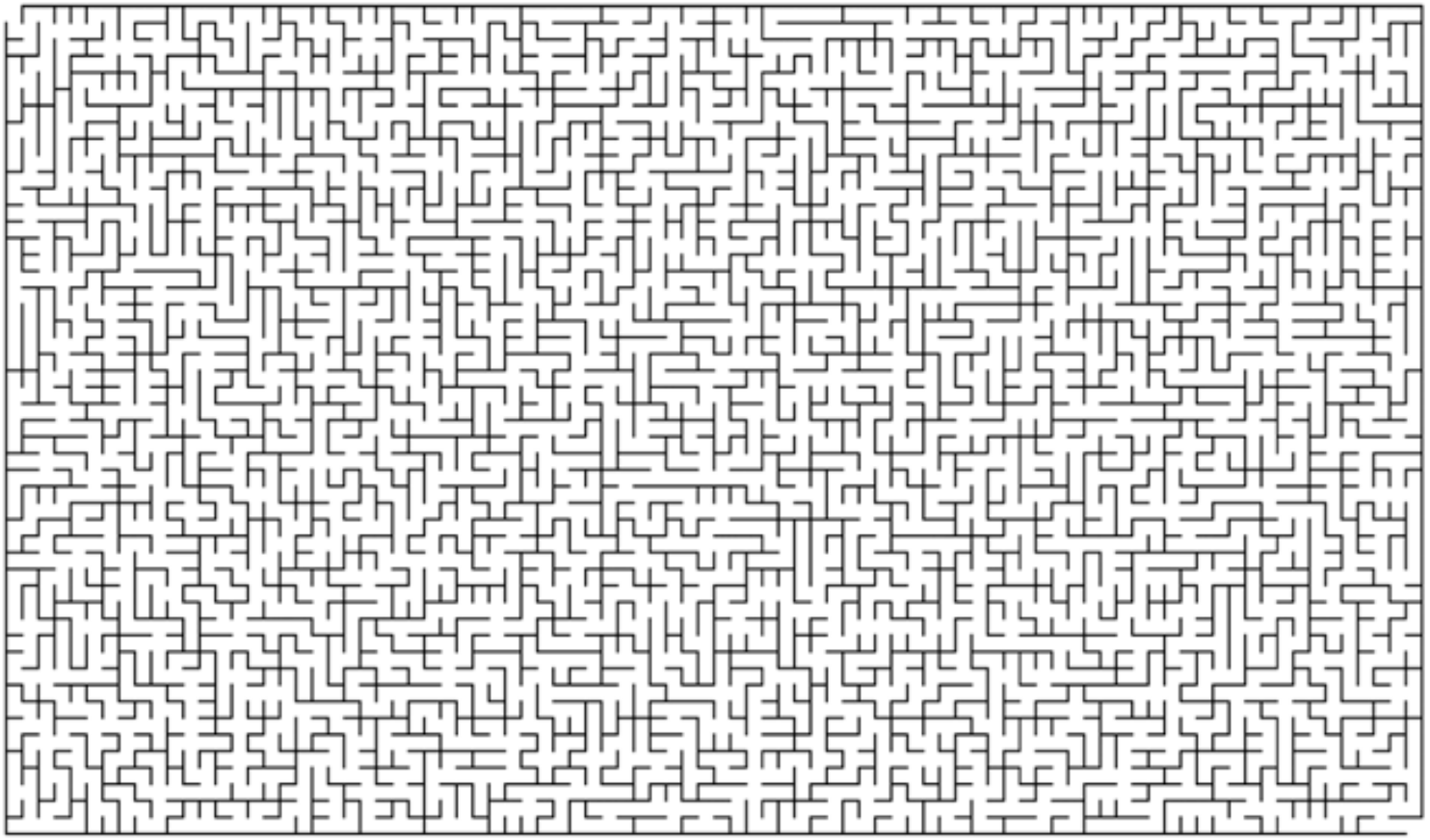
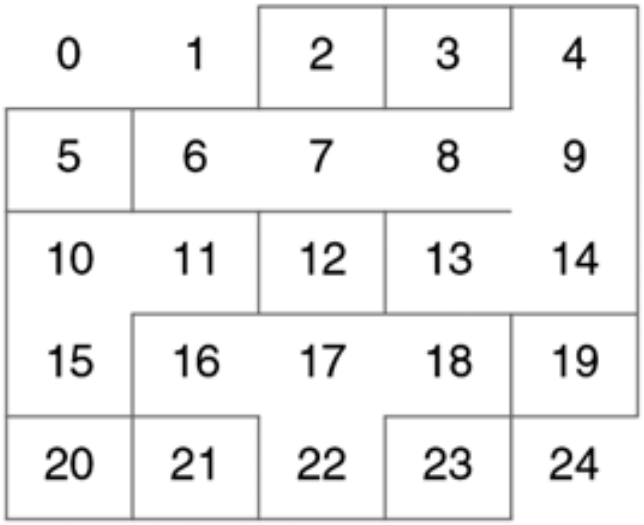
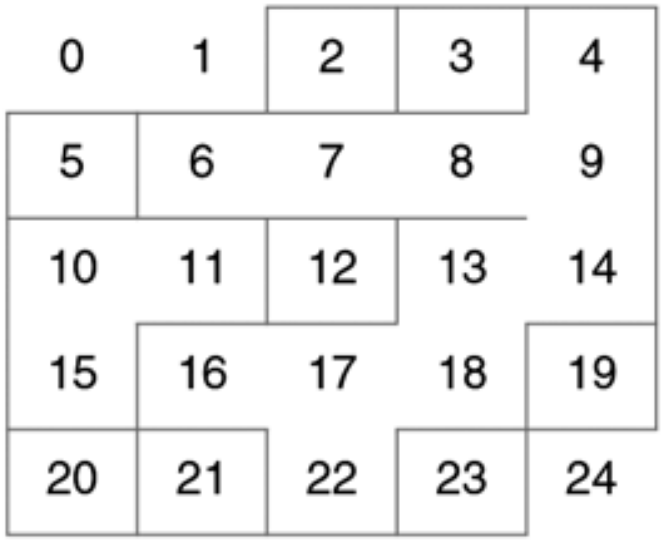


0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

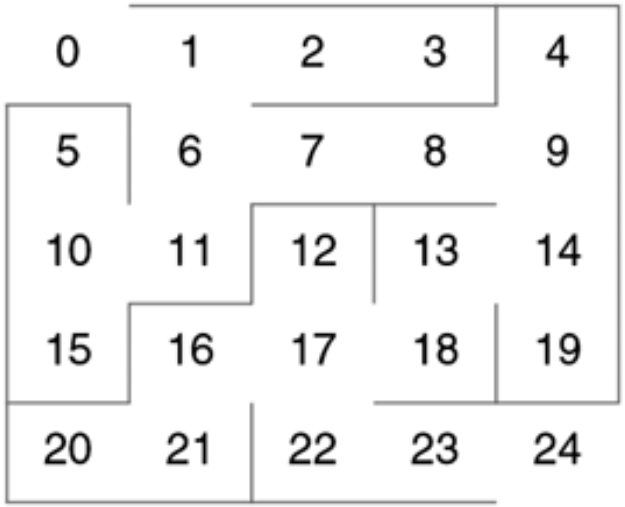
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14}
{15} {16} {17} {18} {19} {20} {21} {22} {23} {24}



{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12}
{16, 17, 18, 22} {19} {20} {21} {23} {24}

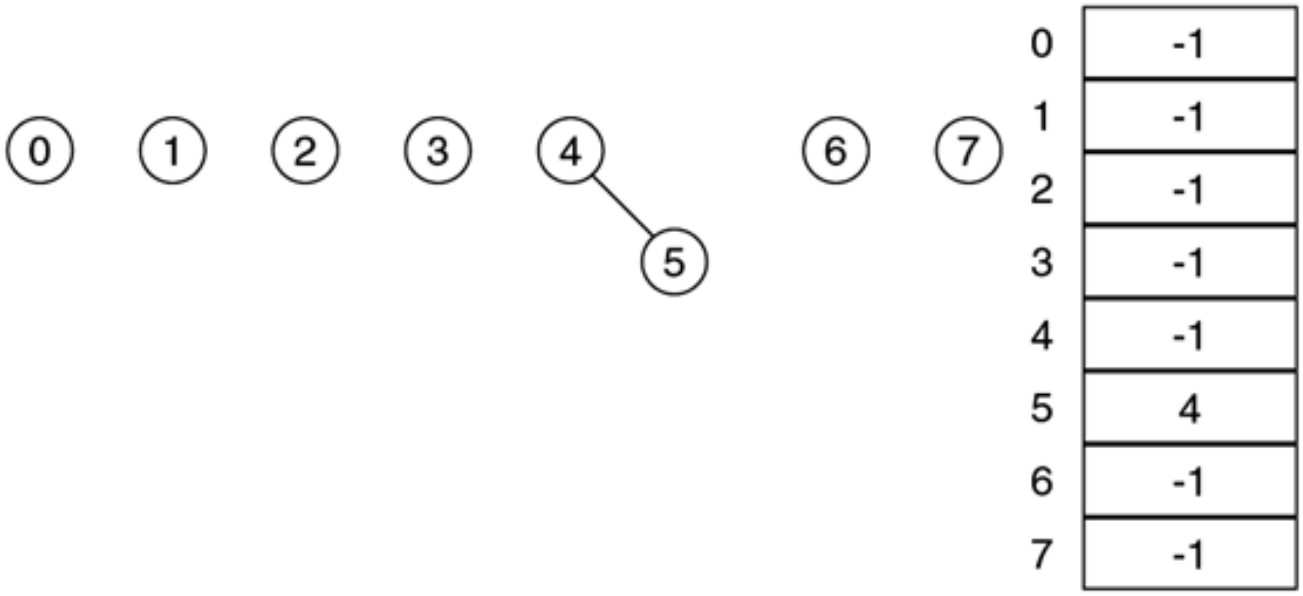


{0, 1} {2} {3} {5} {10, 11, 15} {12}
{4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {19} {20} {21} {23} {24}

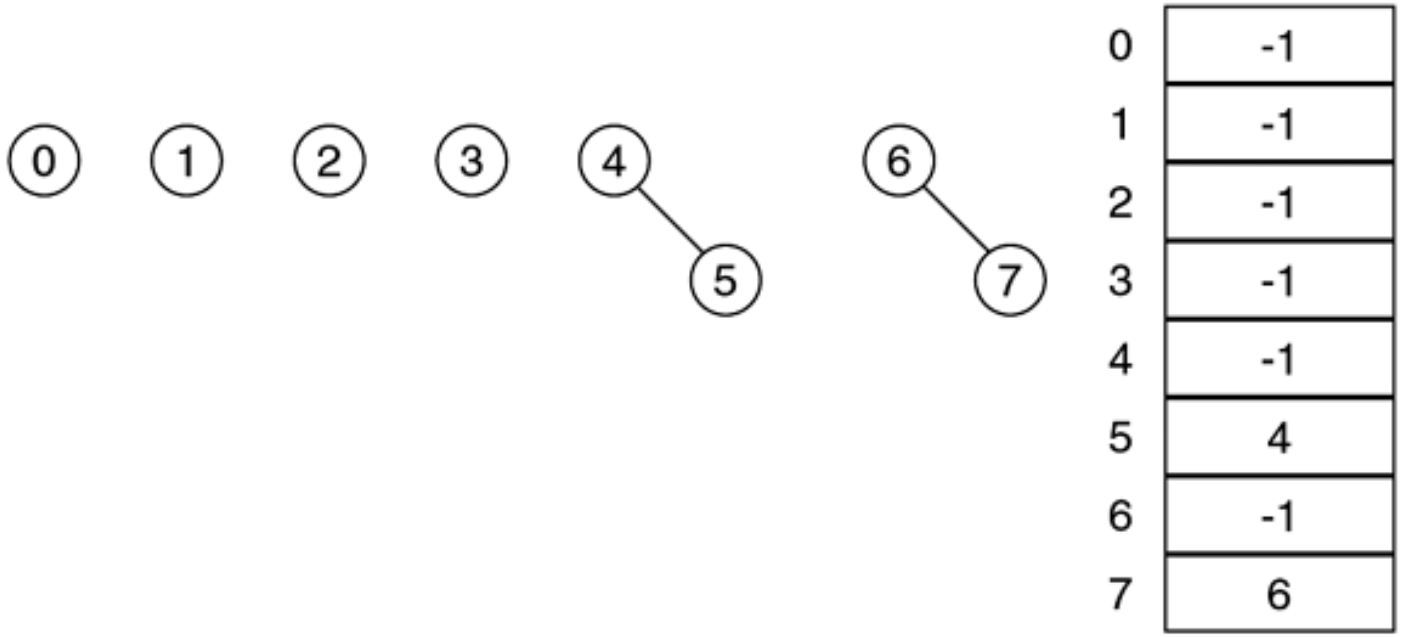


{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

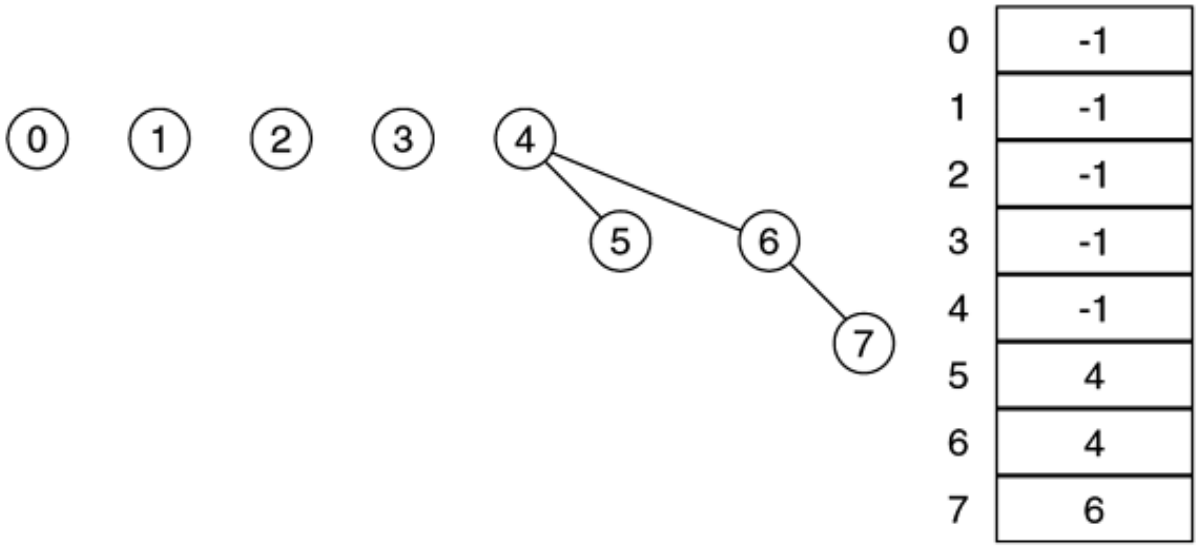
A forest of 8 trees



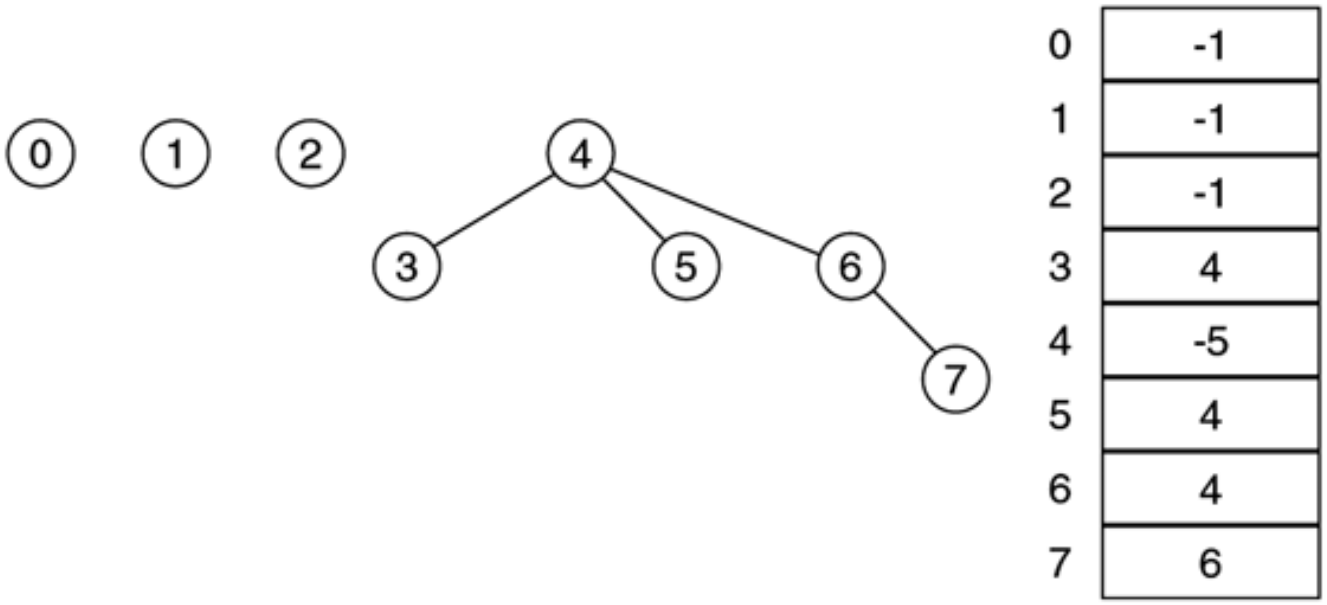
The forest after the union of trees with 4 and 5



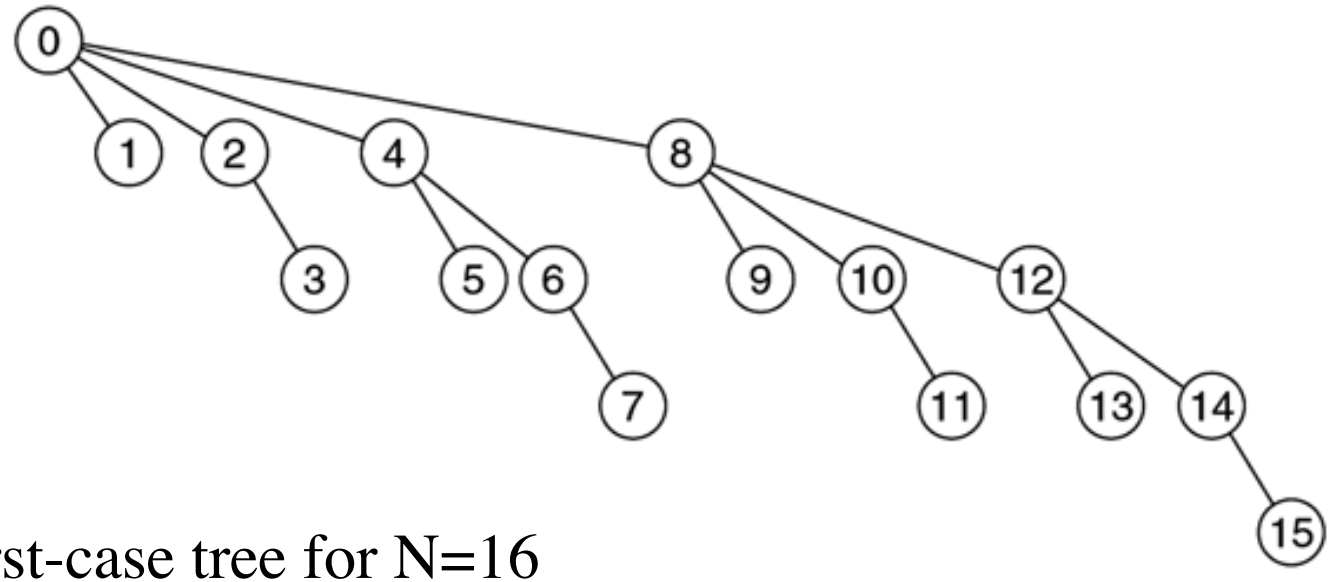
The forest after the union of trees with roots 6 and 7



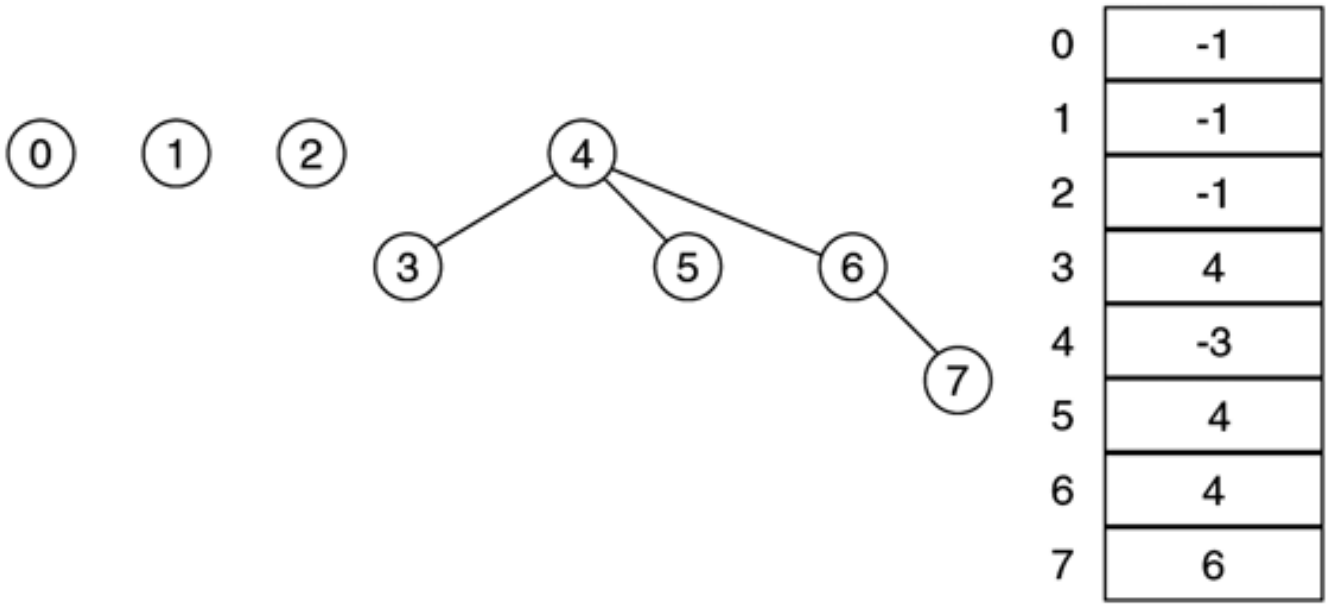
The forest after the union of trees with roots 4 and 6



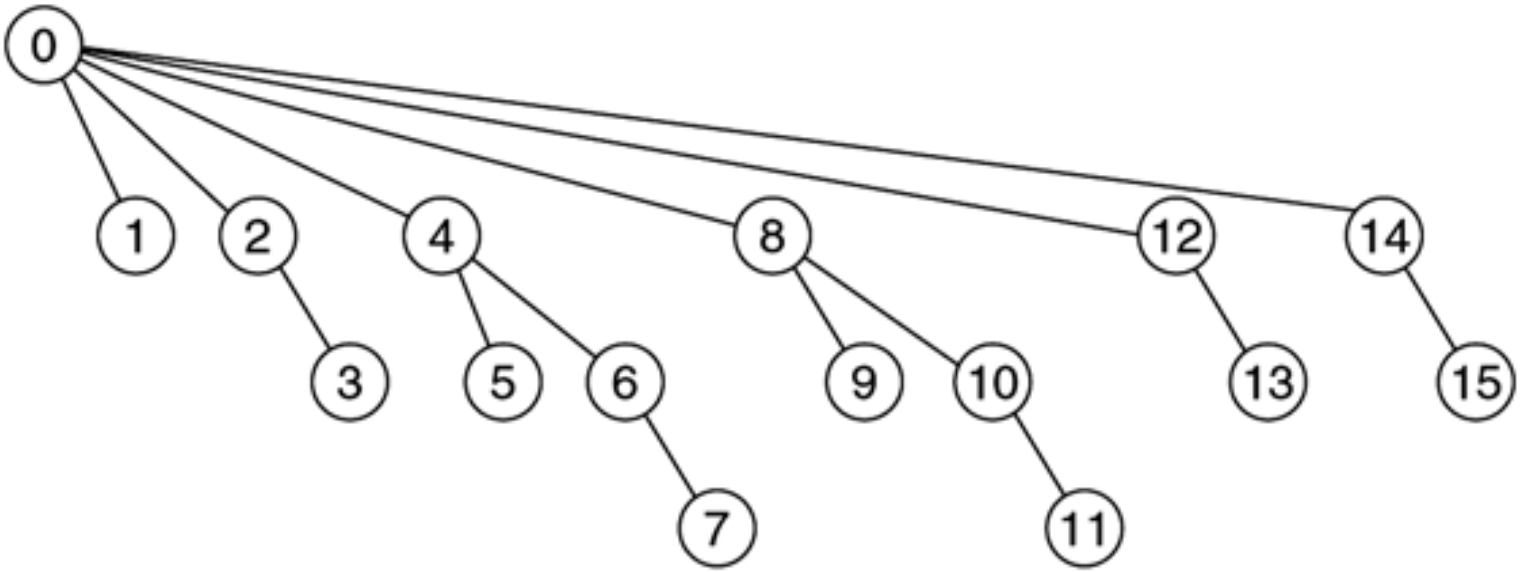
The forest formed by union-by-size, with the size encoded as negative numbers



Worst-case tree for N=16



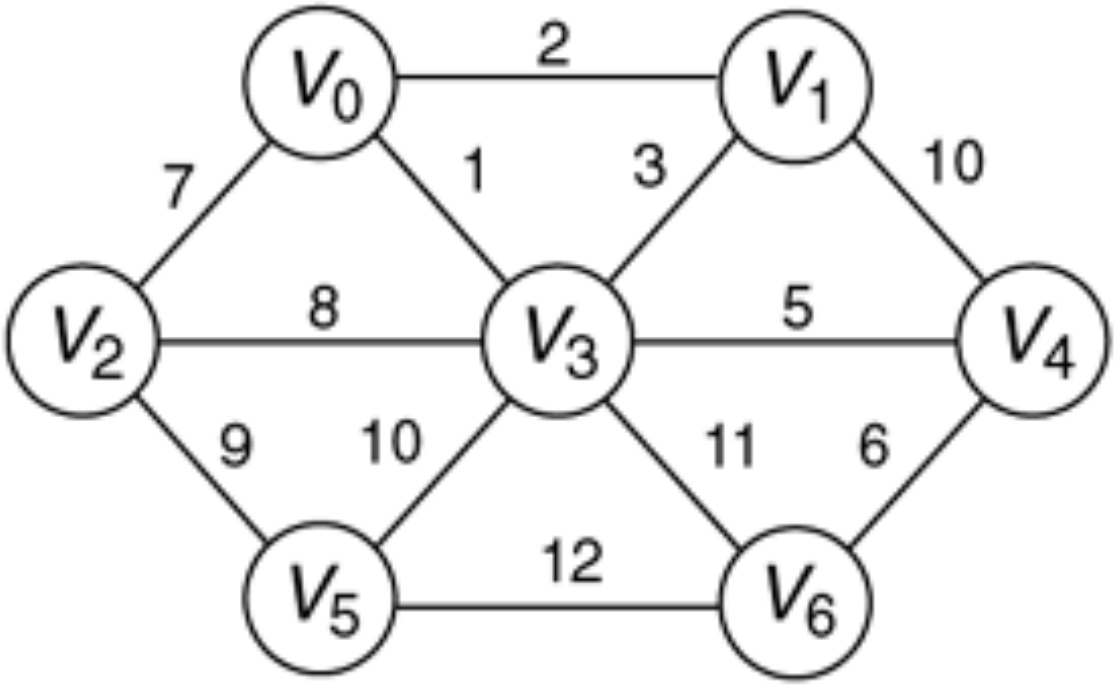
The forest formed by union-by-height, with the height encoded as negative numbers

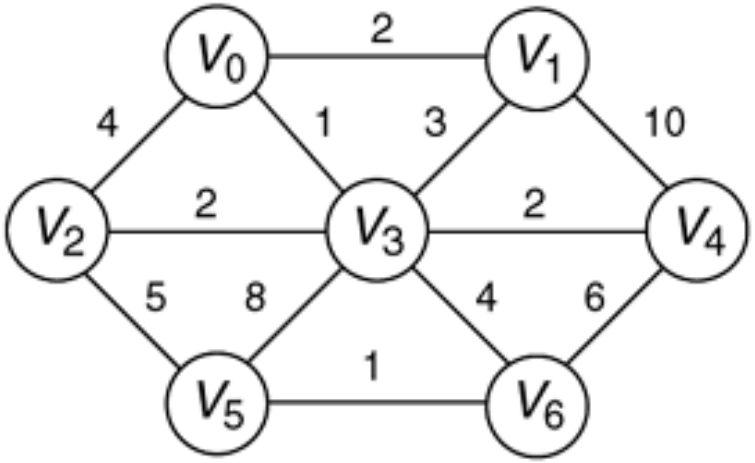


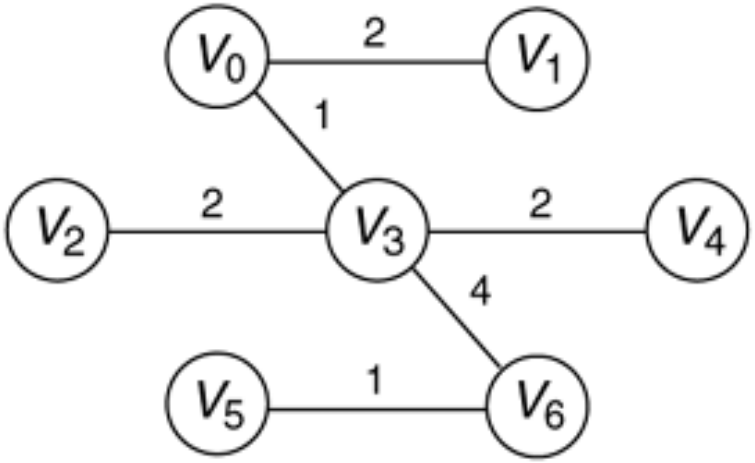
Path compression resulting from a find(14) on the tree

```
1  package weiss.nonstandard;
2
3  // DisjointSets class
4  //
5  // CONSTRUCTION: with int representing initial number of sets
6  //
7  // ******************PUBLIC OPERATIONS*********************
8  // void union( root1, root2 ) --> Merge two sets
9  // int find( x )              --> Return set containing x
10 // ******************ERRORS********************************
11 // Error checking or parameters is performed
12
13 public class DisjointSets
14 {
15     public DisjointSets( int numElements )
16       { /* Figure 24.21 */ }
17
18     public void union( int root1, int root2 )
19       { /* Figure 24.21 */ }
20
21     public int find( int x )
22       { /* Figure 24.21 */ }
23
24     private int [ ] s;
25
26
27     private void assertIsRoot( int root )
28     {
29         assertIsItem( root );
30         if( s[ root ] >= 0 )
31             throw new IllegalArgumentException( );
32     }
33
34     private void assertIsItem( int x )
35     {
36         if( x < 0 || x >= s.length )
37             throw new IllegalArgumentException( );
38     }
39 }
```

```
1       /**
2        * Construct the disjoint sets object.
3        * @param numElements the initial number of disjoint sets.
4        */
5       public DisjointSets( int numElements )
6       {
7           s = new int[ numElements ];
8           for( int i = 0; i < s.length; i++ )
9               s[ i ] = -1;
10      }
11
12      /**
13       * Union two disjoint sets using the height heuristic.
14       * root1 and root2 are distinct and represent set names.
15       * @param root1 the root of set 1.
16       * @param root2 the root of set 2.
17       * @throws IllegalArgumentException if root1 or root2
18       * are not distinct roots.
19       */
20      public void union( int root1, int root2 )
21      {
22          assertIsRoot( root1 );
23          assertIsRoot( root2 );
24          if( root1 == root2 )
25              throw new IllegalArgumentException( );
26
27          if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
28              s[ root1 ] = root2;        // Make root2 new root
29          else
30          {
31              if( s[ root1 ] == s[ root2 ] )
32                  s[ root1 ]--;          // Update height if same
33              s[ root2 ] = root1;        // Make root1 new root
34          }
35      }
36
37      /**
38       * Perform a find with path compression.
39       * @param x the element being searched for.
40       * @return the set containing x.
41       * @throws IllegalArgumentException if x is not valid.
42       */
43      public int find( int x )
44      {
45          assertIsItem( x );
46          if( s[ x ] < 0 )
47              return x;
48          else
49              return s[ x ] = find( s[ x ] );
50      }
```

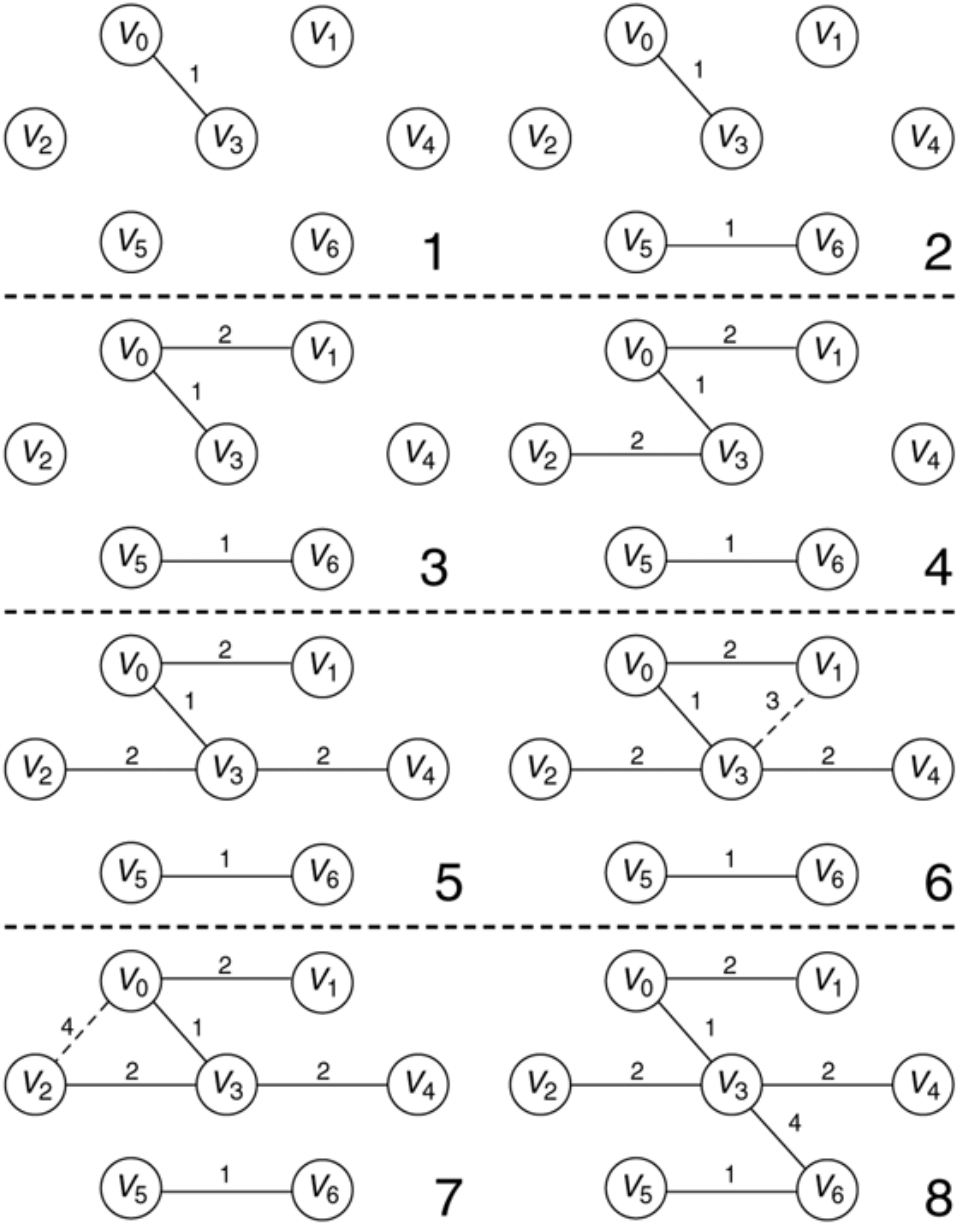

V_0
V_1
V_2
V_3
V_4
V_5
V_6
2
7
1
3
10
8
5
9
10
11
6
12

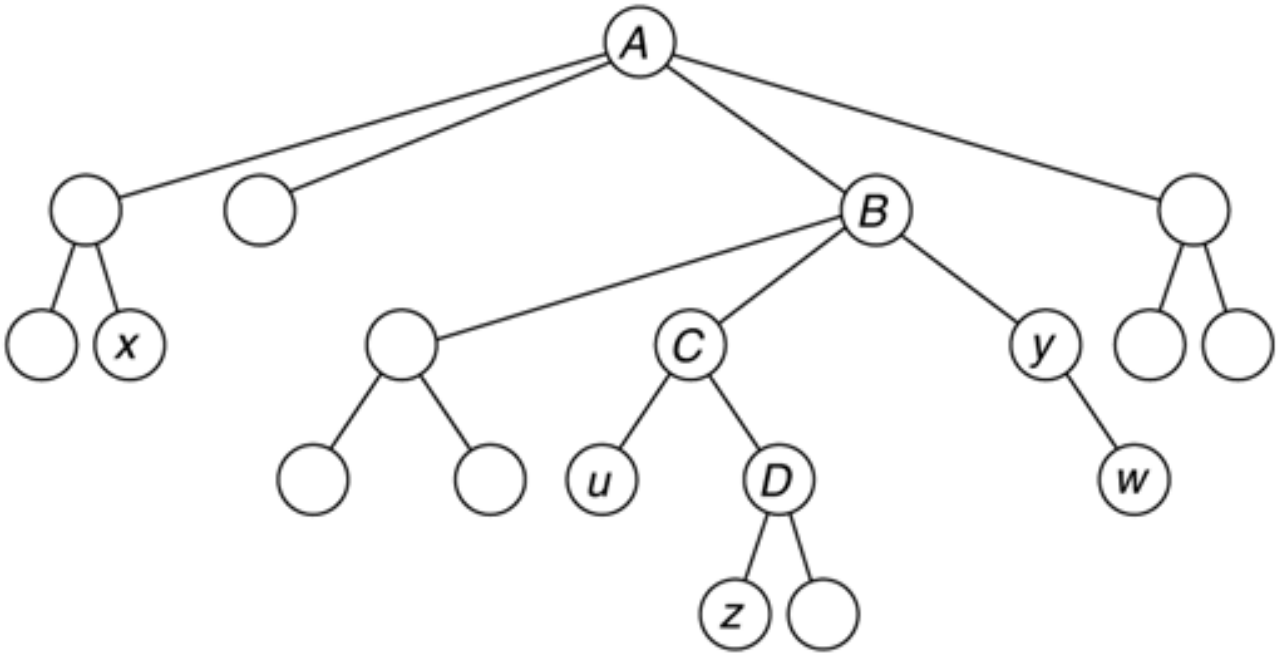




Minimum spanning tree

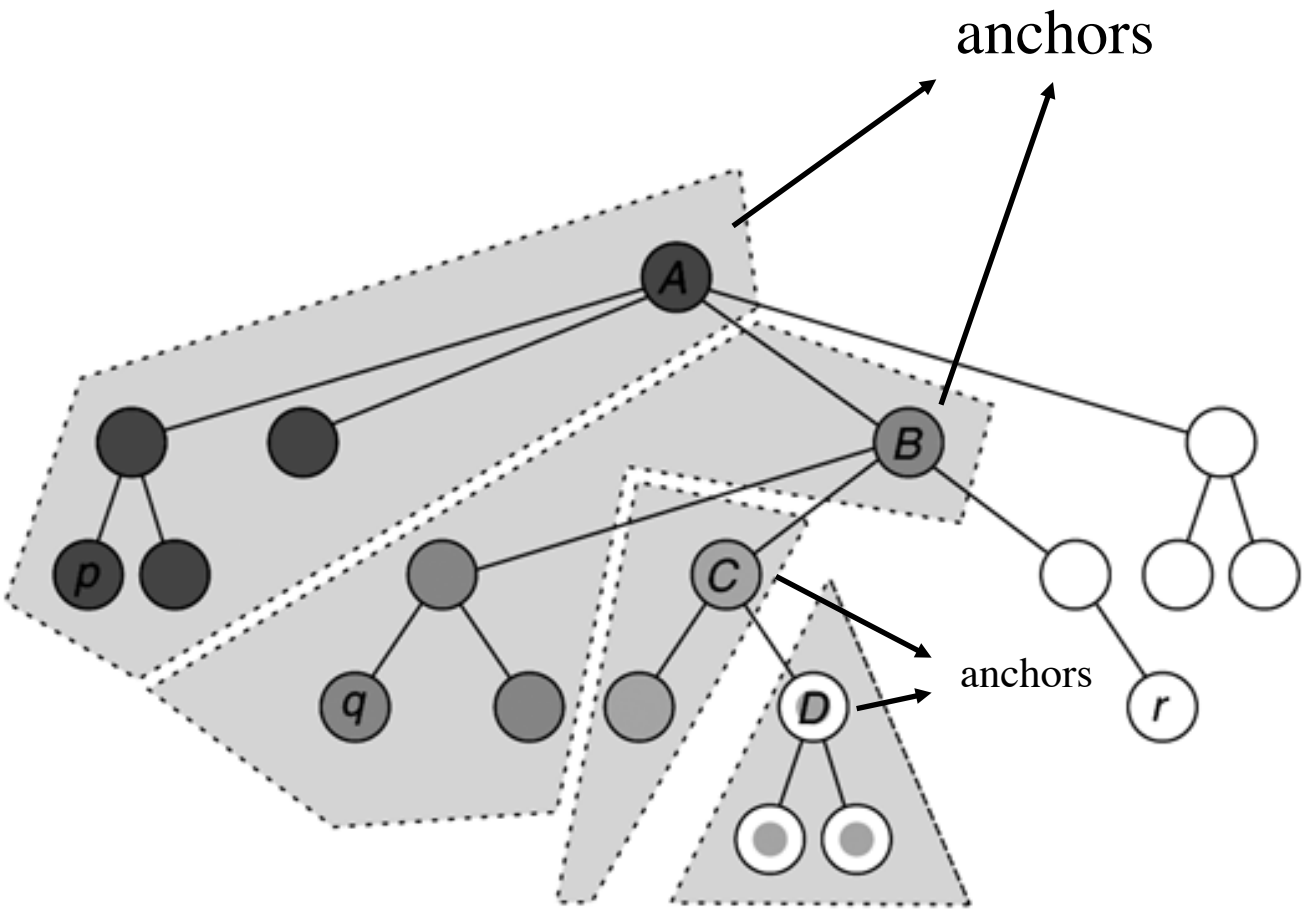
Kruskal's algorithm



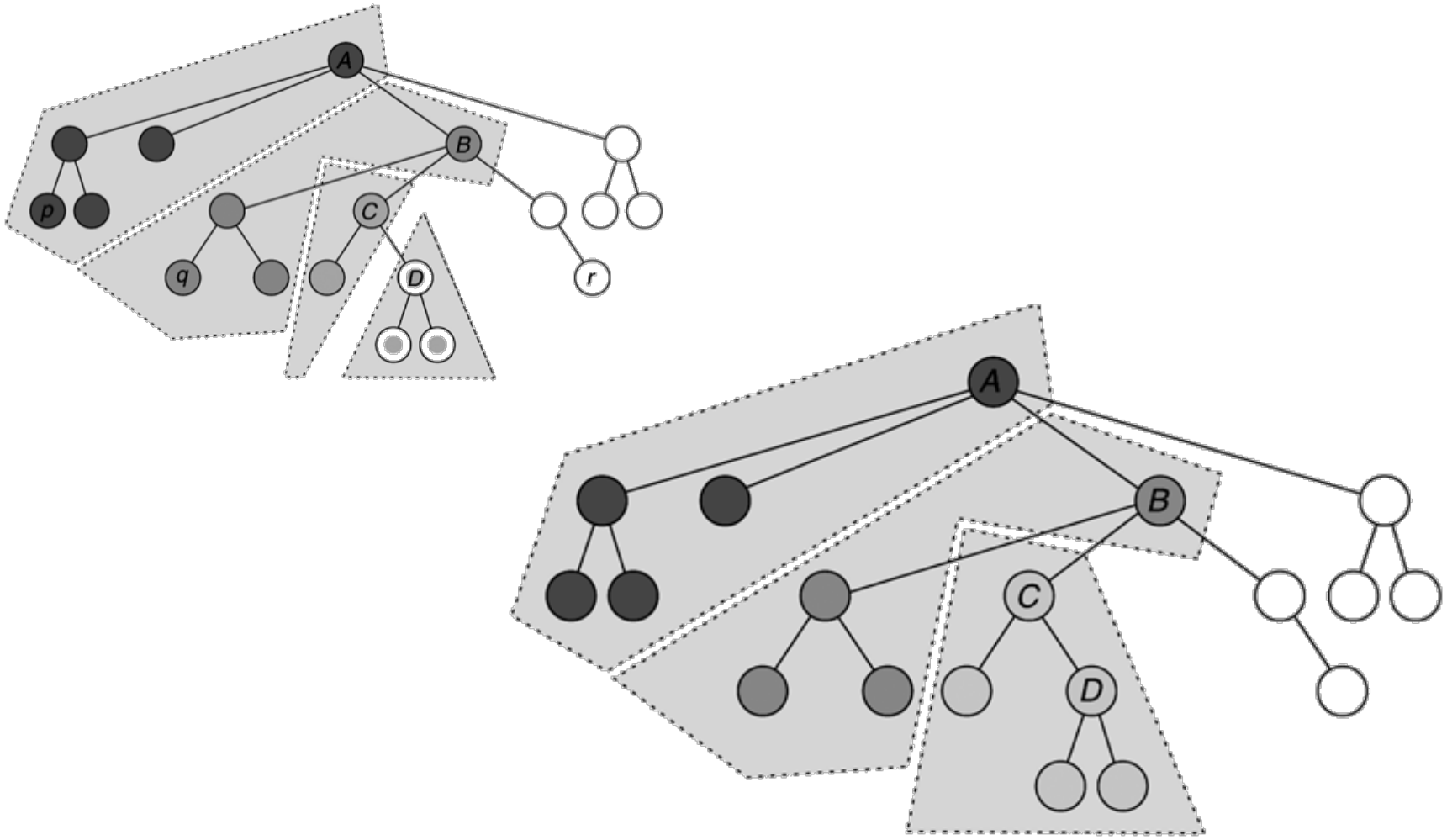


The nearest common ancestor (NCA) for each request in the pair sequence

- NCA(x, y) is A
- NCA(x(u, z) is C
- NCA(x(w, x) is A
- NCA(x(z, w) is B
- NCA(x(w, y) is y



Before we return from D in post-order traversal, this is how the disjoint sets look like. Anchors (A, B, C, D) are nodes in stack.



After we return from D in post-order traversal, we union(C, D) and we can answer NCA(D, x) for all x that has been visited, such as NCA(D, p) and NCA(D, q) but not NCA(D, r)