Disjoint Sets

Introduction to Disjoint Sets

- Data structure for problems requiring equivalence relations
  - Example: Are two elements in the same equivalence class.
- Disjoint sets provide a simple, fast solution
  - Simple: array-based implementation
  - Fast: O(1) per operation average case
- Disjoint-set also known as “union-find”

Equivalence Relations

- A relation R on set S if for every pairs of elements (a, b),a, b $\in$ S ; (a R b) is either true for false. If (a R b) is true, then we say that a is related to b.
- An equivalence relation is a relation R that satisfied three properties.
  - Reflexive – a ~ a, for all a
  - Symmetric – a ~ b if and only if b ~ a.
  - Transitive – a ~ b and b ~ c implies that a ~ c.

Equivalence Classes

- Given an equivalence relation R, decide whether a pair of elements a, b belongs to S is such that (a R b).
- The equivalent class of an element **a** is the subset of S of all elements related to **a**.
- The subsets that represent the equivalence classes will be "disjoint"
- Disjoint sets are sets such that: $S_i \cap S_j = \phi$.
  - Example:
  - S1 {a, b, c} and S2 {d, e} are disjoint
  - But S3 {x, y, z} and S4 {t, u, x} are not disjoint

Equivalence classes –

- Given: S = {1, 2, 3, 4, 5, 6}
- Given R = {(1, 1), (1, 3), (1, 5), (3, 1), (3, 3), (3, 5), (5, 1), (5, 3), (5, 5), (2, 2), (2, 6), (6, 2), (6, 6), (4, 4)}

[1] = [3] = [5] = {(1, 1), (1, 3), (1, 5), (3, 1), (3, 3), (3, 5), (5, 1), (5, 3), (5, 5), }
= **{1, 3, 5}**

[2] = [6] = {(2, 2), (2, 6), (6, 2), (6, 6)} = **{2, 6}**

[4] = {(4, 4)} = **{4}**

{1, 3, 5}, {2, 6} and {4} are equivalence classes

Equivalence Relation Application

- Suppose we have an application involving N distinct items. We will not be adding new items, nor deleting any items. Our application requires us to use an equivalence relation to partition the items into a collection of equivalence classes (subsets) such that:
  - Each item is in a set
  - No item is in more than one set

The Disjoint Set ADT

- A disjoint set data structure keeps nodes of a set of non-overlapping subsets.
- It is a data structure that helps us solve the dynamic equivalence problem.

Disjoint Set Operations

- We identify a set by choosing a **representative element** of the set. It doesn't matter which element we choose, but once chosen, it can't change.
- There are operations of interest:
  - find(x) – determine which set that x is in. The return value is the representative element of that set.
  - union(x, y) – make one set out of the sets containing x and y.

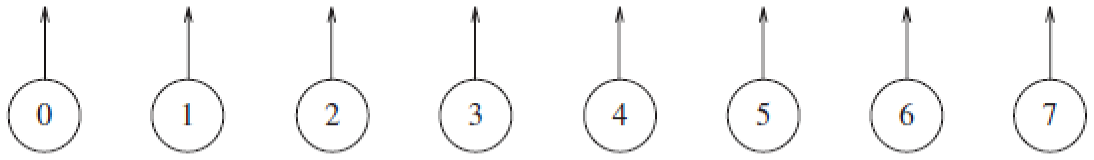
Disjoint Set Operations (cont.)

- If we want to add a pair (a, b) to the list of relations, we:
  - Determine whether a and b are related. This is done by performing finds on both a and b and checking whether they are in the same equivalence class.
  - If they are not, apply union. This operation merges the two equivalence classes containing a and b into a new equivalence class.

- From a set point of view, the result of union (∩) is to create a new set, destroying the originals and preserving the disjointness of all the sets. The algorithm to do this is frequently known as the **disjoint set union/find algorithm** for this reason.
- This algorithm is **dynamic** because, during the course of the algorithm, the sets can change via the union operation.

Basic Data Structure

- Using tree (up-tree) to represent each set, since each element in a tree has the same root. The root can be used to name the set.
- We will represent each set by a tree. Initially, each set contains one element. The trees we will use are not necessary binary trees, but their representation is easy because the only information we will need is a parent link.
- Example: {0} {1} {2} {3} {4} {5} {6} {7}



Eight elements, initially in different sets. The vertical line represents the root's parent.

Implementation

```
class DisjSets
{
  public:
    explicit DisjSets( int numElements );

    int find( int x ) const;
    int find( int x );
    void unionSets( int root1, int root2 );

  private:
    vector<int> s;
};
```

```
/**
 * Construct the disjoint sets object.
 * numElements is the initial number of disjoint sets.
 */
DisjSets::DisjSets( int numElements ) : s( numElements )
{
    for( int i = 0; i < s.size( ); i++ )
        s[ i ] = -1;
}
```

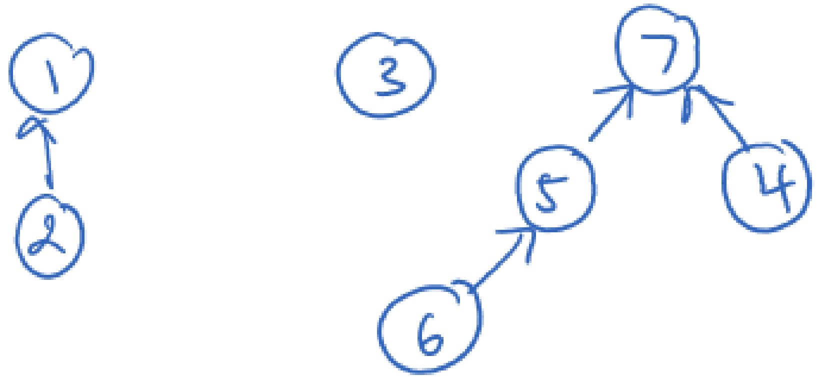
```cpp
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    s[ root2 ] = root1;
}
```

```cpp
/**
 * Perform a find.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
 */
int DisjSets::find( int x ) const
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}
```

Basic Data Structure – find(x)

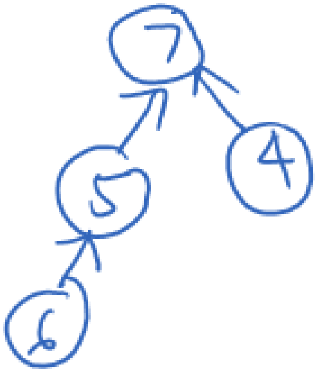
initial state

After several unions



Find Operation

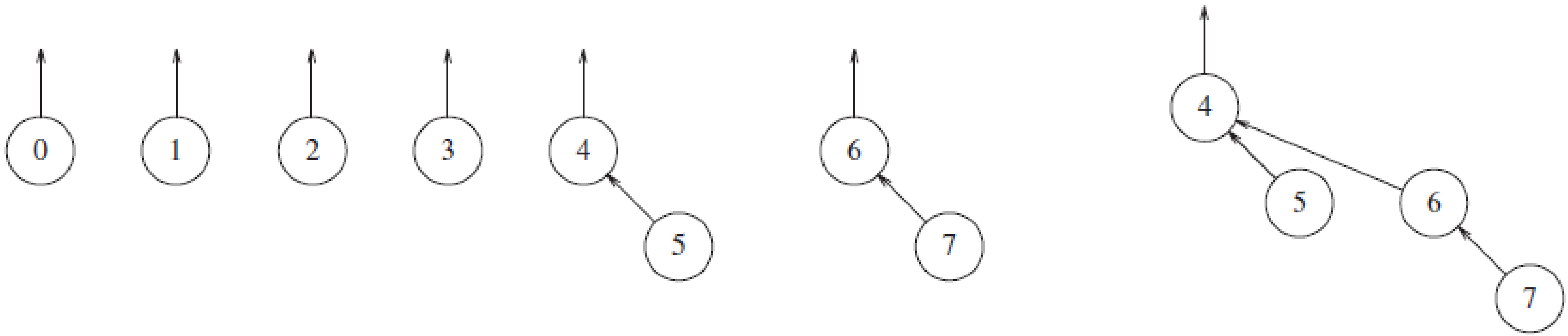
Find(x) - follow x to the root and return the root



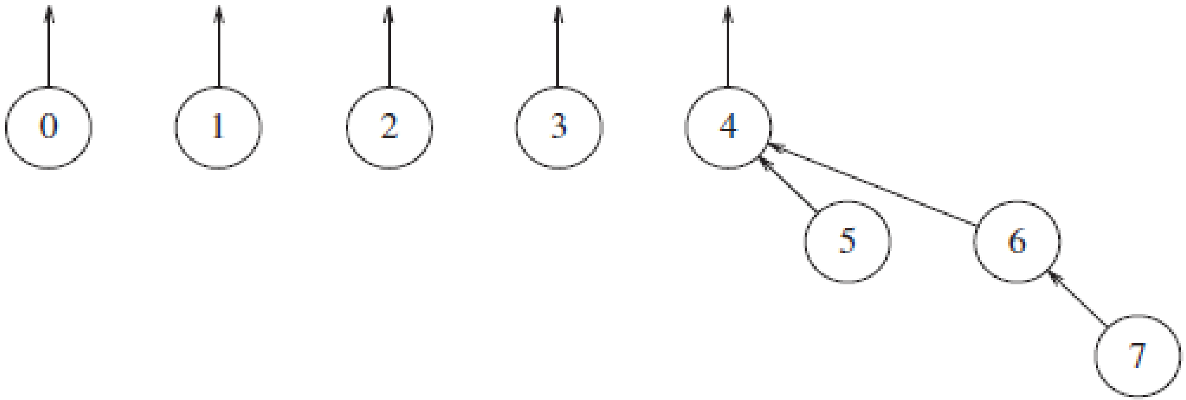
find (6) = 7

Basic Data Structure – union(x, y)

- To perform a union of two sets, we merge the two trees by making the parent link of one tree's root link to the root node of the other tree.
- Union (x, y) = y points to x;
- Example: union(4, 5), union (6, 7), union(4, 6)



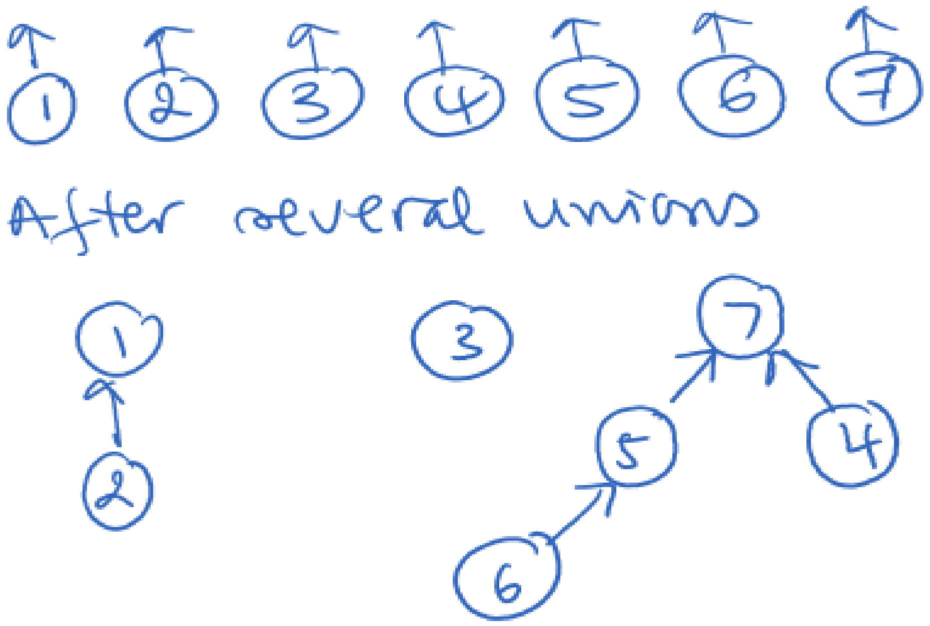
Array Representation of Tree (up-tree)



–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

- Assume each element is associated with an integer i=0 …n-1.
- Create an integer array, s[n]
- An array entry is the element’s parent
- If i is a root, then s[i] = -1
- The highlight is element of each set.
- {0} {1} {2} {3} {4} are the root (-1).
- 4 is parent of set {4, 5}
- 4 is parent of set {4, 6}
- 6 is parent of set {6, 7}

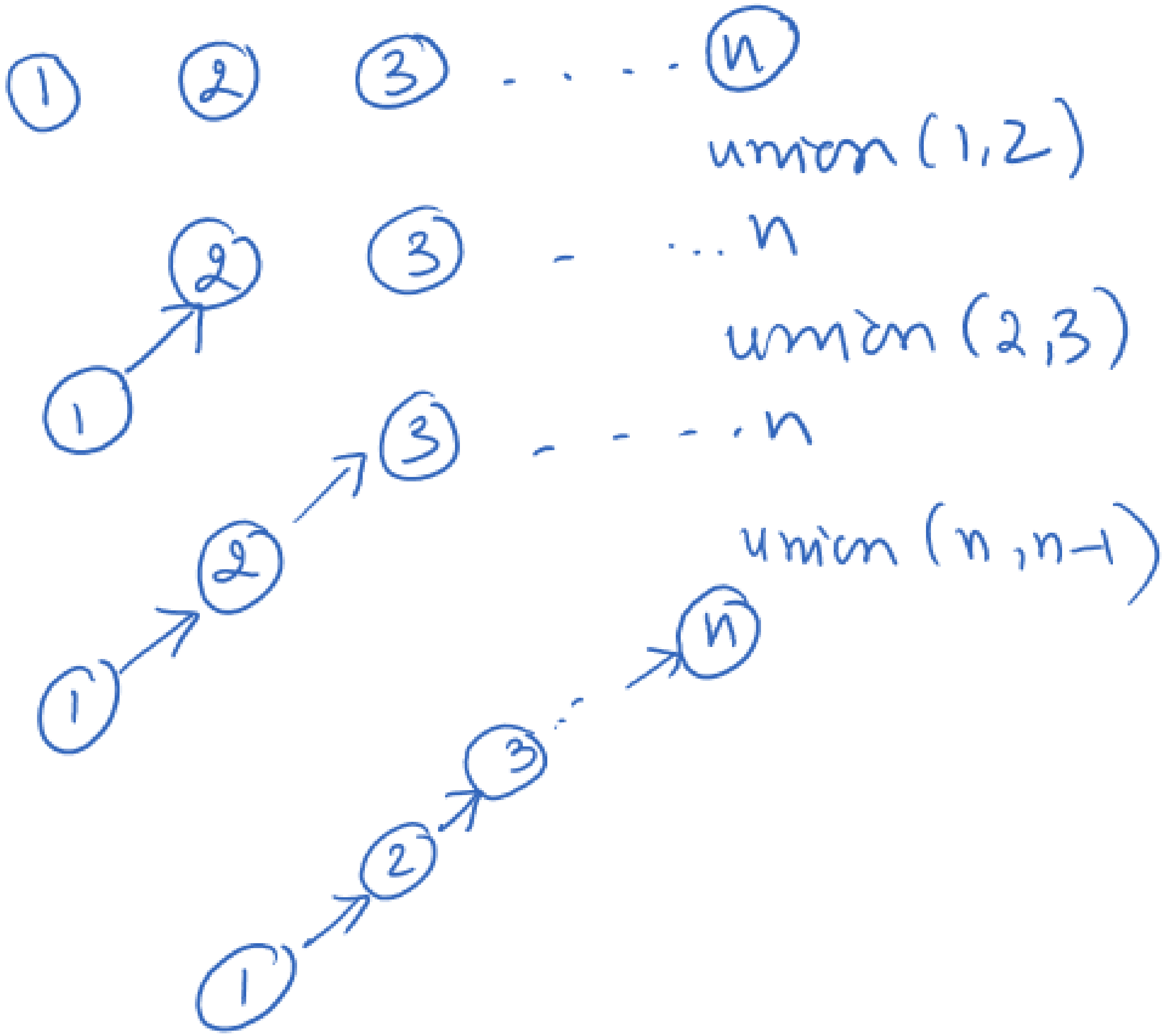
Array Implementation



Root	-1	1	-1	7	7	5	-1
	1	2	3	4	5	6	7

- {1}, {3} and {7} are the root (-1).
- 1 is parent of set {1, 2}
- 7 is parent of set {7, 4}, {7, 5}
- 5 is parent of set {5, 6}

A Bad Case



- Find(x) = n steps!

Improving Performance

- Can we do better? Yes
- Improve union so that **find** only take O(log n) – Smart union
    - Union by weight by size
    - Union by height or by rank

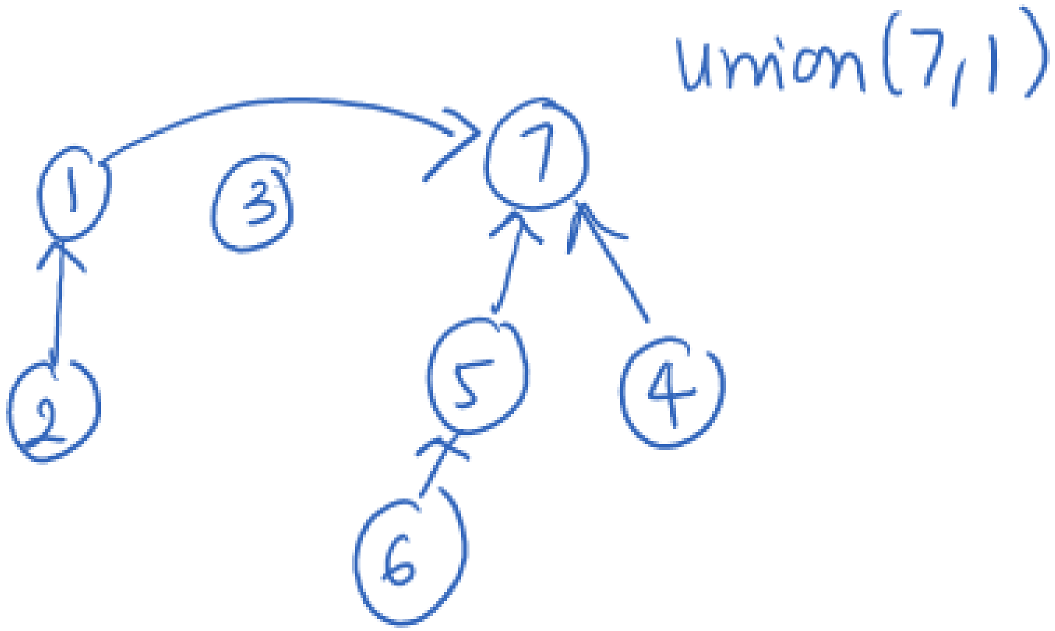
  Reduces complexity to O(m log n)

- Improve find so that it becomes even better!
  - Path compression on find
  - Reduces complexity to almost O(m+n)

Smart Union Algorithms

- A simple improvement is always to link the smaller tree to the larger one. We called this approach **union-by-size**.
- We can prove that unions are done by size, the depth of any node is never more than log N. Note that a node is initially at depth 0. When its depth increases as a result of a union, it is placed in a tree that is at least twice as large as before. Thus its depth can be increased at most log N times.
- This implies the running time for a find operation is O(log N), and a sequence of M operations take O(M log N).

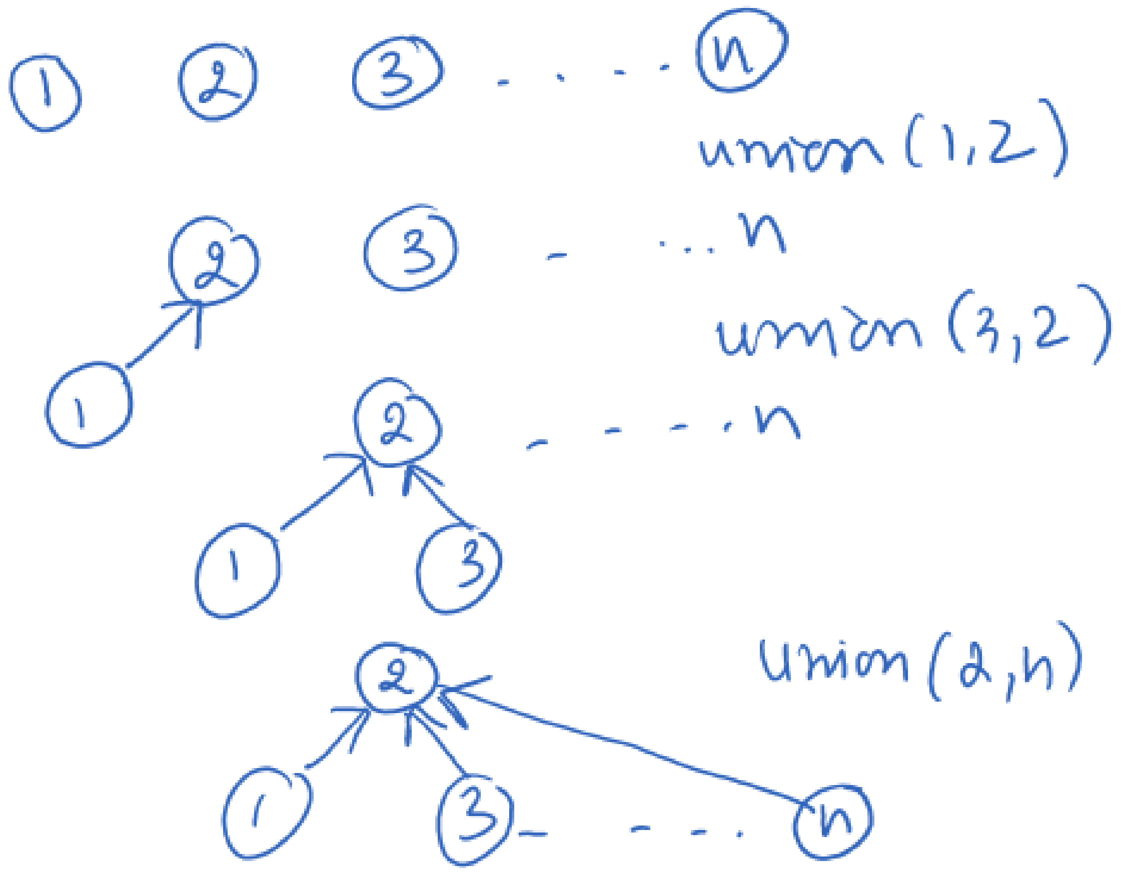
Smart Union – union by size / weight



Root	7	1	-1	7	7	5	-1
	1	2	3	4	5	6	7

to link the smaller tree to the larger one.

Improving the bad case



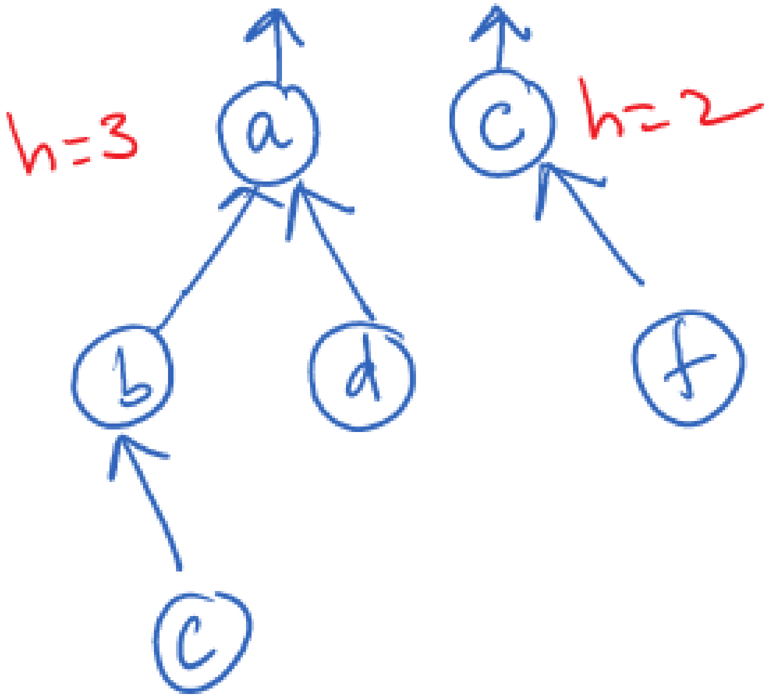
Find(1) = constant time !

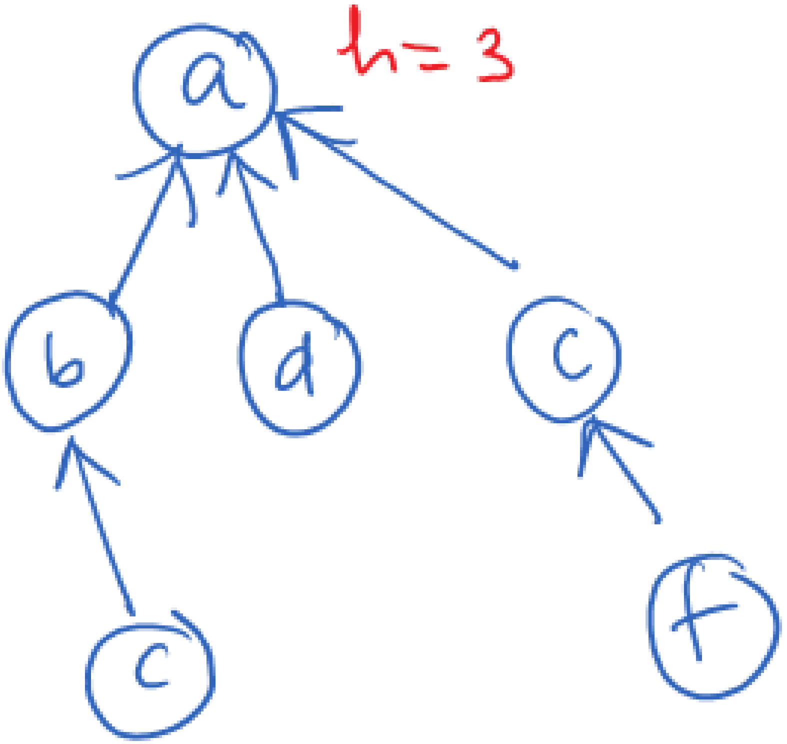
Find(1) at constant time

Smarter Union – Union by Height (rank)

- Each root stores the height of its respective tree
- If one root has greater height than the other, it becomes the parents.
- If the roots show equal height, pick either one as the parent.

Union by Height (rank)





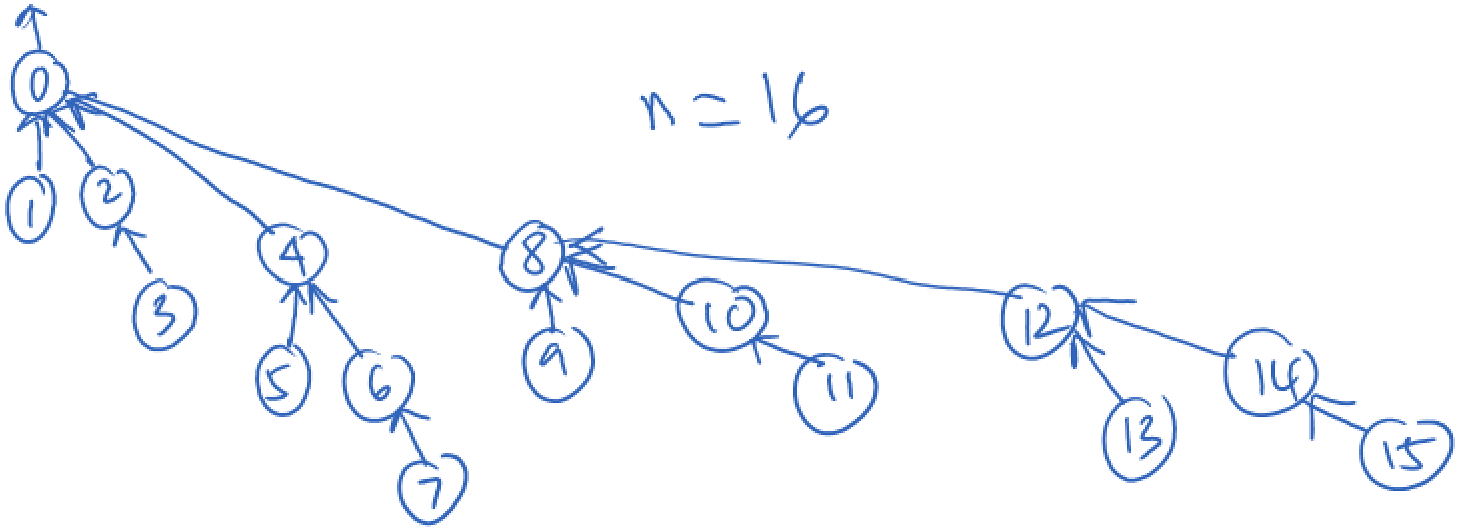
Root (a) has greater height than root (c), (a) becomes the parents.

Smart Union by Height - Implementation

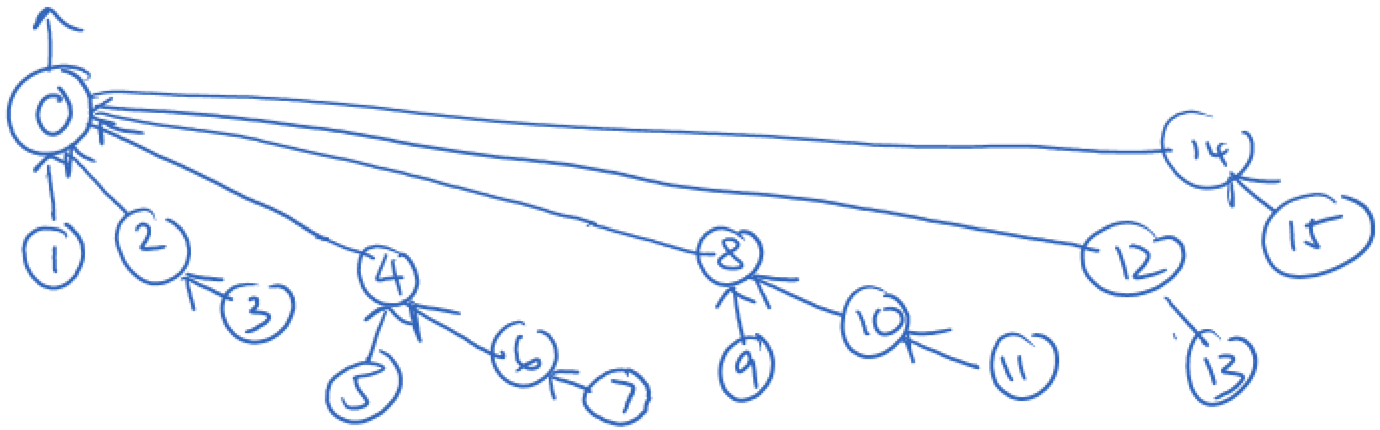
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
        s[ root1 ] = root2;        // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;        // Make root1 new root
    }
}
```

Path Compression

- Smart union achieves O(M) time for M operations (average case)
- But still O(M log N) in the worst case
- Path compression is performed during a find operation.
- All nodes accessed during a find(x) are linked directly to the root
- Path compression without smart union still O(M log N) worst case.



- Worst case tree for N = 16



- The effect of path compression after find(14).

Path Compression Implementation

```
/**
 * Perform a find with path compression.
 * Error checks omitted again for simplicity.
 * Return the set containing x.
 */
int DisjSets::find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

Path Compression with Smart Union

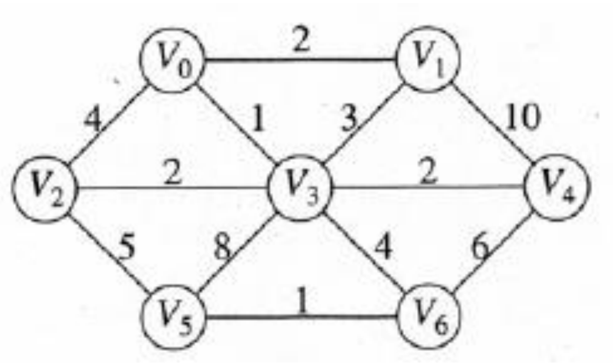
- Path compression works as is with union-by-size (tree sizes don’t change)
- Path compression with union by height requires re-computation of heights.
- Path compression does not change average case time, but does reduce worst case time.

Applications of Disjoint Sets

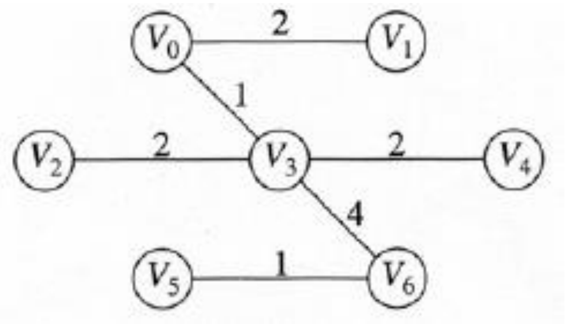
- Kruskal's minimum spanning tree
- Generating mazes
- Network connectivity
- Grid percolation
- Image processing

Minimum Spanning Trees

- A spanning tree of an undirected graph is a tree formed by graph edges that connect all the vertices of the graph.
- A minimum spanning tree is a connected subgraph of G that spans all vertices at minimum cost.
  - The number of edges in the minimum spanning tree is |V| - 1



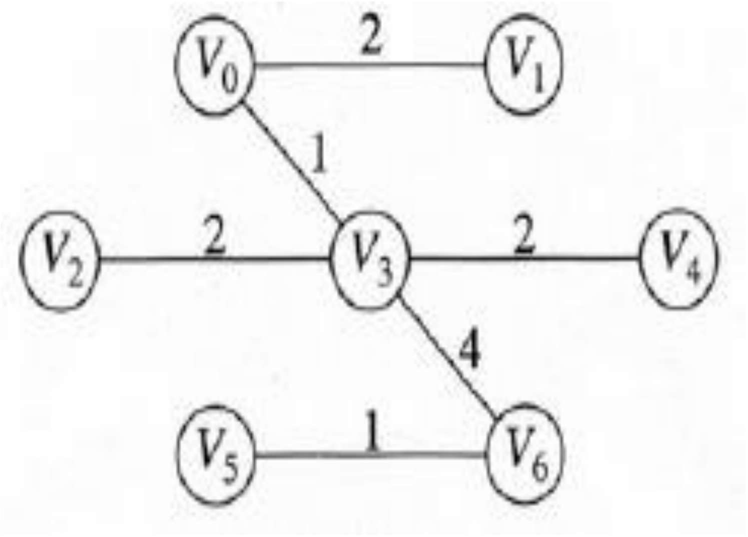
Graph



Minimum spanning tree

Kruskal’s algorithm

- Kruskal’s algorithm, uses to find the minimum spanning tree, is simple
  - IT continually selects edges in order of smallest weight to add to the tree if it does not cause a cycle.
  - Notice that the edges (v1, v3) and (v0, v2) are rejected because either would cause a cycle.



Disjoint Set Example

- Given a set of cities, C, and a set of roads, R, that connect two cities (x, y) determine if it is possible to travel from any given city to another given city.

```
for (each city in C)
    put each city in its own set
for (each road (x,y) in R)
    if (find9x) != find(y))
        union (x, y)
```

Now we can determine if it’s possible to travel by road between two cities c1 and c2 by testing

```
        find (c1) == find (c2)
```

Kruskal’s algorithm

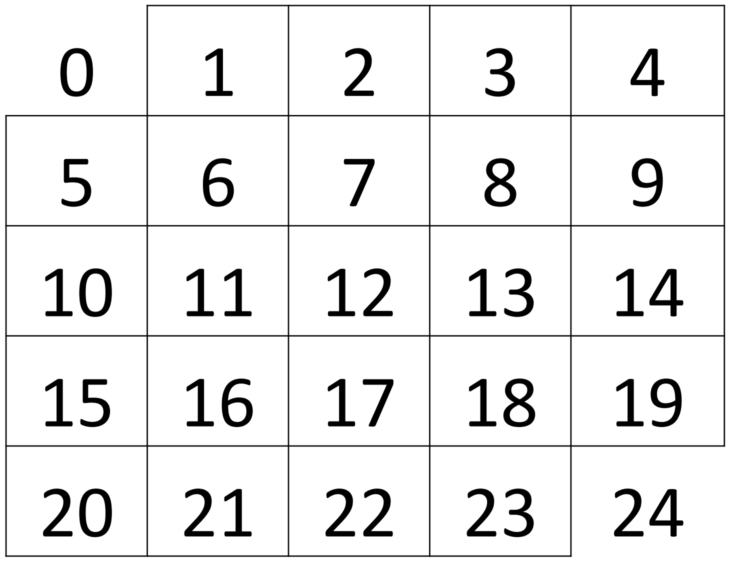
- How do we determine whether an edge (u, v) should be accepted or rejected?
  - Maintain each connected component in the spanning forest as a disjoint set
  - If u and v are in the same disjoint set, as determined by two find operations, the edge is rejected because u and v are already connected.
  - Otherwise, the edge is accepted and a union operation is performed on the two disjoint sets containing u and v, in effect, combining the connected components.

Maze Generator - A union-find application

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 random maze generator can use union-find. Consider a 5x5 maze.
- Initially, 25 cells, each isolated by walls from the others.
- This corresponds to an equivalence relation – two cells are equivalent if they can reached from each other (walls been removed so there is a path from one to the other).

Maze Generator – (cont.)



- To start, choose an entrance and exit.
- Randomly remove walls until the entrance and exit cells are in the same set.
- Removing a wall is the same as doing a union operation.
- Do not remove a randomly chosen wall if the cells are already in the same set.

MakeMaze

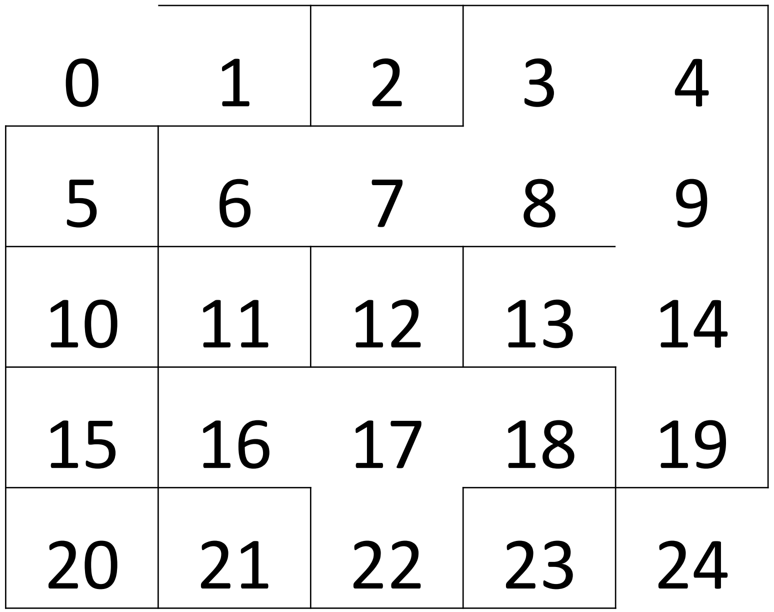
```
MakeMaze(int size)
{ entrance = 0; exit = size - 1;
While (find(entrance) != find(exit)) {
    cell1 = a randomly chosen cell
    cell2=  a randomly chosen adjacent cell
    if (find(cell) != find (cell2)
        union (cell1, cell2)
} // end while
} //end MakeMaze
```

Initial State

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}

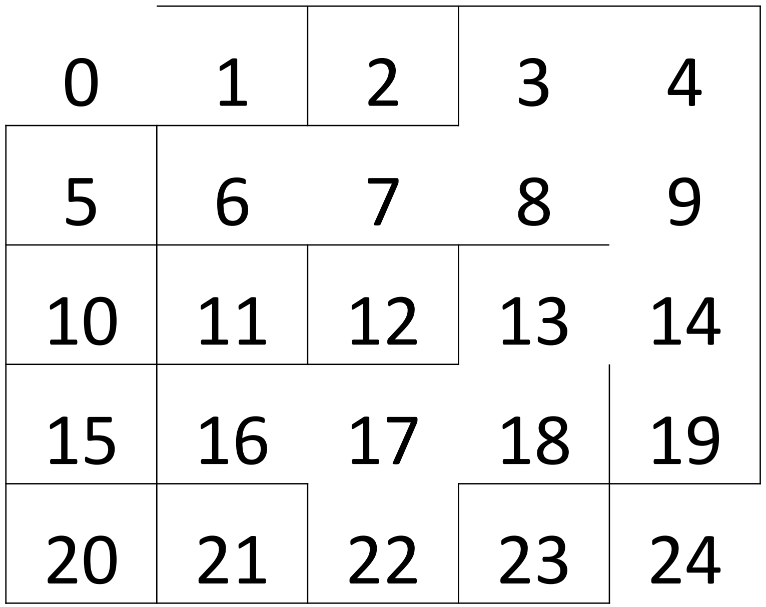
Intermediate State



- Algorithm selects wall between 18 and 13.

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

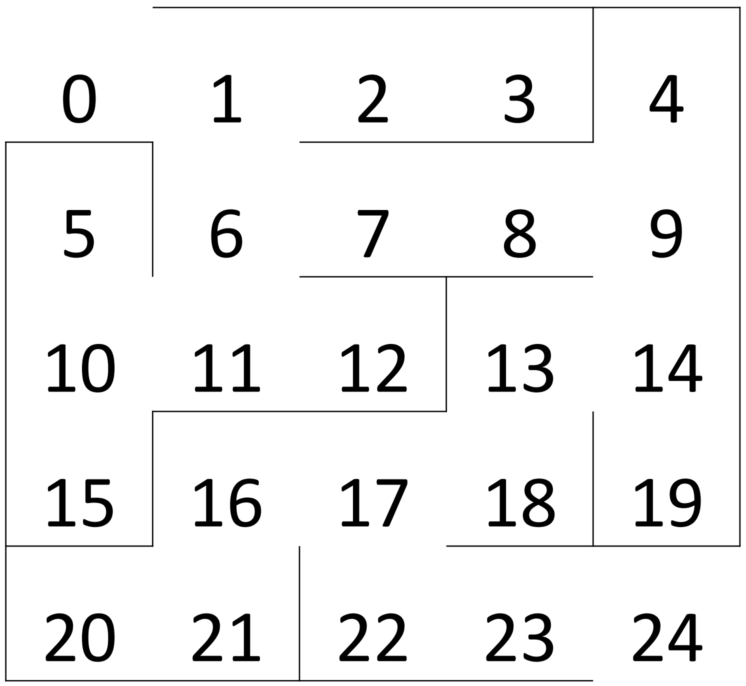
A Different Intermediate State



- Algorithm selects wall between 18 and 13.

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

Final State



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

Time Complexity of Disjoint Sets

Simple Scheme

Find(x): O(n) for sets of cardinality n in the worst case.

Union(x,y): O(1) for root element, O(n) worst case

Better scheme

Find(x): at most O(log n) time.

Union(x, y): constant time if x and y are roots.

The End!