CSCI 335
Software Design and Analysis III
Lecture 20:
The Disjoint Sets Class

Professor Anita Raja

11-14-22

Agenda

- Equivalence Relations
  - Union-Find
  - Smart-Union
  - Union by size
  - Union by height
  - Path compression
  - Union by rank
- Analysis
- Application

Equivalence Relations

- **Relation R** on a set S: for every pair (a,b), a, b ∈ S, a R b is either T or F.
  - If a R b is T, we say a is related to b.
  - Examples:

    Relation < on the set {0,1,2,3,4,5,6}

    ....Many more....

- **Equivalence** relation R has 3 properties
  - a R a, for all a (reflexive)
  - a R b => b R a (symmetric)
  - a R b and b R c => a R c (transitive)

- Examples (which ones of the following are equivalence relations?):
  - = ?
  - ≤ ?
  - Two cities in the same country ?

Dynamic Equivalence Problem

- Given set S={a1,a2,a3,a4,a5,a6}
  - 36 pairs of elements that are either related or not

- Given equivalence relation ~
  - a1~a2, a2~a3, a5~a6
  - ai ~ ai for i=1,..,6 (from reflexivity)
  - Symmetry => a2~a1, a3~a2, a6~a5
  - Transitivity => a1 ~ a3, a3 ~ a1

- Equivalence class of an element *a* ∊ S: is subset of S ∊ all elements related to *a*
  - The equivalence classes form a partition of S.
  - Every member of S appears in exactly one equivalence class.
  - To decide if a ~ b, we need only check whether a and b are in the same equivalence class
  - {a1,a2,a3},{a4},{a5,a6} are the <u>equivalence classes</u> in above example

Disjoint sets

- Input is initially a collection of N sets, each with one element.
- Initial representation is that all relations (except reflexive relations) are false.
- Each set has a different element, so that $S_i \cap S_j = \emptyset$

=> Disjoint sets

Union-Find

- Find:
  - given an element, find the equivalence class it is in
- Union:
  - Add the relation a~b:
    - Perform a Find on a and b.
    - If already related, do nothing.
    - Else Union the equivalence classes of a and b into a new class.
  - $S_i \cup S_j = S_k$

Union-Find

- **Dynamic:** the equivalence classes can change with time,
  - i.e. Unions are applied at any time during the course of the algorithm.

- **On-line:** a union or find “arrives” at each instance of time and needs to be processed.
  - i.e. when a find is performed, it must give an answer before continuing.
  - Like an oral exam

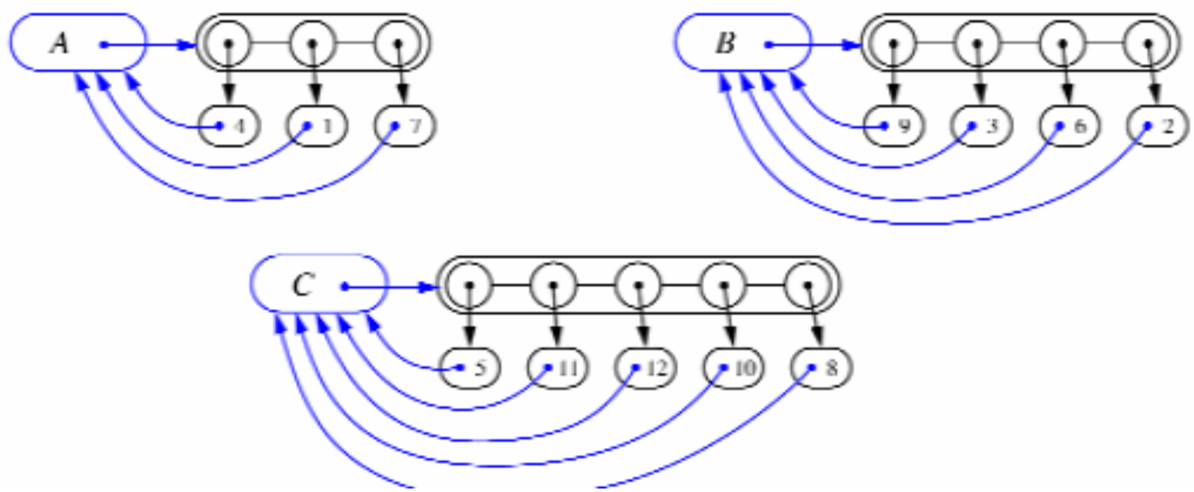
- **Off-line:** the sequence of unions and finds is given, and you need to calculate final sets (not dynamic).
  - Like a written exam

Union-Find

- Assume set S = {0,1,2,…,N-1} (can be hashed)
- Name of set returned by find is irrelevant
- Solutions?
- O(1) find?
    - Union ?

List-based Implementation

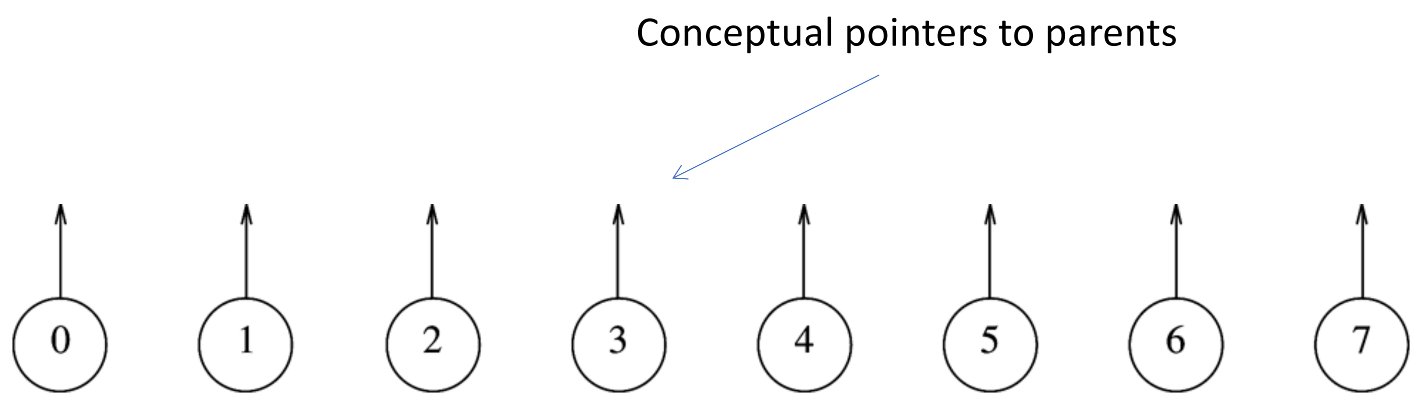
- Each set is stored in a sequence represented with a linked-list.
- Each node should store an object containing the element and a reference to the set name



Analysis of List-based representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set.
  - Thus, an element can be moved at most O(log N) times.
- Total time needed to do M finds and N-1 unions is O(M+NlogN)
- O(NlogN)

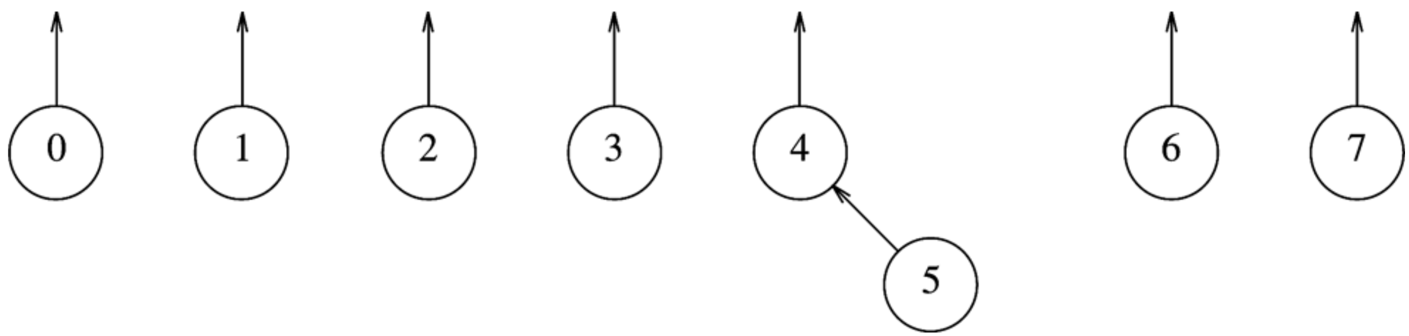
Better



Each equivalence class is represented by a tree.
The name of the class is the root (i.e. name of the representative).
Initially each element is one class containing itself (reflexivity).
Can be implemented by array

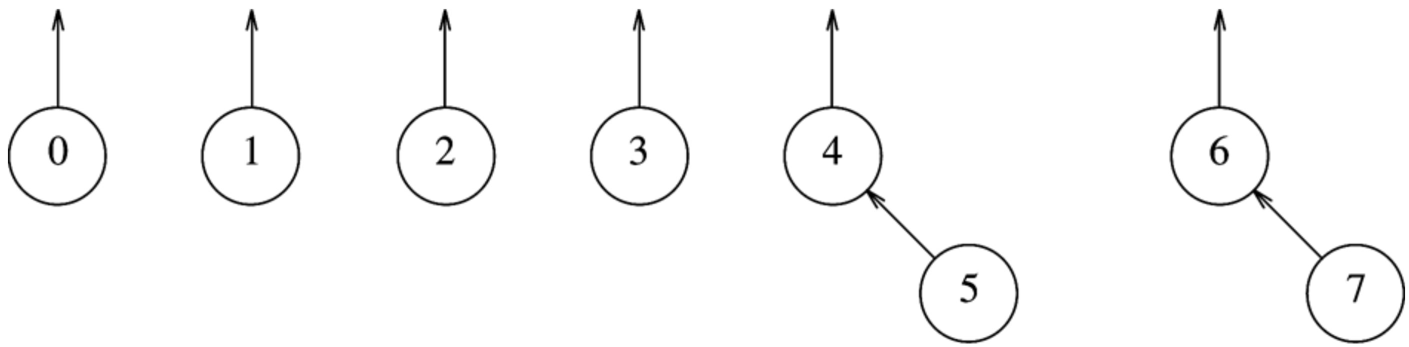
Better

After Union(4,5)



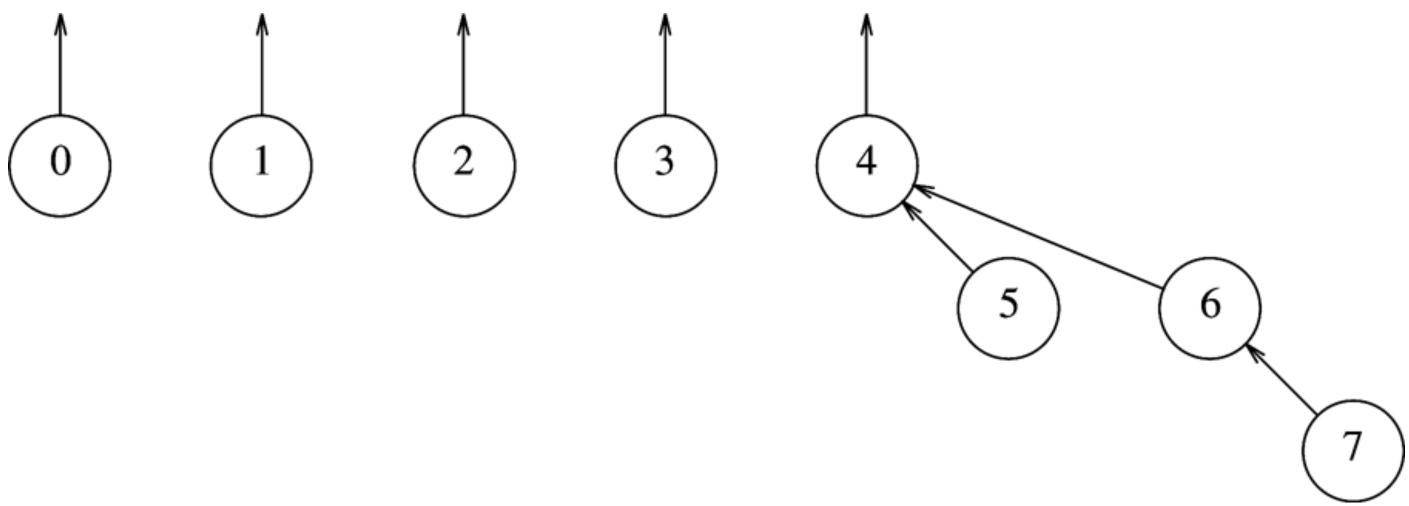
Better

After Union(6,7)



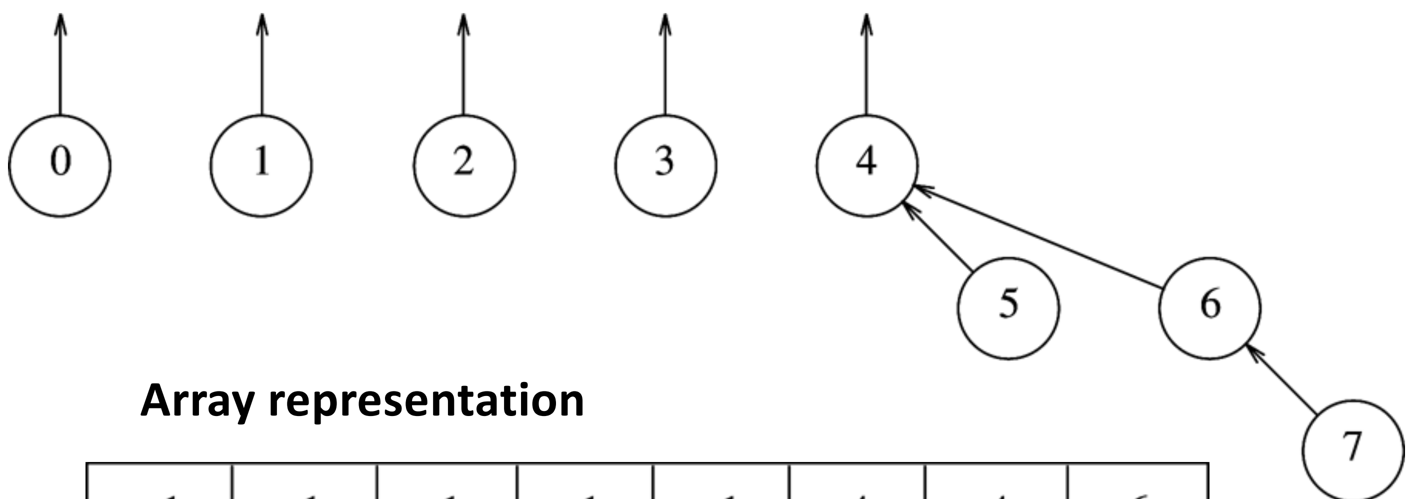
Better

After Union(4,6)



Better

After Union(4,6)



Array representation

–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

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;
};
```

Analysis

- Union's cost?

Analysis

- Union's cost?
- Union: Fast O(1)

Analysis

- Union's cost?
- Union: Fast O(1)
- Find's cost?

Analysis

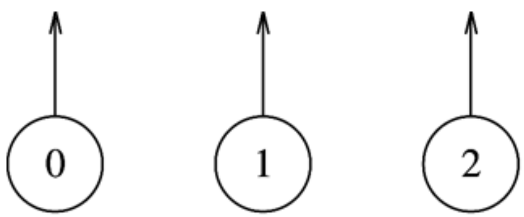
- Union's cost?
- Union: Fast O(1)
- Find's cost ?
- Find: Worst-case depth of deeper node O(N)

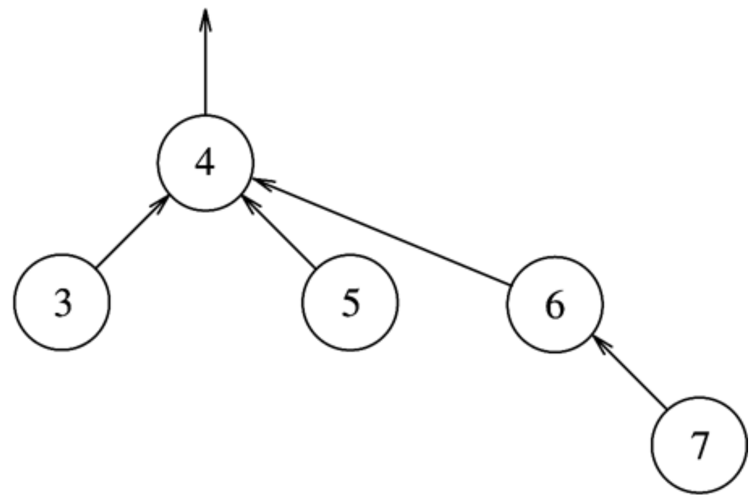
=> O(MN) for M mixed find/union operations

- Average running time: depends on what is *average:*
  - *$\Theta(M)$ or $\Theta(MN)$ or $\Theta(M\log N)$ for M unions*

Smart Union Algorithms

- Union by size: Smaller tree becomes subtree of larger

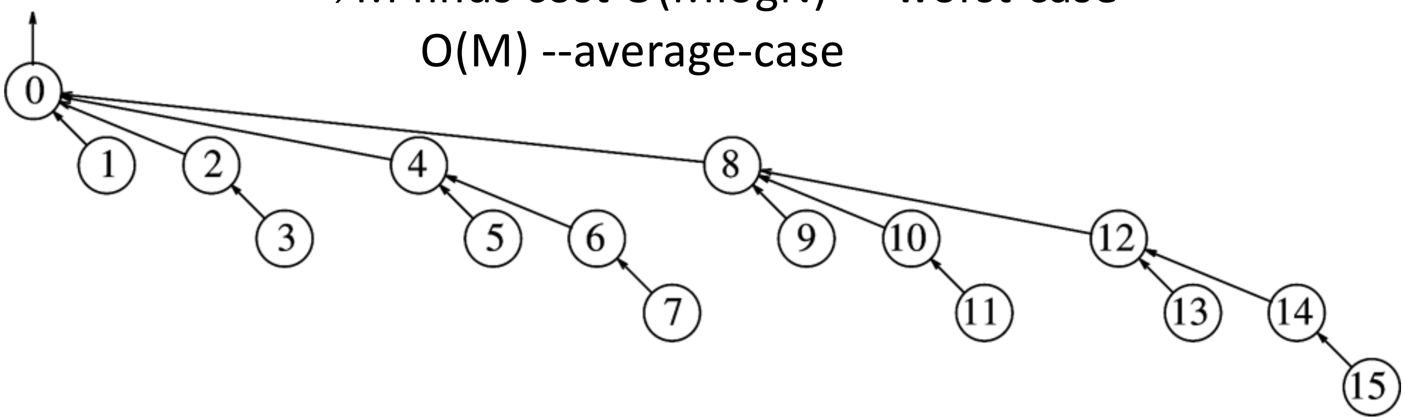




After Union(3,4)

Smart Union Algorithms

Depth of any node can’t be more than logN
⇒Find costs O(logN) --worst-case
⇒M finds cost O(MlogN) --worst-case
O(M) --average-case

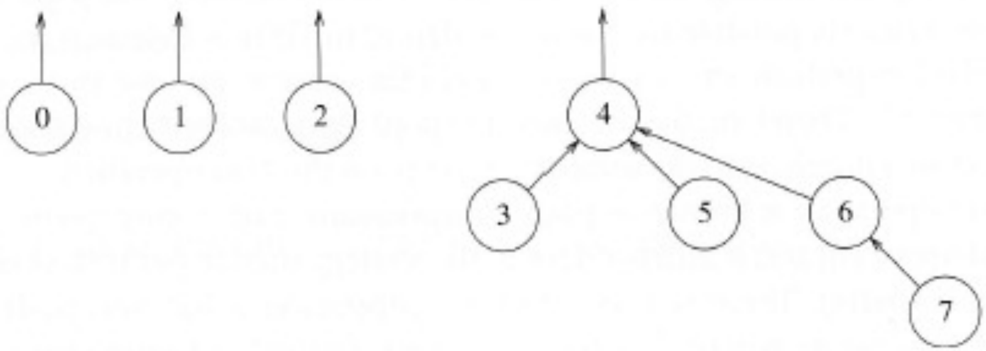


Worst-case for union when N=16 (binomial tree) [result after 16 unions]
[union of equal size trees]

Smart Union Algorithms

- **Union-by-height**: shortest tree under tallest
- Height is increased by one **iff** when two trees of equal height are combined.
- Again O(logN) find

Smart Union Algorithms



–1	–1	–1	4	–5	4	4	6
0	1	2	3	4	5	6	7

Union-by-size

–1	–1	–1	4	–3	4	4	6
0	1	2	3	4	5	6	7

Union-by-height

Smart Union Algorithms

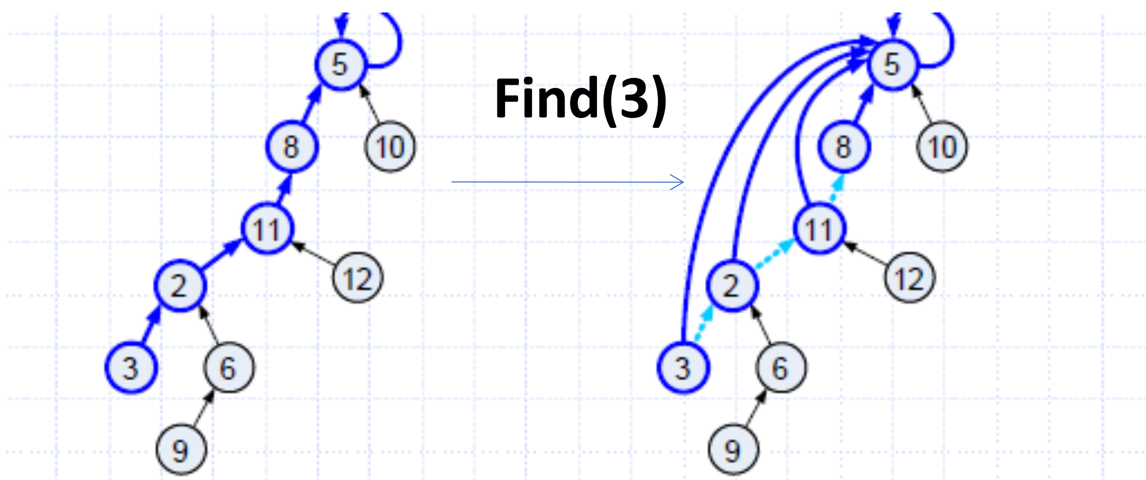
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
       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

- Improve on the O(MlogN) worst-case find() cost for M finds
- Ideas?

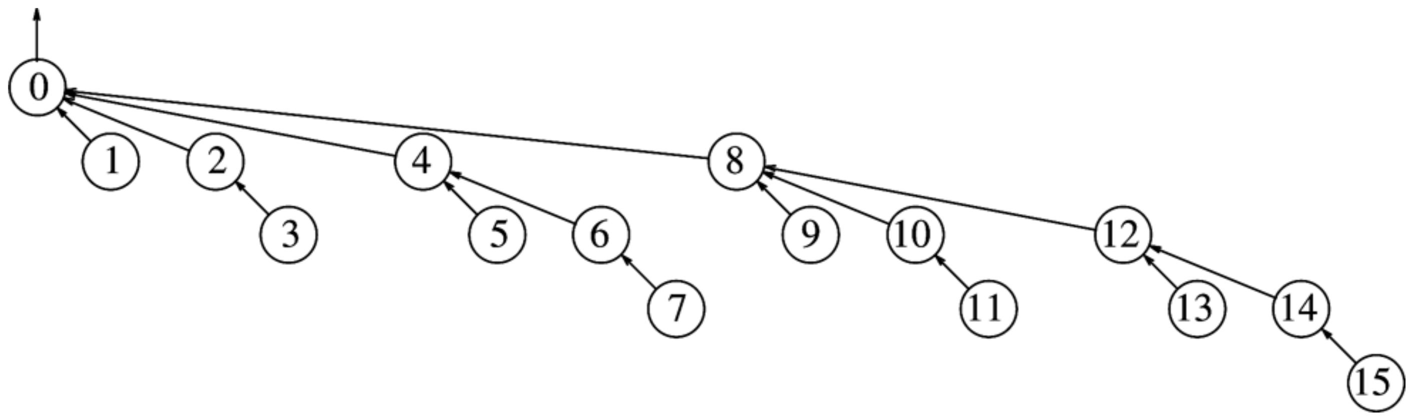
Path Compression

- Improve on the O(MlogN) worst-case find() cost for M finds
- It can appear frequently
- **Path compression:** During find(x) make all nodes on the path from x to the root to be children of the root.

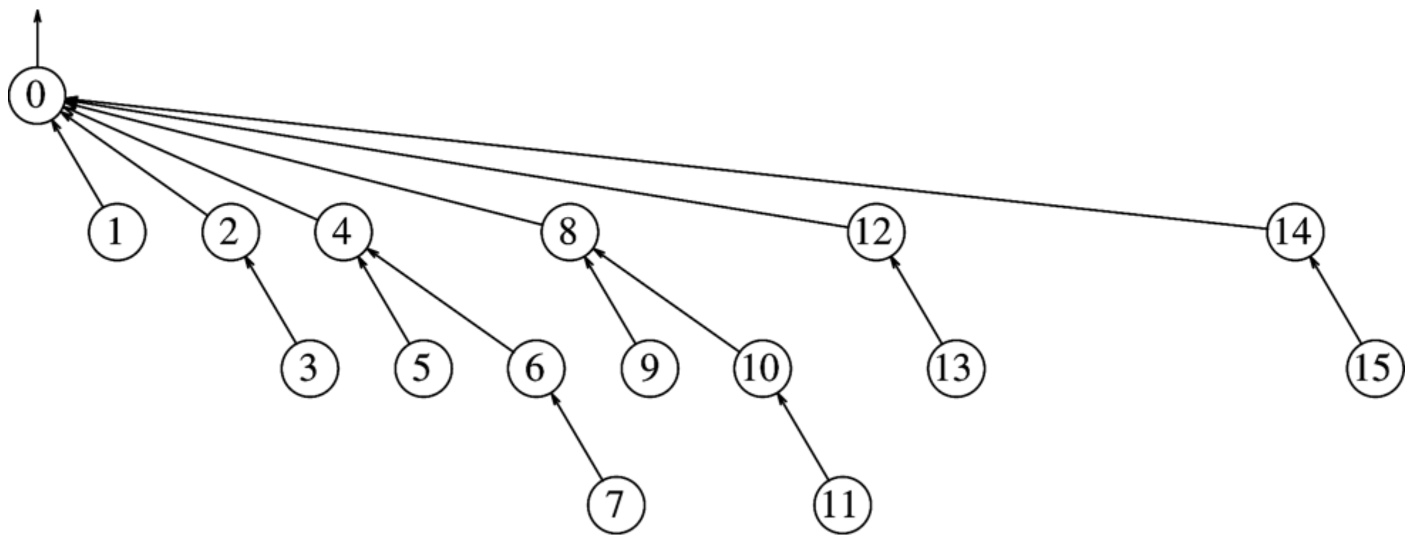


Path Compression

Apply find (14) in this set



Path Compression



Path Compression

```
/**
 * 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 ] );
}
```

Union-by-rank

- **Union-by-rank:** after compression the height of tree may not be accurate.
- It is now called **rank** => it is an upper bound on the actual height.

Path Compression

(A) Union-by-size + Path Compression

or

(B) Union-by-rank + Path Compression

Average case: Not sure whether Path Compression helps

Worst case: Path Compression helps a lot

Union-by-rank is preferred because it requires fewer updates on heights

Worst-Case Analysis of Union-Find

- Definition: **log*N** is number of times logarithm of N needs to be applied until it gets to ≤ 1.
- For example: log*65536=4, because ($65536=2^{16}$)

  log log log log 65536 = 1
- Example: $\log^* 2^{65536}$ = ? (2^{65536} has 20,000 digits in decimal form)
- log*N grows very slowly as N becomes bigger !
  - Log*2 = 1
  - Log*4 = 2
  - Log*16 = 3
  - Log*65536 = 4
  - $\log^* 2^{65536} = 5$

Analysis of Union-Find

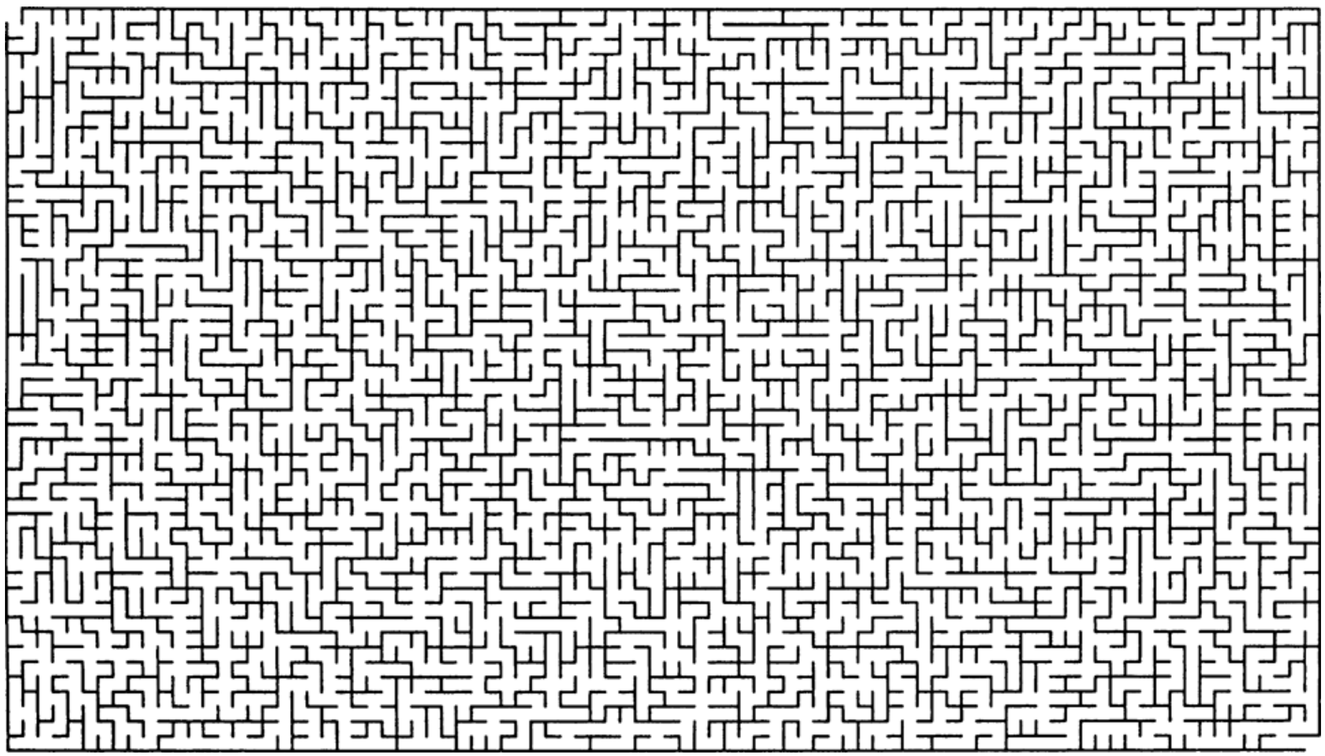
Any sequence of $M=\Omega(N)$ union/find operations takes a total of $O(M \log^* N)$ running time.

Model:

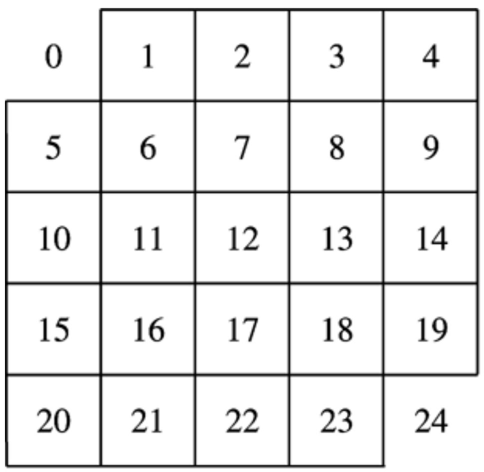
Union/Finds in any order

Union-by-rank **with** path compression

A fun application

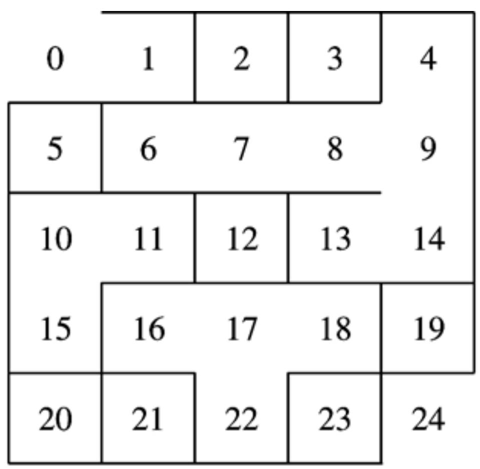


A fun application



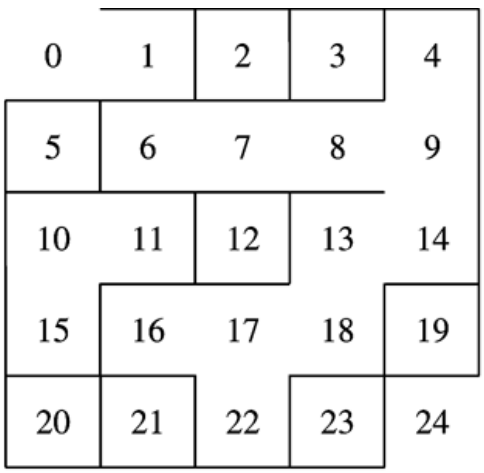
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

A fun application



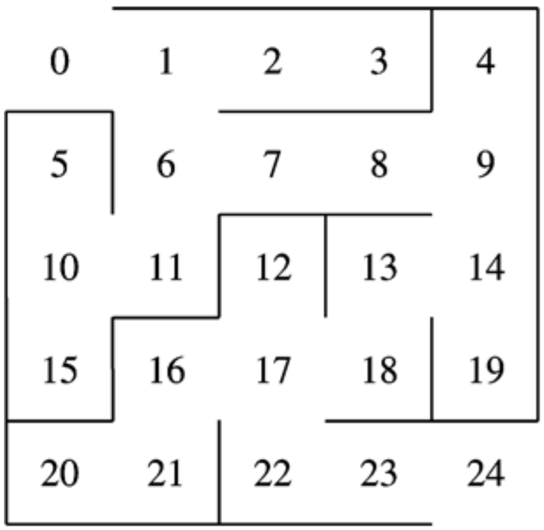
{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

A fun application



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

A fun 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}

Summary

- Simple data structure to maintain disjoint sets
- Important for graph theoretical problems
- Union step is flexible and so we get a much more effective algorithm
- Path compressions – earliest forms of self-adjustment
  - Seen in splay trees and skew heaps
  - Use extremely interesting because from theoretical pov one of the first algorithms that was simple with a not-so-simple worst case analysis.
- Any sequence of $M=\Omega(N)$ union/find operations takes a total of $O(M \log^* N)$ running time.

Model: Union/Finds in any order

Union-by-rank **with** path compression