# Union-Find: A Data Structure for Disjoint Set Operations

# The Union-Find Data Structure

- Purpose:
  - To manipulate disjoint sets (i.e., sets that don’t overlap)
  - Operations supported:

| | |
|---|---|
| ***Union ( x, y )*** | Performs a union of the sets containing two elements x and y |
| ***Find ( x )*** | Returns a pointer to the set containing element *x* |

Q) Under what scenarios would one need these operations?

# Some Motivating Applications for Union-Find Data Structures

*Given a set S of n elements, [$a_1 \ldots a_n$], compute all its equivalent classes*

Example applications:

- Electrical cable/internet connectivity network
- Cities connected by roads
- Cities belonging to the same country

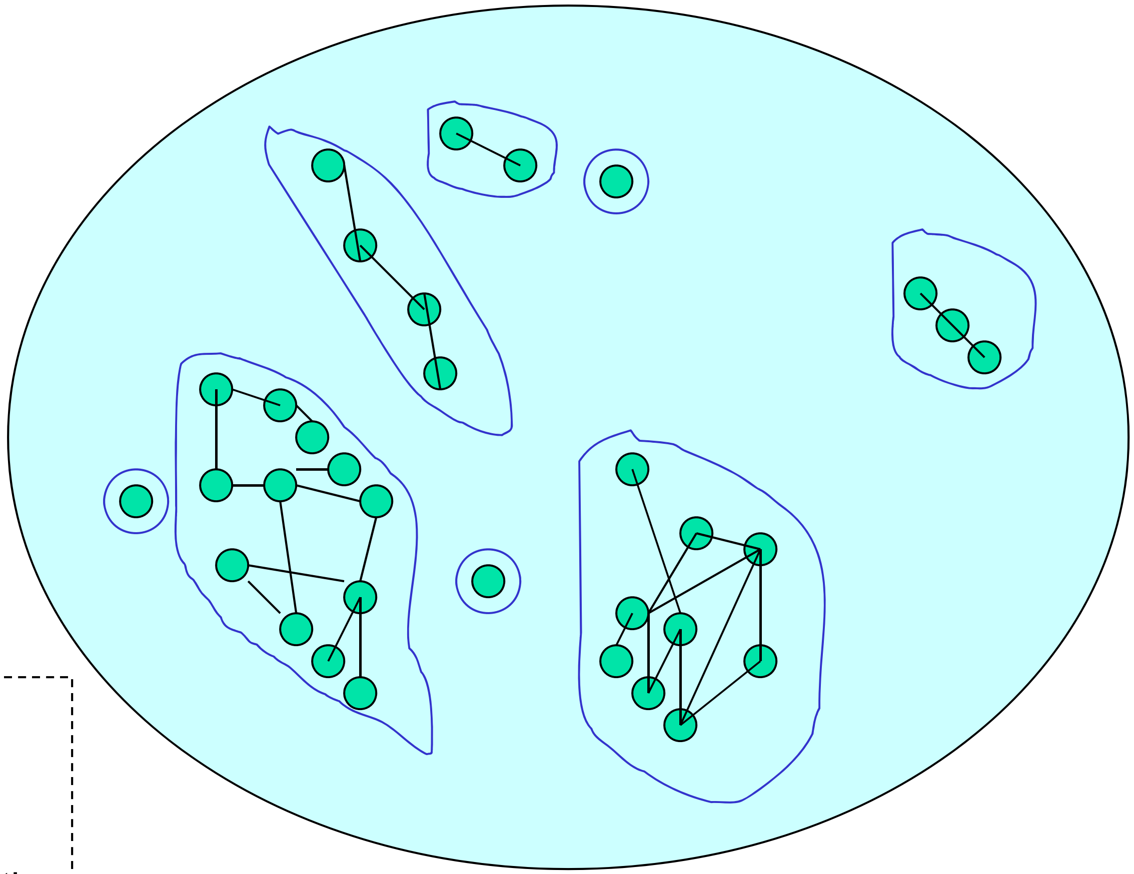
# Equivalence Relations

- An *equivalence relation* *R* is defined on a set *S*, if for every pair of elements *(a,b)* in *S*,
  - a R b is either false or true
- *a R b* is true iff:
  - (*Reflexive*) a R a, for each element a in S
  - (*Symmetric*) a R b if and only if b R a
  - (*Transitive*) a R b and b R c implies a R c
- The *equivalence class of an element a* (in S) is the subset of S that contains all elements *related* to *a*

# Properties of Equivalence Classes

- An observation:
  - Each element must belong to exactly one equivalence class
- Corollary:
  - All equivalence classes are mutually disjoint
- What we are after is the set of all equivalence classes

# Identifying equivalence classes



Legend:

Equivalence class

Pairwise relation

# Disjoint Set Operations

- To identify all equivalence classes
  1. Initially, put each each element in a set of its own
  2. Permit only two types of operations:
     - **<u>Find(x):</u>** Returns the current equivalence class of x
     - **<u>Union(x, y):</u>** Merges the equivalence classes corresponding to elements x and y (assuming x and y are related by the eq.rel.)

This is same as:

**unionSets( Find(x), Find(y) )**

# Steps in the Union (x, y)

1. $EqClass_x$ = Find (x)
2. $EqClass_y$ = Find (y)
3. $EqClass_{xy} = EqClass_x \cup EqClass_y$

union

# A Simple Algorithm to ComputeEquivalence Classes

1. Initially, put each element in a set of its own

   i.e., $EqClass_a = \{a\}$, for every $a \in S$

$\Theta(n^2)$ iterations

2. FOR EACH element pair (a,b):
   1. Check [a R b == true]
   2. IF a R b THEN
      1. $EqClass_a$ = Find(a)
      2. $EqClass_b$ = Find(b)
      3. $EqClass_{ab} = EqClass_a \cup EqClass_b$

*"Union(a,b)"*

# Specification for Union-Find

- *Find(x)*
  - Should return the id of the equivalence set that currently contains element x
- *Union(a,b)*
  - If a & b are in two different equivalence sets, then Union(a,b) should merge those two sets into one
    - Otherwise, no change

# How to support Union() and Find() operations efficiently?

- Approach 1
  - *Keep the elements in the form of an array, where:*
    *A[i] stores the current set ID for element i*
  - Analysis:
    - Find() will take O(1) time
    - Union() could take up to O(n) time
    - Therefore a sequence of m (union and find) operations could take O(m n) in the worst case
      - This is bad!

# How to support Union() and Find() operations efficiently?

- Approach 2
  - *Keep all equivalence sets in separate linked lists:*
    *1 linked list for every set ID*
  - Analysis:
    - *Union()* now needs only O(1) time
      (assume doubly linked list)
    - However, *Find()* could take up to O(n) time
      - Slight improvements are possible (think of Balanced BSTs)
    - A sequence of m operations takes $\Omega$(m log n)
    - Still bad!

# How to support Union() and Find() operations efficiently?

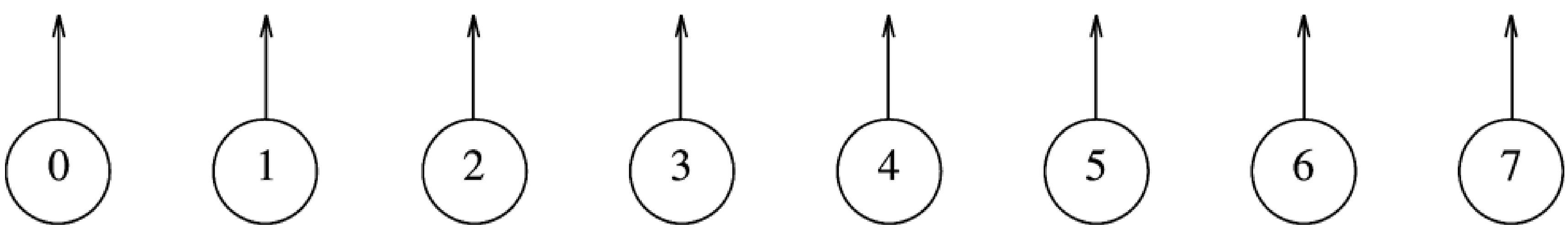
- Approach 3
  - *Keep all equivalence sets in separate trees:*
    *1 tree for every set*
  - *Ensure (somehow) that Find() and Union() take very little time (<< O(log n))*

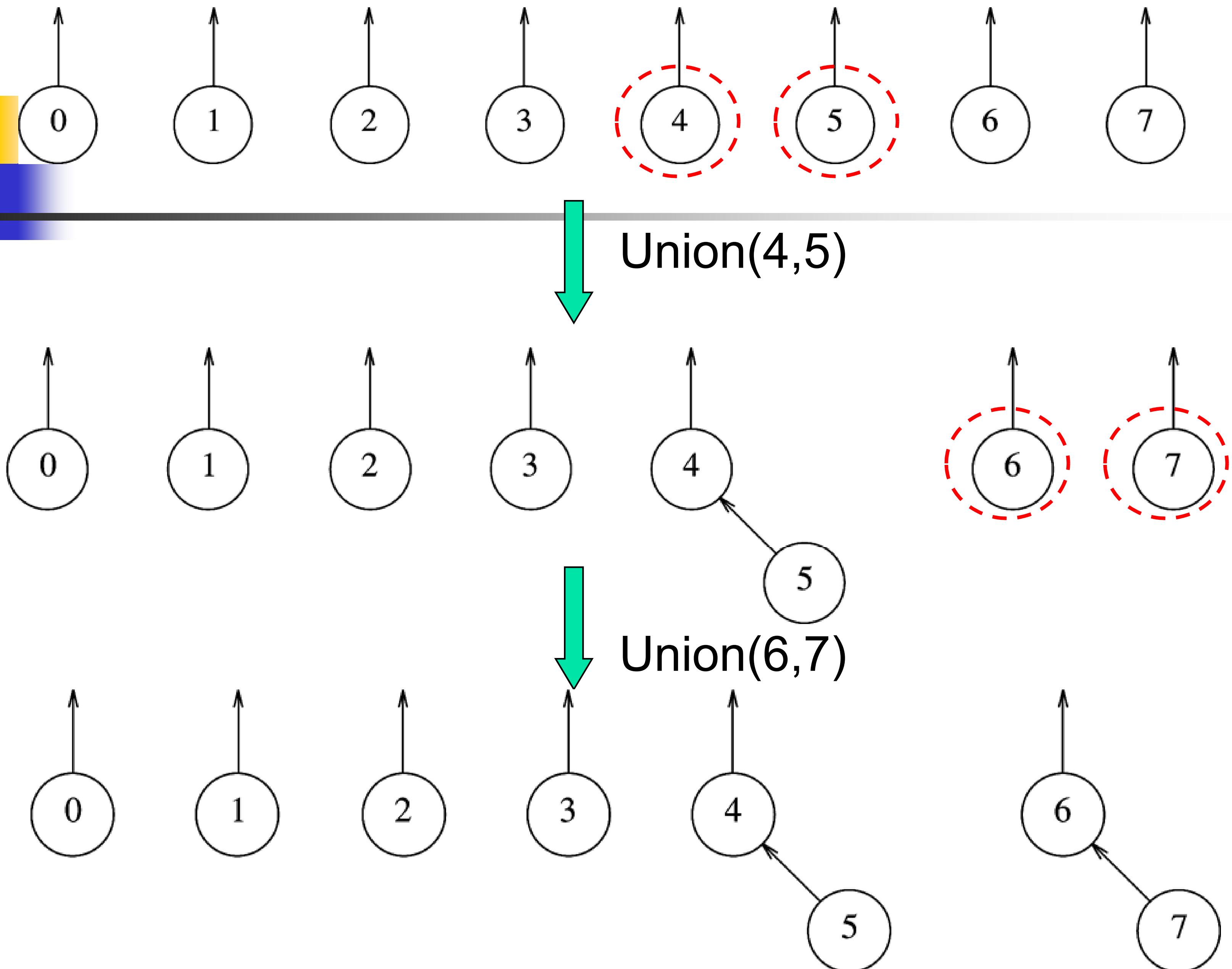
  - That is the Union-Find Data Structure!

*The Union-Find data structure for n elements is a forest of k trees, where $1 \leq k \leq n$*

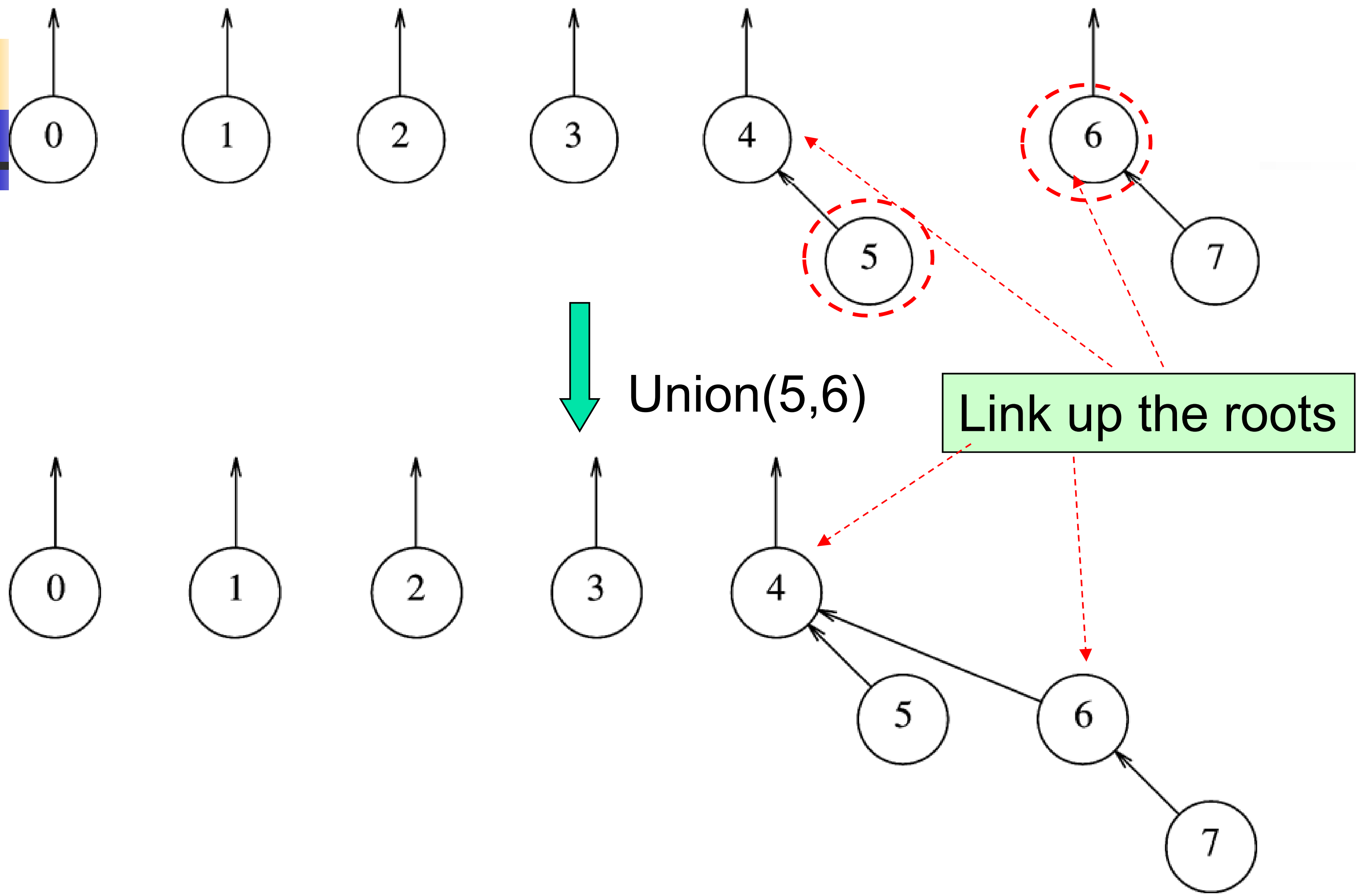
# Initialization

- Initially, each element is put in one set of its own
  - Start with *n* sets == *n* trees





0
1
2
3
4
5
6
7
Union(4,5)
0
1
2
3
4
5
6
7
Union(6,7)
0
1
2
3
4
5
6
7


0
1
2
3
4
6
5
7
Union(5,6)
Link up the roots
0
1
2
3
4
5
6
7

# The Union-Find Data Structure

- **Purpose:** To support two basic operations efficiently
  - *Find (x)*
  - *Union (x, y)*
- Input: An array of n elements
- Identify each element by its array index
  - Element label = array index

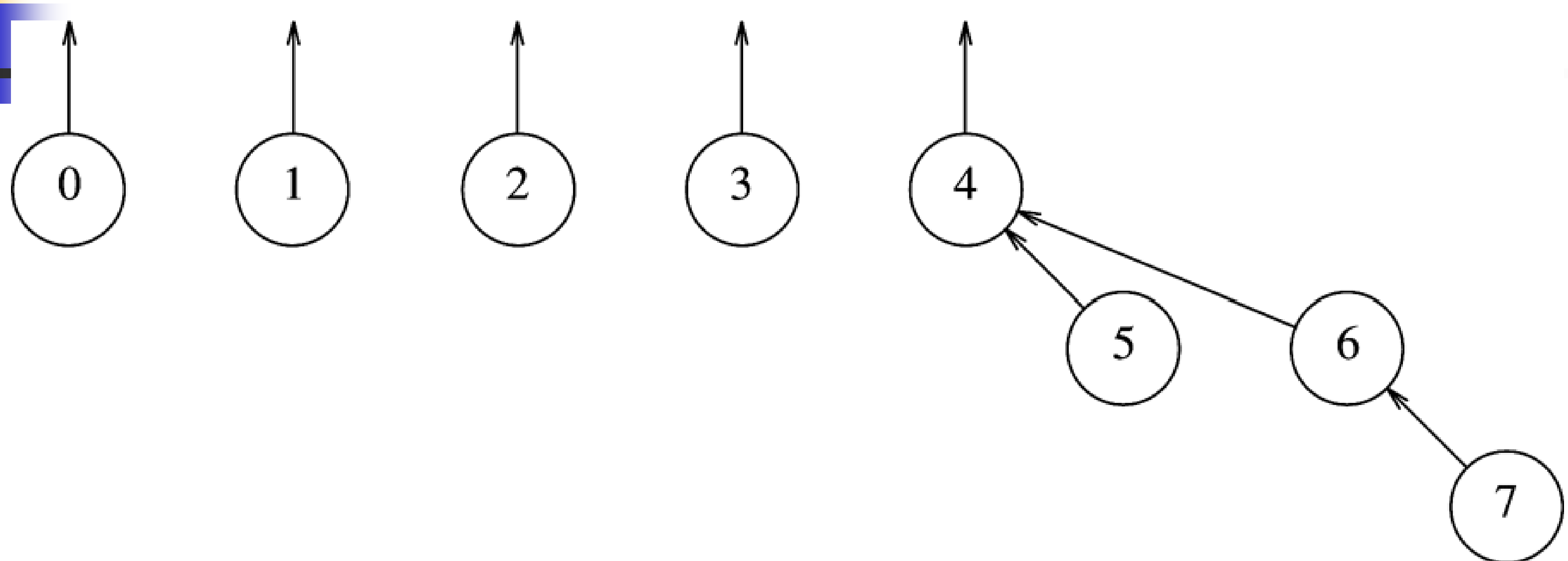
# Union-Find Data Structure

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

    int find( int x ) const;
    int find( int x );
    void unionSets( int root1, int root2 );
    void union(int x, int y);
  private:
    vector<int> s;
};
```

Note: This will always be a vector<int>, regardless of the data type of your elements.     WHY?

# Union-Find D/S: Implementation




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| −1 | −1 | −1 | −1 | −1 | 4 | 4 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Entry s[i] points to $i^{th}$ parent
- -1 means root

This is WHY vector<int>

# Union-Find: “Simple Version”

```
/**
 * 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;
}
```

“Simple Find” implementation

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

Union performed arbitrarily

```
/**
 * 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;
}
```

This could also be:
s[root1] = root2
(both are valid)

a & b could be arbitrary elements (need not be roots)

```
void DisjSets::union(int a, int b)
{
  unionSets( find(a), find(b) );
}
```

# Analysis of the simple version

- Each unionSets() takes only O(1) in the worst case
- Each Find() could take O(n) time
  - ➔ Each Union() could also take O(n) time

- Therefore, m operations, where m>>n, would take O(m n) in the worst-case

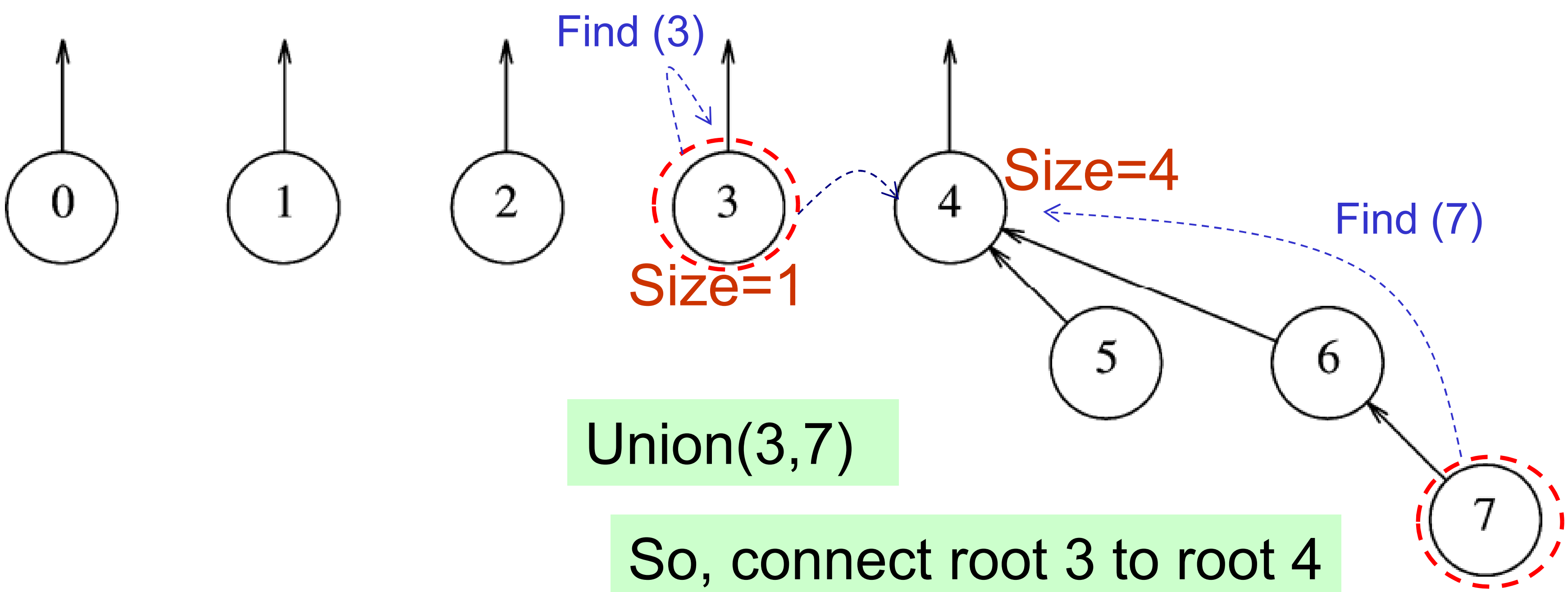
Pretty bad!

# Smarter Union Algorithms

- Problem with the arbitrary root attachment strategy in the simple approach is that:
  - The tree, in the worst-case, could just grow along one long (O(n)) path
- <u>Idea:</u> Prevent formation of such long chains
  - => Enforce Union() to happen in a “balanced” way

# Heuristic: *Union-By-Size*

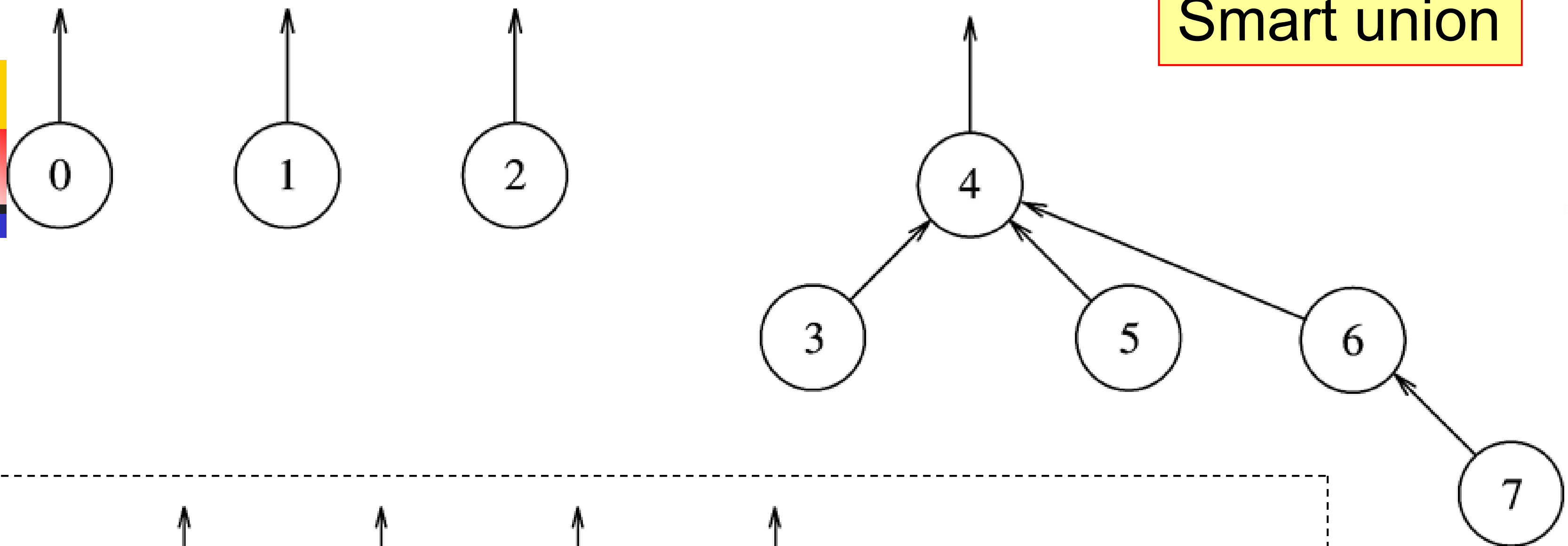
- *Attach the root of the "smaller" tree to the root of the "larger" tree*



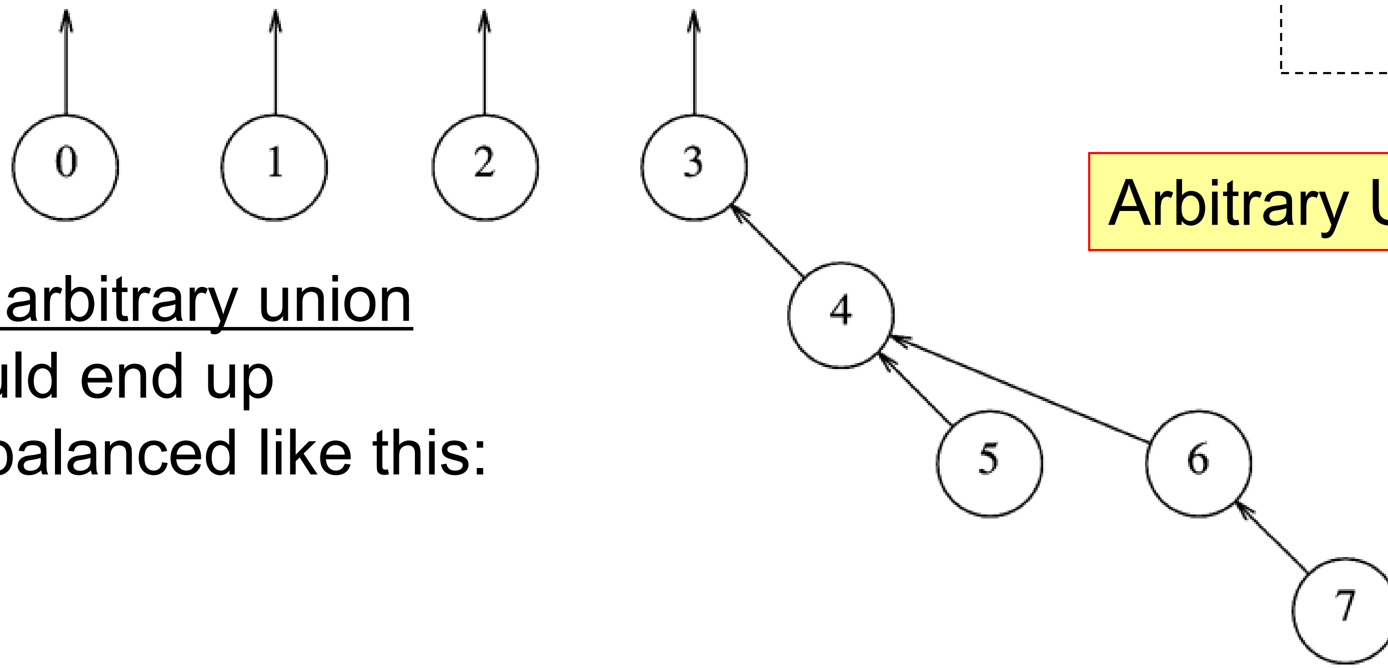

Union(3,7)

So, connect root 3 to root 4

## Union-By-Size:




Smart union



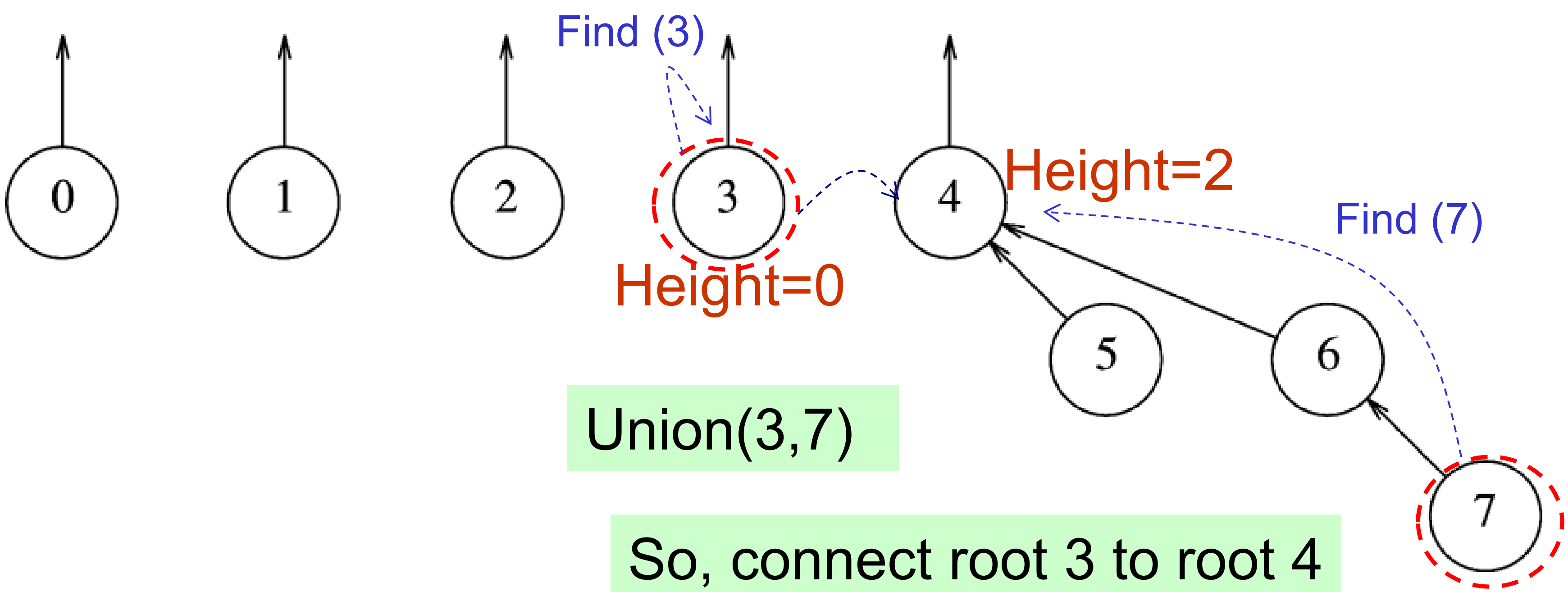

Arbitrary Union

An arbitrary union could end up unbalanced like this:

Also known as "*Union-By-Rank*"

# Another Heuristic: *Union-By-Height*

- *Attach the root of the "shallower" tree to the root of the "deeper" tree*

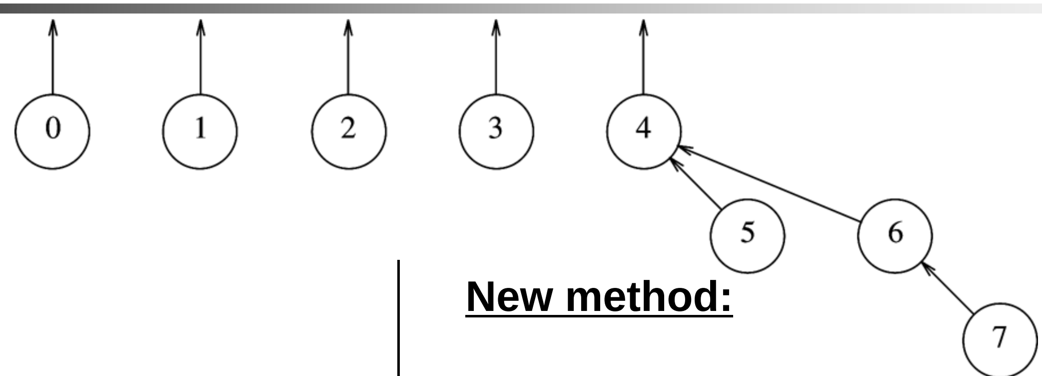



Union(3,7)

So, connect root 3 to root 4

Let us assume union-by-rank first

# How to implement smart union?




**Old method:**

| -1 | -1 | -1 | -1 | -1 | 4 | 4 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

But where will you keep track of the heights?

- s[i] = parent of i
- S[i] = -1, means root

**New method:**

What is the problem if you store the height value directly?

| -1 | -1 | -1 | -1 | **-3** | 4 | 4 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- instead of roots storing -1, let them store a value that is equal to: -1-(tree height)

# New code for union by rank?

```
void DisjSets::unionSets(int root1,int root2) {

    // first compare heights

    // link up shorter tree as child of taller tree
    // if equal height, make arbitrary choice

    // then increment height of new merged tree if height
    has changed – will happen if merging two equal
    height trees

}
```

# New code for union by rank?

```
void DisjSets::unionSets(int root1,int root2) {
        assert(s[root1]<0);
        assert(s[root2]<0);
        if(s[root1]<s[root2])        s[root2]=root1;
        if(s[root2]<s[root1])        s[root1]=root2;
        if(s[root1]==s[root2]) {
          s[root1]=root2;
          s[root2]--;
}
```

Note: All nodes, except root, store parent id.
The root stores a value = negative(height) -1

# Code for Union-By-Rank

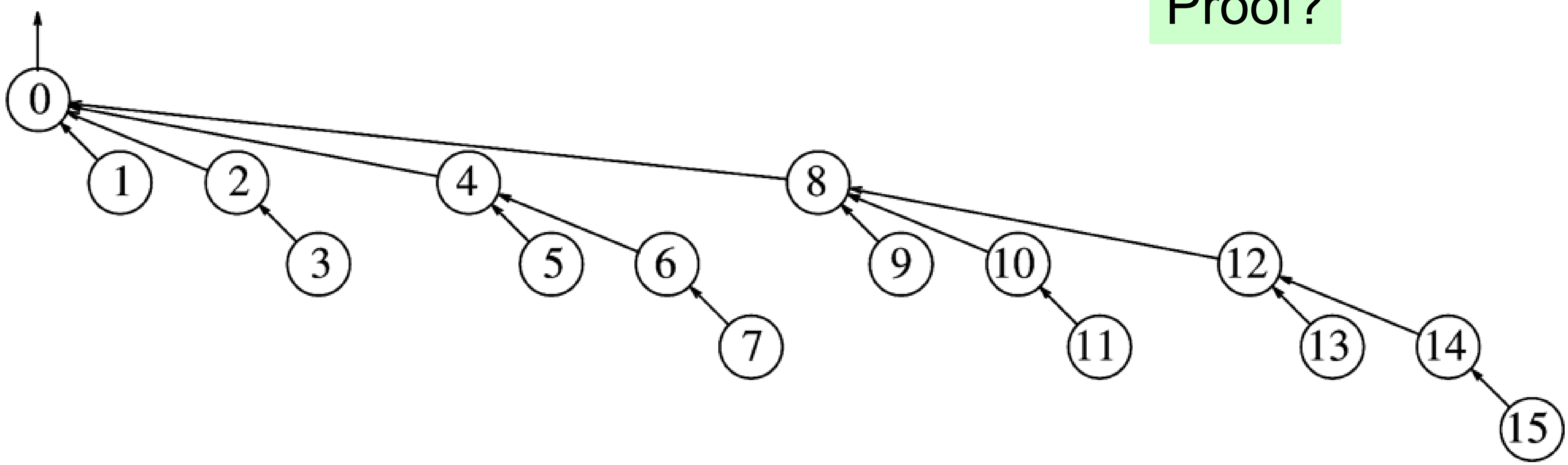
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * root1 is the root of set 1.
 * root2 is the root of set 2.
 */
void DisjSets::unionSets( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
       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

Similar code for union-by-size

# How Good Are These Two Smart Union Heuristics?

- Worst-case tree

Proof?




Maximum depth restricted to O(log n)

# Analysis: Smart Union Heuristics

- For smart union (by rank or by size):
  - Find() takes O(log n);
    - ==> union() takes O(log n);
  - unionSets() takes O(1) time
- For m operations: O(m log n) run-time

- Can it be better?
  - What is still causing the (log n) factor is the distance of the root from the nodes
  - Idea: Get the nodes as close as possible to the root

Path Compression!

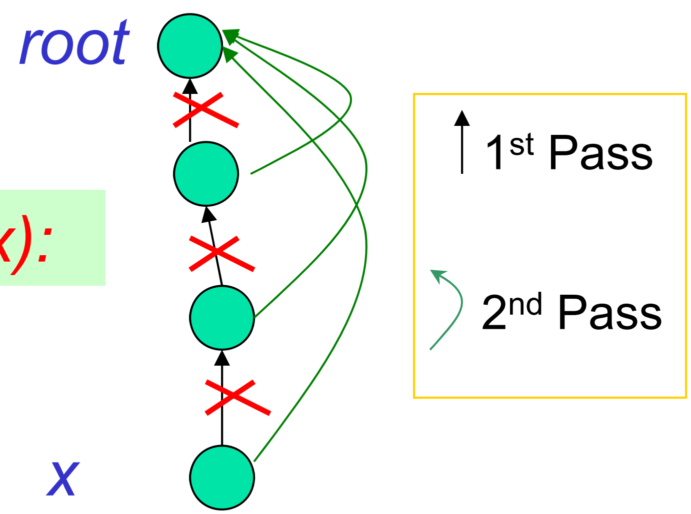
# *Path Compression* Heuristic

- During find(x) operation:
  - Update all the nodes along the path from x to the root point directly to the root
  - A two-pass algorithm

How will this help?

Any future calls to find on x or its ancestors will return in constant time!

*find(x):*



# New code for find() using path compression?

```
void DisjSets::find(int x) {

?

}
```

# New code for find() using path compression?

```
int DisjSets::find(int x) {

    // if x is root, then just return x
    if(s[x]<0) return x;

    // otherwise simply call find recursively, but..
    //      make sure you store the return value (root index)
    //              to update s[x], for path compression

    return s[x]=find(s[x]);
}
```

# Path Compression: Code

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

It can be proven that path compression alone ensures that find(x) can be achieved in O(log n)

Spot the difference from old find() code!

# Union-by-Rank & Path-Compression: Code

## Init()

```
/**
 * 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;
}
```

## Union()

```
void DisjSets::union(int a, int b)
{
  unionSets( find(a), find(b) );
}
```

## unionSets()

**Smart union**

```
/**
 * 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
    }
}
```

## Find()

**Smart find**

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

Amortized complexity for m operations:
O(m Inv. Ackerman (m,n)) = O(m log*n)

# Heuristics & their Gains

| | Worst-case run-time for m operations |
|---|---|
| Arbitrary Union, Simple Find | O(m n) |
| Union-by-size, Simple Find | O(m log n) |
| Union-by-rank, Simple Find | O(m log n) |
| Arbitrary Union, Path compression Find | O(m log n) |
| **Union-by-rank, Path compression Find** | **O(m Inv.Ackermann(m,n)) = O(m log*n)** |

Extremely slow Growing function

# What is Inverse Ackermann Function?

- $A(1,j) = 2^j$ for $j>=1$
- $A(i,1)=A(i-1,2)$ for $i>=2$
- $A(i,j)= A(i-1,A(i,j-1))$ for $i,j>=2$

- InvAck(m,n) = min{i | A(i,floor(m/n))>log N}

- InvAck(m,n) = O(log*n) (pronounced "log star n")

Even Slower!

A very slow function

# How Slow is Inverse Ackermann Function?

- What is log*n?

A very slow function

- log*n = log log log log ……. n
  - How many times we have to repeatedly take log on n to make the value to 1?
- log*65536=4, but log*$2^{65536}$=5

# Some Applications

# A Naïve Algorithm for Equivalence Class Computation

1. Initially, put each element in a set of its own

   i.e., $EqClass_a = \{a\}$, for every $a \in S$

2. FOR EACH element pair (a,b):
   1. Check [a R b == true]
   2. IF a R b THEN
      1. $EqClass_a = Find(a)$
      2. $EqClass_b = Find(b)$
      3. $EqClass_{ab} = EqClass_a \cup EqClass_b$

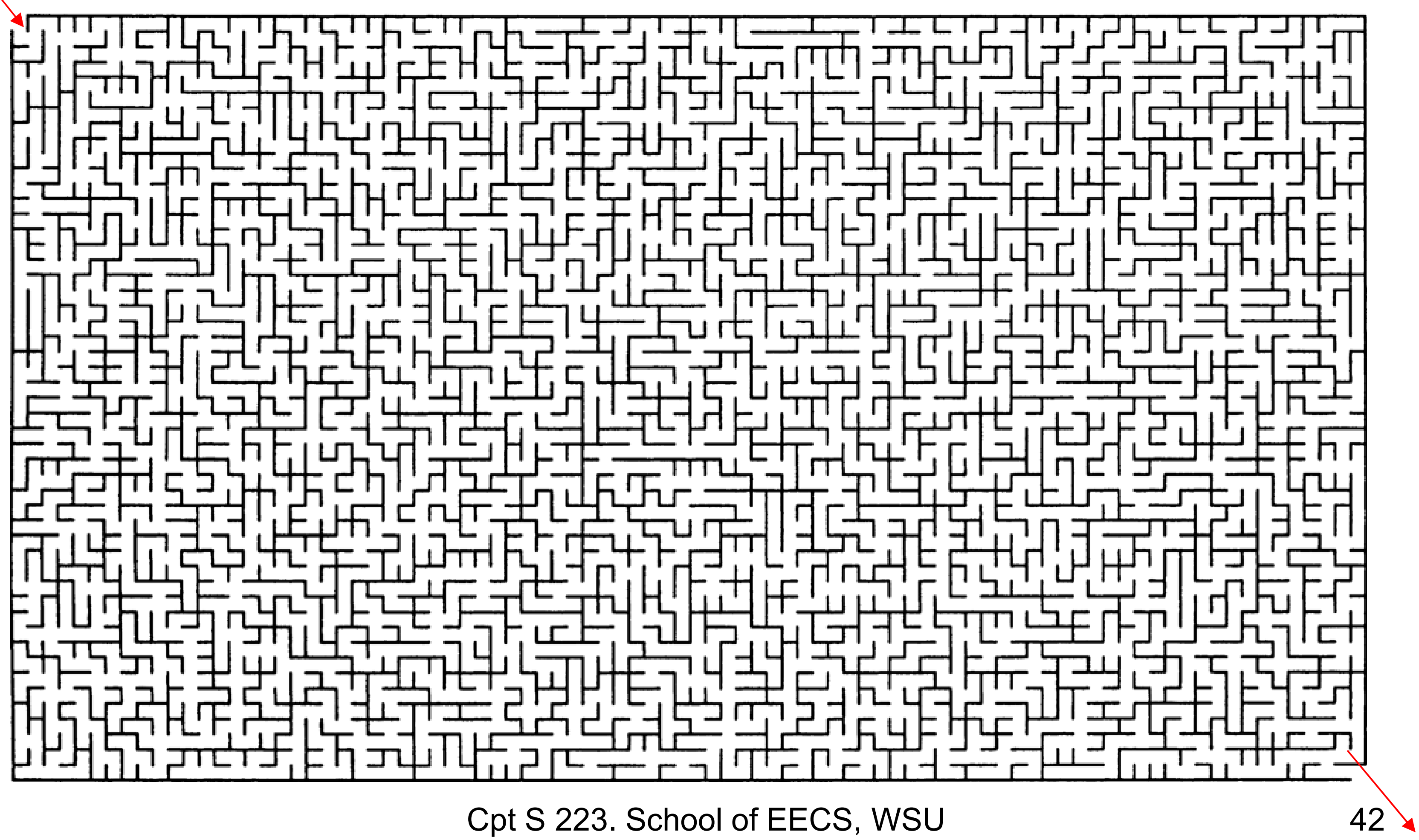
$\Theta(n^2)$ iterations

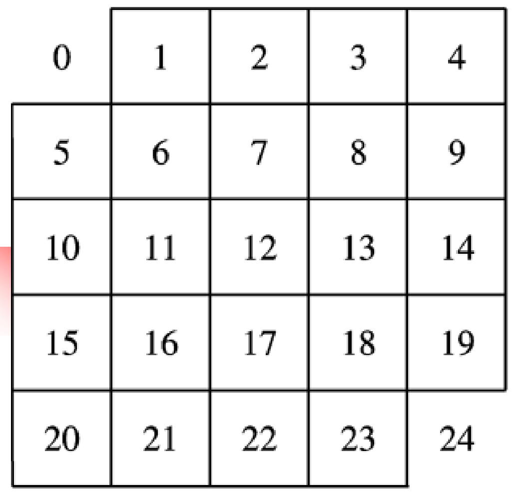
O(log *n) amortized

Run-time using union-find: $O(n^2 \log^* n)$

Better solutions using other data structures/techniques could exist depending on the application

# An Application: Maze



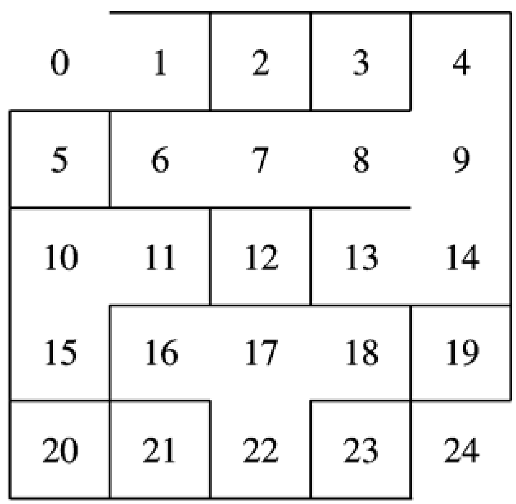



## Strategy:

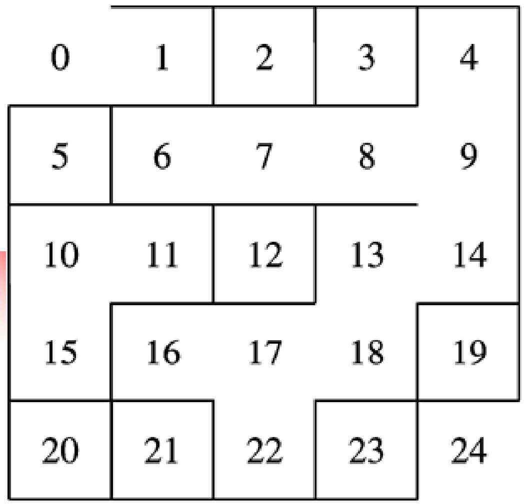
- As you find cells that are connected, collapse them into equivalent set
- If no more collapses are possible, examine if the Entrance cell and the Exit cell are in the same set
  - If so => we have a solution
  - O/w => no solutions exists

{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}

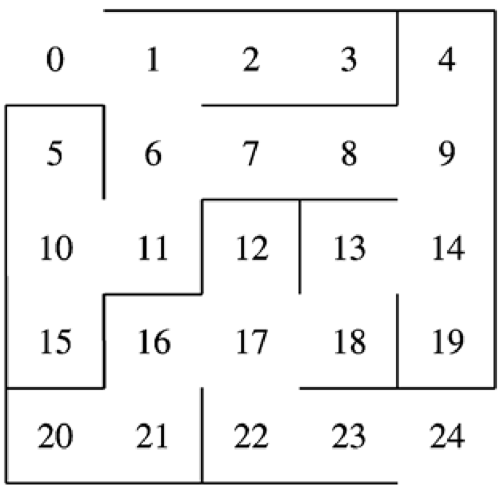



## Strategy:

- As you find cells that are connected, collapse them into equivalent set
- If no more collapses are possible, examine if the Entrance cell and the Exit cell are in the same set
  - If so => we have a solution
  - O/w => no solutions exists

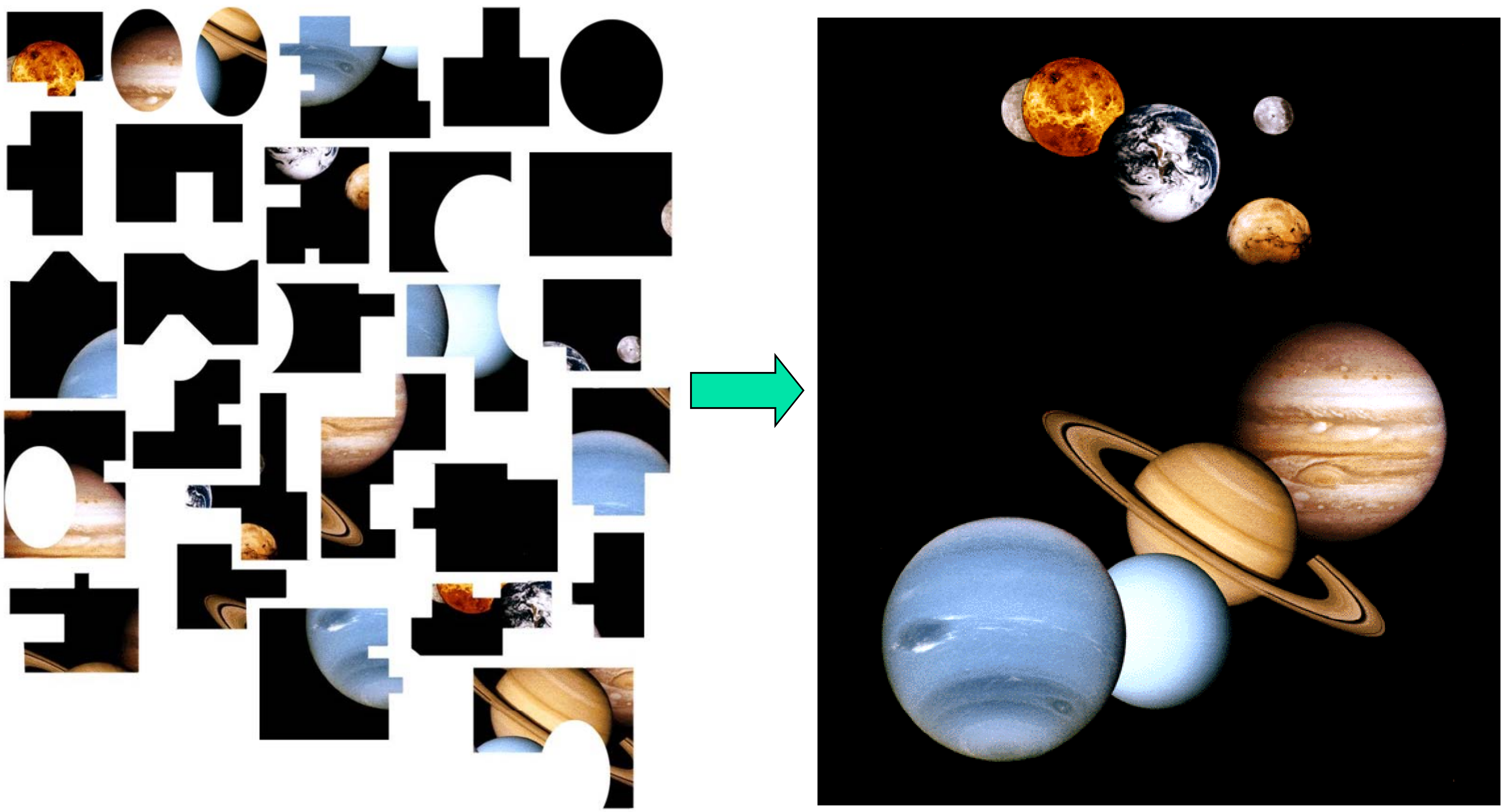
{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}

---




{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}

# Another Application: Assembling Multiple Jigsaw Puzzles at once



Merging Criterion: Visual & Geometric Alignment

Picture Source: *http://ssed.gsfc.nasa.gov/lepedu/jigsaw.html*

# Summary

- Union Find data structure
  - Simple & elegant
  - Complicated analysis
- Great for disjoint set operations
  - Union & Find
  - In general, great for applications with a need for "clustering"