WASHINGTON STATE UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Disjoint Sets (Union-Find)

Union-find algorithm

- Purpose:
  - To manipulate disjoint sets (i.e., sets that do not 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?

WSU

Union-find: motivation

- Given a set S of n elements, [a1…an], compute all its equivalent classes
- Example applications:
  - Electrical cable/internet connectivity network
  - Cities connected by roads
  - Cities belonging to the same country

WSU

Equivalent relations

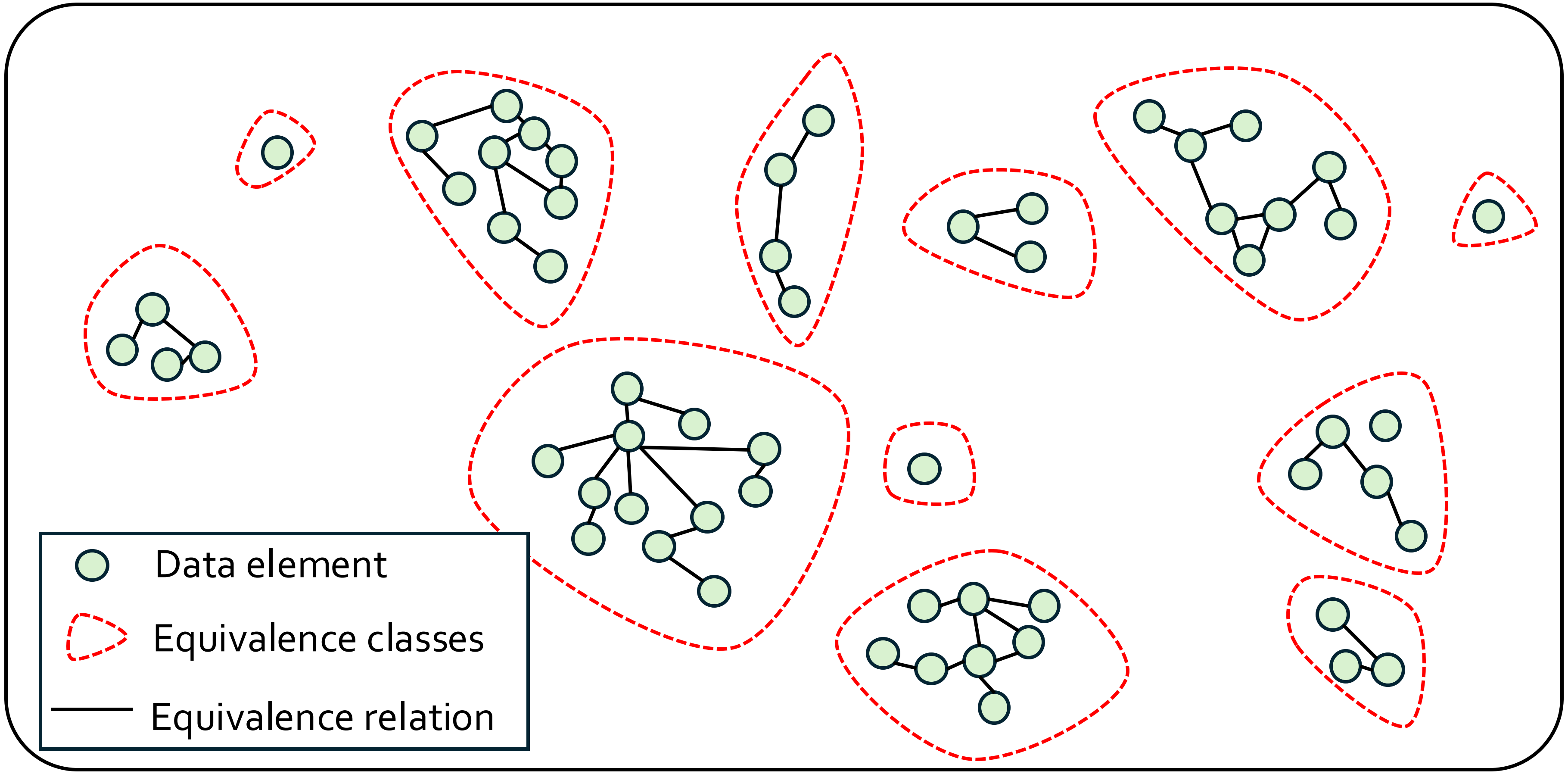
- An equivalence relation R is defined on a set S, if for every pair of elements (a,b) in S,
  - a R b is either false or true
- a R b is true iff:
  - (Reflexive) a R a, for each element a in S
  - (Symmetric) a R b if and only if b R a
  - (Transitive) a R b and b R c implies a R c
- The equivalence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

Equivalence classes: properties

- 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

WSU

Equivalence classes: example



WSU

Disjoint set operations

- To identify all equivalence classes:

1. Initially, put each element in a set of its own
2. Permit only two types of operations:
   - find(x): Returns the current equivalence class of x
   - union(x, y): Merges the equivalence classes corresponding to elements x and y (assuming x and y are related by the eq.rel.)

union(x, y) is equivalent to:
unionSets(find(x), find(y))

union() calls find()

Steps in union(x, y):

1. EqClassx = Find (x)
2. EqClassy = Find (y)
3. EqClassxy = EqClassx U EqClassy

U: union of two sets

Compute equivalence classes

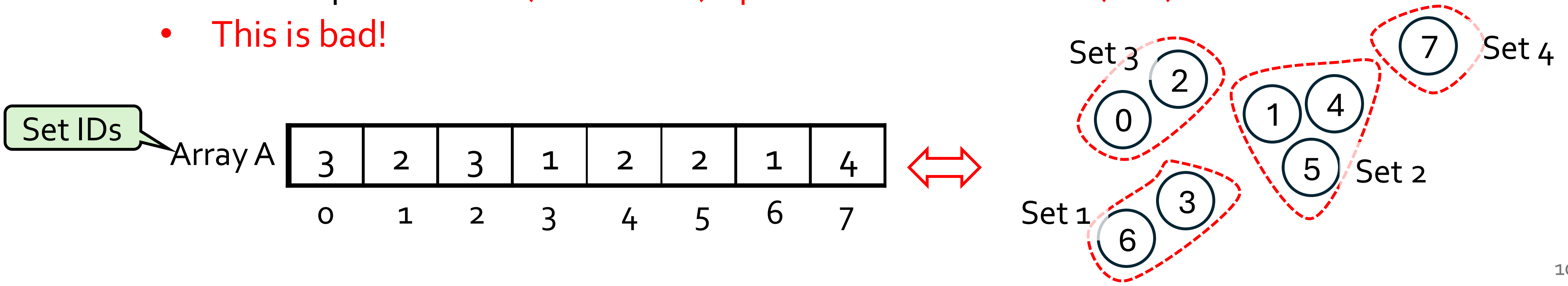
- Initially, put each element in a set of its own
  - i.e., EqClassa = {a}, for every a $\in$ S
- FOR EACH element pair (a,b):
  1. Check [a R b == true]
  2. IF a R b THEN
     1) EqClassa = Find(a)
     2) EqClassb = Find(b)
     3) EqClassab = EqClassa U EqClassb

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

Union-find: efficient methods

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



Union-find: efficient methods

- Approach 2: using linked list
  - Keep all equivalence sets in separate linked lists: a linked list for every set ID
- Analysis:
  - union(): $O(1)$ time (assume doubly linked list)
  - find(): $O(n)$ time
  - Improvements are possible (e.g., balanced BSTs) → $O(\log(n))$
  - A sequence of m operations takes $\Omega(m \log(n))$
  - Still not good!

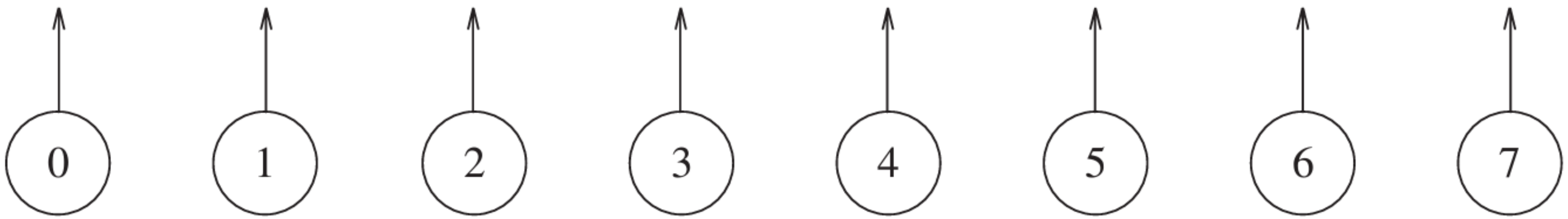
Union-find: efficient methods

- Approach 3: using a forest
- Keep all equivalence sets in separate trees: 1 tree for 1 set
- Goal: ensure (somehow) that find() and union() take very little time
  - time << O(log n)
- That is the union-find data structure (or disjoin-set data structure)

The union-find data structure for n elements is a forest of k trees, where $1 \le k \le n$

Basic union-find: initialization

- Initially, each element is put in one set of its own
  - Start with n sets == n trees
  - Each tree represents an equivalence set



Underlying implementation

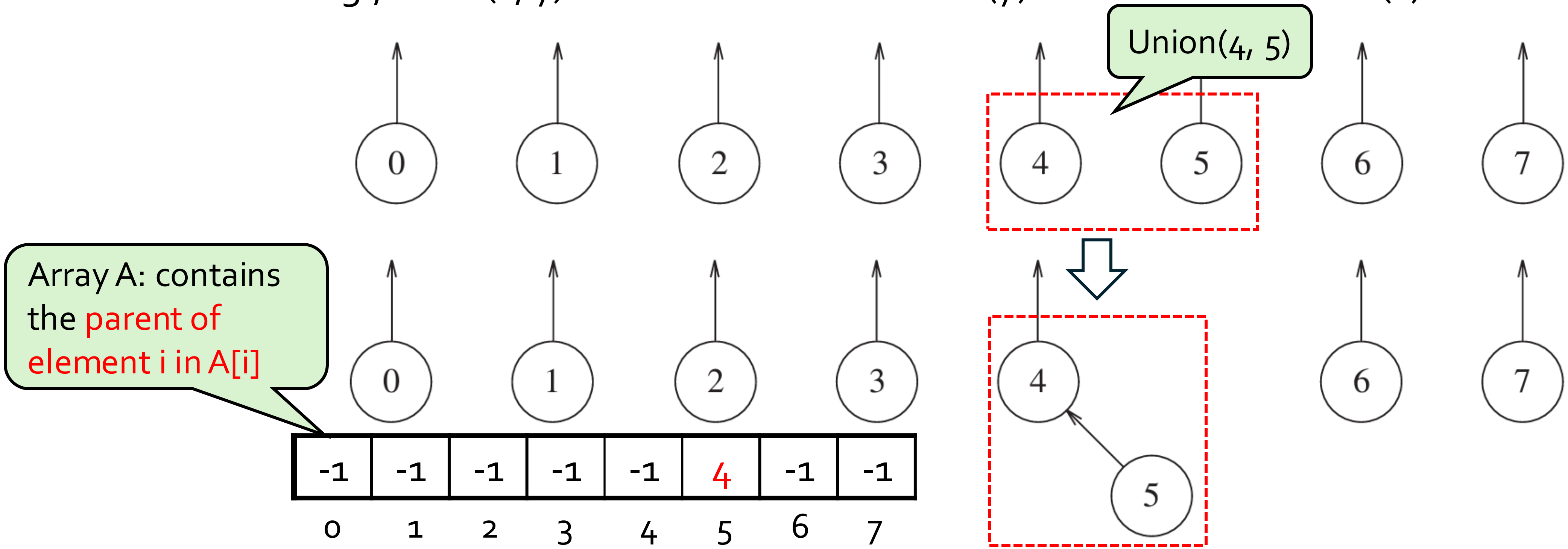
Array A: contains the parent of element i in A[i]
If i is the root (no parent): set -1

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

WSU

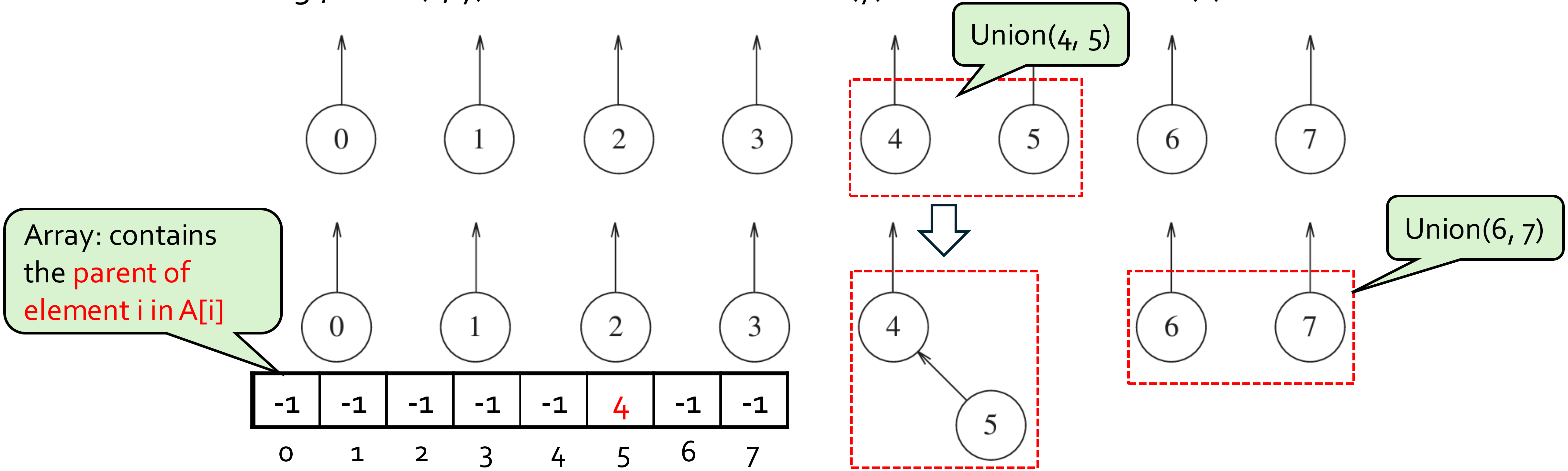
Basic union-find: union

- Union two sets by merging two trees
  - e.g., union(x, y) → make the second tree (y) a subtree of the first (x)



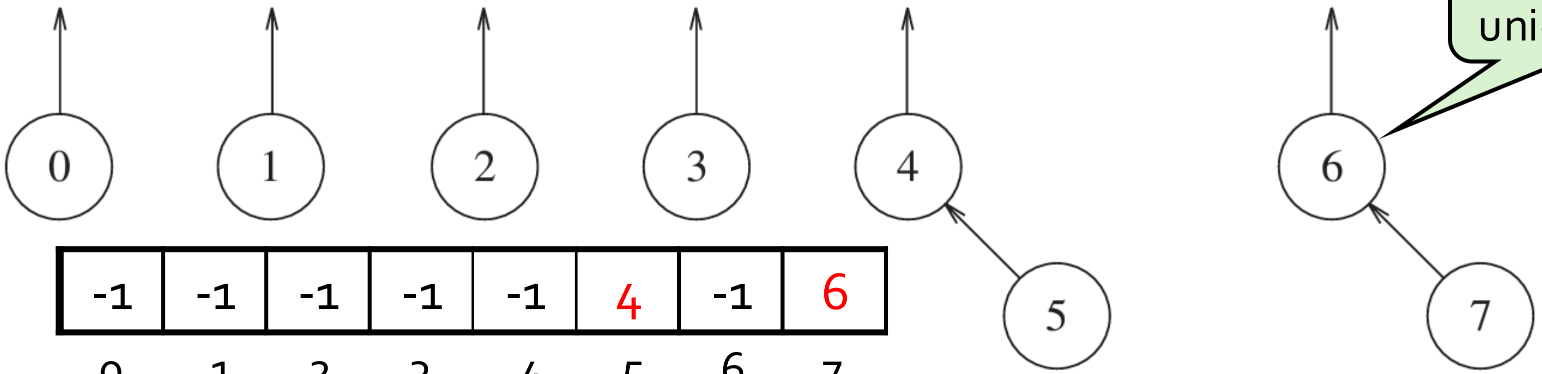
Basic union-find: union

- Union two sets by merging two trees
  - e.g., union(x, y) → make the second tree (y) a subtree of the first (x)

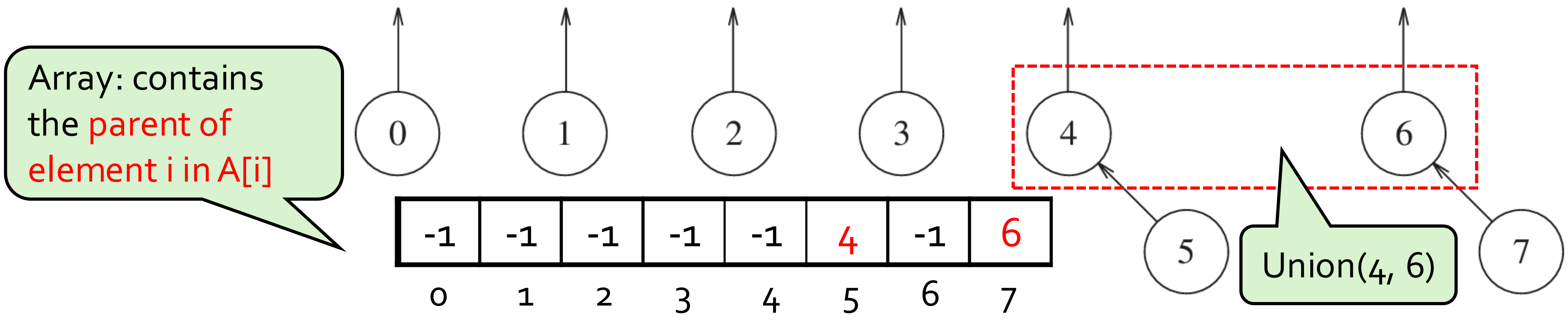


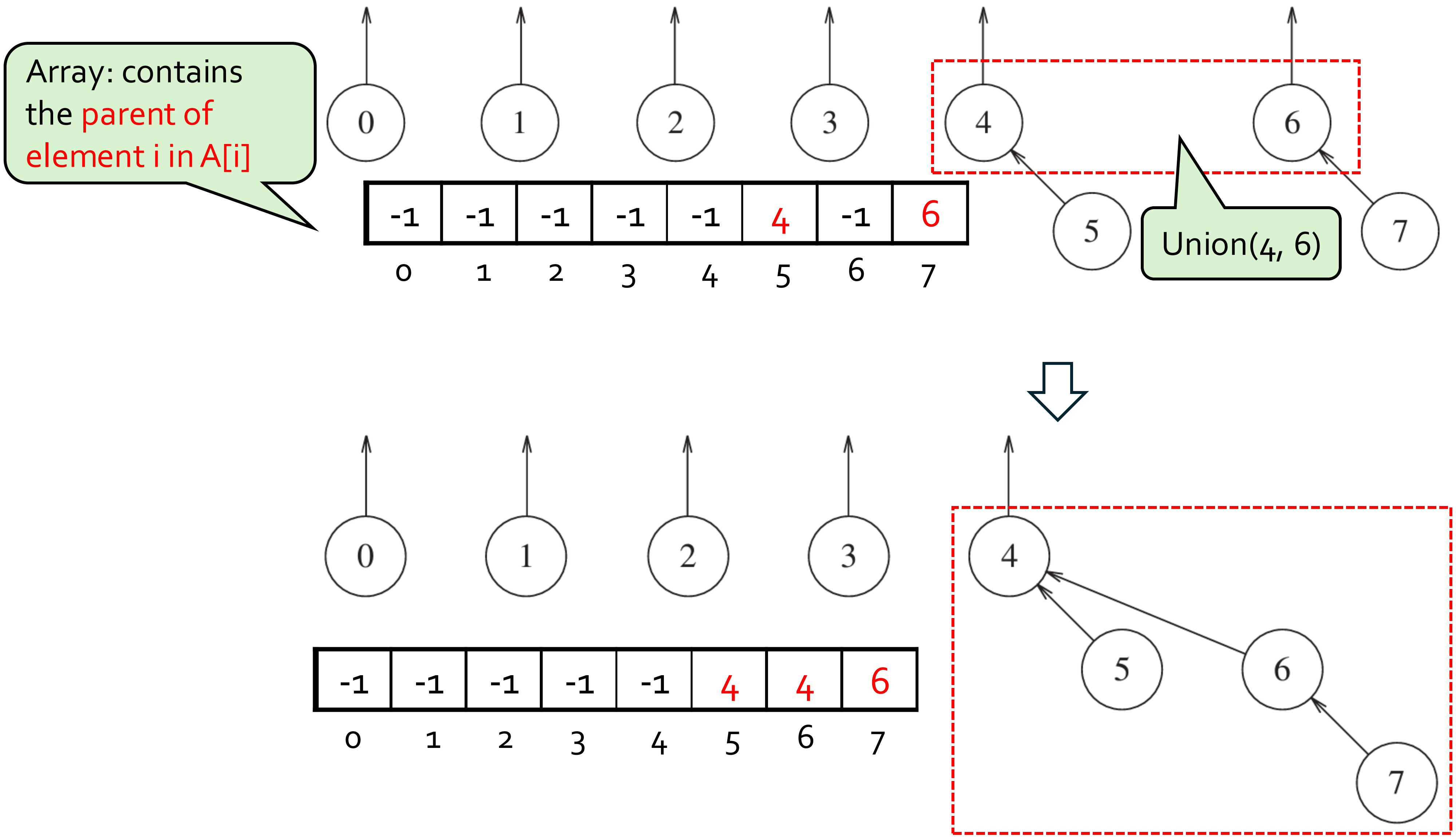
Array: contains the parent of element i in A[i]

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



After union(6, 7)


Array: contains the parent of element i in A[i]
0
1
2
3
4
6
5
7
-1 -1 -1 -1 -1 4 -1 6
0 1 2 3 4 5 6 7
Union(4, 6)


Array: contains the parent of element i in A[i]
-1 -1 -1 -1 -1 4 -1 6
0 1 2 3 4 5 6 7
Union(4, 6)
-1 -1 -1 -1 -1 4 4 6
0 1 2 3 4 5 6 7

Basic union-find: 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;
};
```

Array s: contains the parent of element i in s[i]
Initialization: i is root, setting s[i]=-1

Figure 8.6 Disjoint sets class interface

Basic union-find: implementation

```
/**
 * Construct the disjoint sets object.
 * numElements is the initial number of disjoint sets.
 */
DisjSets::DisjSets( int numElements ) : s{ numElements, - 1 }
{
}
```

Initialized with -1

Figure 8.7 Disjoint sets initialization routine

Basic union-find: 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 )
{
    s[ root2 ] = root1;
}
```

Figure 8.8 union (not the best way)

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

make the second tree a subtree of the first

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

WSU

Basic union-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 ] );
}
```

Recursive call of find

Figure 8.9 A simple disjoint sets `find` algorithm

Basic union-find: analysis

- 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
    - because union() calls find()
- Therefore, m operations, where m>>n, would take O(m n) in the worst-case
- Pretty bad!

find() is the bottleneck

Union-find: smarter version

- Problem in basic union-find: arbitrary root attachment strategy

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

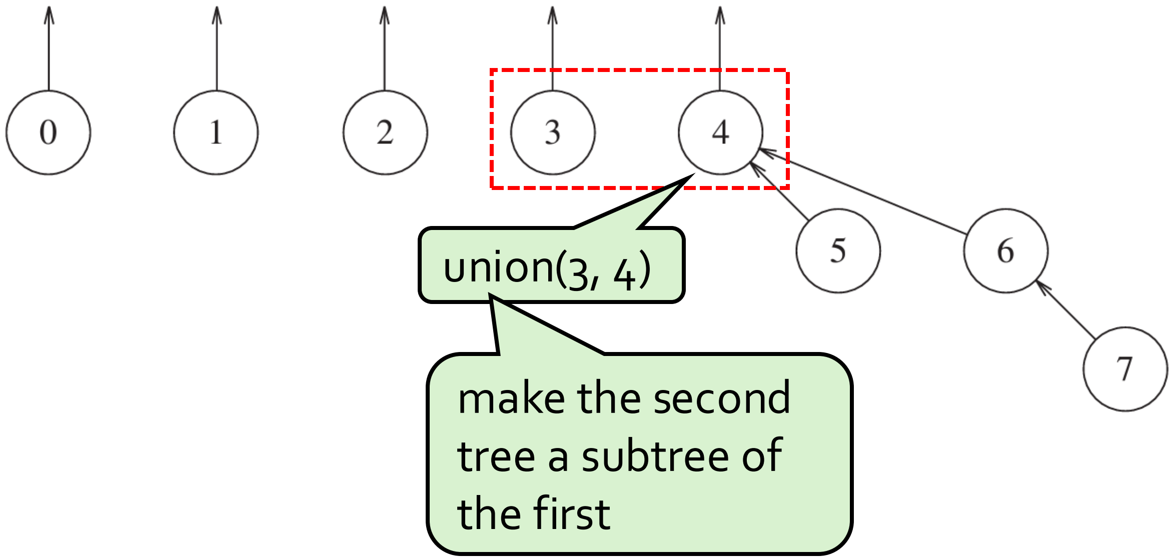
Figure 8.8 union (not the best way)

Union-find: smarter version

- Problem in basic union-find: arbitrary root attachment strategy
- The tree, in the worst-case, could just grow along one long path (O(n))

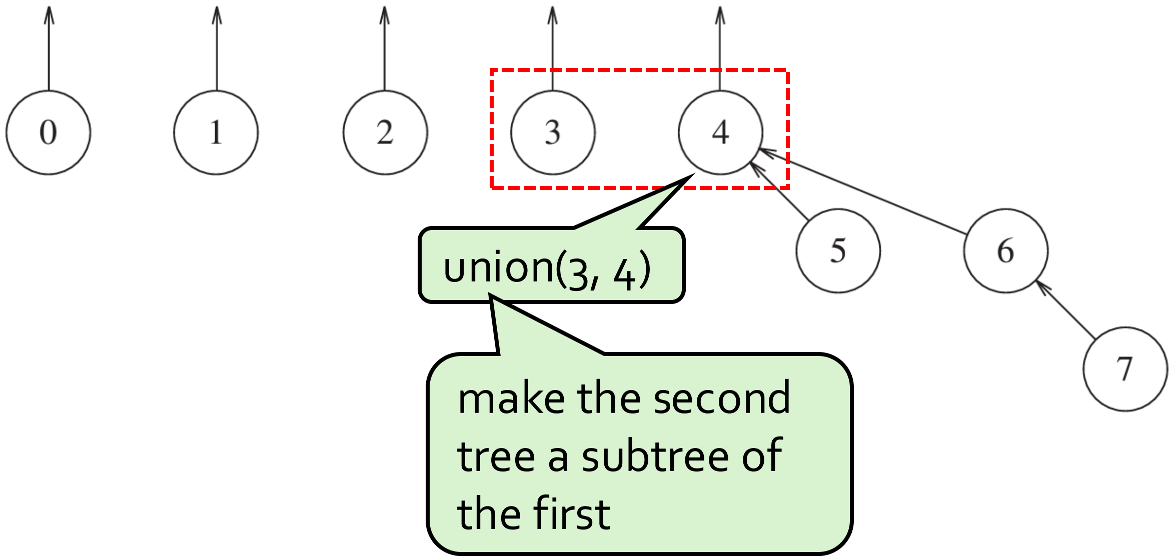
Union-find: smarter version

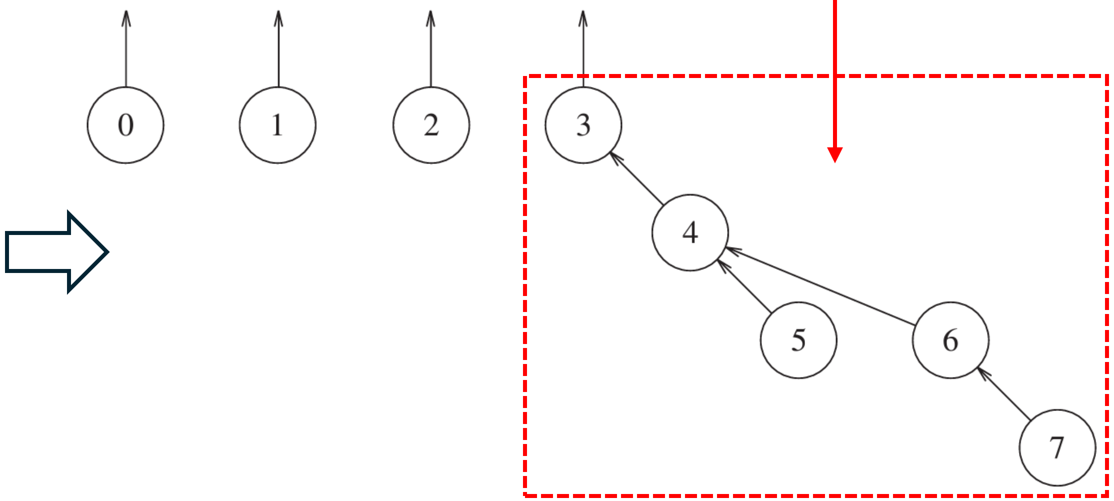
- Problem in basic union-find: arbitrary root attachment strategy
- The tree, in the worst-case, could just grow along one long path (O(n))



Union-find: smarter version

- Problem in basic union-find: arbitrary root attachment strategy
- The tree, in the worst-case, could just grow along one long path (O(n))





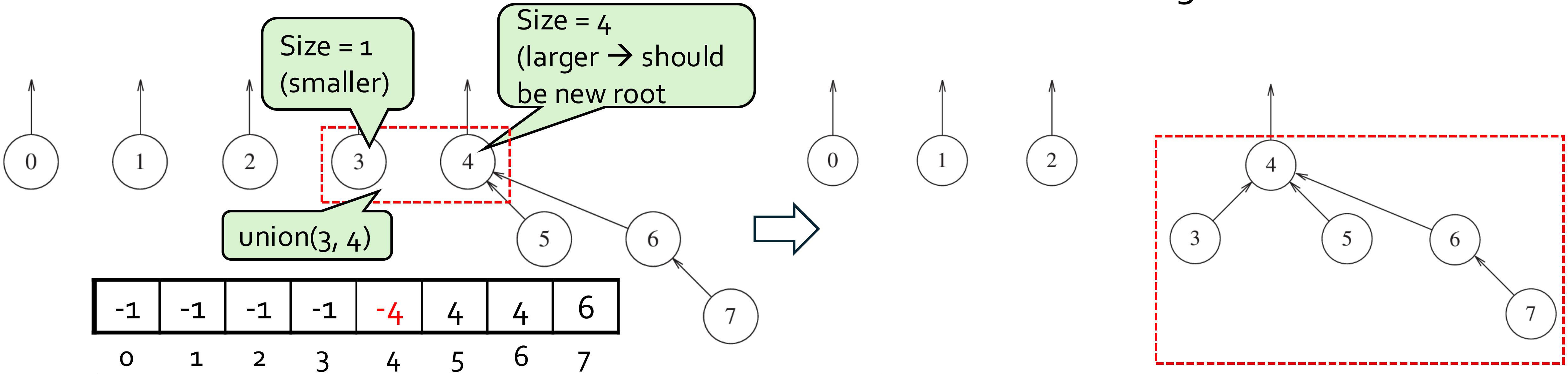
Union-find: smarter version

- Problem in basic union-find: arbitrary root attachment strategy
- The tree, in the worst-case, could just grow along one long path (O(n))
- Idea: Prevent formation of such long chains
- → Enforce union() to happen in a “balanced” way

WSU

Smart union by size

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



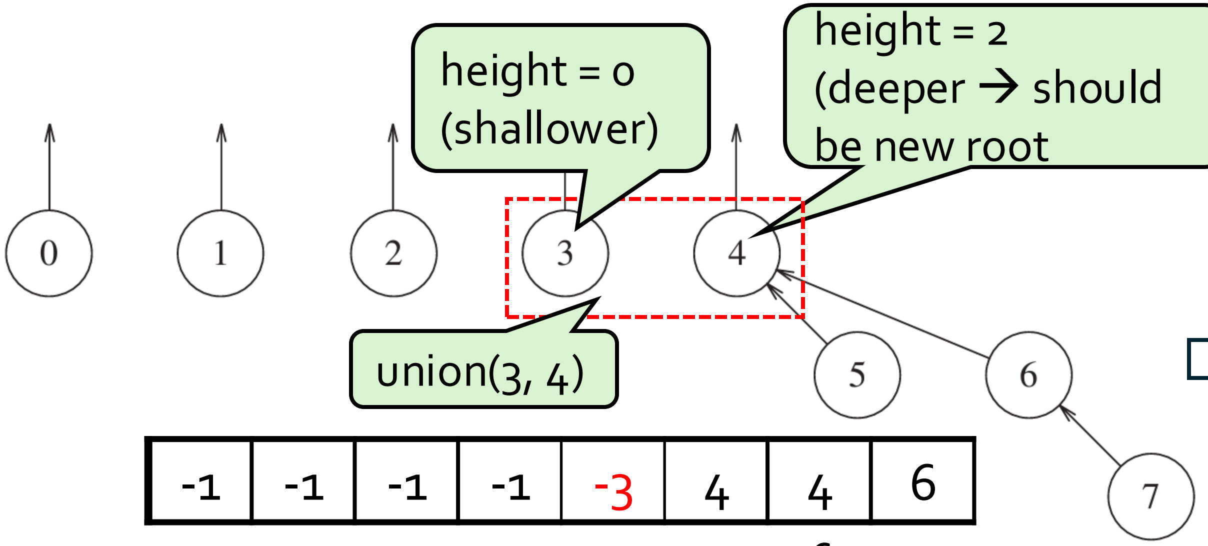
-1	-1	-1	-1	-4	4	4	6
0	1	2	3	4	5	6	7

Array:
1. If element i is root : s[i] = - (tree size)
2. If element i is not root: s[i] = index of parent of element i

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

Smart union by height

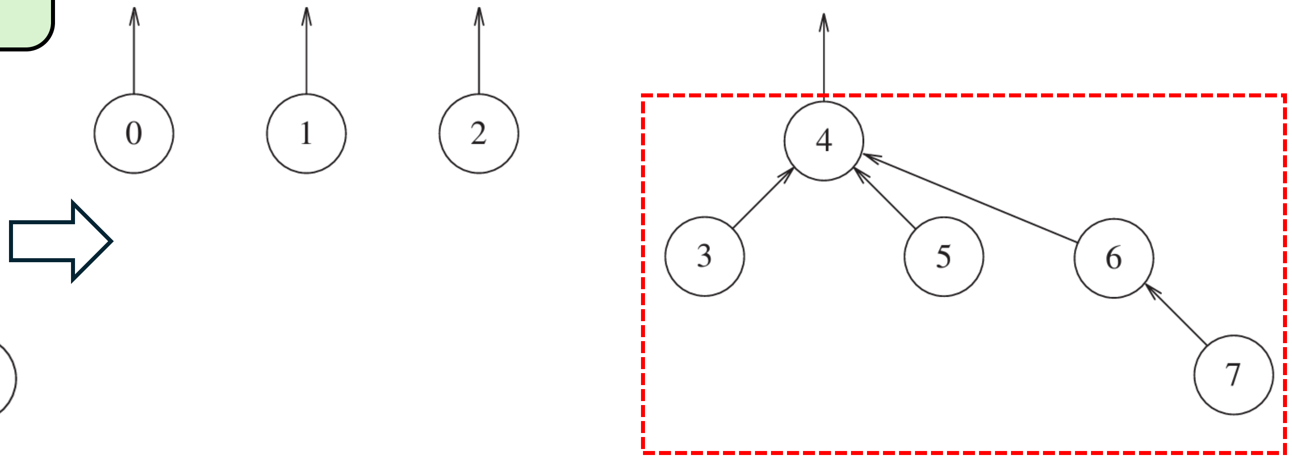
- Attach the root of the “shallower” tree to the root of the “deeper” tree



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

Array:

1. If element i is root: s[i] = - (tree height + 1)
2. If element i is not a root: s[i] = index of parent of element i



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

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

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

Similar code for union-by-size

Figure 8.14 Code for union-by-height (rank)

WSU

Smart union: analysis

- Worst-case tree

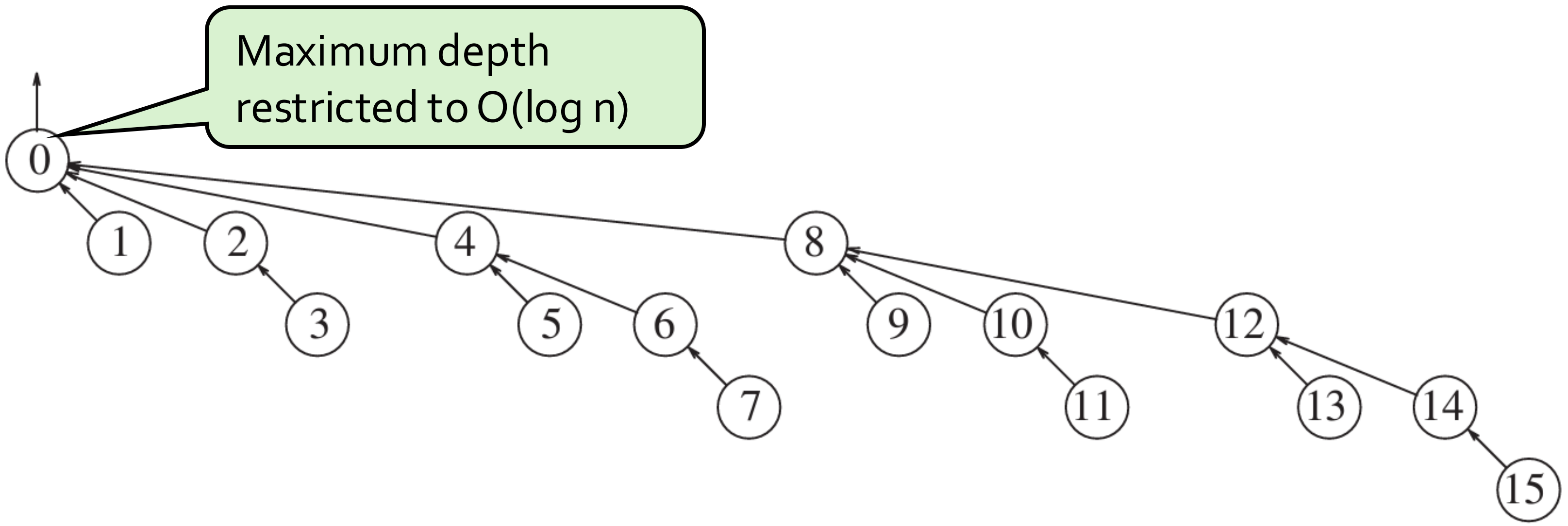


Figure 8.12 Worst-case tree for $N = 16$

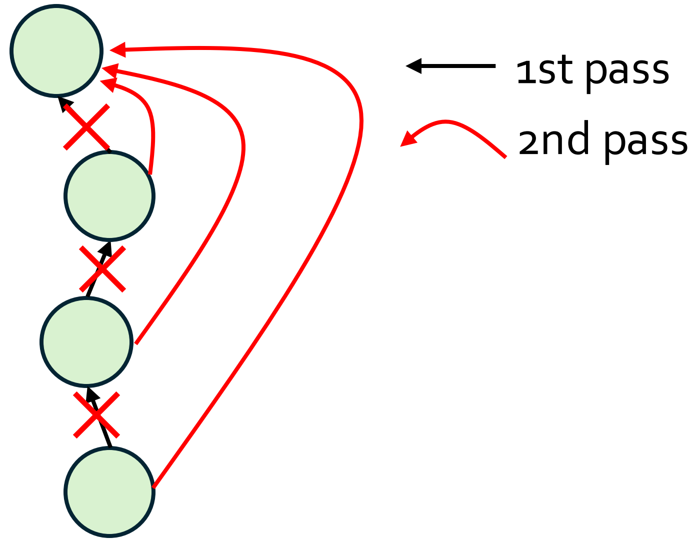
Smart union: analysis

- 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
  - Solution: path compression

Path compression

- 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

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



WSU

Find() using 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 ] );
}
```

Figure 8.16 Code for disjoint sets `find` with path compression

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

s[x] is the index of x's parent: now it is x's root

Heuristics and their gains

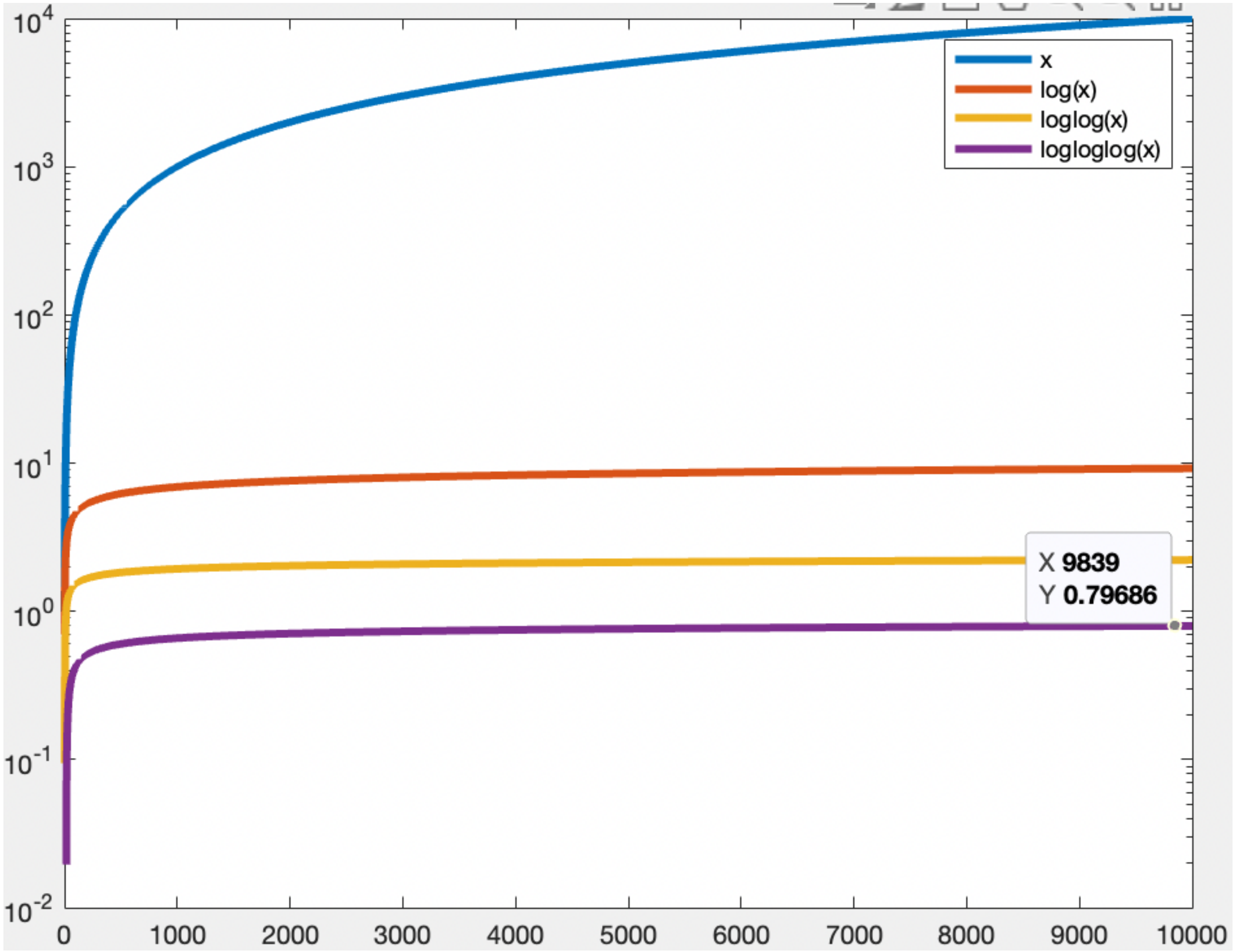
	Worst-case run-time for m operations
Arbitrary Union, Simple Find	O(m n)
Union-by-size, Simple Find	O(m log(n))
Union-by-rank, Simple Find	O(m log(n))
Arbitrary Union, Path compression Find	O(m log(n))
Union-by-rank, Find using path compression	O(m inverse_Ackermann(m, n)) = O(m log*(n))

Inverse Ackermann function

- Definition of inverse Ackermann function
  - A(1,j) = 2j 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")
- log*n = log log log log ……. N
- log*65536 = 4 v.s. log2(65536) = 16
- log*265536 = 5 v.s. log2(265536) = 18

Very slow growth rate

Inverse Ackermann function



WSU

Application: maze generation

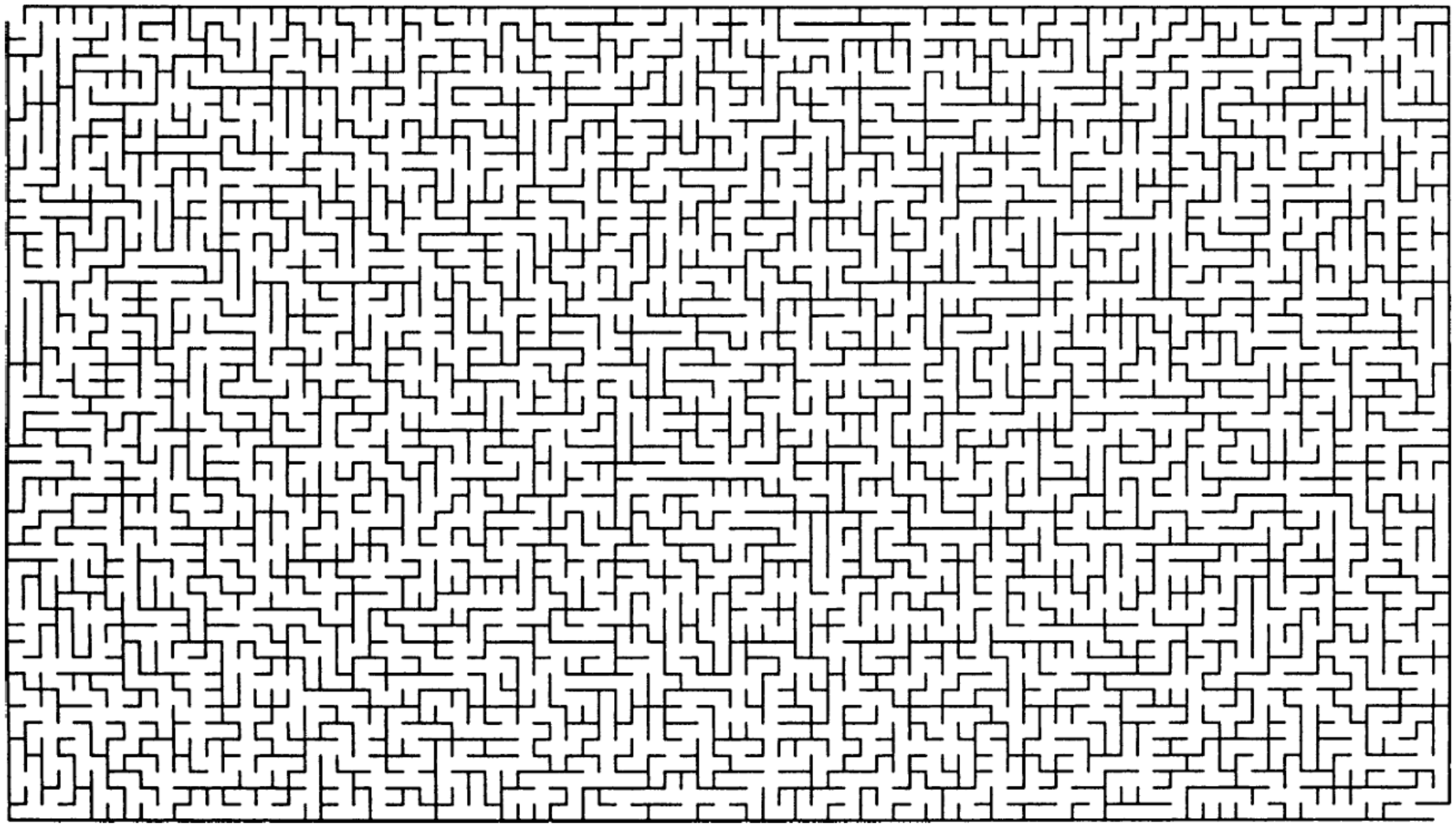
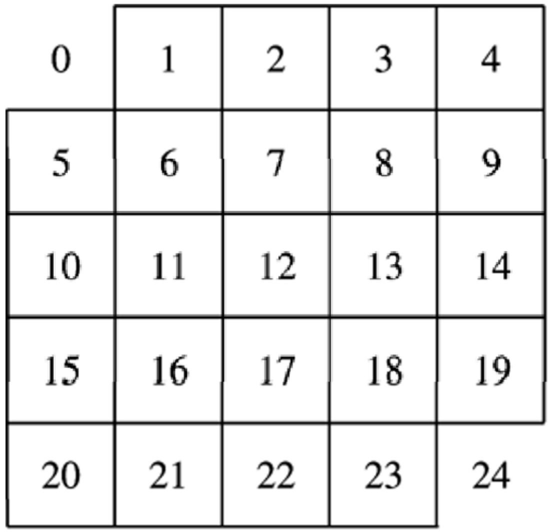


Figure 8.25 A 50-by-88 maze

Union-find algorithm



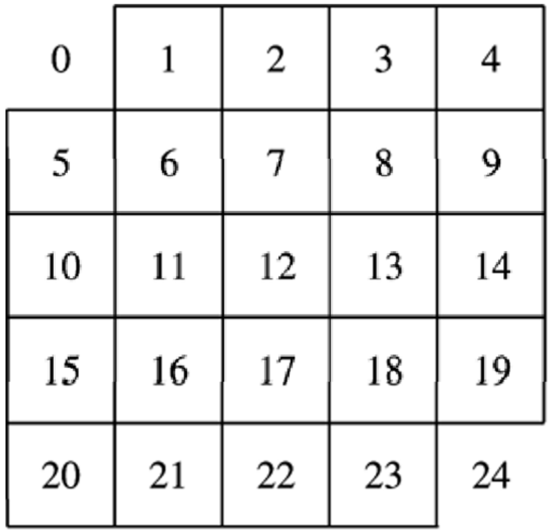
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14}
{15} {16} {17} {18} {19} {20} {21}{22} {23} {24}

Strategy:

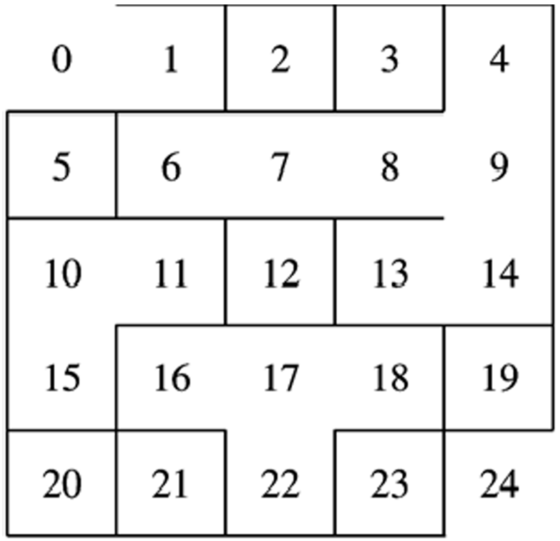
1. As you find cells that are connected, collapse them into equivalent set
2. If no more collapses are possible, examine if the Entrance cell and the Exit cell are in the same set
   - If so → we have a valid solution
   - Otherwise → no valid solutions exists

WSU

Union-find algorithm



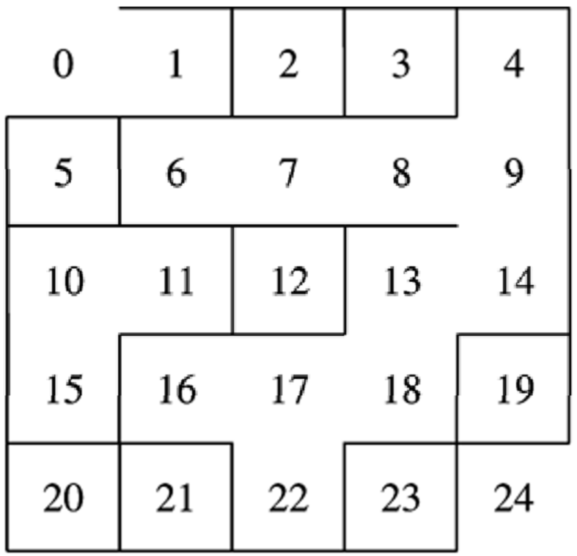
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14}
{15} {16} {17} {18} {19} {20} {21}{22} {23} {24}



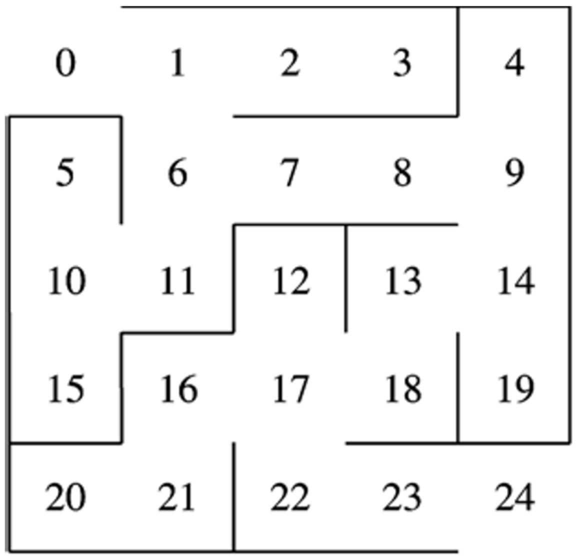
{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5}

{10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

Union-find algorithm



{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}

Union-find: 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”