THE UNIVERSITY OF TEXAS AT DALLAS

CS3345 Data Structures & Algorithm Analysis

Lecture Notes 22

Omar Hamdy
Associate Professor
Department of Computer Science

Equivalence Relations

- A relation *R* is defined on a set S if for every pair of elements $(a, b), a, b \in S$, aRb is either true or false. If aRb is true, then we say that a is related to b
- An equivalence relation is a relation *R* that satisfies three properties:
  - (Reflexive) *aRa*, for all a ∈ S
  - (Symmetric) *aRb* if and only if *bRa*
  - (Transitive) *aRb* and *bRc* implies that *aRc*
- Is ≥ an equivalence relationship?
- Equivalence relationship examples include electrical connectivity, relating cities by the country, or connecting cities through bi-directional roads.

Dynamic Equivalence

- We use ~ to indicate an equivalence relationship between a, b (a ~ b)
- It is easy then to store all the explicit equivalence relationships (a ~ b)
- The challenge is in the transitive property, where there could be additional implicit relationships, which we need to infer quickly.
- Example: assume the set {a1, a2, a3, a4, a5}, and the equivalence relationships (a1~a2), (a3~a4) and (a5~a1). What are other implicit equivalence relationships?
  - (a2~a5)
- What if we also have (a4~a2)?
  - All elements will have equivalence relationships with each other.

The Equivalence Class

- We define the equivalence class of an element a ∈ S as the subset of S that contains all the elements that are related to a.
- Notice that every element ∈ S appears in exactly one equivalence class (why?)
- Therefore, to decide if a ~ b, we only need to check if they both belong to the same equivalence class.
- For the initial collection N, we form the equivalence classes with each set containing one element only. (Is that true?)
  - Yes, because it represents the reflexive relationship, which is always true.
  - Each set contains a different element, and hence $S_i \cap S_j = \emptyset$, and hence sets are **disjoint**
- Only two operations are allowed: find a specific equivalence class and add a relationship to an equivalence class.

union/find Algorithm

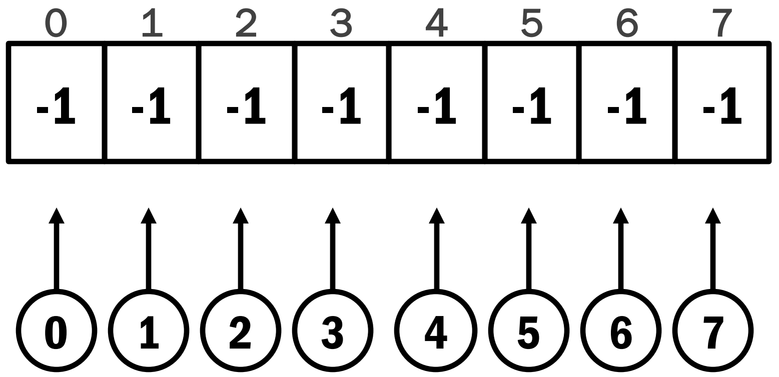
- find: this operation returns the name of the equivalence class set that contains a given element.
- union: this operation is used to add a new a ~ b equivalence relationship:
  - First, we find if a and b are in two different sets.
  - If yes, then union operation merges the two sets of a and b into a new equivalence class.
  - The new $S_k = S_i \cup S_j$
  - Note, by destroying the two sets after the union preserves the disjoint property
- The union/find algorithm is what causes the dynamic equivalence relationships.

union/find Algorithm

- Since the actual value of each element is irrelevant to the find/union algorithm, we can assume that all the elements have been numbered sequentially from 0 to N − 1 through some hashing scheme.
- Therefore, initially we can say $S_k = \{i\}$ for i = 0 to N-1
- The goal is to efficiently perform these two operations.
- Unfortunately, there is no easy way to guarantee constant worse-case time for both operations; rather it will be one or the other.
- find: to make find operation fast, we can store the name of the set of each element in an array. Hence, find becomes an O(1) lookup operation.
- However, this will make union to be Θ(N) (why?)
  - The algorithm will need to scan through all elements in both sets and change the name of their sets to the new name (or simply one of them by adding one set to the other)

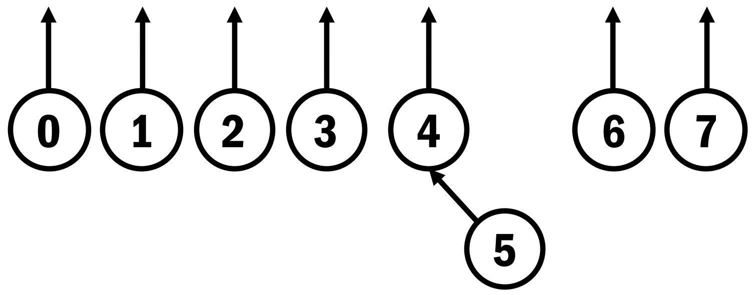
Enhanced Data Structure

- Given the name of each set is not relevant to the find operation, then one simple idea is to use trees to represent sets, with the root of each tree as the name of the set.
- The only required information from this tree is the parent link.
- Find is to find the tree root, and Union is to merge the two tree roots.
- We can use an array to store the parent's name of each node. Such that s[i] represents the parent of element i. If i is a root, s[i] = -1
- For example, an initial collection of 8 elements can be represented as:



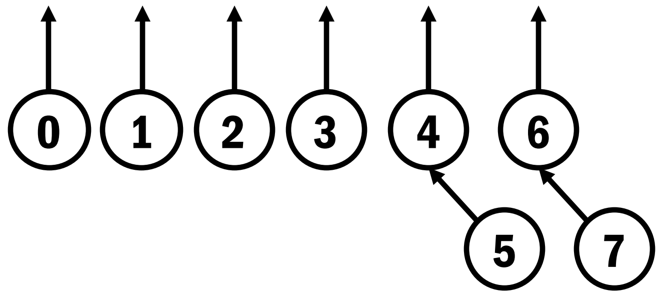
Basic Data Structure

- After union(4,5), forest and array will look like this:



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

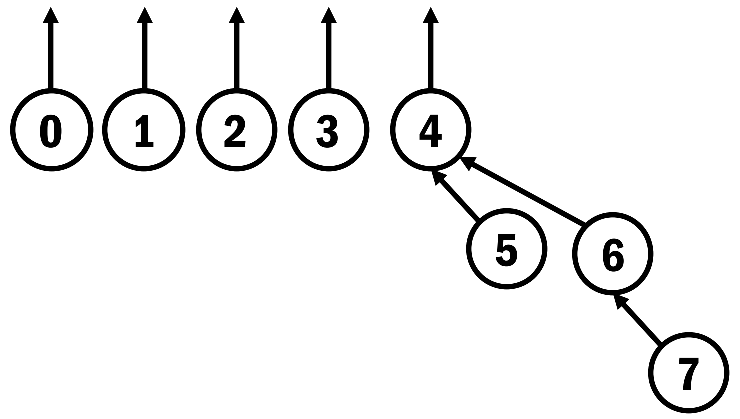
- After union(6,7), forest and array will look like this:



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

Basic Data Structure

- After union(4,6), forest and array will look like this:



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

- Note that, we are simply following a root selection such that root of the union(x,y) is x

Simple Algorithm Implementation

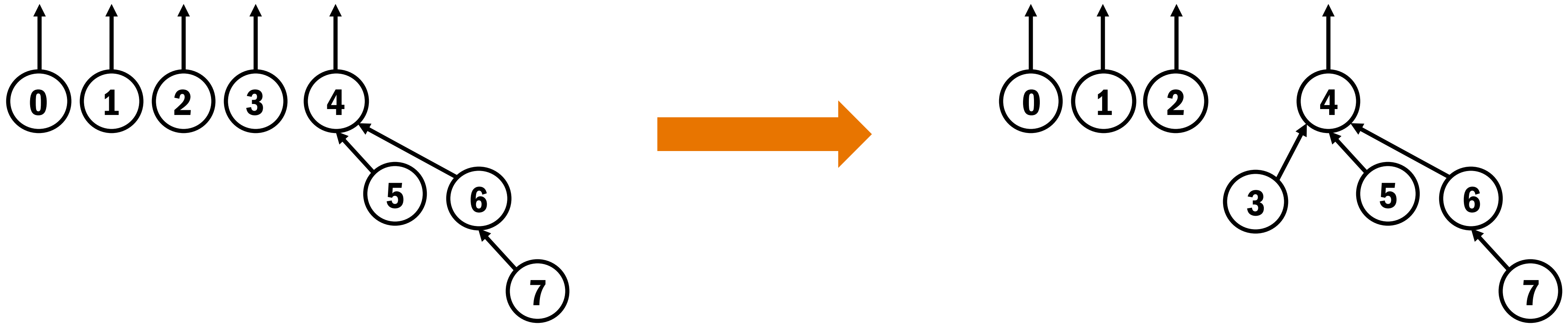
```
public DisjSets( int numElements )
{
    s = new int [ numElements ];
    for( int i = 0; i < s.length; i++ )
        s[ i ] = -1;
}
```

```
public void union( int root1, int root2 )
{
    s[ root2 ] = root1;
}
```

```
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}
```

Smart Union Algorithm

- In the simple algorithm, the root selection in union(x,y) is x.
- This could cause a high-depth tree, and hence find becomes O(N).
- A better approach is to make the smaller tree a subtree of the larger (union-by-size).
- Example, union(3,4) from previous example would yield the following:

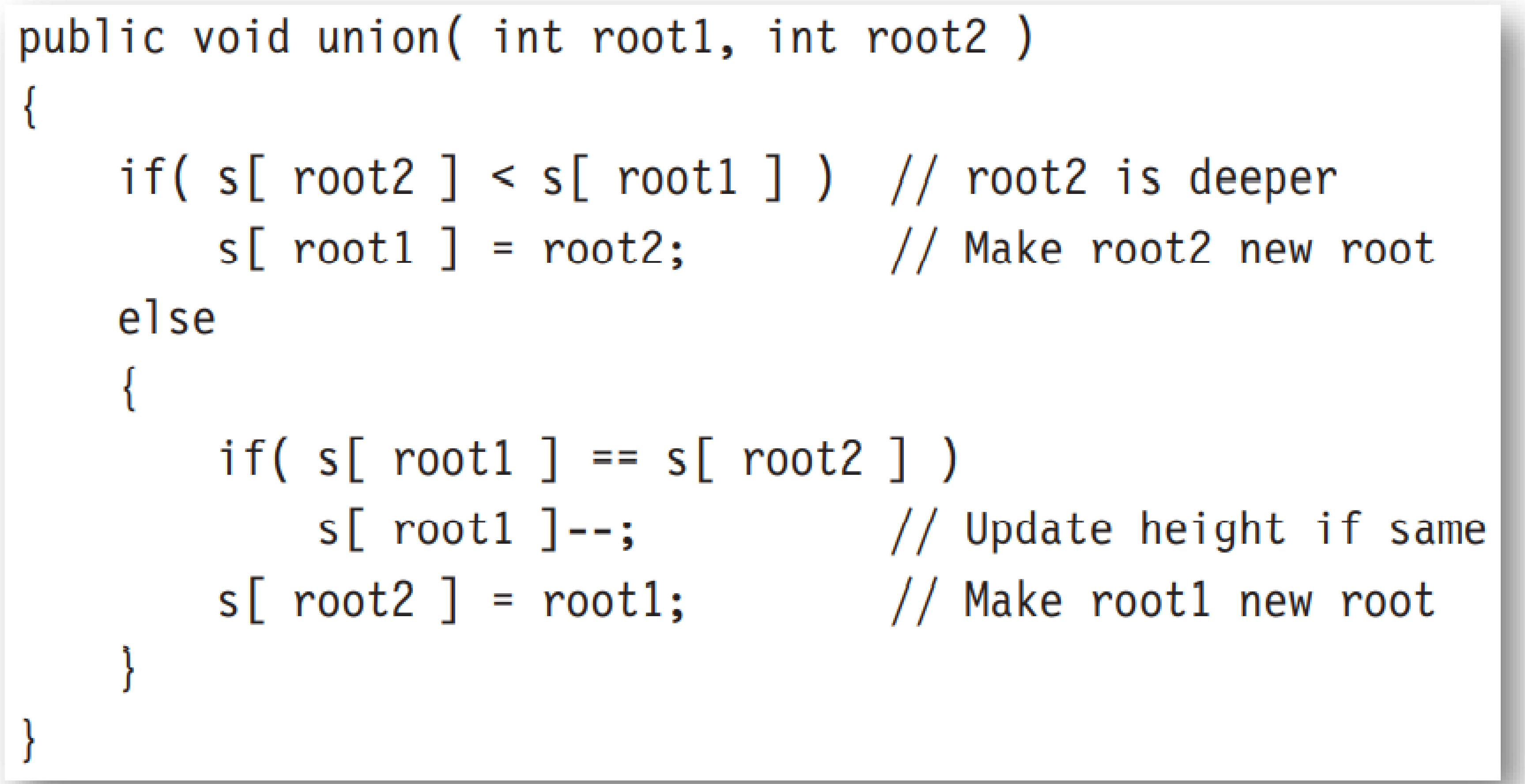


- What is the forest structure using the simple algorithm?

Smart Union Algorithm

- Union by size guarantees that tree depth will never exceed log N. This can be shown by example when trying the worst-scenario in all unions (what is that?)
- Observe that a node or a sub-tree joins another (as a result of union), if the other sub-tree is of equal or larger size, which results in a max height increase by 1. This can be used to prove that find operation complexity is O(log N)
- The implementation can simply be -1 (which indicates a root node in the array) multiplied by the size of the tree, which gives a negative number.
- An alternative approach is union-by-height, which makes the shallow tree a sub-tree of the deeper one.
- Worst case is when trying to union two equally height trees (binomial trees), which increases the height by 1, and hence find operation complexity is O(log N)
  - This is implemented by storing negative of heights -1

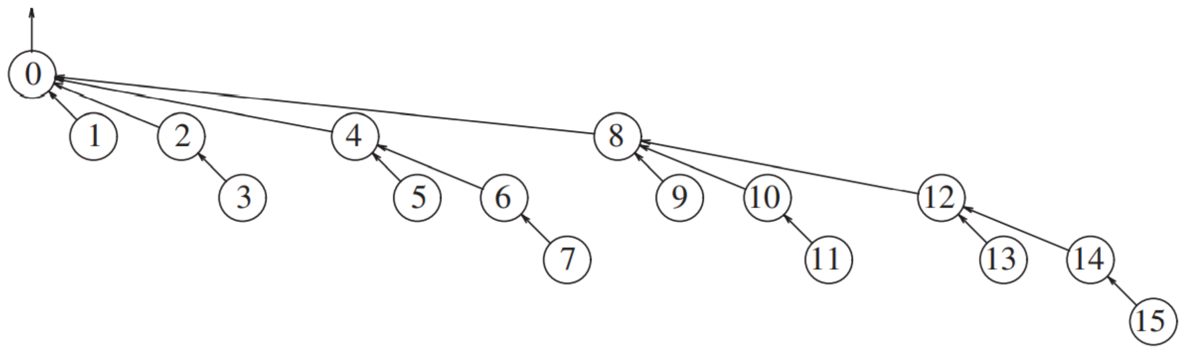
Union-by-Height Implementation

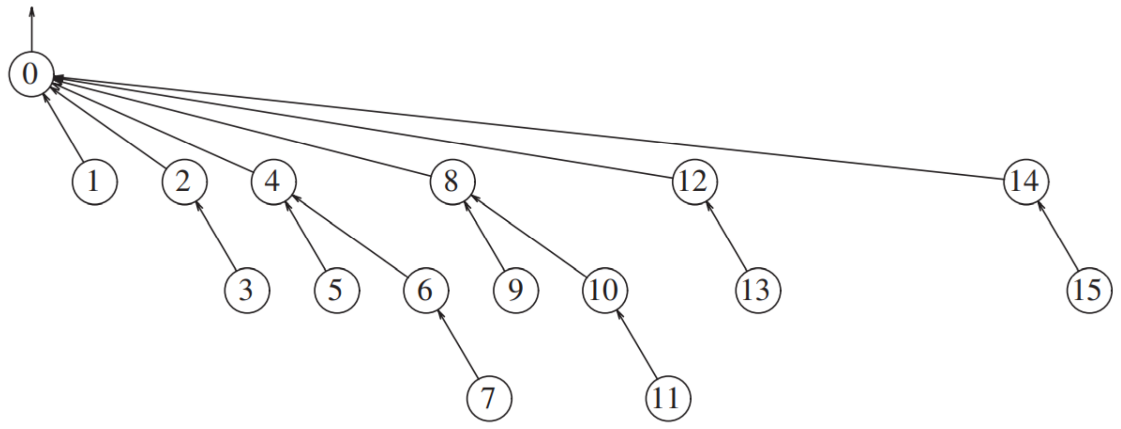


```
public void union( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
        s[ root1 ] = root2;        // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;        // Make root1 new root
    }
}
```

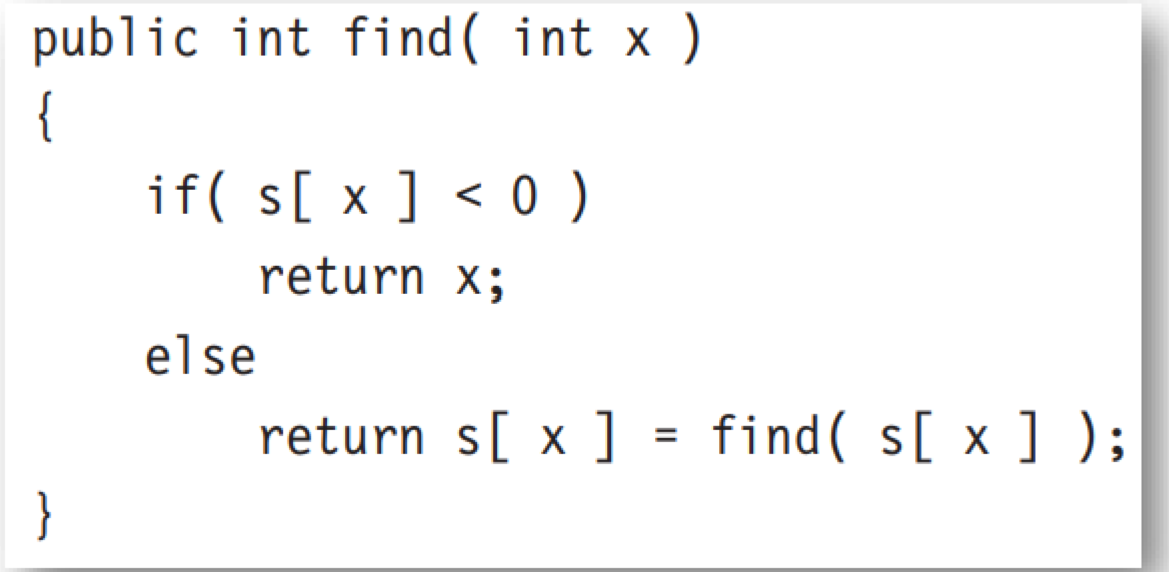
Path Compression

- Path compression is an enhanced find algorithm to provide better overall performance. Recall for M operations, the worst-case overall performance can be O(M log N).
- The modified find(x) algorithm changes the parent of each node on the path from x to the root to be linked to the root
- For example, find(14) on the follow tree





Path Compression Implementation



```
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

- Path compression is compatible with union-by-size. However, it is not compatible with union-by-height (why)?

To Do List

- ➢Review lecture notes
- ➢Study chapter 8 and review the Maze application (8.7)