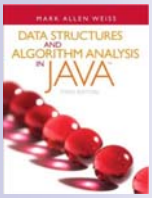
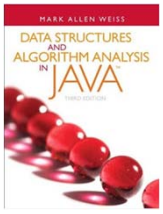


FIU FLORIDA INTERNATIONAL UNIVERSITY

Some Problems on Graphs (V)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

Module #7: Some Problems on Graphs (part V)

Outline:

- **Union-Find Data Structures:**
  - **Disjoint set class**
    - **Find and Union operations**

Disjoint Set Definition

- Suppose we have a **collection** of *n* **distinct** items/elements - **set**.
- We want to make the **partition** of the set into a **collection** of sets such that:
  - Each item is in a set
  - No item is in more than one set
- The resulting sets are said to be **disjoint sets**
- **Examples:**
  - {1, 2, 3} and {4, 5, 6} are disjoint sets.
  - {**x**, y, z} and {t, u, **x**} are **not** disjoint sets.

3

Disjoint Set Terms

- We **can** identify a **set** by selecting a **representative element** of the set.
- It doesn't matter which element we choose, but once chosen, **it can't change.**
- **Example:**
  - Let *S* = {1,2,3,4,5,6,7,8,9}
  - Let initial partition be (will highlight/underline representative elements)

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

4

Disjoint Set Terms

- **Three operations** of interest:
  - **make-set(x)** - create a new set with only x. Assume x is not already in some other set.
  - **find(x)** - return the **representative** of the **set** containing x.
  - **union(x,y)** - combine the two sets containing x and y into one new set. A new representative is selected.

5

Disjoint Set. Example #1

- Let *S* = {1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Let **initial partition** be (**make-set**(x), for each x in *S*):

    {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}
  - **union**(2,5):

    {1}, {2, 5}, {3}, {4}, {6}, {7}, {8}, {9}
  - **find**(4) = 4, **find**(2) = 2, **find**(5) = 2
  - **union**(4,6), **union**(2,7)

    {1}, {2, 5, 7}, {3}, {4, 6}, {8}, {9}
  - **find**(4) = 6, **find**(2) = 2, **find**(5) = 2
  - **union**(2,6)

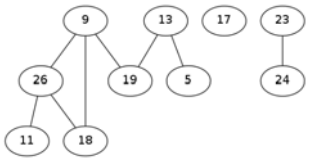
    {1}, {2, 4, 5, 6, 7}, {3}, {8}, {9}

6

Disjoint Set. Example #2

- Find the **connected components** of the undirected graph G=(V,E) (maximal **subgraphs** that are **connected**).

```
CONNECTED-COMPONENTS
    for (each vertex v in V)
        make-set(v)
    for (each edge (u,v) in E)
        if (find(u) != find(v))
            union(u,v)
```



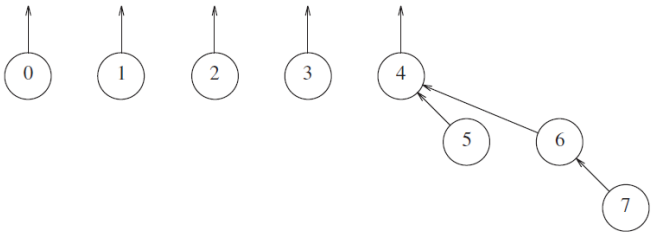
- By using the **find** operation we can verify if two vertices, u and v, are in the same **connected component** by testing:

```
SAME-COMPONENT(u,v)
  if find(u) = find(v)
    return TRUE
  else
    return FALSE
```

7

Disjoint set implementation. Basic ideas

- **Up-tree implementation**
  - Basic idea is to use a **tree** to represent each **set**.
  - Every item/element is in a **tree**
  - The **root** of the **tree** is the **representative element** of all items in that tree i.e., the root can be use to name the set.
  - In this **up-tree implementation**, every node (except the root) has a **pointer** pointing to its **parent**.
  - The **root** element has a **pointer** pointing to "empty" parent (or pointing to itself).



8

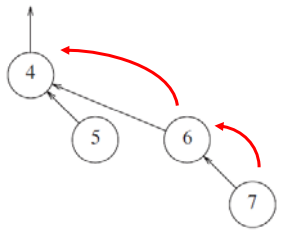
Disjoint set implementation: find(x)

- **find(x)**
  - Start at **x** and follow parent pointers to root
  - Return the root

Pseudocode:

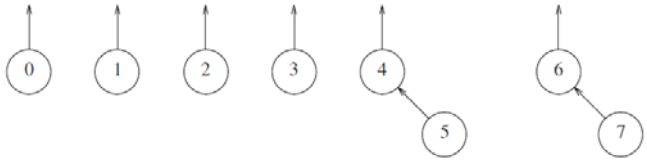
```
find(x)
while (x != "empty" parent)
    x = x -> parent;
return x;
```

find(7) = 4

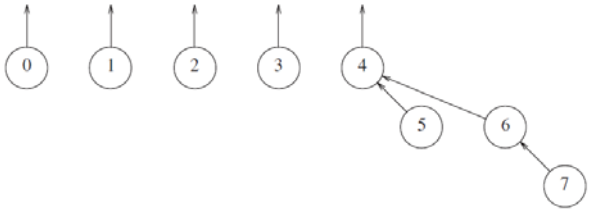


Disjoint set implementation: union(x,y)

- **union(x,y)**
  - Assume **x** and **y** are roots
    - (else find the roots of their trees)
  - Assume distinct trees (else do nothing)
  - Change **root** of one to have parent be the root of the other
- **Example:**
  - Before…

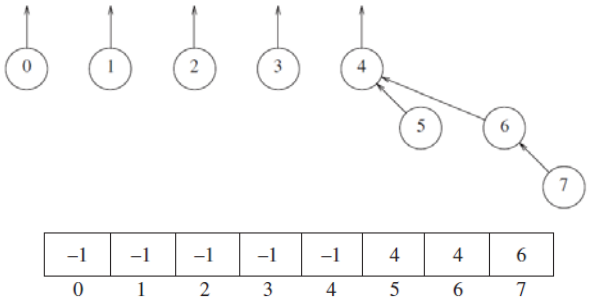


  - After **union(4,6)**



Disjoint set. Simple implementation

- If elements are **contiguous** numbers (e.g., 0,1, 2,…, *n*-1), use an array of length *n* called **up**
  - Put in array index of parent, with **-1** (or **0**) for a root
- **Example:**



–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

- If set elements are **not contiguous** numbers, could have a separate dictionary to map elements (keys) to numbers (values)

11

Disjoint set class.

- **The skeleton:**

```
public class DisjSets
{
    public DisjSets( int numElements )
      { /* Figure 8.7 */ }
    public void union( int root1, int root2 )
      { /* Figures 8.8 and 8.14 */ }
    public int find( int x )
      { /* Figures 8.9 and 8.16 */ }

    private int [ ] s;
}
```

12

Disjoint set class.

- **Initialization routine:**

```
/**
 * Construct the disjoint sets object.
 * @param numElements the initial number of disjoint sets.
 */
public DisjSets( int numElements )
{
    s = new int [ numElements ];
    for( int i = 0; i < s.length; i++ )
        s[ i ] = -1;
}
```

13

Disjoint set class.

- **Find Algorithm:**

```
/**
 * Perform a find.
 * Error checks omitted again for simplicity.
 * @param x the element being searched for.
 * @return the set containing x.
 */
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}
```

- Worst-case run-time for **find**? ***O*(*n*)**

14

Disjoint set class.

- **Union Algorithm (not the best):**

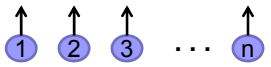
```
/**
 * Union two disjoint sets.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 */
public void union( int root1, int root2 )
{
    s[ root2 ] = root1;
}
```

- Worst-case run-time for **union**? ***O*(1)**

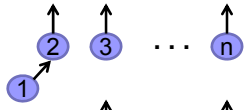
15

Union operation

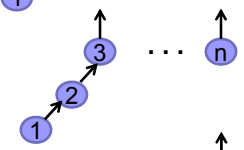
- Example of the **bad** case:



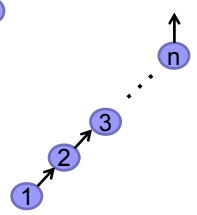
Can we do better? Yes!!!



After **union(1,2)**



After **union(2,3)**



After **union(n-1,n)**

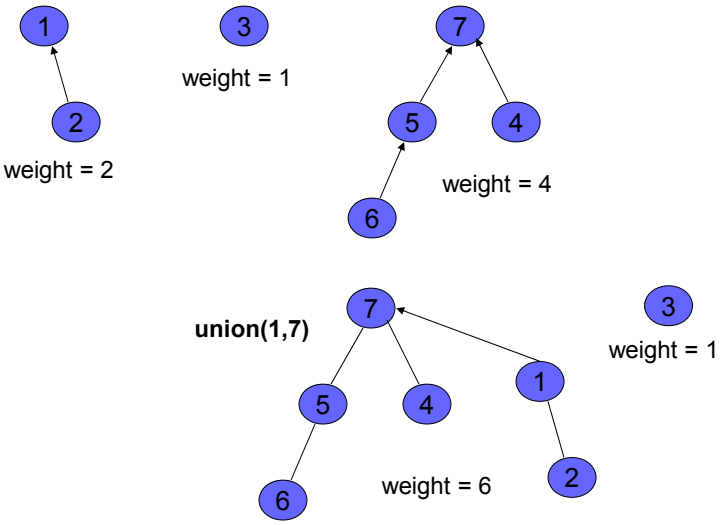
find(1)? *O*(n) steps!

16

Optimization of the union operation

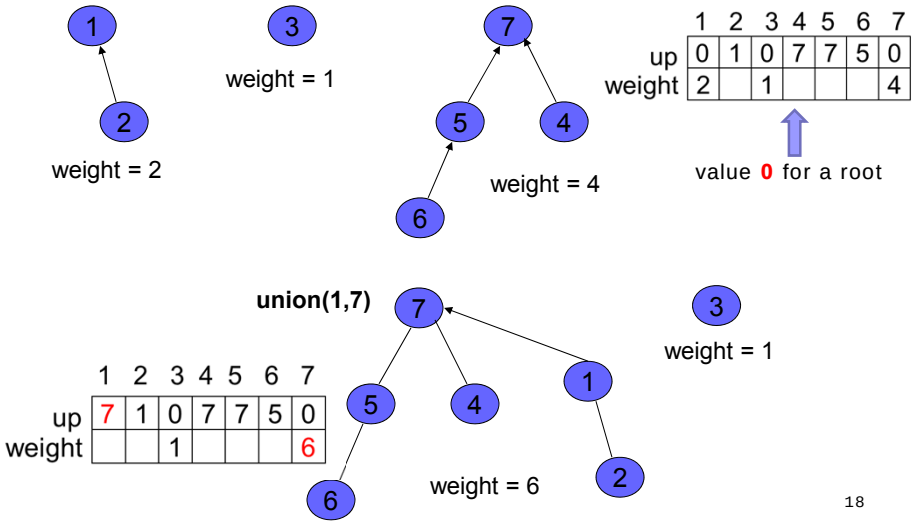
Union by weight:

- **Strategy**: Always point the **smaller tree** (by total number of nodes) **to the root of the larger tree**



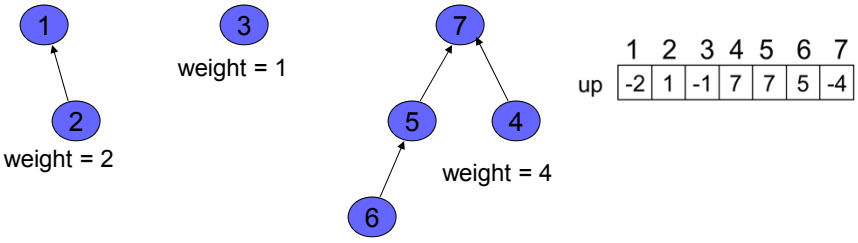
Optimization of the union operation

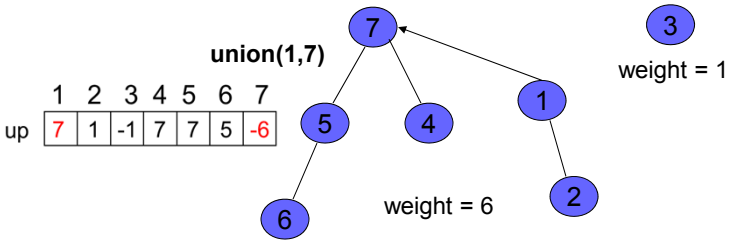
- **Union by weight. Array Implementation**
- Keep the **weight** (number of nodes in a **second array**)



Union by weight. Array Implementation

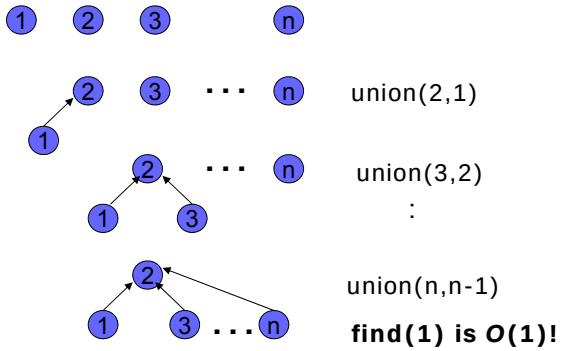
- **We do not need a second array!**
  - Instead of storing **0** for a root, store **negation of weight**
  - So up **value < 0 means a root**





Optimization of the union operation

- Using **union by weight…**



- Then:
  - **union** is still ***O*(1)**
- **What is worst-case complexity of find(x)?**

Theorem: **Union by weight** guarantees that all **up-tress** will have height al most ***O*(log n)** then **find is *O*(log n)**.

Java implementation

- For a full version of the Disjoint set class implementation please consult:

https://users.cs.fiu.edu/~weiss/dsaajava2/code/DisjSets.java

(Authors: Mark Weiss)

21