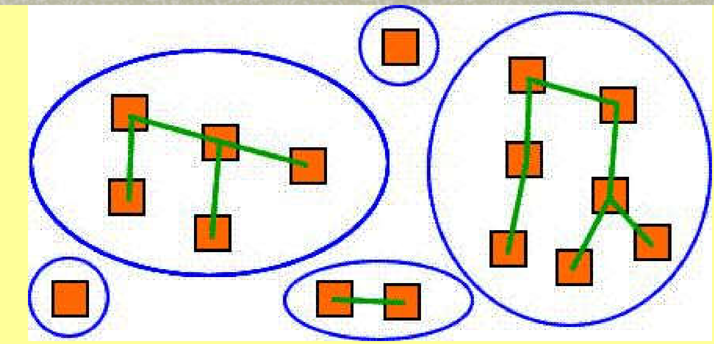
Analysis & Design of Algorithms (CSCE 321)

Prof. Amr Goneid
Department of Computer Science, AUC

Part R4. Disjoint Sets

Disjoint Sets



- What are Disjoint Sets?
- Tree Representation
- Basic Operations
- Parent Array Representation
- Simple Find and Simple Union
- Disjoint Sets Class
- Some Applications

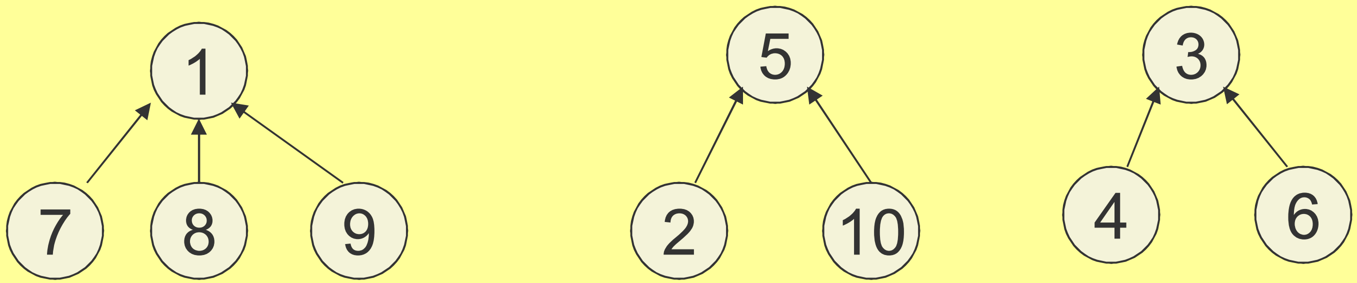
What are Disjoint Sets?

- A set S is a collection of elements of the same type
- We assume that the elements are the numbers 1,2,.. n. In practice, these numbers may be indices to a symbol table containing the names of the sets.
- Disjoint sets are collections of elements with no common elements between the sets. If S_i and S_j, $i \neq j$ are two sets, then $S_i \cap S_j = Ø$
- Examples:

  S1 = {1,7,8,9},   S2 = {5,2,10},   S3 = {3,4,6}

Union-Find Data Structure: Tree Representation

- One possible representation is a tree where a set can be identified by a *parent* node and *children* nodes.
- In this representation, children point to their parents, rather than the reverse:



S1 = {1,7,8,9} S2 = {5,2,10} S3 = {3,4,6}

Basic Operations

- **Find(i):**

  Given the element (i), find the set containing (i), i.e. find the root of the tree.

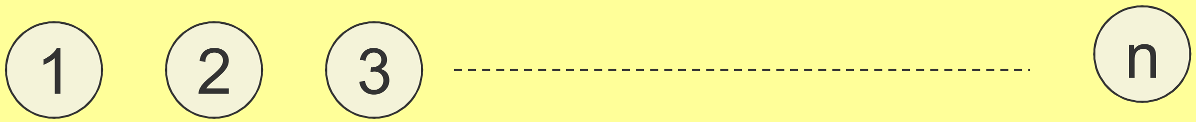
- **Union(i,j):**

  Given two disjoint sets S_i and S_j, obtain a set containing the elements in both sets, i.e.,

  $S_i \cup S_j$

Parent Array Representation

- n disjoint nodes can be represented as n disjoint sets where each node is its own parent, i.e. p[i] = -1:



i	1	2	3				n
p[i]	-1	-1	-1				-1

Basic Operations

- **Find(i):**

  Finding out which subset we are in is simple, for we keep traversing up the parent pointers until we hit the root.

Simple Find

- The value in p[i] represents the parent of node (i). For the sets S1,S2,S3 shown before, we have:

i	1	2	3	4	5	6	7	8	9	10
p[i]	-1	5	-1	3	-1	3	1	1	1	5

- <u>Algorithm for Simple find:</u>

  Find the set containing node (i) = find the parent of (i):

```
int find(int i)
{       while (p[i] >= 0) i = p[i];
        return i;
}
```

find (1) → 1
find (4) → 3
find (10)→ 5

Basic Operations

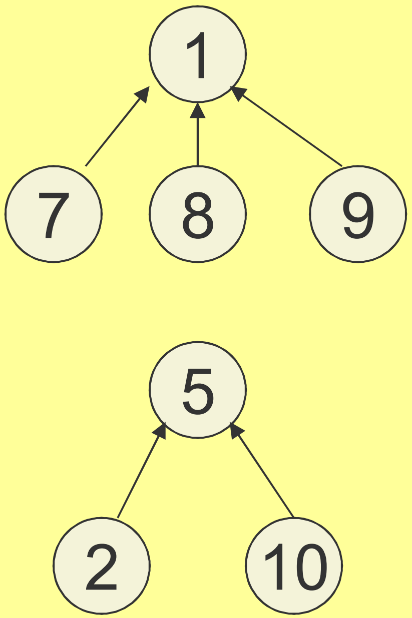
- **Union(i,j):**

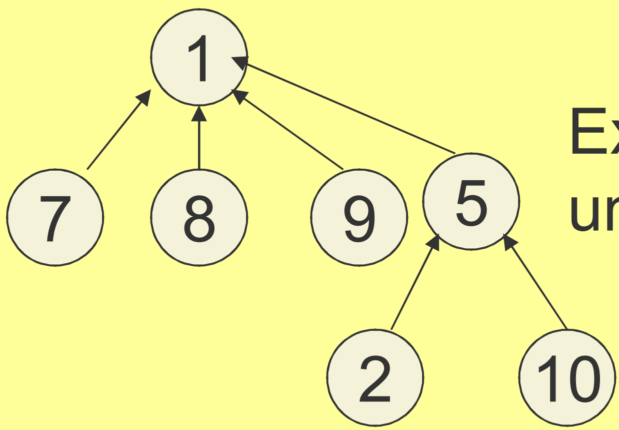
  Making a union between two subsets is also easy. Just make the root of one of two trees point to the other, so now *all* elements have the same root and thus the same subset name.

Simple Union

- **Simple (disjoint) Union:**

  Make set (i) the child of set (j) = union (i,j)





Example:
union(5,1)

Simple Union

- The parent array would change to:

i	1	2	3	4	5	6	7	8	9	10
p[i]	-1	5	-1	3	1	3	1	1	1	5

- Algorithm:

```
void union (int i, int j)
{ p [i] = j ; }
```

Disjoint sets class

```
// Make set(i) the child of set(j)
   void simpleUnion(int i, int j)
   {    p[i] = j;      }

// Find the parent set of subset(i)
   int simpleFind(int i)
   {    while (p[i]>=0) i = p[i];
        return i;
   }
```

Analysis

- **<u>Union:</u>** takes constant time, i.e., ***O(1)***
- **<u>Find:</u>** Suppose we do the following sequence of unions:

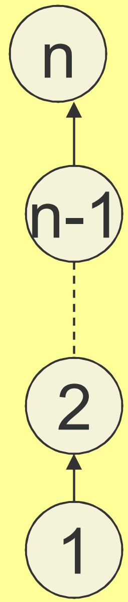
  union(1,2) , union(2,3),...., union(n-1,n)

  The time of find parent of node (i) will be i.

  To process all finds, we need

  (1+2+....+n) steps = $O(n^2)$

  Hence, the average find cost is ***O(n)***



Improvement

- We can improve the find cost by modifying the <u>union</u> algorithm to be:

  "*make the root of the smaller tree point to the root of the bigger tree" (known as **"union by size"**)*
- In this case, the cost of <u>find</u> is ***O(log n)***
- <u>**Implementation:**</u>

  Store the node count in each root. Modify the count when making a union.

Note: there is also **"union by rank"**

Disjoint sets class

```
// Make a union between set(i) and set(j)
    void  simpleUnion(int i, int j)
    {
            int sum = c[i] + c[j];
            if (c[i] > c[j]) {p[j] = i; c[i] = sum;}
            else { p[i] = j; c[j] = sum;}
    }

// Find the parent set of subset(i)
    int  simpleFind(int i)
    {       while (p[i]>=0) i = p[i];
            return i;
    }
```

Path compression

```
function Find(x)
        if x.parent != -1                // if not root
                x.parent := Find(x.parent)
        return x.parent
```

This is valid, since each element visited on the way to a root is part of the same set. The resulting flatter tree speeds up future operations.

Check the wiki page: both operations are semi-linear: O(a*n)

Application example

MST with Kruskal

```
KRUSKAL(G):
    A = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by weight(u, v), increasing:
        if FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```