Disjoint Sets

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Disjoint Sets

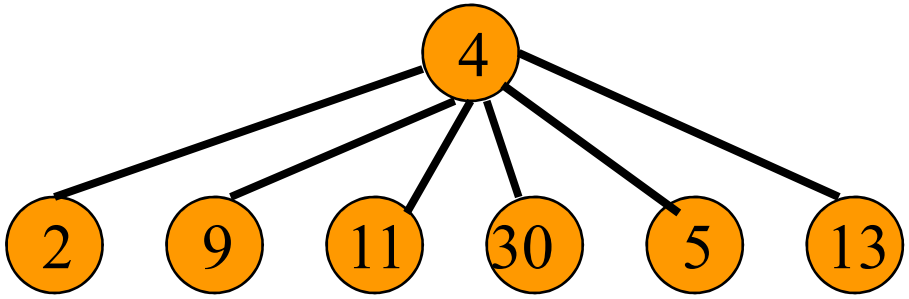
- Given a set {1, 2, …, n} of n elements.
- Initially each element is in a different set.
  - {1}, {2}, …, {n}
- An intermixed sequence of union and find operations is performed.
- A *union* operation combines two sets into one.
  - Each of the n elements is in exactly one set at any time.
- A *find* operation identifies the set that contains a particular element.

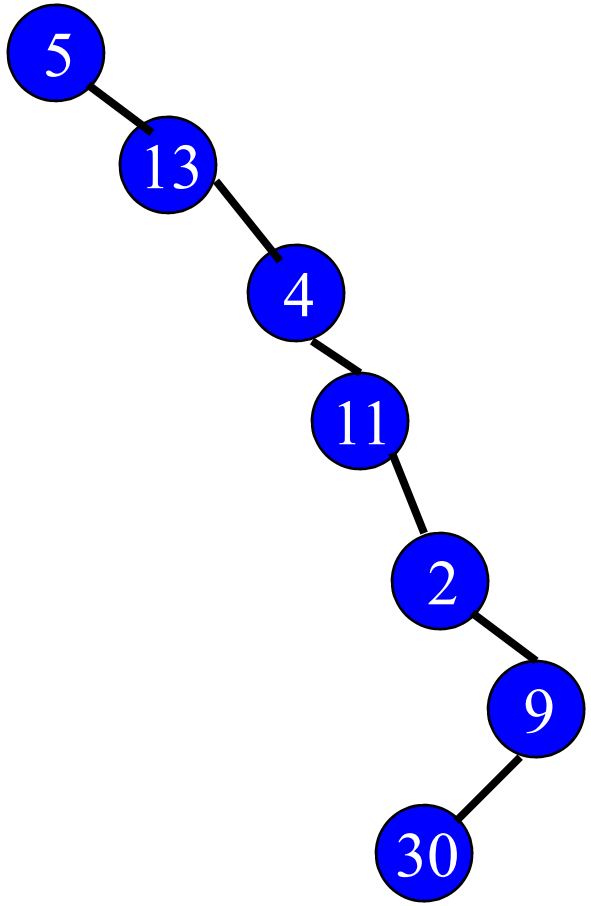
Time Complexity

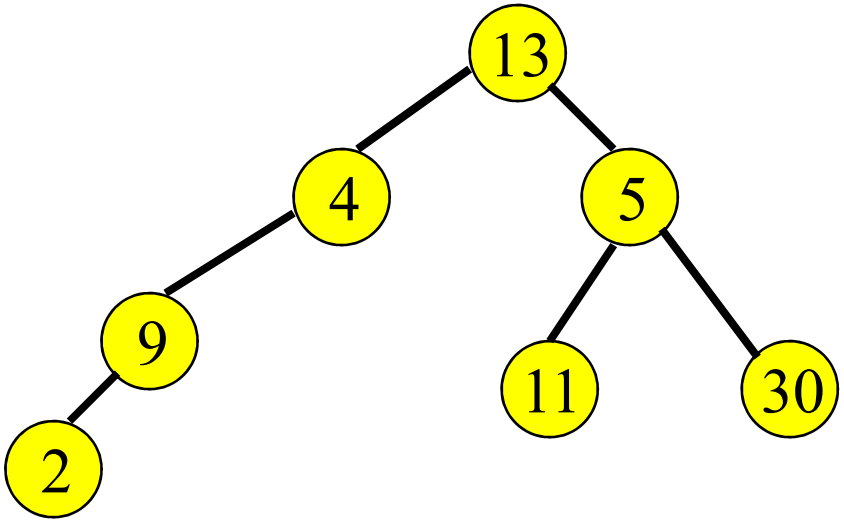
- Let u and f be the number of union and find operations, respectively
- Using a tree (not a binary tree) to represent a set, the time complexity becomes almost O(n + f) (assuming at least n/2 union operations).

A Set as a Tree

- S = {2, 4, 5, 9, 11, 13, 30}
- Some possible tree representations:

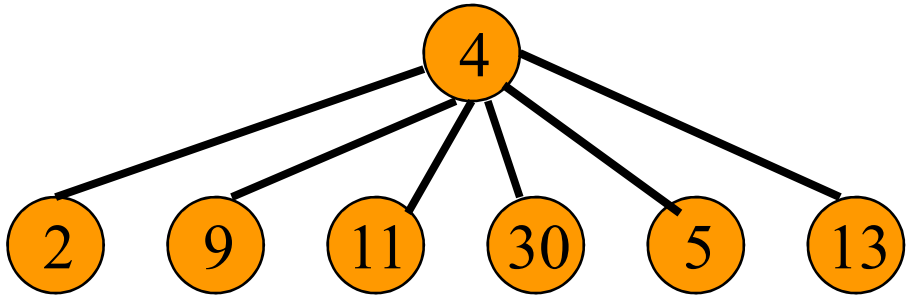






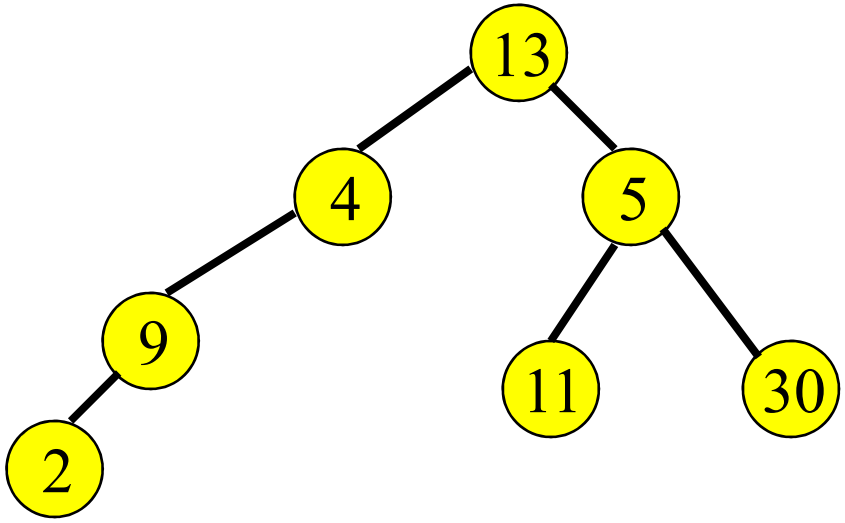
Result of a Find Operation

- Find(i) is to identify the set that contains element i.
- In most applications of the union-find problem, the user does not provide set identifiers.
- The requirement is that Find(i) and Find(j) return the same value iff elements i and j are in the same set.



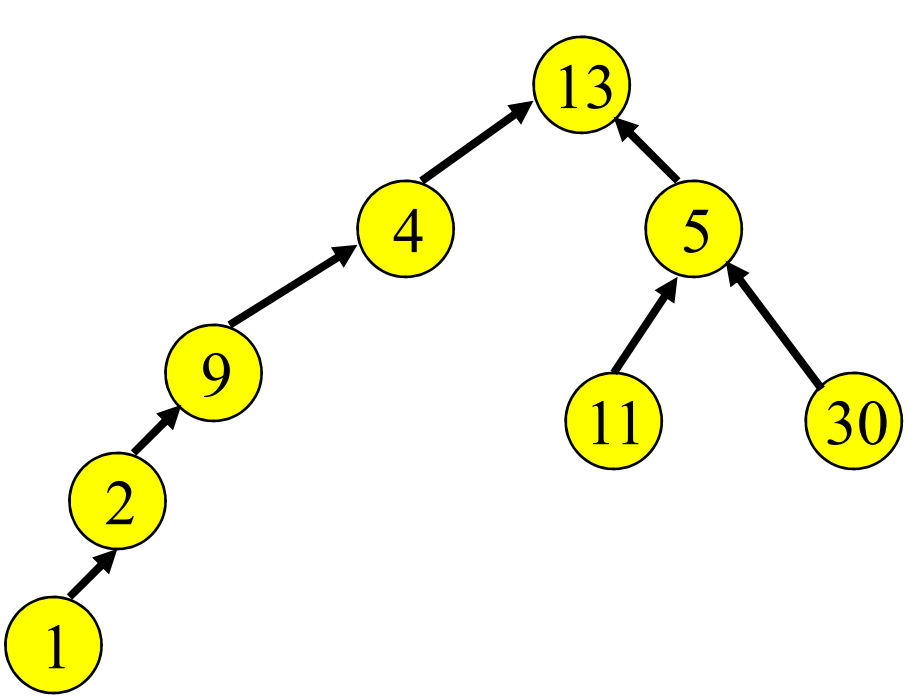
Find(i) will return the element that is in the tree root.

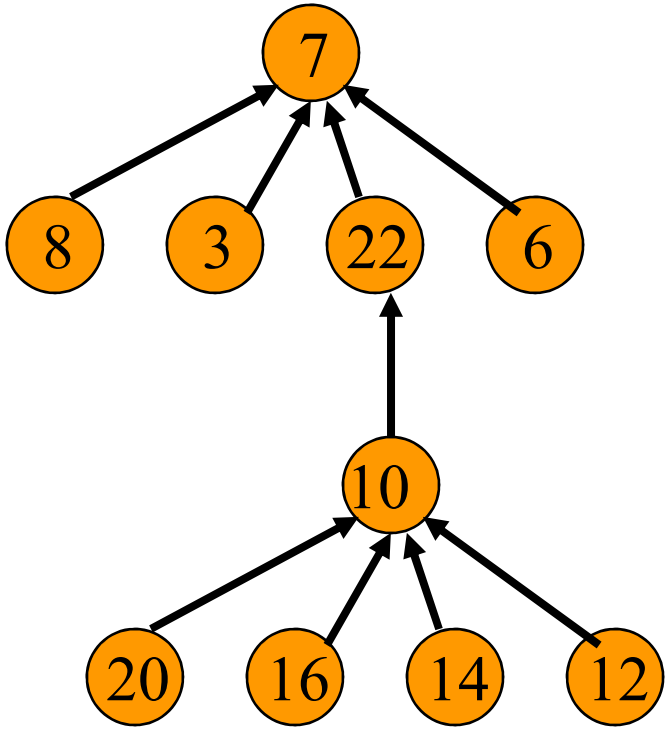
Strategy for Find(i)



- Start at the node that represents element i and climb up the tree until the root is reached.
- Return the element in the root.
- To climb the tree, each node must have a parent pointer.

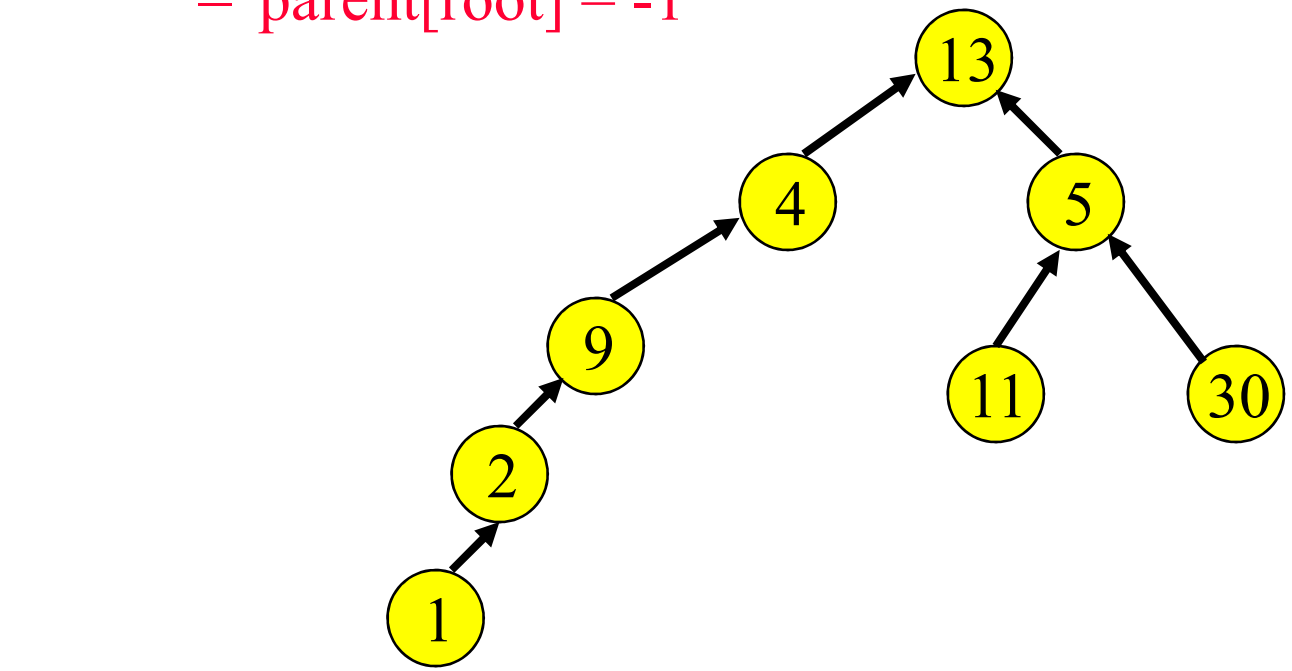
Trees with Parent Pointers

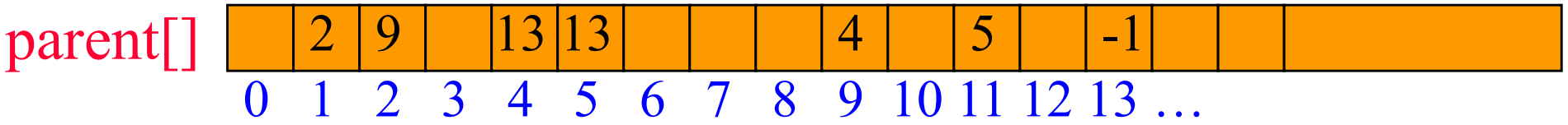




Representation

- Use an integer array parent[] such that parent[i] is the parent of an element i
  - parent[root] = -1

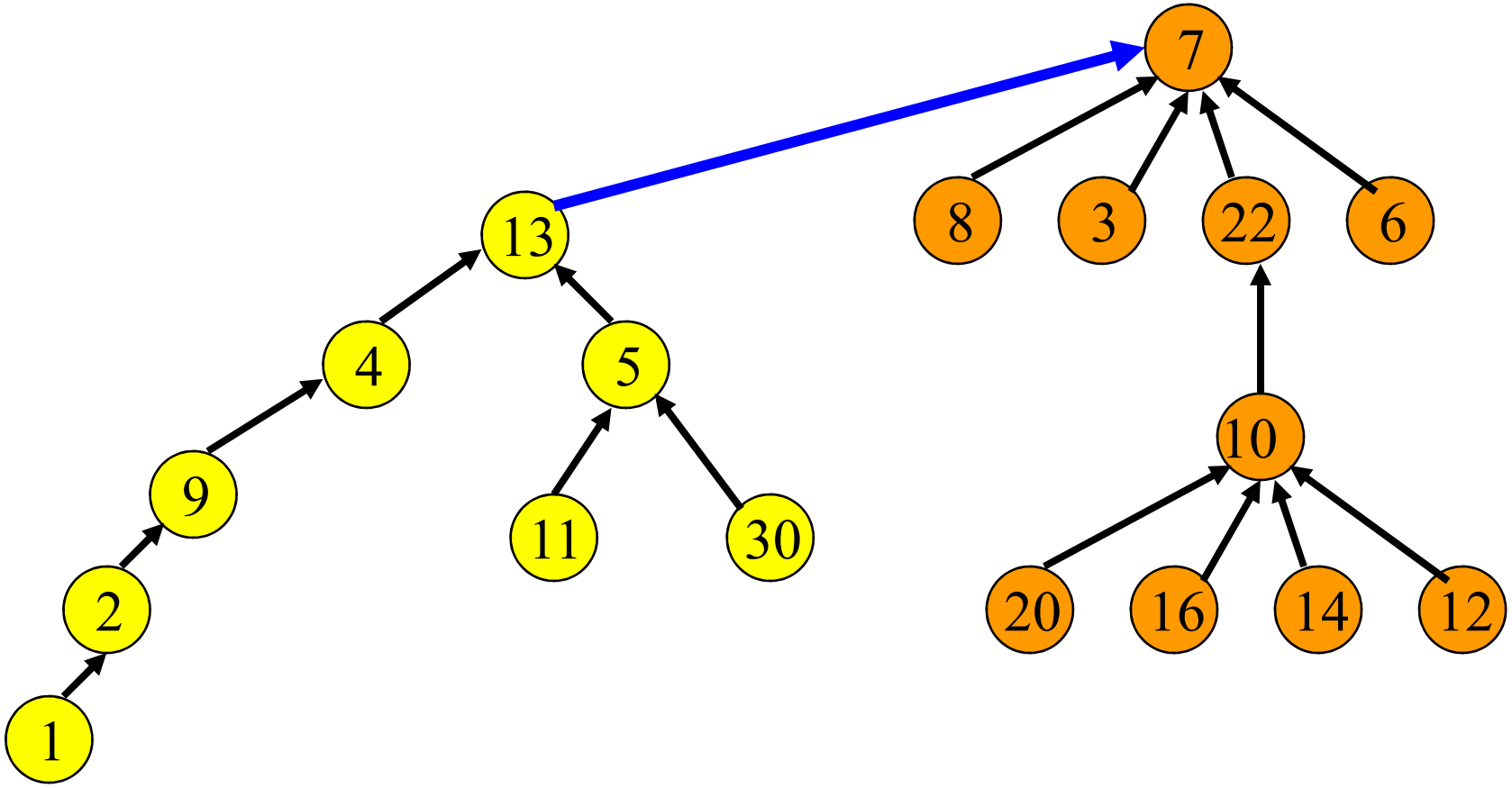




Union Operation

- Union(i, j)
  - i and j are the roots of two different trees, i != j.
- To unite the trees, make one tree a subtree of the other.
  - parent[j] = i

Union Example



- Union(7,13) ➔ parent[13] = 7

The Union Method

```
void SimpleUnion(int i, int j)
    {parent[i] = j;}
```

Time Complexity of SimpleUnion()

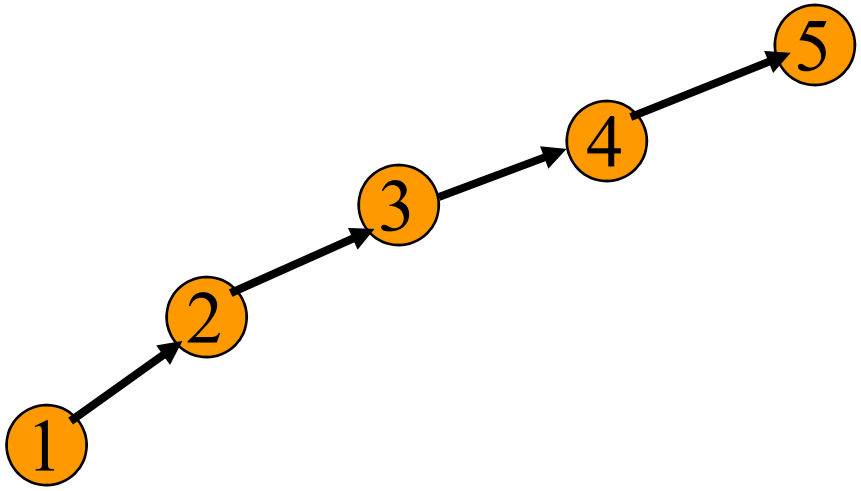
- O(1)

The Find Method

```
int SimpleFind(int i)
{
     while (parent[i] >= 0)
        i = parent[i];  // move up the tree
     return i;
}
```

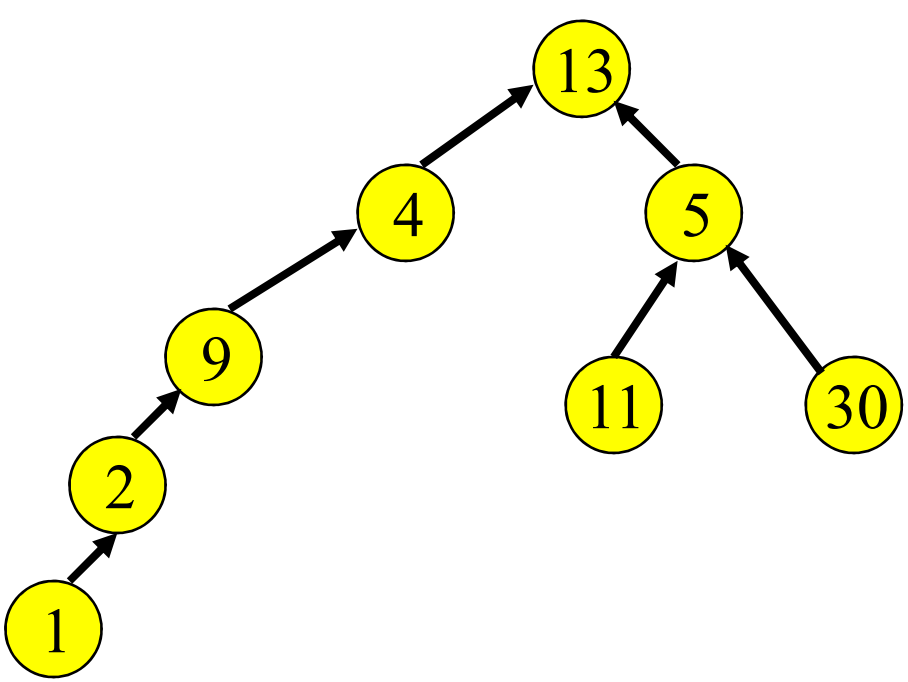
Time Complexity of SimpleFind()

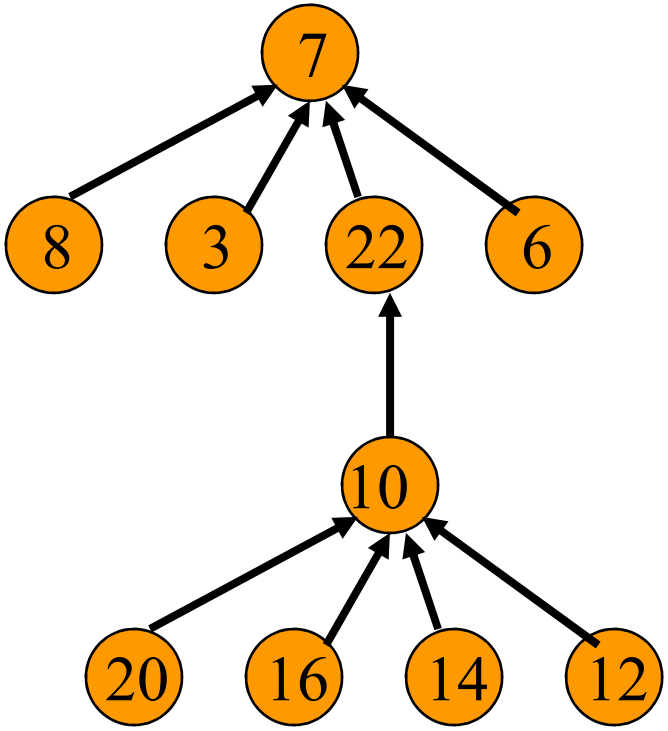
- Tree height may equal number of elements in tree.
  - Union(2,1), Union(3,2), Union(4,3), Union(5,4)…



So complexity is O(u).

Smart Union Strategies

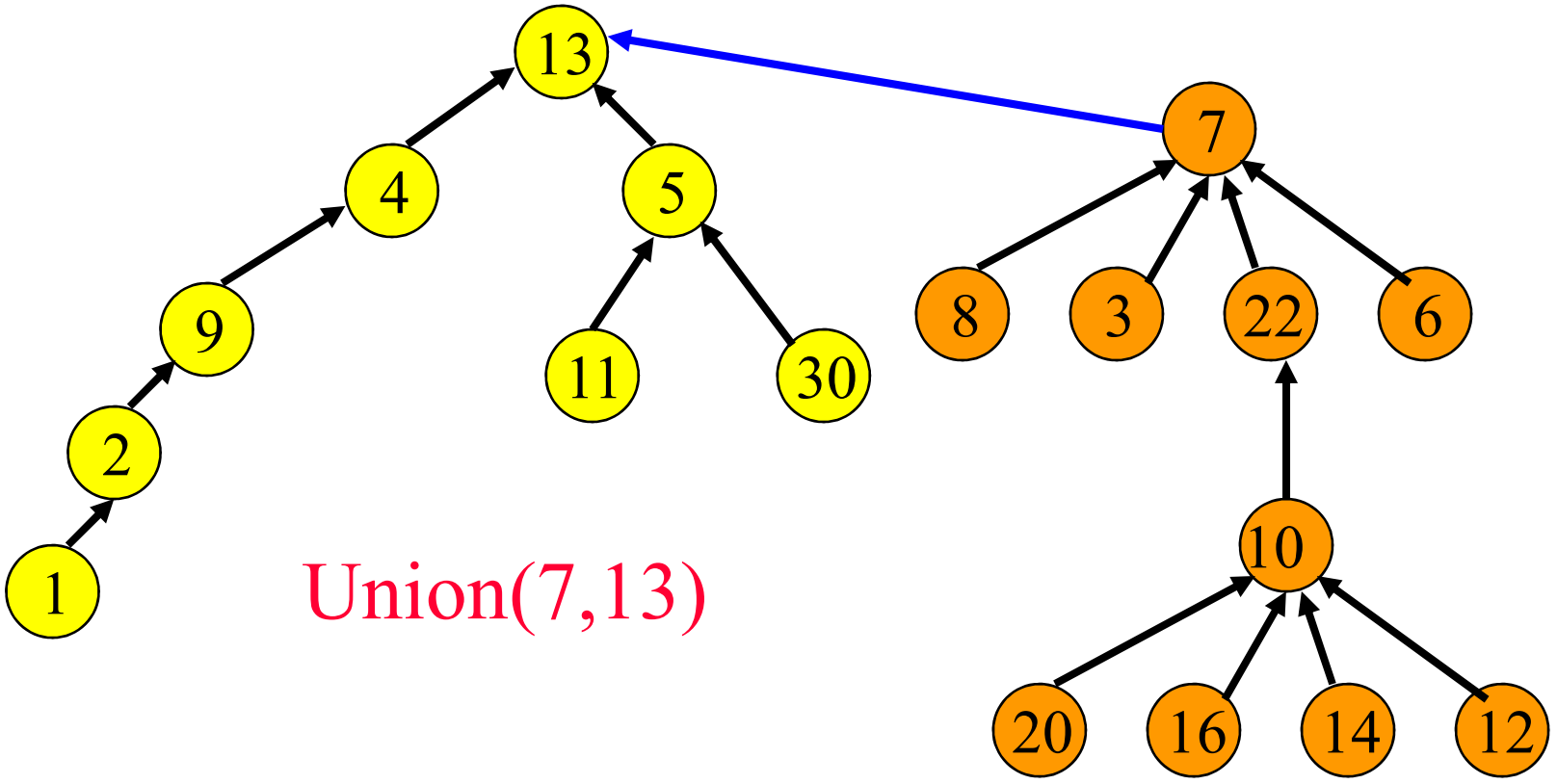




- Union(7,13)
- Which tree should become a subtree of the other?

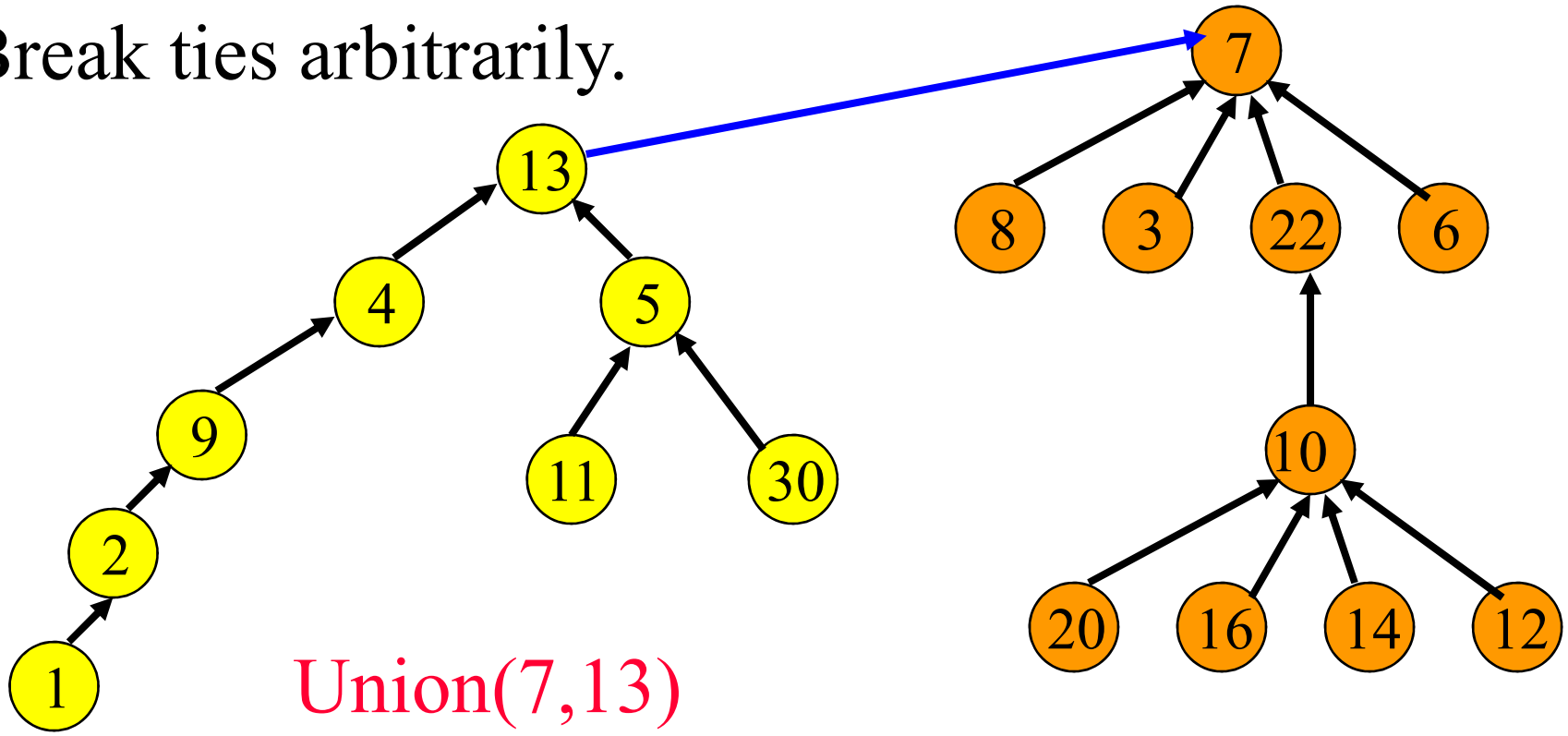
Height Rule

- Make tree with smaller height a subtree of the other tree.
- Break ties arbitrarily.

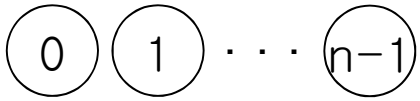


Weight Rule

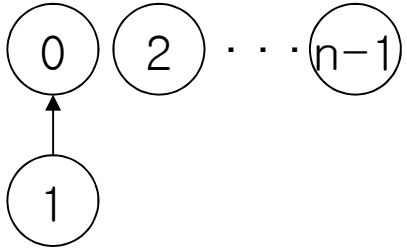
- Make tree with fewer number of elements a subtree of the other tree.
- Break ties arbitrarily.



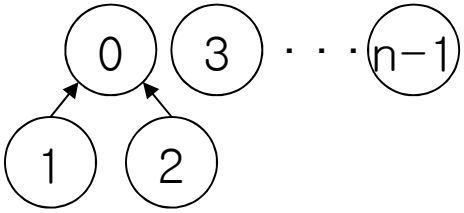
Example



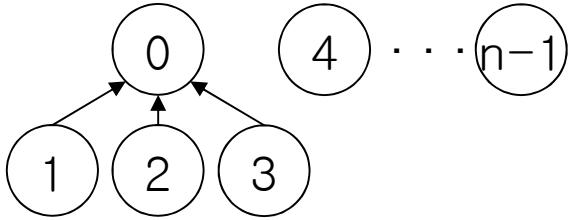
initial



Union(0,1)

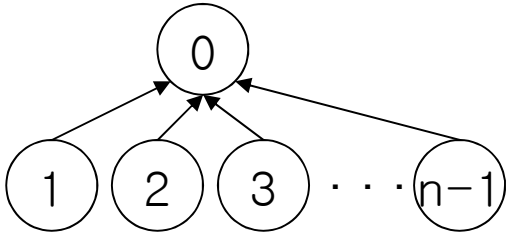


Union(0,2)



Union(0,3)

· · ·



Union(0,n−1)

Height of a Tree

- Suppose we start with single element trees and perform unions using either the height or the weight rule.
- The height of a tree with n elements is at most floor ($\log_2 n$) + 1.
- Proof is by induction on n. See text.
- So the time complexity of SimpleFind() is O(log n).

Implementation

- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

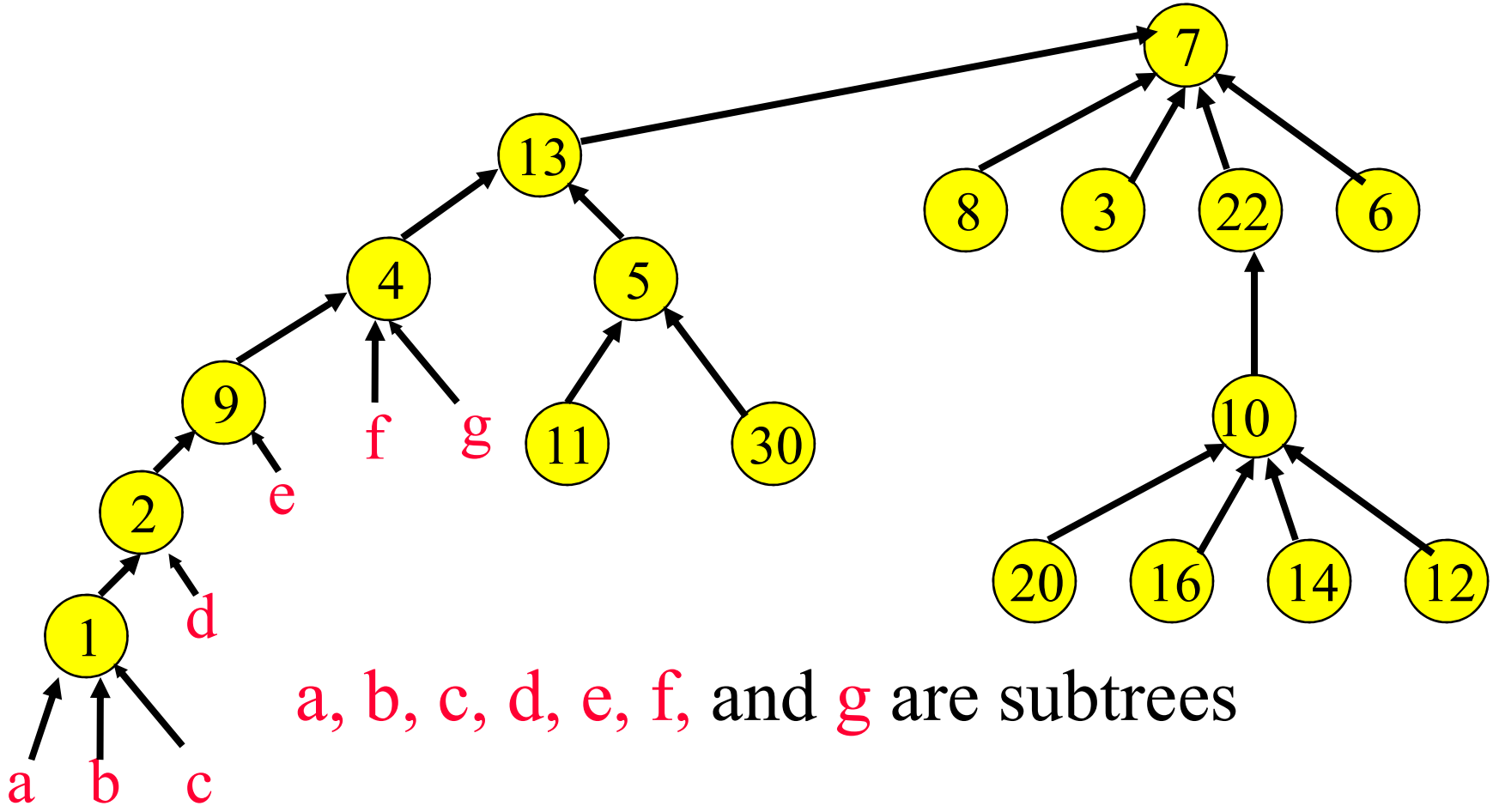
Implementation (cont.)

- When we use the weight rule, we maintain a *count* field in the root of every tree
- Since all nodes other than the roots of trees have a non-negative number in the *parent* field, we can maintain the count in the parent field of the roots as a negative number
- Initially, the *parent* field of all nodes contains -1

Union Function with Weight Rule

```
Program 5.25:Union functions with weighting rule
================================================
void Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i != j, using the weighting rule.
// parent[i]=-count[i] and parent[j]=-count[j].
{
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) { // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
    }
    else { // j has fewer nodes (or i and j have the same number of nodes)
        parent[j] = i;
        parent[i] = temp;
    }
}
================================================
```

Sprucing up the Find Method

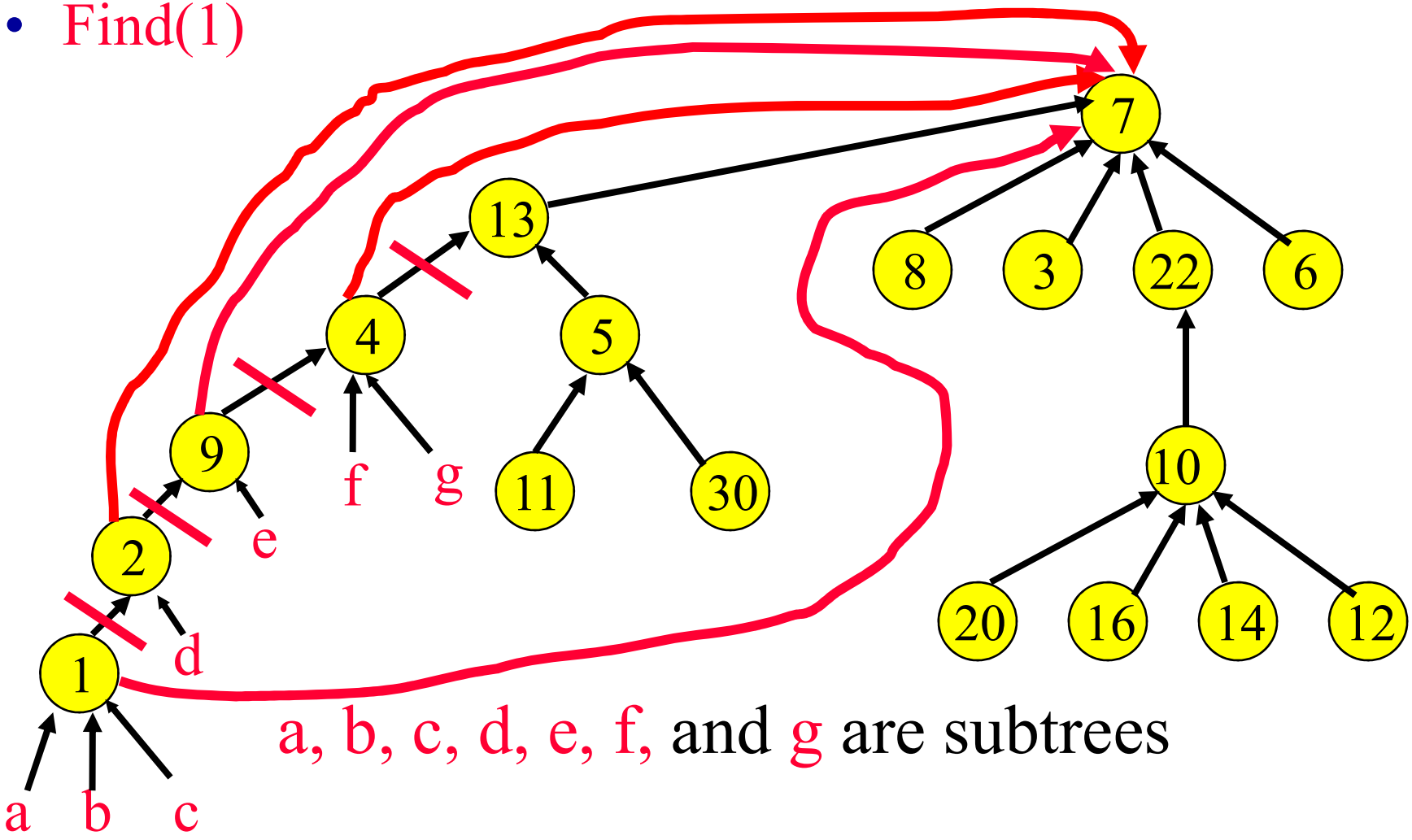


a, b, c, d, e, f, and g are subtrees

- Find(1)
- Do additional work to make future finds easier.

Path Compaction

- Make all nodes on find path point to tree root.
- Find(1)



a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree.

Collapsing Find

```
Program 5.26:Collapsing rule
================================================
int Sets::CollapsingFind(int i)
{// Find the root of the tree containing element i.
// Use the collapsing rule to collapse all nodes from i to the root.
   for (int r = i; parent[r] >= 0; r = parent[r]); // find root
   while (i != r) {// collapse
      int s = parent[i];
      parent[i] = r;
      i = s;
   }
   return r;
}
================================================
```

The time complexity becomes almost O(n + f)

- n: the number of elements
- f: the number of find operations

Time Complexity

- Ackermann's function.
  - $A(i, j) = 2^j$, $i = 1$ and $j >= 1$
  - $A(i, j) = A(i-1, 2)$, $i >= 2$ and $j = 1$
  - $A(i, j) = A(i-1, A(i, j-1))$, $i, j >= 2$
- Examples
  - $A(2, 1) = A(1, 2) = 2^2$
  - $A(2, 2) = A(1, A(2, 1)) = A(1, 2^2) = 2^{2^{\wedge}2}$

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	2^1	2^2	2^3	2^4
$i = 2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	2^{2^2}	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$

Time Complexity

- Inverse of Ackermann's function
  - $\alpha(p, q) = \min\{z \geq 1 \mid A(z, p/q) > \log_2 q\}, p \geq q \geq 1$

- $\alpha(p, q) = O(\log^* q)$
  - log*n (pronounced "log star n")
  - log*n = log log log ... log n
    - How many times we have to repeatedly take log on *n* to make the value to 1?
  - $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$
  - $\log^* 2^{65536} = 5$

Time Complexity

- Ackermann's function grows very rapidly as i and j are increased.
  - $A(2,4) = 2^{65,536}$
- The inverse function grows very slowly.
  - $\alpha(p,q) < 5$ until $q = 2^{A(4,1)}$
  - $A(4,1) = A(2,16) >>>> A(2,4)$
- In the analysis of the union-find problem, q is the number, n, of elements; $p = n + f$; and $u >= n/2$.
- For all practical purposes, $\alpha(p, q) < 5$.

Time Complexity

Lemma 5.6 [Tarjan and Van Leeuwen]

Let $T(f, u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u >= n/2$.

$$k_1*(n + f*\alpha(f+n, n)) <= T(f, u) <= k_2*(n + f*\alpha(f+n, n))$$

where k_1 and k_2 are constants.

These bounds apply when we start with singleton sets and use either the weight or height rule for unions and any one of the path compression methods for a find.