Union-Find Problem

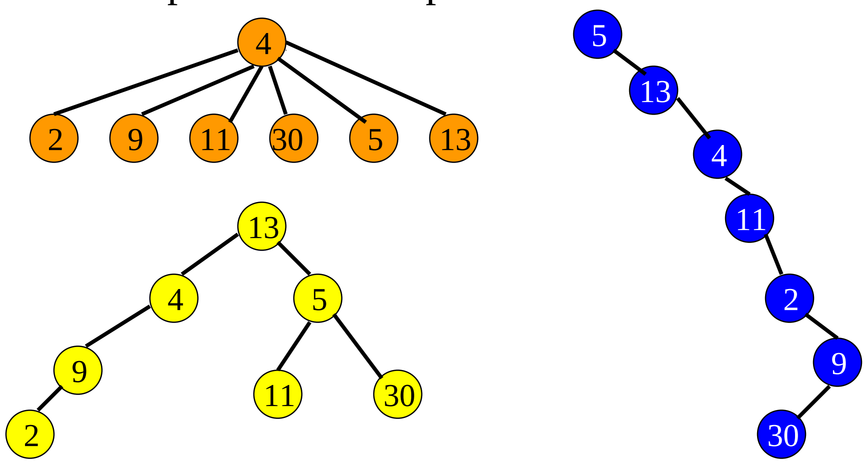

- Given a set {1, 2, …, n} of n elements.
- Initially each element is in a different set.
  - {1}, {2}, …, {n}
- An intermixed sequence of union and find operations is performed.
- A union operation combines two sets into one.
  - Each of the n elements is in exactly one set at any time.
- A find operation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

Using Arrays And Chains

- See Section 7.7 for applications as well as for solutions that use arrays and chains.
- Best time complexity obtained in Section 7.7 is O(n + u log u + f), where u and f are, respectively, the number of union and find operations that are done.
- 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).

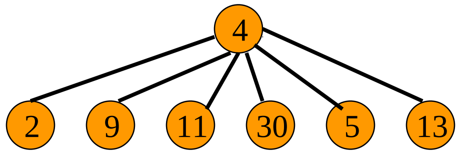
A Set As A Tree

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



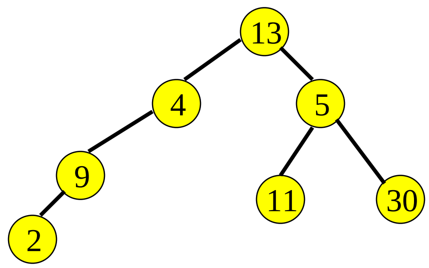
Result Of A Find Operation

- 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.



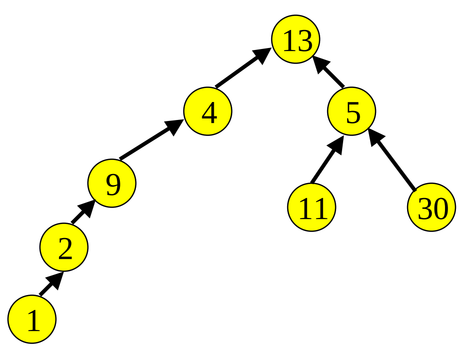
find(i) will return the element that is in the tree root.

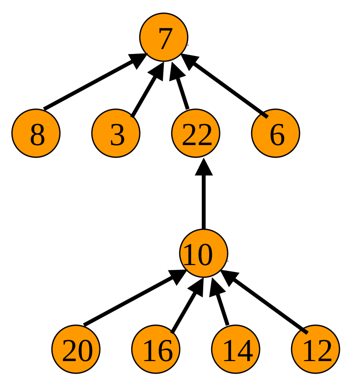
Strategy For find(i)



- 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

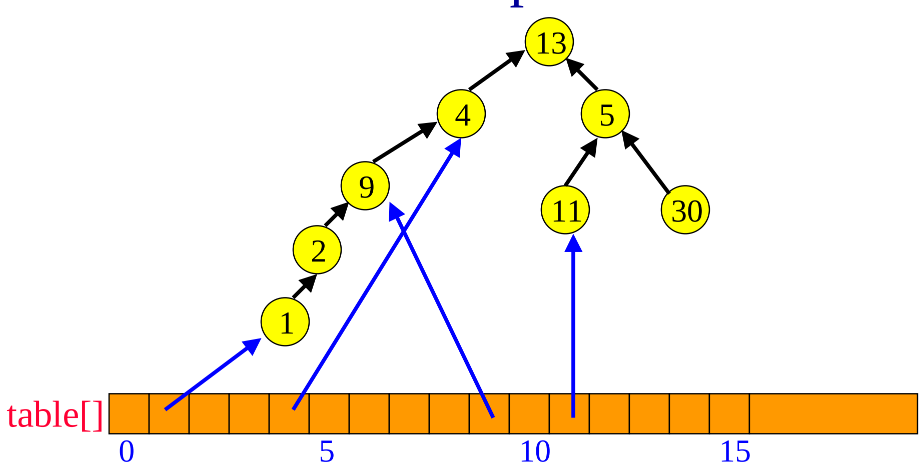




Possible Node Structure

- Use nodes that have two fields: element and parent.
  - Use an array table[] such that table[i] is a pointer to the node whose element is i.
  - To do a find(i) operation, start at the node given by table[i] and follow parent fields until a node whose parent field is null is reached.
  - Return element in this root node.

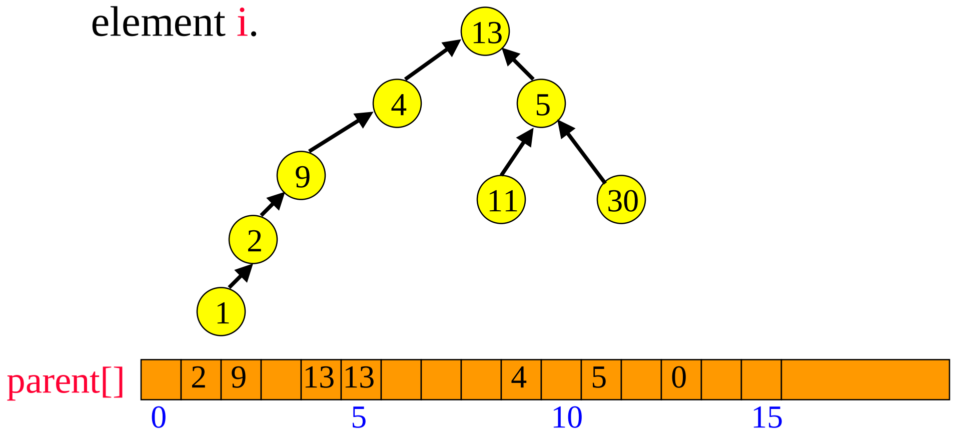
Example



(Only some table entries are shown.)

Better Representation

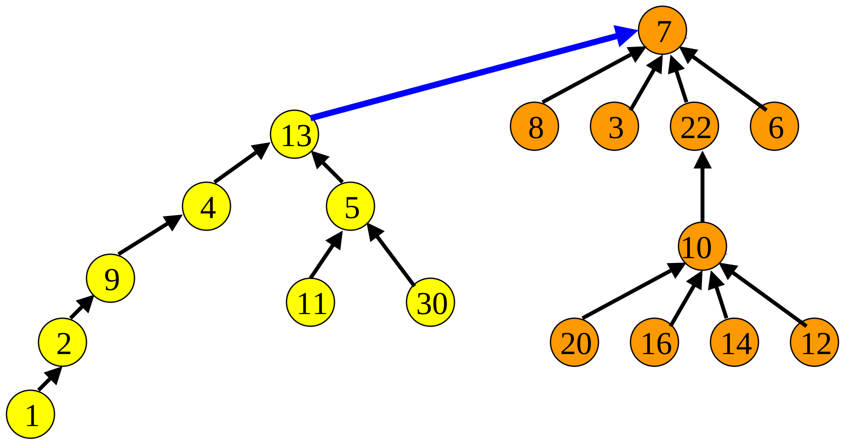
- Use an integer array parent[] such that parent[i] is the element that is the parent of element i.



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



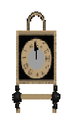
- union(7,13)

The Find Method

```
public int find(int theElement)
{
    while (parent[theElement] != 0)
      theElement = parent[theElement];  // move up
    return theElement;
}
```

The Union Method

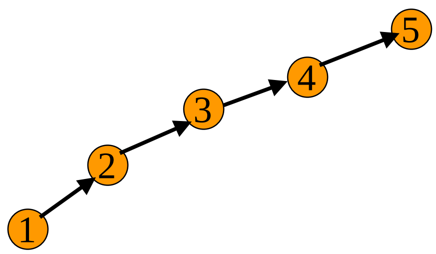
```
public void union(int rootA, int rootB)
      {parent[rootB] = rootA;}
```

Time Complexity Of union() 

- O(1)

Time Complexity of find()

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

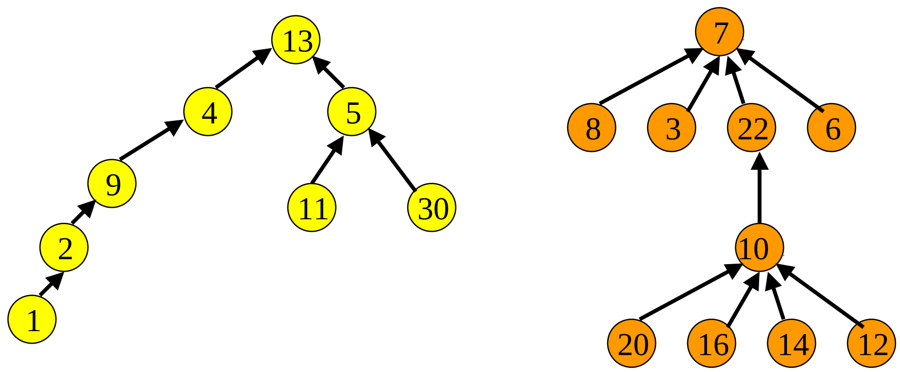


So complexity is O(u).

u Unions and f Find Operations

- O(u + uf) = O(uf)
- Time to initialize parent[i] = 0 for all i is O(n).
- Total time is O(n + uf).
- Worse than solution of Section 7.7!
- Back to the drawing board.

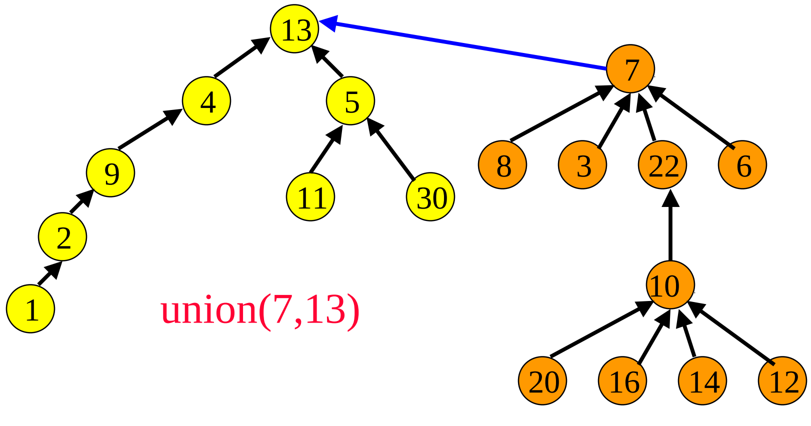
Smart Union Strategies



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

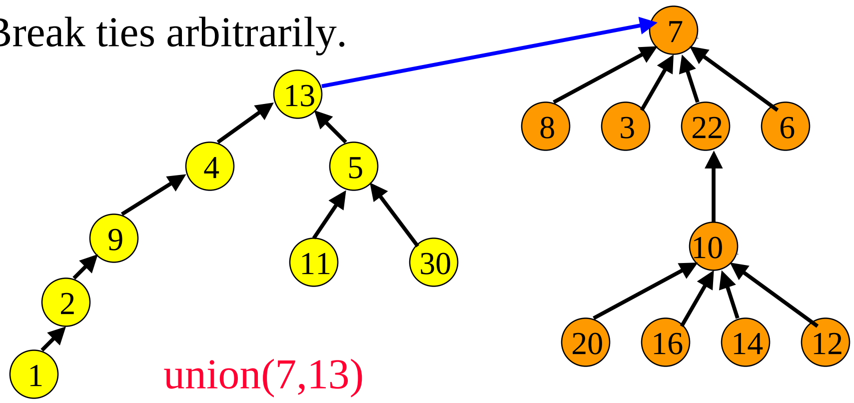
Height Rule

- 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.



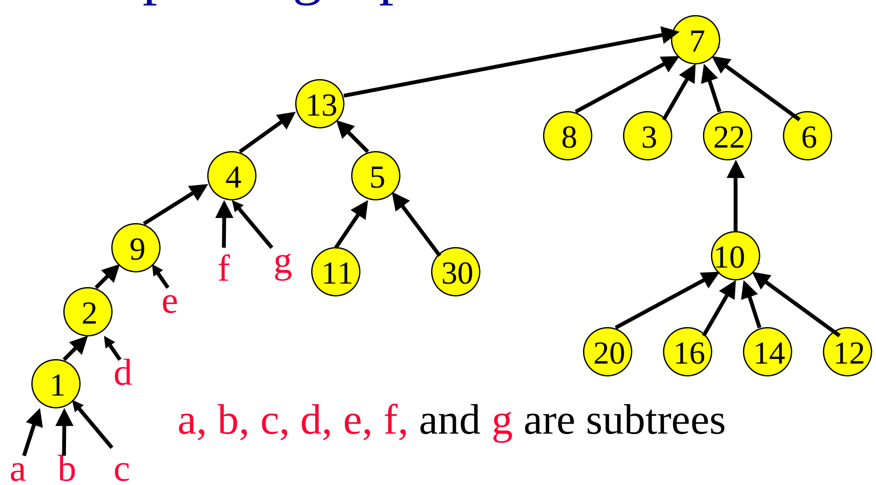
Implementation

- 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.

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 p elements is at most floor $(\log_2 p) + 1$.
- Proof is by induction on p. See text.

Sprucing Up The Find Method

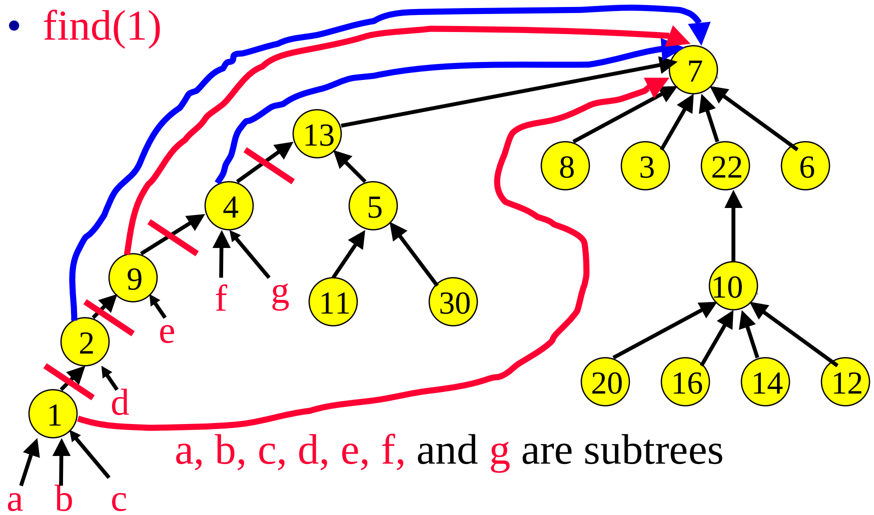


a, b, c, d, e, f, and g are subtrees

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

Path Compaction

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

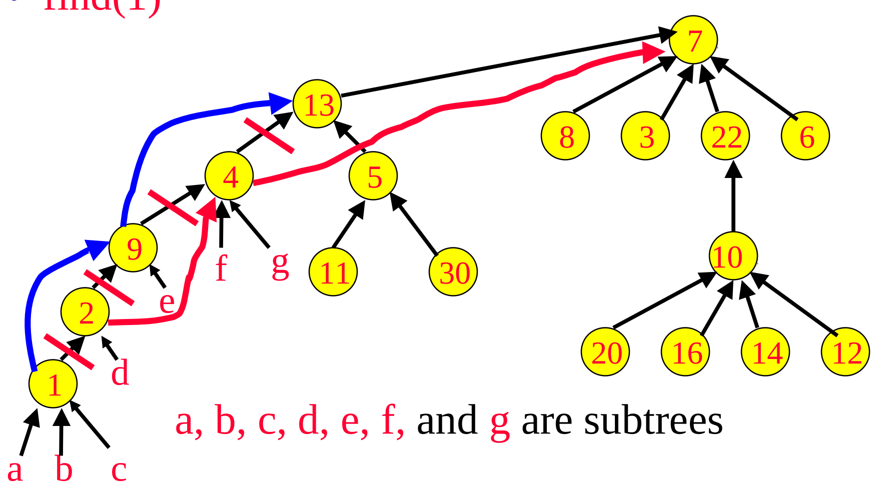


a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree.

Path Splitting

- Nodes on find path point to former grandparent.
- find(1)

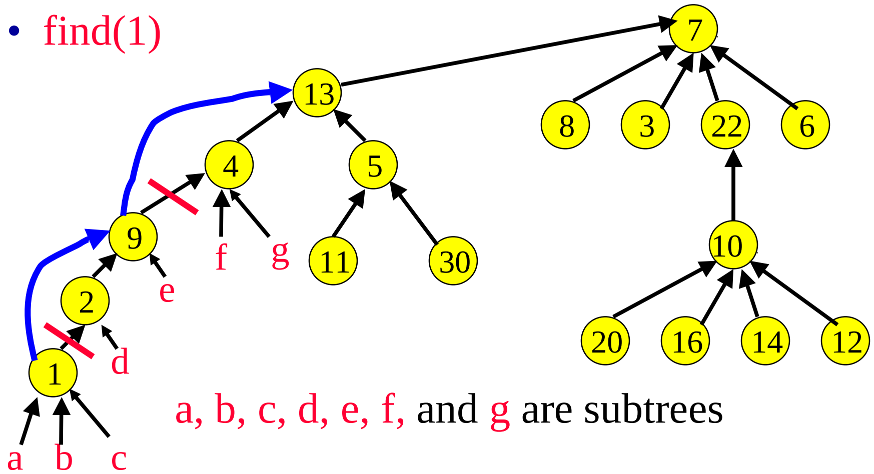


a, b, c, d, e, f, and g are subtrees

Makes only one pass up the tree.

Path Halving

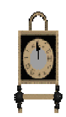
- Parent pointer in every other node on find path is changed to former grandparent.
- find(1)



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

Changes half as many pointers.

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

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

Theorem 12.2 [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.

a*(n + f*alpha(f+n, n)) <= T(f,u) <= b*(n + f*alpha(f+n, n))

where a and b 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.