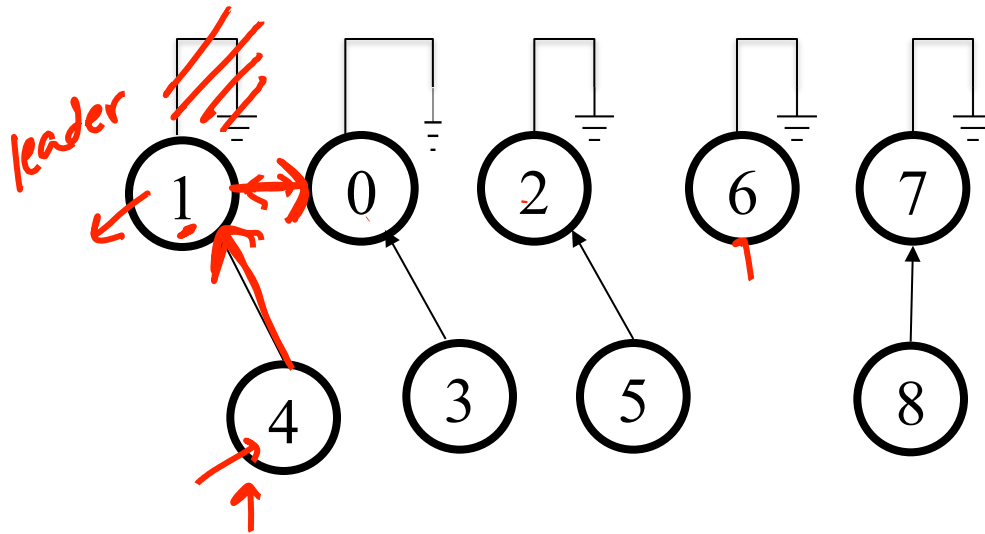


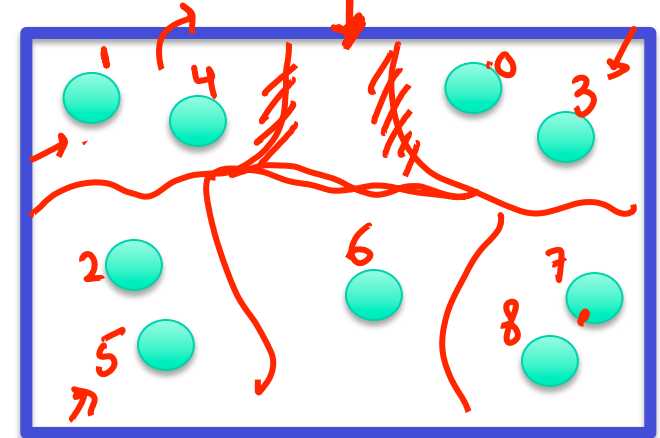
CSE 100

Disjoint Set, Union Find

Union-find using up-trees



The union-find data structure



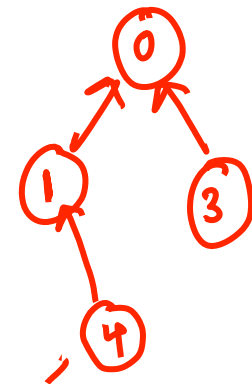
Perform these operations:

Find(4) ~~1~~

Find(3) ~~0~~

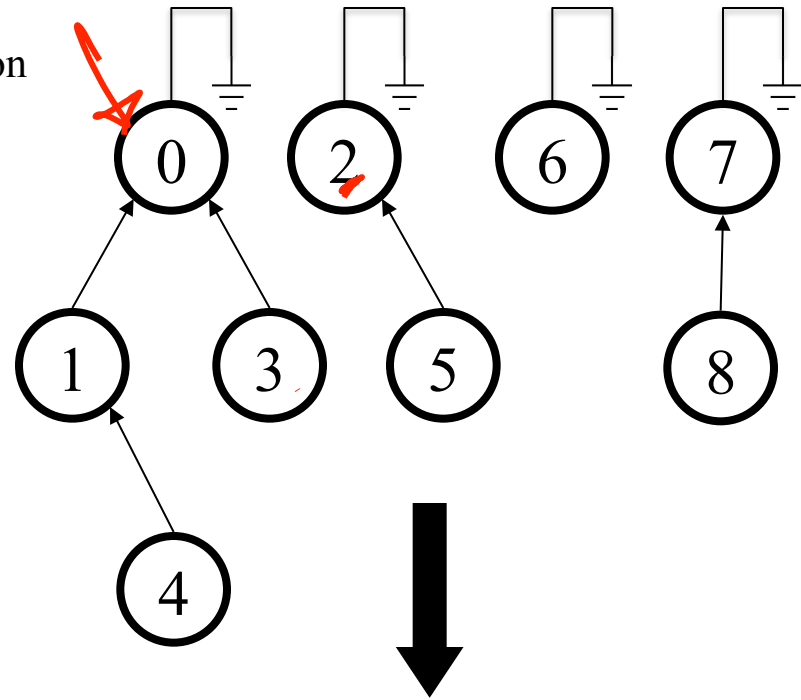
Union(1, 0) ~~0~~

Find(4) ~~0~~

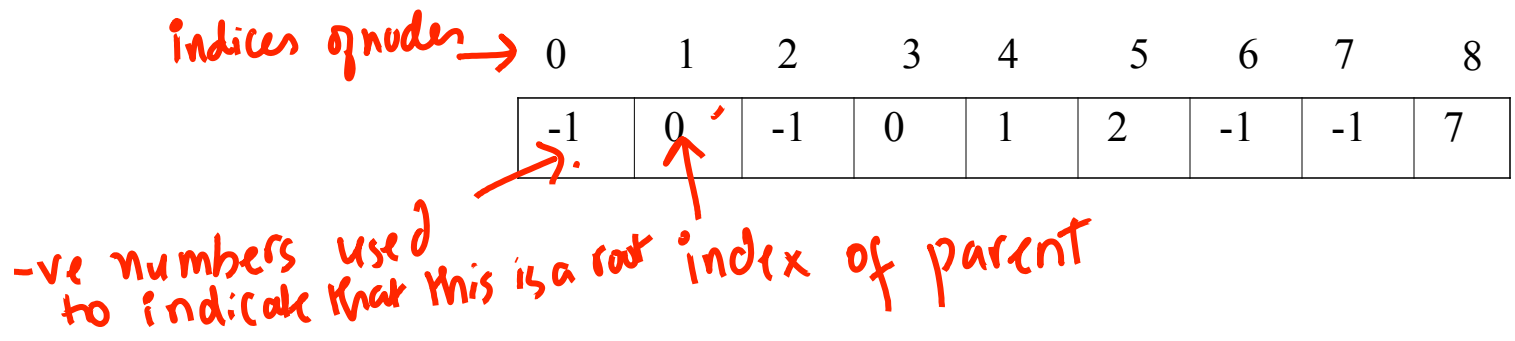


Array representation of Up-trees

- A compact and elegant implementation
- Each entry is the up index
- -1 for the roots
- Write the forest of trees, showing parent pointers and node labels, represented by this array



Find(4) = 0



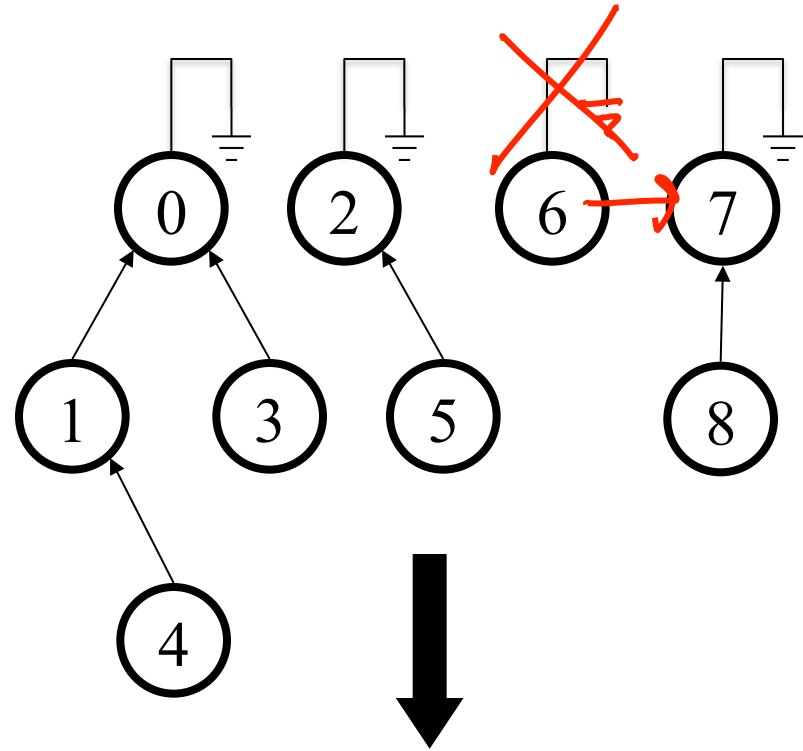
Performing a union operation

Union(6,7)

Fill in 6, 7 and 8 in the array

union(6,7)

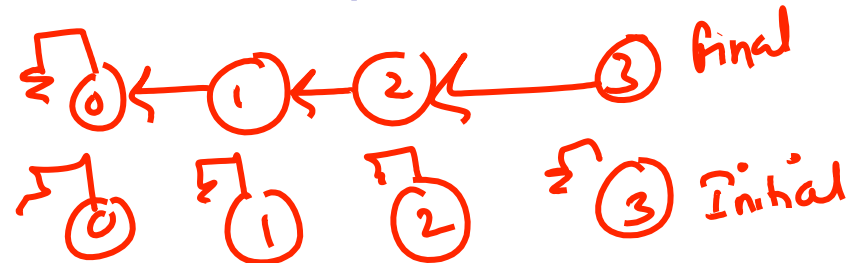
*make 7 the parent of
6 (this choice right now
is arbitrary)*



0	1	2	3	4	5	<u>6</u>	7	8
-1	0	-1	0	1	2	-1	-1	7
-1	0	-1	0	1	2	7	-1	7

Disjoint subsets using up-trees (Simple Union and Find)

Start with 4 items: 0, 1, 2, 3



Suppose **Union**(*i*, *j*) makes the node **Find**(*i*) the parent of the node **Find**(*j*)

Perform these operations:

0	1	2	3
-1	-1	-1	-1

Union(2, 3)

Union(1, 2)

Find(0) =

Find(3) =

Union(0, 1)

Find(1) =

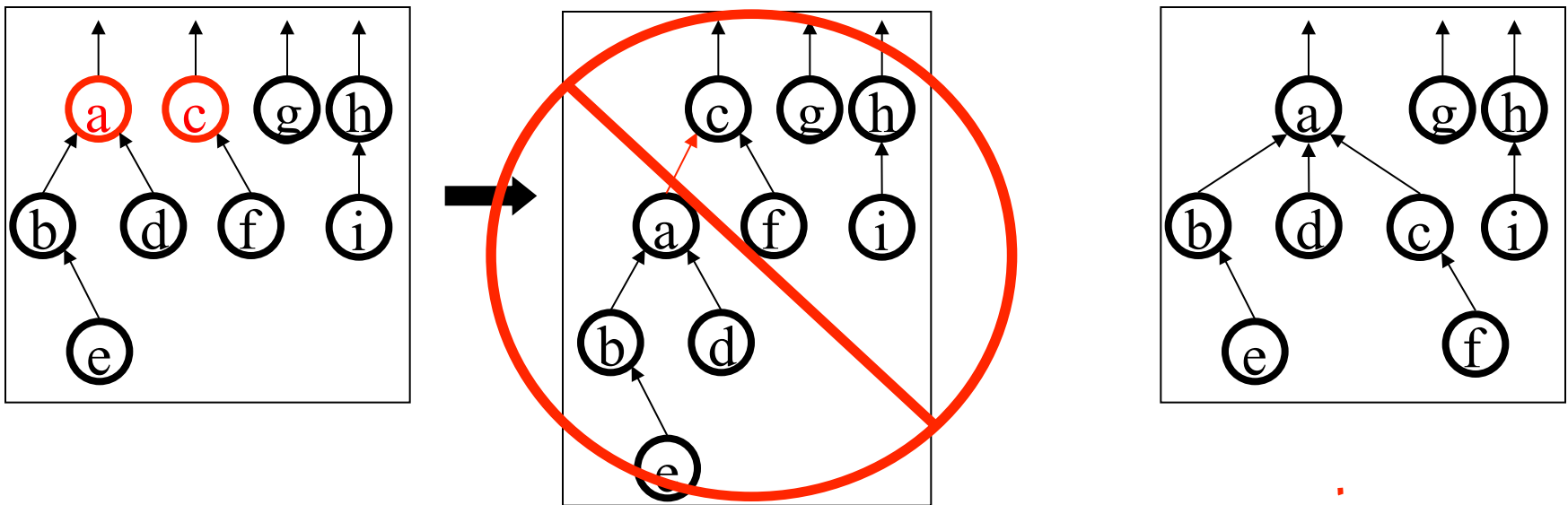
Run time of Simple union and find (using up-trees)

If we have no rule in place for performing the union of two up trees, what is the worst case run time of find and union operations in terms of the number of elements N ?

- A. find: $O(N)$, union: $O(1)$
- B. find: $O(1)$, union: $O(N)$
- C. find: $O(\log_2 N)$, union: $O(1)$
- D. find: $O(1)$, union: $O(\log_2 N)$

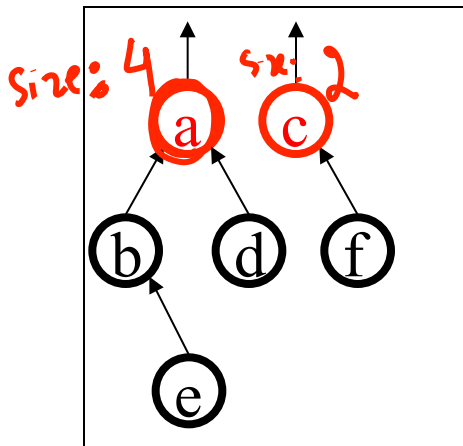
Improving Union operations

- In the Union operation, a tree becomes like a linked list if, when joining two trees, the root of the smaller tree (e.g. a single node) always becomes the root of the new tree
- We can avoid this by making sure that in a Union operation, the *larger* of the two trees' roots becomes the root of the new tree (ties are broken arbitrarily)

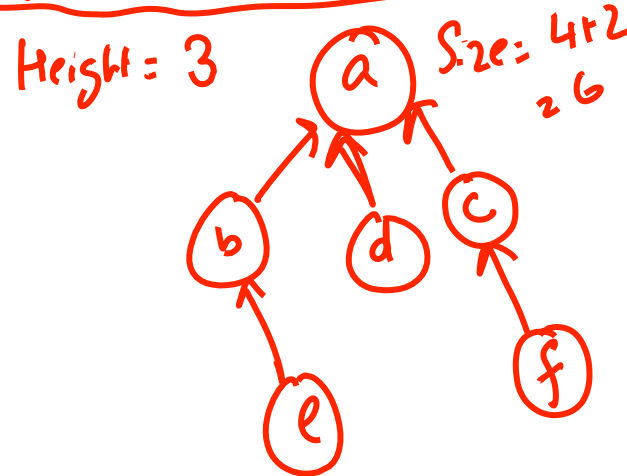


Smarter Union operations

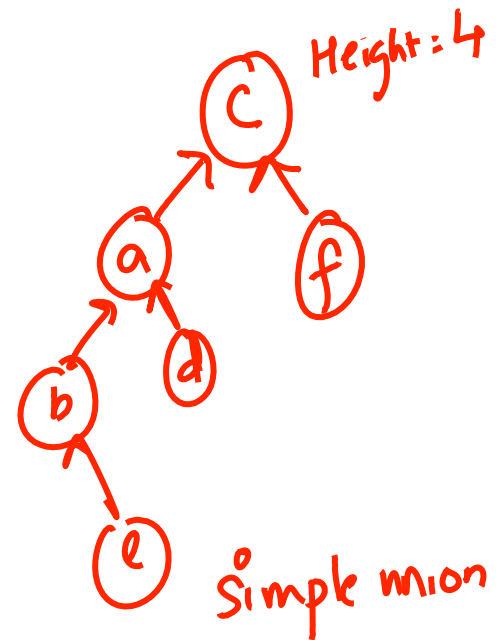
- Avoid this problem by having the *larger* of the 2 trees become the root
 - **Union-by-size**
 - Each root stores the size (# nodes) of its respective tree
 - The root with the larger size becomes the parent
 - Update its size = sum of its former size and the size of its new child
 - Break ties arbitrarily



Union by size
results in Smaller height



union by size



Simple union

Disjoint subsets using trees (Union-by-size and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that **Union**(*i*, *j*) makes the root of the smaller tree a child of the root of the larger tree.



Perform these operations:

Union(2, 3)

Union(1, 2)

Find(0) 0

Find(3) 2

Union(0, 2)

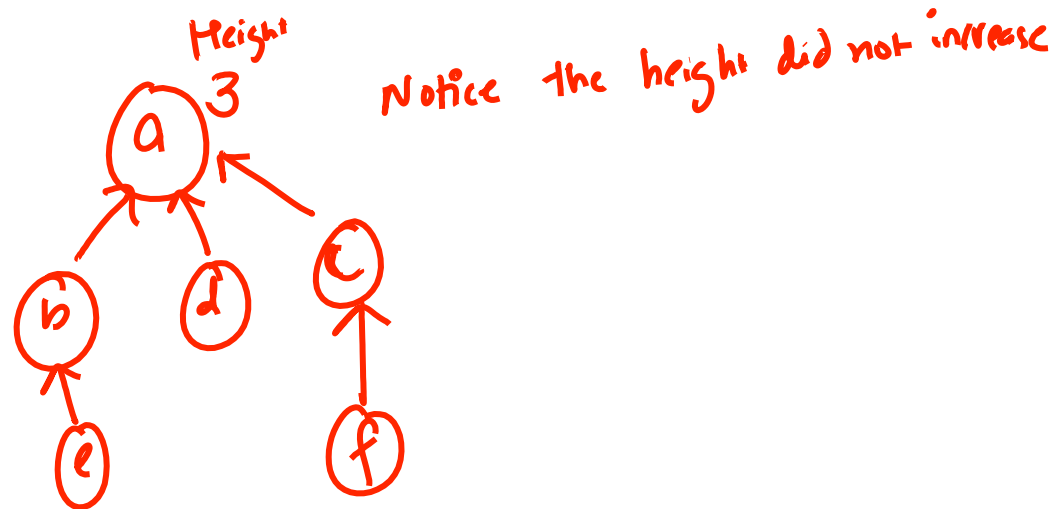
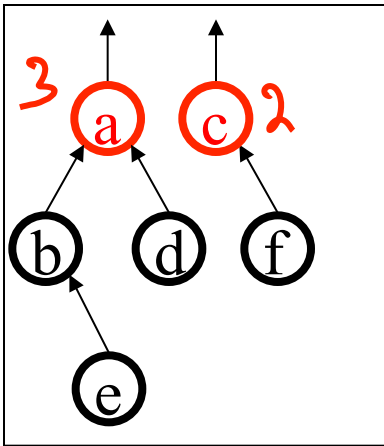
Find(1) 2

Find(0) 2

	0	1	2	3
Initial	-1	-1	-1	-1
After Union(2, 3)	-1	-1	-2	2
After Union(1, 2)	-1	2	-3	2
After Union(0, 2)	2	2	-4	2

Smarter Union operations

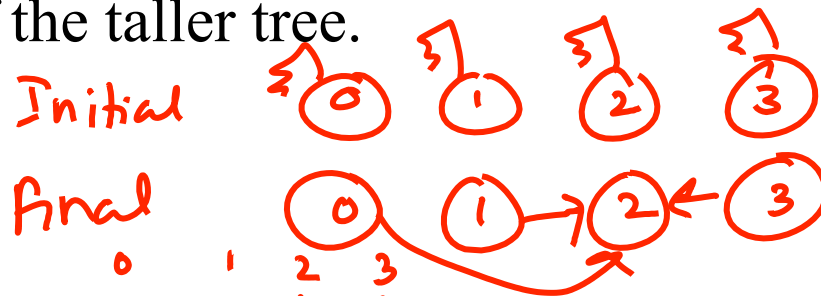
- Avoid this problem by having the *larger* of the 2 trees become the root
 - **Union-by-size**
 - **Union-by-height (also called union by rank)**
 - Each root stores the height of its respective tree
 - If one root has a greater height than the other, it becomes the parent. Its stored height doesn't need to be updated
 - If the roots show equal height, pick either one as the parent
 - Its stored height should be increased by one
 - Break ties arbitrarily



Disjoint subsets using trees (Union-by-height and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that **Union**(i, j) makes the root of the shorter tree a child of the root of the taller tree.



Perform these operations:

Union(2, 3)

Union(1, 2)

Find(0) 0

Find(3) 2

Union(0, 2) =

Find(1) 2

Find(0) 2

0 1 2 3
-1 -1 -1 -1

-1 -1 -2 2
-1 2 (-2) 2

2 2 -2 2

Notice height of 2 is unchanged

Cost of disjoint subsets operations with smarter Union and simple Find

Either union-by-size or union-by-height will guarantee that the height of any tree is no more than $\log_2 N$, where N is the total number of nodes in all trees

union by size

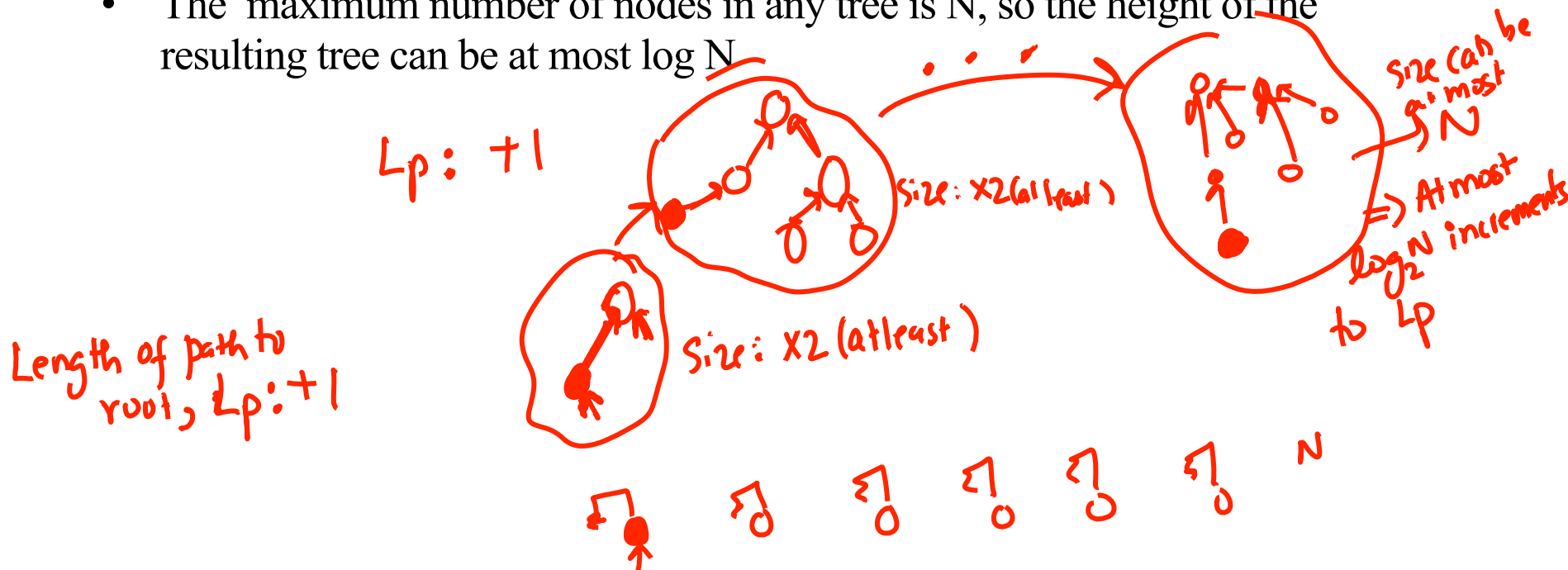
Q: What is the worst case run time of find and union operations in terms of the number of elements N ?

- A. find: $O(N)$, union: $O(1)$
- B. find: $O(1)$, union: $O(N)$
- ☒ C. find: $O(\log_2 N)$, union: $O(1)$
- D. find: $O(1)$, union: $O(\log_2 N)$

Solo vote!

Bounding the height of the up-tree using union by size.

- Initially all the nodes are in singleton trees (with height 1)
- Take the perspective of a single node.
- The only time the height of the tree which the node is part of increases by 1 is when the node joins a larger group i.e. if the height of the node's up-tree increases by 1, the number of nodes in that tree would have at least doubled
- The maximum number of nodes in any tree is N , so the height of the resulting tree can be at most $\log N$



Cost of disjoint subsets operations with smarter Union and simple Find

Q: What is the worst case run time of find and union operations in terms of the number of elements N ?

A. find: $O(N)$, union: $O(1)$

Discuss and revote

B. find: $O(1)$, union: $O(N)$

 C. find: $O(\log_2 N)$, union: $O(1)$

D. find: $O(1)$, union: $O(\log_2 N)$

- Therefore, doing $N-1$ union operations (the maximum possible) and M find operations takes time $O(N + M \log_2 N)$ worst case
- With simple unions the complexity was:
- This is a big improvement; but we can do still better, by a slight change to the Find operation:
adding *path compression* (next lecture)

Kruskal's algorithm run time with smart union path compression find:

1. Sort edges in increasing order of cost

$$O(|E| \log |E|)$$

2. Set of edges in MST, $T = \{\}$

3. For $i = 1$ to $|E|$ |E| iterations

3a. $S1 = \text{find}(u); S2 = \text{find}(v)$

$$O(\log |V|) \xrightarrow{\text{total}} O(|E| \log |V|)$$

3b. if $(S1 \neq S2) \{ // \text{If } T \cup \{e_i = u, v\} \text{ has no cycles}$

$$\longrightarrow O(|E|)$$

Add e_i to T

$$O(1)$$

union($S1, S2$)

$$O(1)$$

total

$$O(|V|)$$

$$O(|E| \log |E| + |E| \log |V| + |E| + |V|) = O(|E| \log |V| + |V|) = O(|E| \log |V|)$$

|V| < |E| for connected graphs

What is the improvement from simple union find?

Union-Find Code

- Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT's methods is compact

```
class DisjSets{
    int *array;

    /**
     * Construct the disjoint sets object
     * numElements is the initial number of disjoint sets
     */
    DisjSets( int numElements )

        { array = new int [ numElements ];
          for( int i = 0; i < numElements; i++ )
              array[ i ] = -1;
          }
}
```

Union-by-height

```
/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set labels.
 * root1  is the root of set 1
 * root2  is root of set 2
 * returns the root of the union
 */
int union (      int  root1, int  root2 ){

    if( array[ root2 ] < array[ root1 ] )
    {
        array[ root1 ] = root2;      // root2 is higher
        return root2;                // Make root2 new root
    } else {
        if( array[ root1 ] == array[ root2 ] )
            array[ root1 ]--;        // Update height if same
        array[ root2 ] = root1;      // Make root1 new root
        return root1;
    }
}
```