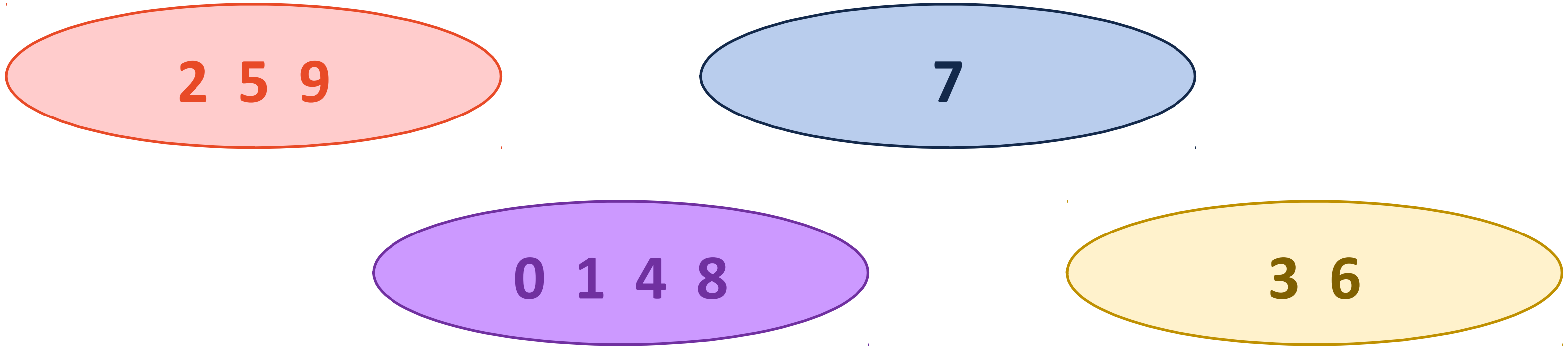
CS 400

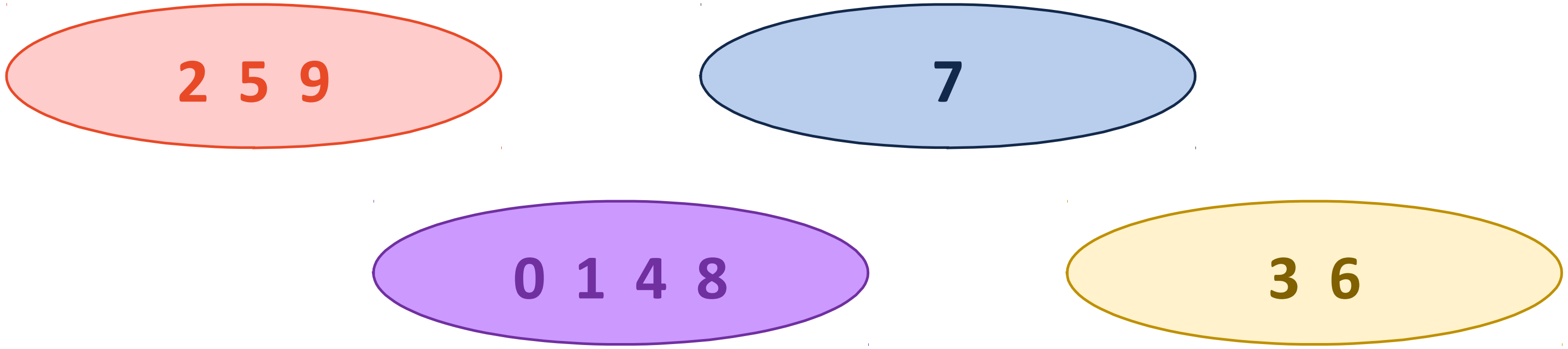
Disjoint Sets

ID: 11-01

Disjoint Sets

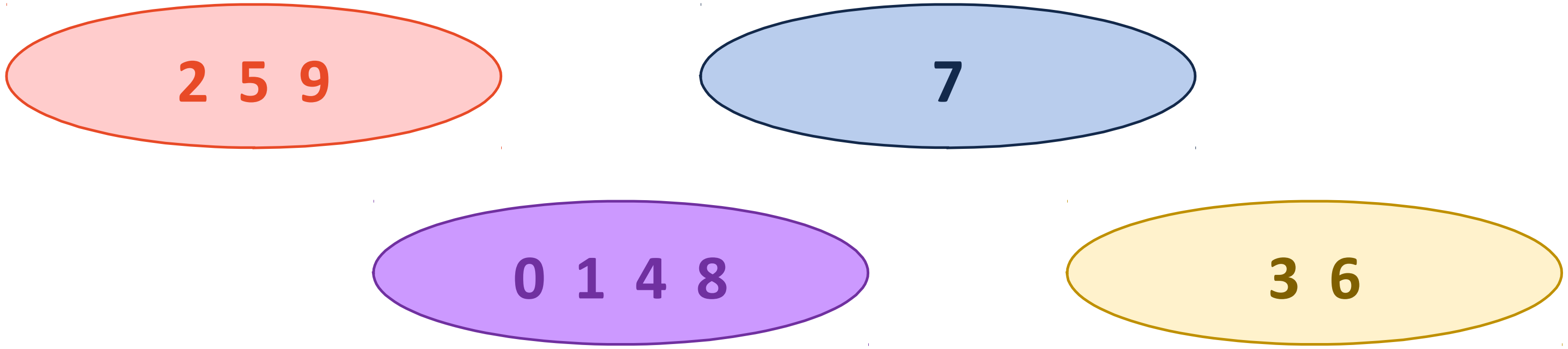


Disjoint Sets



Operation: find(4)

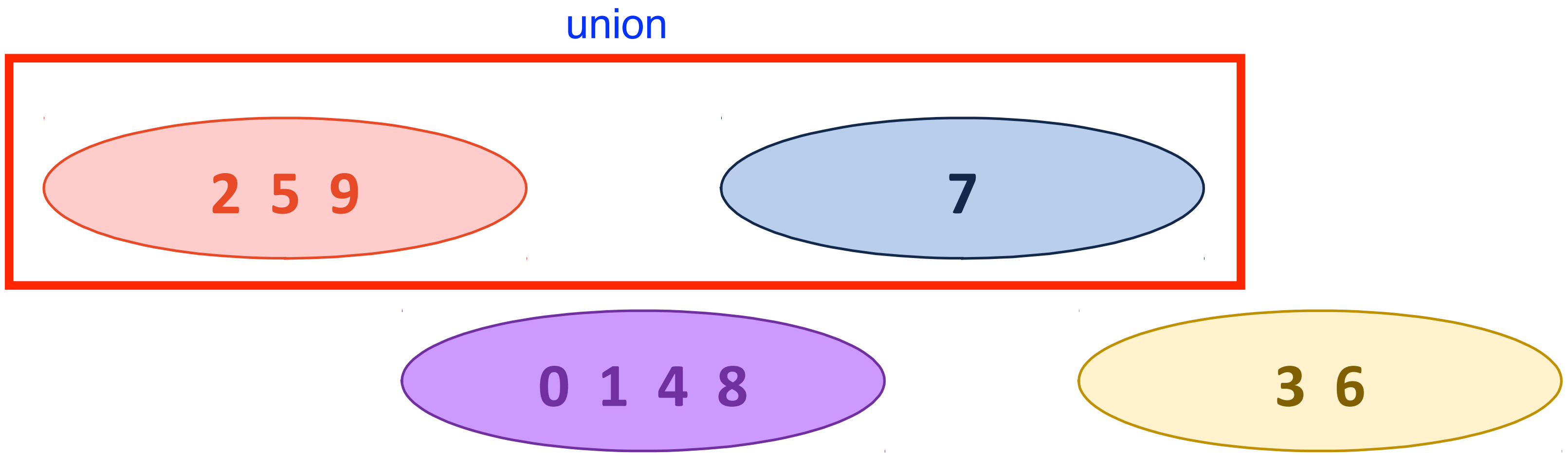
Disjoint Sets



Operation: find(4) == find(8)

0 0

Disjoint Sets



Operation:

```
if ( find(2) != find(7) ) {
    union(   find(2), find(7)    );
}
```

Disjoint Sets ADT

- Maintain a collection S = {s_0, s_1, … s_k}

- Each set has a representative member.

- API:
```
void makeSet(const T & t);
void union(const T & k1, const T & k2);
T & find(const T & k);
```

CS 400

Disjoint Sets: Implementation #1

ID: 11-02

Implementation #1

0 1 4

2 7

3 5 6

0	1	2	3	4	5	6	7
0	0	2	3	0	3	3	2

O(1)

Find(k):

find(4) == 0

Union(k1, k2):

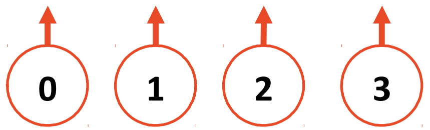
CS 400

Disjoint Sets: UpTrees

ID: 11-03

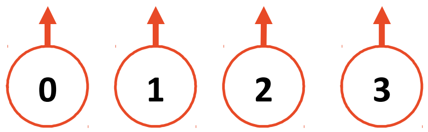
Implementation #2

- We will continue to use an array where the index is the key
- The value of the array is:
  - **-1,** if we have found the representative element
  - **The index of the parent**, if we haven't found the rep. element
- We will call theses **UpTrees**:



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

UpTrees



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

1 U 2

0	1	2	3
-1	-1	1	0

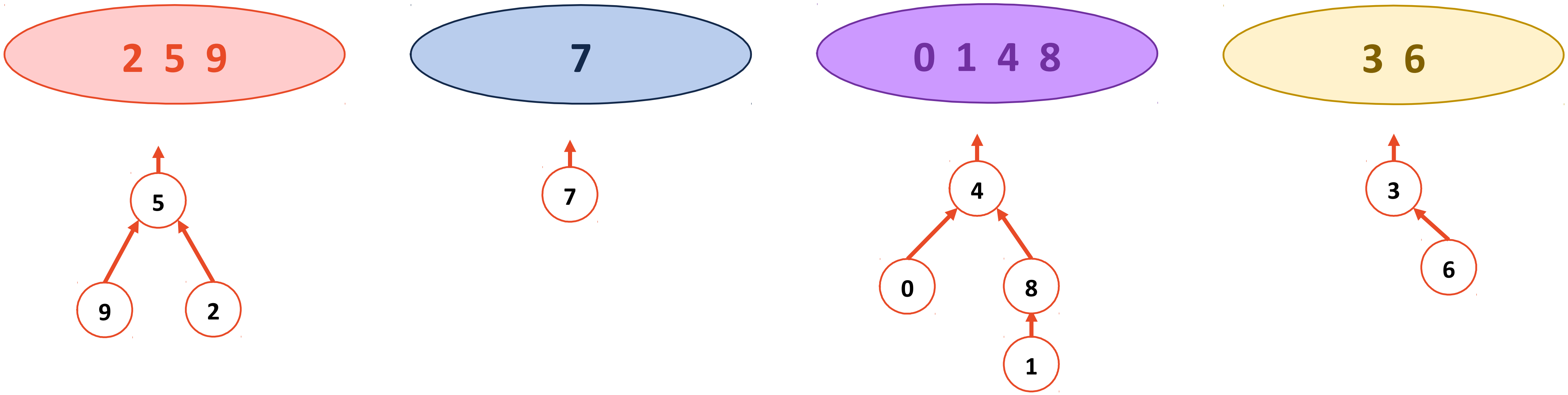
0 U 3

0	1	2	3
-1	-1	-1	0

0 U 1

0	1	2	3
-1	0	1	0

Disjoint Sets



0	1	2	3	4	5	6	7	8	9
4	8	5	6 -1	-1	-1	-1 3	-1	4	5

CS 400

UpTrees: Simple Running Time

ID: 11-04

Disjoint Sets Find

```
int DisjointSets::find() {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

Running time?

O(h) <= O(n)

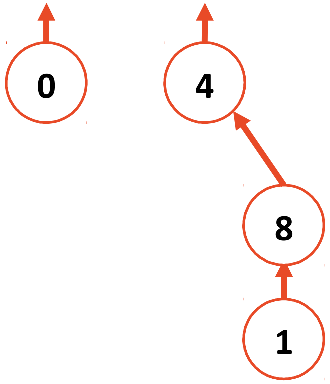
What is the ideal UpTree?

All the children point to the identity node

Disjoint Sets Union

```
void DisjointSets::union(int r1, int r2) {


}
```



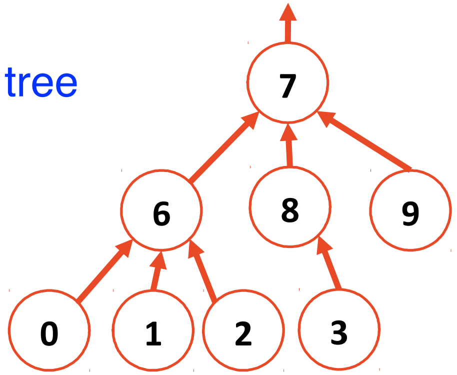
CS 400

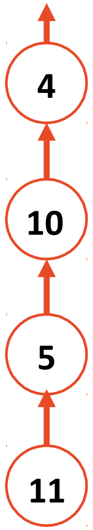
UpTrees: Smart Union and Path Compression

ID: 11-05

Disjoint Sets – Union

close to an ideal tree





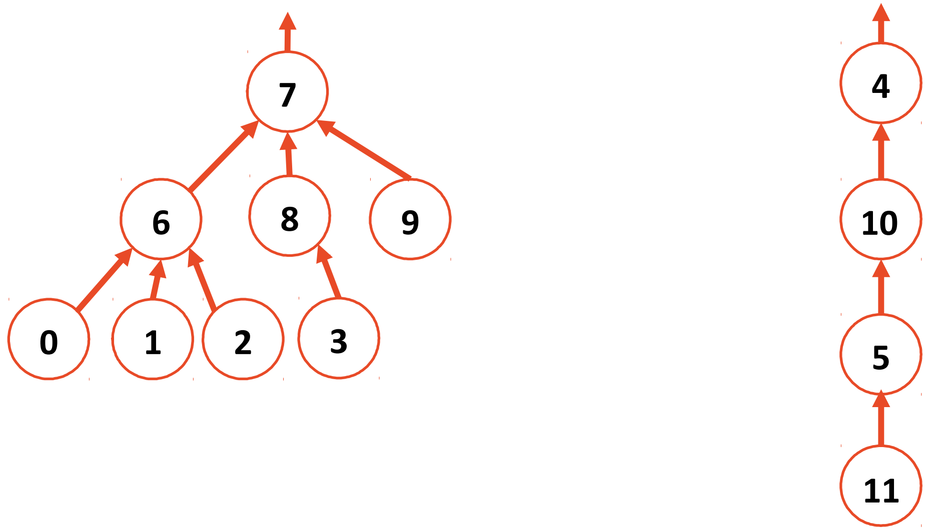
worst possible tree
(a linked list)

instead of representing the root as -1, use -h-1

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5
				-4			-3				

Union by height to prevent limit the growth in height

Disjoint Sets – Smart Union

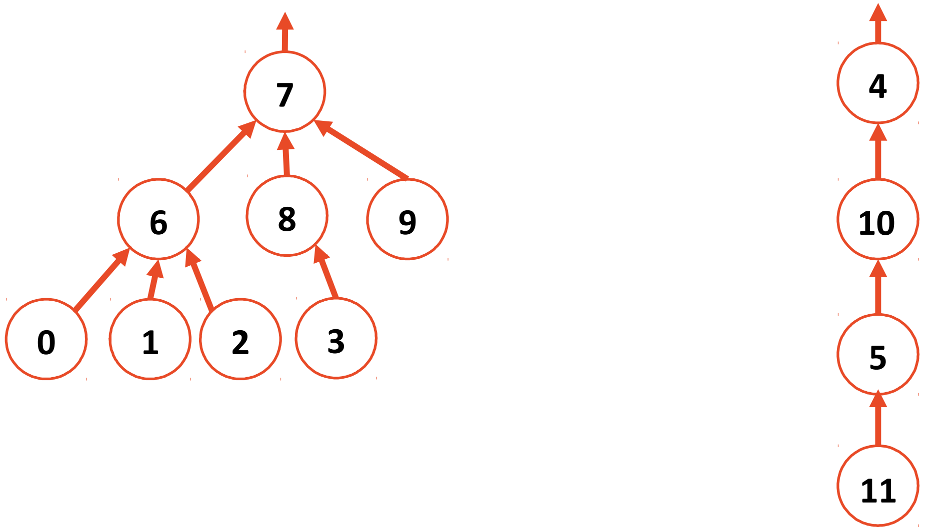


Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Disjoint Sets – Smart Union



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-8	7	7	4	5

***Idea**: Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is: ___log n___.

Disjoint Sets Find

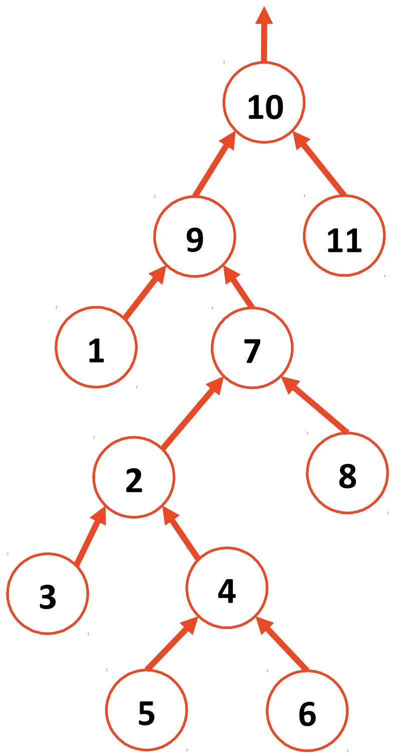
```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Path Compression



point the child nodes to the root as much as possible, especially the ones that have the longest paths

Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

$\log^*(n) =$

$0 \quad , n \leq 1$

$1 + \log^*(\log(n)) \; , n > 1$

What is $\mathbf{lg^*(2^{65536})}$?   5

Disjoint Sets Analysis

In an Disjoint Sets implemented with smart **unions** and path compression on **find**:

Any sequence of **m union** and **find** operations result in the worse case running time of O(m log*(n)______________),

where **n** is the number of items in the Disjoint Sets.

~O(1)