Today's announcements:

MP6 available, due 11/15, 11:59p.

Remembering CS173…

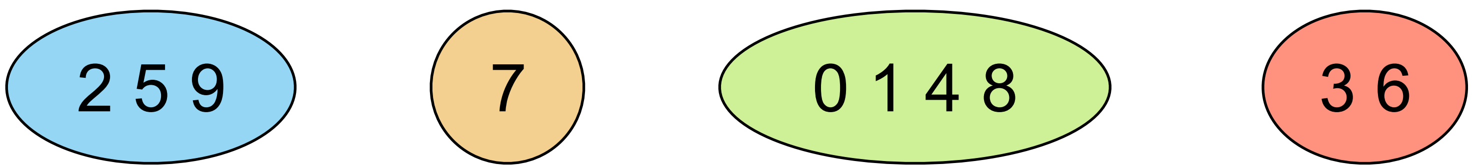
Let *R* be an equivalence relation on the set of students in this room, where $(s,t) \in R$ if *s* and *t* have the same favorite among {A, FB, TR, CC, PMC, ____}.

Notation from math: $[\ \_\_\_\ ]_R = \{x : xR\_\_\_\}$

One big goal for us: Given s and t we want to determine if sRt.

A Disjoint Sets example:

Let R be an equivalence relation on the set of students in this room, where $(s,t) \in R$ if s and t have the same favorite among A, FB, TR, CC, PMC, ____}.



1. Find(4)
2. Find(4)==Find(8)
3. If (!(Find(7)==Find(2)) then Union(Find(7),Find(2))

Disjoint Sets ADT

We will implement a data structure in support of “Disjoint Sets”:

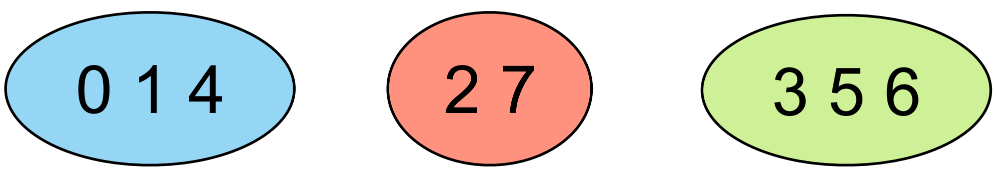
- Maintains a collection $S = \{s_0, s_1, \ldots s_k\}$ of disjoint sets.
- Each set has a representative member.
- Supports functions:

```
void MakeSet(const T & k);

void Union(const T & k1, const T & k2);

T & Find(const T & k);
```

A first data structure for Disjoint Sets:



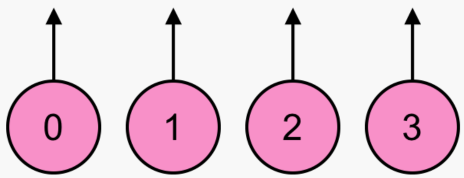
0	1	2	3	4	5	6	7
0	0	2	3	0	3	3	2

Find:

Union:

A better data structure for Disjoint Sets: UpTrees

- if array value is -1, then we’ve found a root, o/w value is index of parent.
- x and y are in the same tree iff they are in the same set.



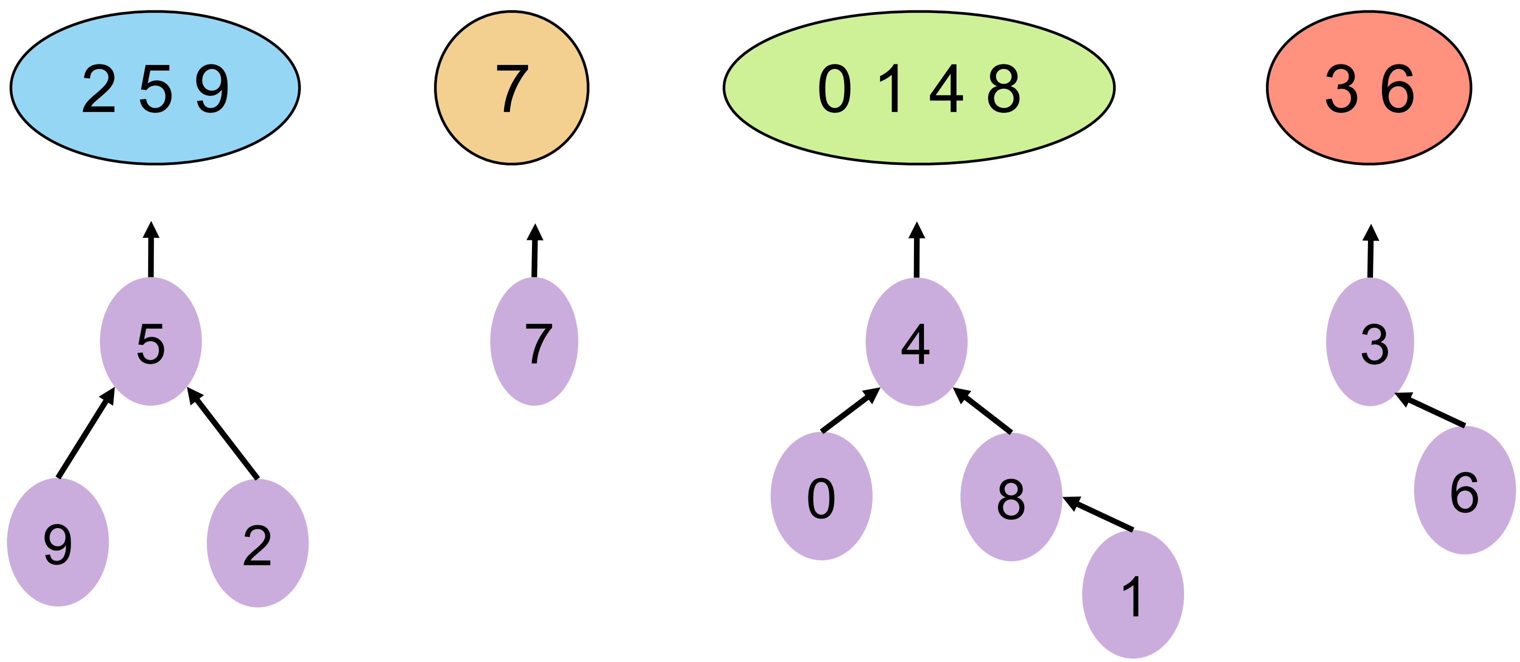
0	1	2	3
-1	-1	-1	-1

0	1	2	3

0	1	2	3

0	1	2	3

Let *R* be an equivalence relation on the set of students in this room, where *(s,t)* $\in$ *R* if *s* and *t* have the same favorite among {AB, FN, DJ, ZH, FB}.



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

A better data structure for Disjoint Sets:

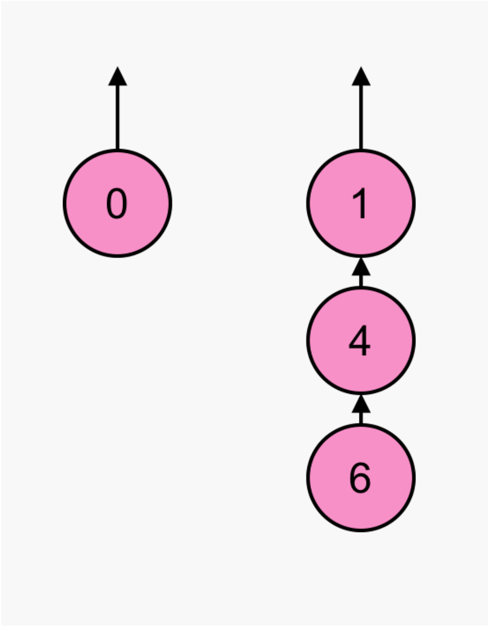
```
int DS::Find(int i) {
   if (s[i] < 0) return i;
   else return Find(s[i]);
}
```

Running time depends on ___________.

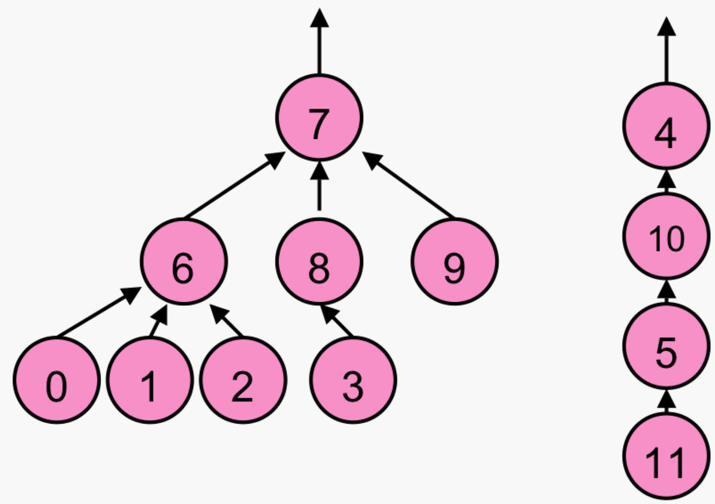
Worst case?

What's an ideal tree?

```
void DS::Union(int root1, int root2) {
   ________________________;
}
```



Smart unions:



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

Keeps overall height of tree as small as possible.

Union by size:

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

Increases distance to root for fewest nodes.

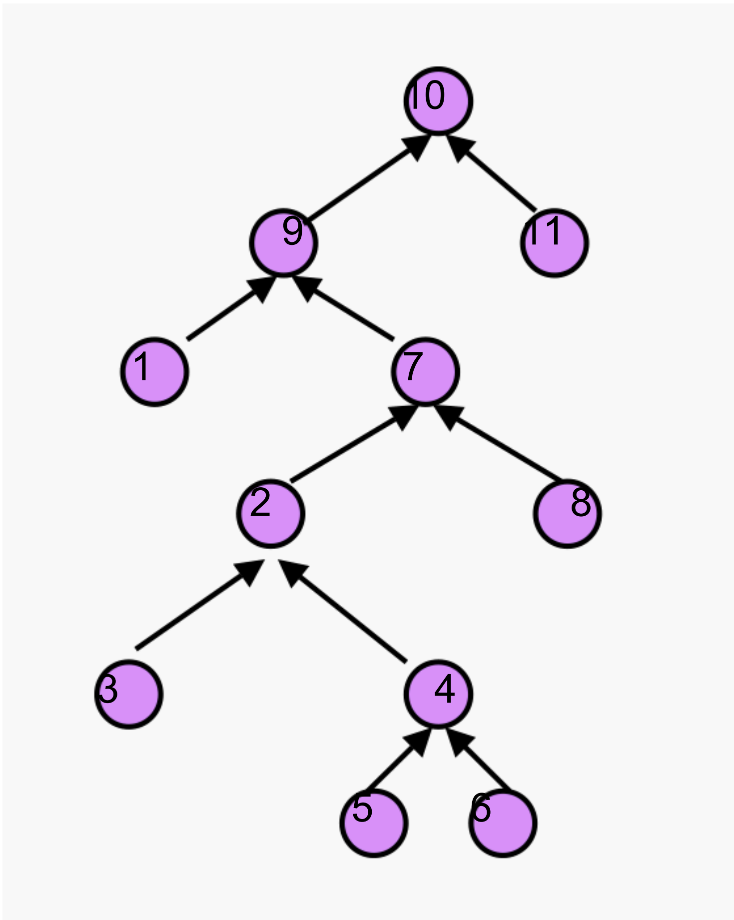
Both of these schemes for Union guarantee the height of the tree is ___________.

Smart unions:

```
int DS::Find(int i) {
   if (s[i] < 0) return i;
   else return Find(s[i]);
}
```

```
void DS::UnionBySize(int root1, int root2) {
   int newSize = s[root1]+s[root2];
   if (isBigger(root1,root2)) {
      s[root2]= root1;
      s[root1]= newSize;
   }
   else {
      s[root1] = root2;
      s[root2]= newSize;
   }
}
```

Path Compression:



Path Compression:

```
int DS::Find(int i) {
   if (s[i] < 0) return i;
   else return      Find(s[i]);
}
```

```
void DS::UnionBySize(int root1, int root2) {
   int newSize = s[root1]+s[root2];
   if (isBigger(root1,root2)) {
      s[root2]= root1;
      s[root1]= newSize;
   }
   else {
      s[root1] = root2;
      s[root2]= newSize;
   }
}
```

Analysis:

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Example:

2^{65536}

Relevant result:

In an upTree implementation of Disjoint Sets using smart `union` and path compression upon `find`...

any sequence of *m* `union` and `find` operations results in worst case running time of O(____________), where *n* is the number of items.

http://research.cs.vt.edu/AVresearch/UF/