CMSC 341
Lecture 20 Disjointed Sets

Prof. John Park

Based on slides from previous iterations of this course

Introduction to Disjointed Sets

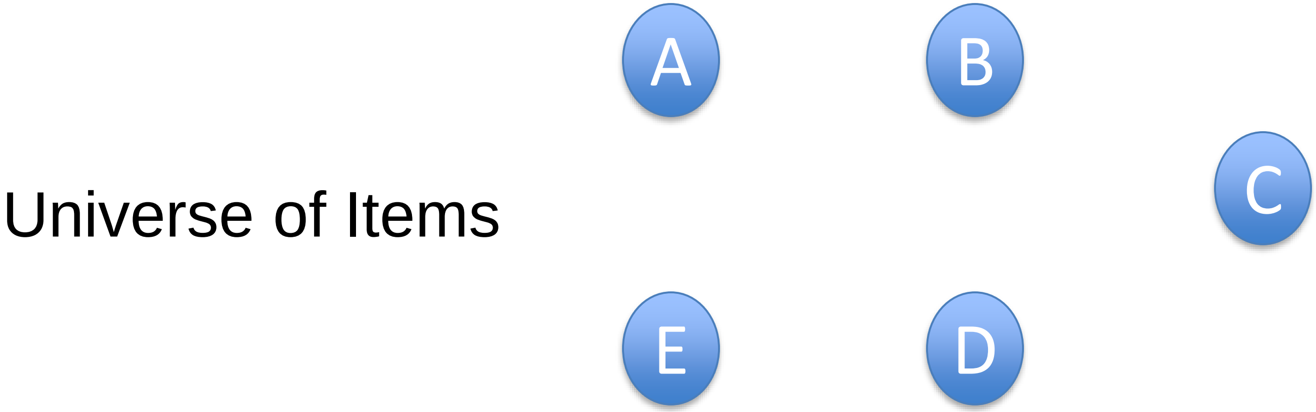
Disjoint Sets

- A data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets

From: https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Universe of Items

- Universal set is made up of all of the items that can be a member of a set

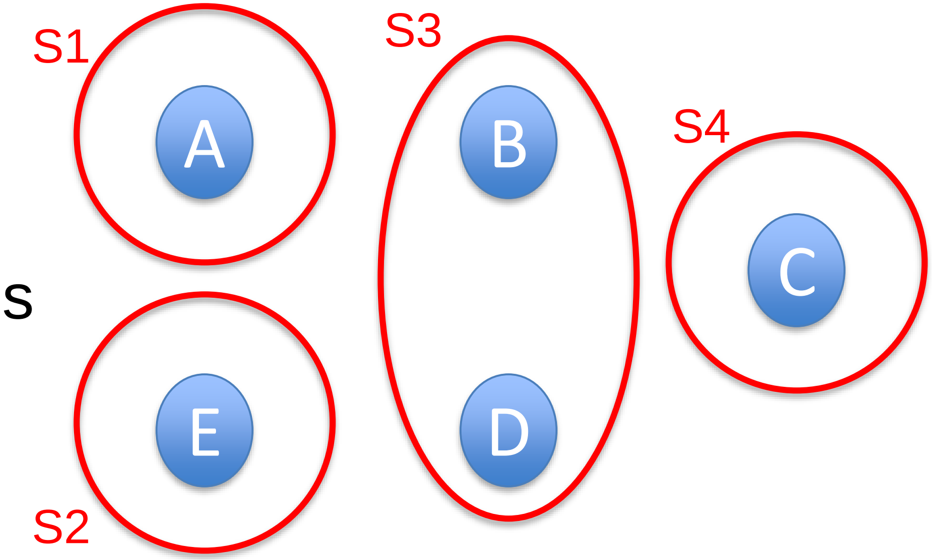


From: https://www.youtube.com/watch?v=UBY4sF86KEY

Disjoint Sets

- A group of sets where no item can be in more than one set

Universe of Items

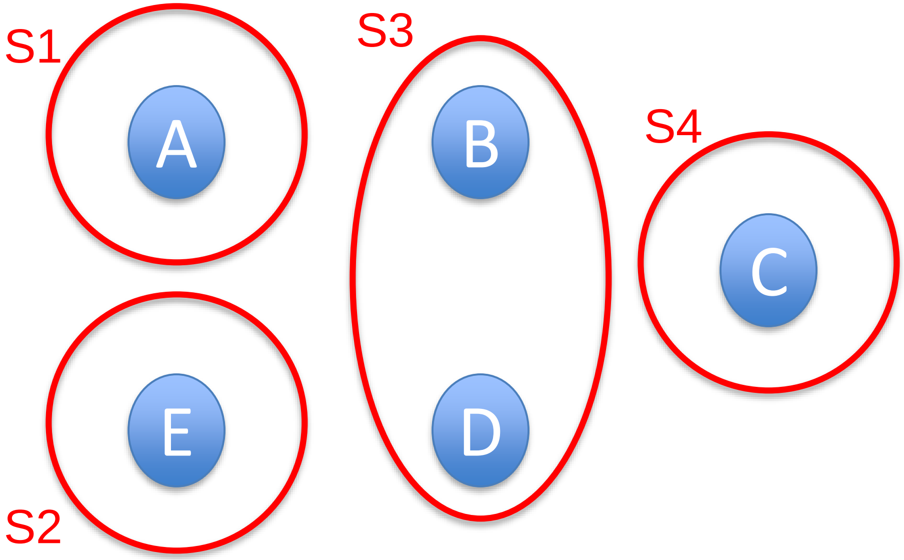


From: https://www.youtube.com/watch?v=UBY4sF86KEY

Disjoint Sets

- A group of sets where no item can be in more than one set

Supported Operations:
`Find()`
`Union()`
`MakeSet()`

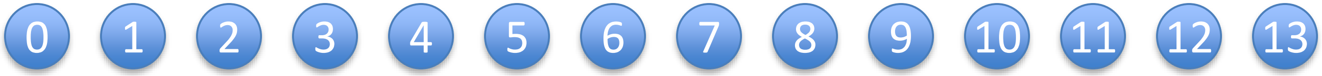


Uses for Disjointed Sets

- Maze generation
- Kruskal's algorithm for computing the minimum spanning tree of a graph
  - Given a set of cities, C, and a set of roads, R, that connect two cities (x, y) determine if it’s possible to travel from any given city to another given city
- **Determining if there are cycles in a graph**

Disjoint Set Example

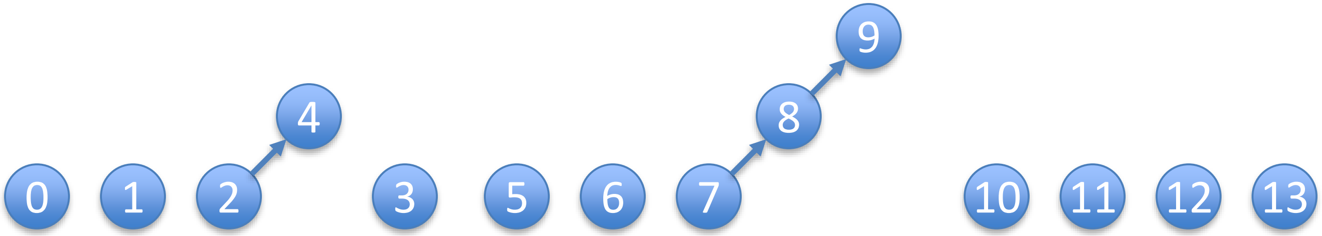
Disjoint Set with No Unions



-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- A negative number means we are at the root
- A positive number means we need to move or “walk” to that index to find our root
- The LONGER the path, the longer it takes to find, and moves farther away from our goal of a constant timed function

Disjoint Set with Some Unions



-1	-1	4	-1	-1	-1	-1	8	9	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Notice:

- Value of index is where the index is linked to

Operations of a Disjoint Set

`Find()`

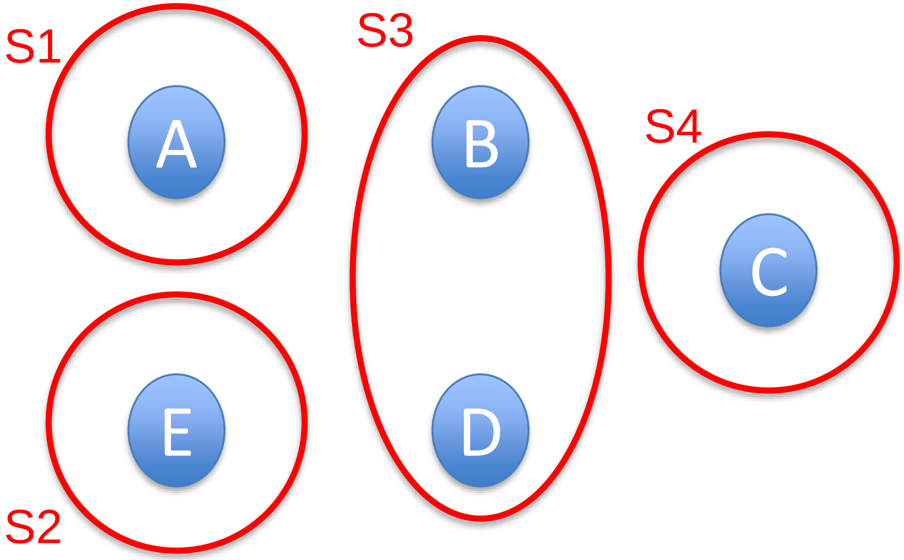
- Determine which subset an element is in
- Returns the name of the subset
- **`Find()`** typically returns an item from this set that serves as its "representative“
  - By comparing the result of two **`Find()`** operations, one can determine whether two elements are in the same subset

`Find()`

- Asks the question, what set does item E belong to currently?

What does **`Find(E)`** return?

Returns S2



From: https://www.youtube.com/watch?v=UBY4sF86KEY

`Union()`

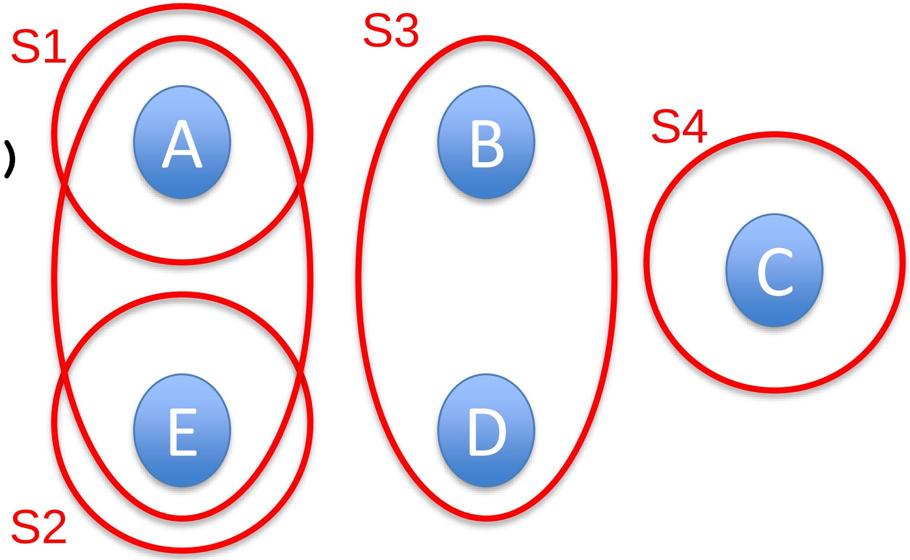
- **`Union()`**
  - Merge two sets (w/ one or more items) together
  - Order can be important
  - One of the roots from the 2 sets will become the root of the merged set

`Union()`

- Join two subsets into a single subset.

Before **`Union(S2, S1)`**

After **`Union(S2, S1)`**



From: https://www.youtube.com/watch?v=UBY4sF86KEY

`MakeSet()`

- Makes a set containing only a given element (a singleton)
- Implementation is generally trivial

Types of Disjoint Sets

Types of Disjoint Sets

- There are two types of disjoint sets
  1. Array Based Disjoint Sets
  2. Tree Based Disjoint Sets
  - (We can also implement with a linked list)

Array Based Disjoint Sets

- We will assume that elements are 0 to n - 1
- Maintain an array **`A`**: for each element **`i`**, **`A[i]`** is the name of the set containing **`i`**

Array Based Disjoint Sets

- **`Find(i)`** returns **`A[i]`**
  - Runs in O(1)
- **`Union(i,j)`** requires scanning entire array
  - Runs in O(n)

```
for (k = 0;k < n; k++) {
  if (A[k] == A[j]) {
    A[k] = A[i]; } }
```

Tree Based Disjoint Sets

- Disjoint-set forests are data structures
  - Each set is represented by a tree data structure
  - Each node holds a reference to its parent node

- In a disjoint-set forest, the representative of each set is the root of that set's tree

Tree Based Disjoint Sets

- **`Find()`** follows parent nodes until it reaches the root

- **`Union()`** combines two trees into one by attaching the root of one to the root of the other

Animation

- Disjoint Sets
- https://www.cs.usfca.edu/~galles/visualization/DisjointSets.html

Optimization of Disjointed Sets

Optimization

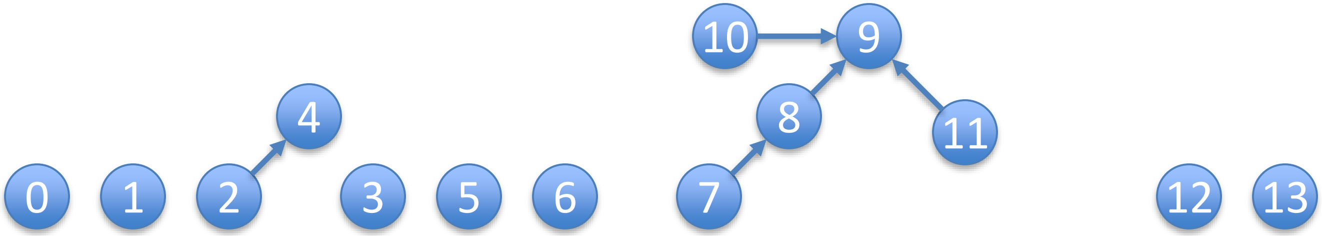
- Three main optimization operations:
  1. Union-by-rank (size)
  2. Union-by-rank (height)
  3. Path Compression

- Be very clear about how the array representations change for different things (union by size, union by height, etc.)

Union-by-Rank (size)

- ***Size*** = number of nodes (including root) in given set
- A strategy to keep items in a tree from getting too deep (large paths) by uniting sets intelligently
- At each root, we record the ***size*** of its sub-tree
  - The number of nodes in the collective tree

Union-by-Rank (size)



-1	-1	4	-1	-2	-1	-1	8	9	-5	9	9	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

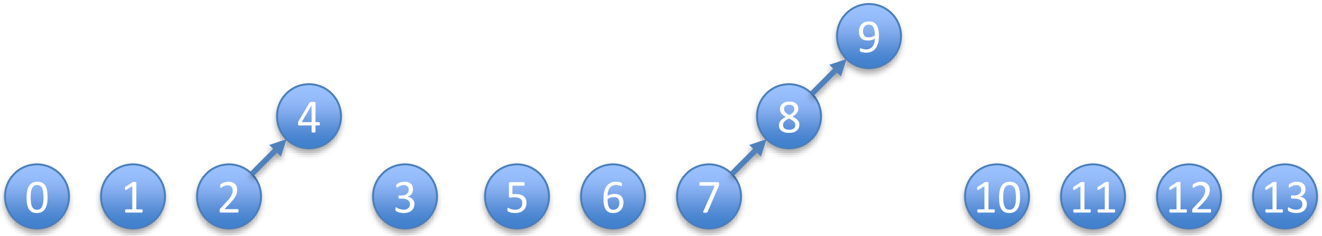
Notice two things:

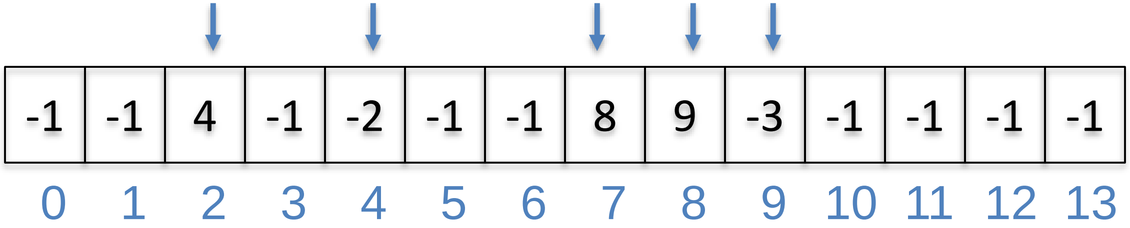
1. Value of index is where the index is linked to
2. As the size of the set increases, the negative number **size** of the root increases (see 4 and 9)

Union-by-Rank (height)

- A strategy to keep items in a tree from getting too deep (large paths) by uniting sets intelligently
- At each root, we record the ***height*** of its sub-tree
- When uniting two trees, make the smaller tree a sub-tree of the larger one
  - So that the tree that is larger does not add **another** level!!

Union-by-Rank (height)



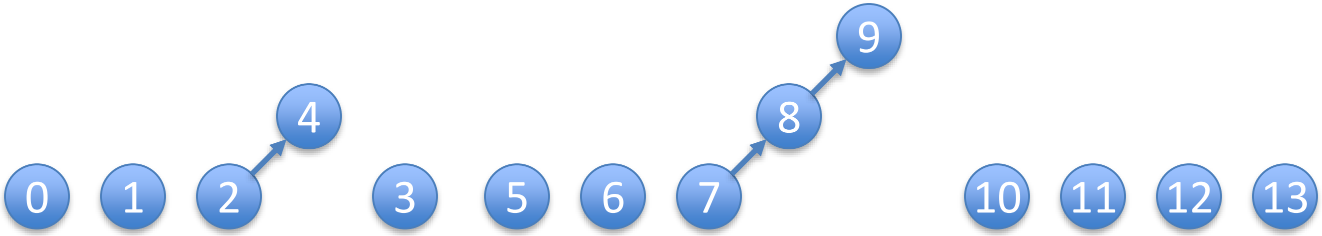


-1	-1	4	-1	-2	-1	-1	8	9	-3	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Notice two things:

1. Value of index is where the index is linked to
2. As the size of the set increases, the negative number **height** of the root increases (see 4 and 9)

Union-by-Rank (height)

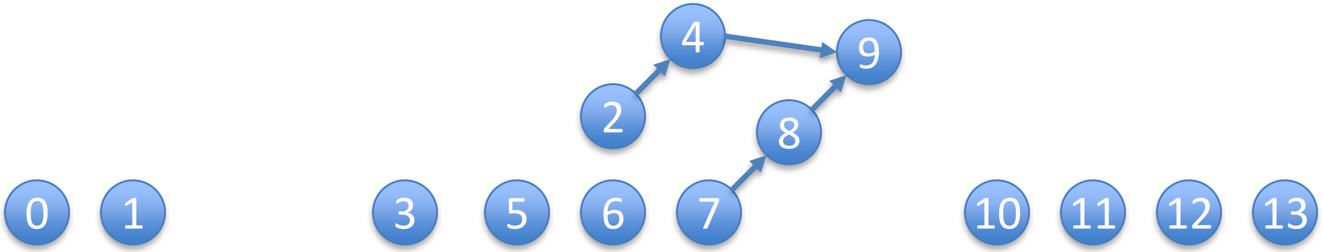


-1	-1	4	-1	-2	-1	-1	8	9	-3	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

What if we merge {2,4} with {7, 8, 9}?

Because 9 has a greater height than 4, 4 would be absorbed into 9.

Union-by-Rank (height)



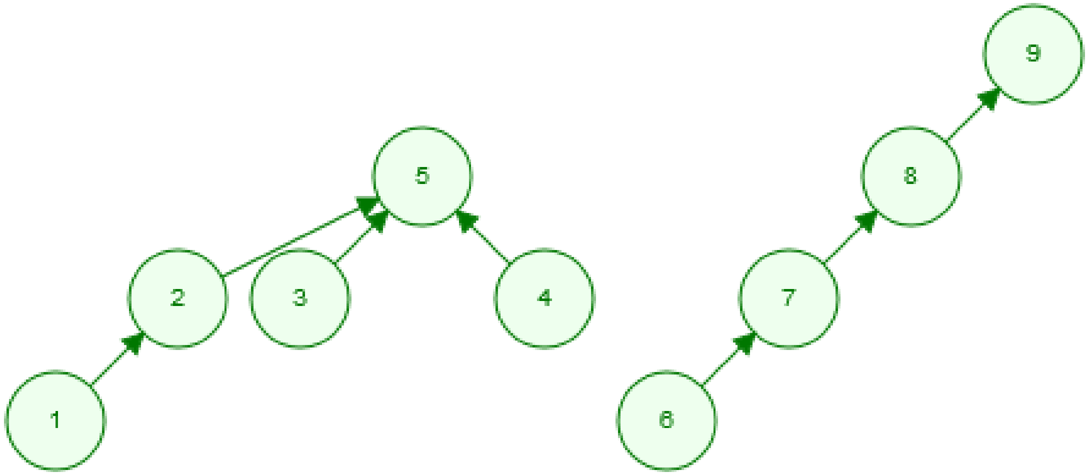
Update 4 to point to 9

-1	-1	4	-1	-2 9	-1	-1	8	9	-3	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13

When uniting two trees, make the smaller tree a sub-tree of the larger one so that the one tree that is larger does not add **another** level!!

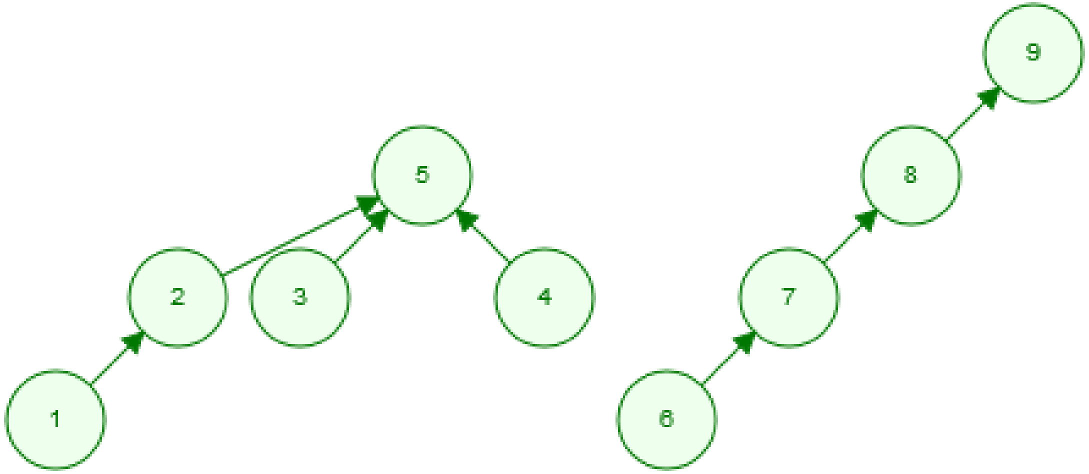
Example of Unions

- If we union 5 and 9, how will they be joined?



Example of Unions

- By rank (size)?
  - 9 becomes a child of 5
- By rank (height)?
  - 5 becomes a child of 9



Path Compression

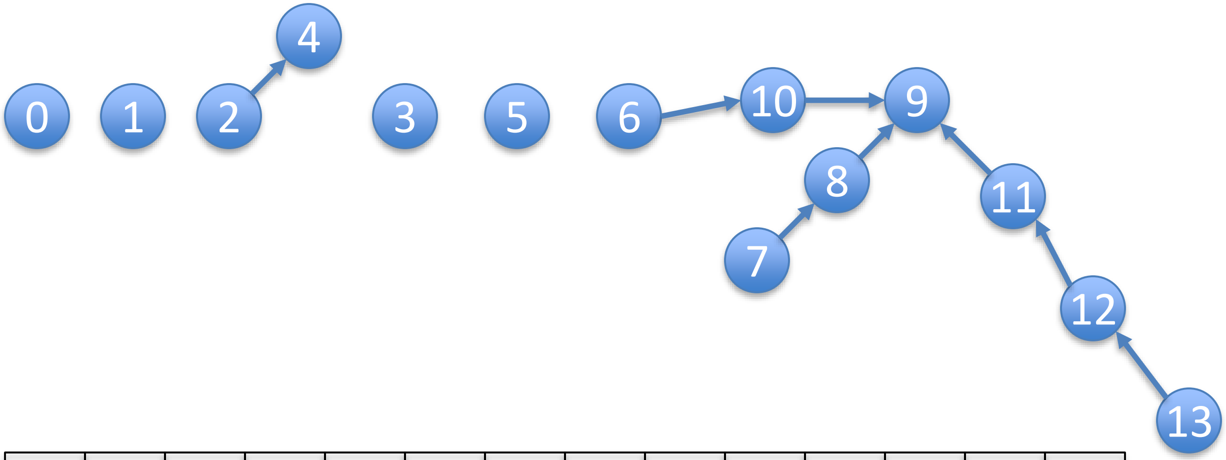
- If our path gets longer, operations take longer
- We can shorten this (literally and figuratively) by updating the element values of each child <u>directly</u> to the root node value
  - No more walking through to get to the root
- Done as part of **`Find()`**
  - So the speed up will be eventual

Path Compression

- Theoretically flattens out a tree
- Uses recursion
- Base case
  - Until you find the root
  - Return the root value
- Reassign as the call stack collapses

Path Compression

Before Path Compression



0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	4	-1	-1	-1	10	8	9	-1	9	9	11	12

During a **`Find()`**, we update the index to point to the root

Path Compression

After Path Compression

-1	-1	4	-1	-1	-1	9	9	9	-1	9	9	9	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13

After we run **`Find(6)`** we update it to point to 9
After we run **`Find(13)`** we update it to point to 9
Along with all other nodes between 13 and 9!

Code for Disjoint Sets

Generic Code

```
function MakeSet(x)
     x.parent := x

function Find(x)
     if x.parent == x
        return x
     else
        return Find(x.parent)

function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     xRoot.parent := yRoot
```

C++ Implementation

```
class UnionFind {
  int[] u;

  UnionFind(int n) {
    u = new int[n];
    for (int i = 0; i < n; i++)
      u[i] = -1;
  }

  int find(int i) {
    int j,root;
    for (j = i; u[j] >= 0; j = u[j]) ;
    root = j;
    while (u[i] >= 0) { j = u[i]; u[i] = root; i = j; }
    return root;
  }

  void union(int i,int j) {
    i = find(i);
    j = find(j);
    if (i !=j) {
      if (u[i] < u[j])
        { u[i] += u[j]; u[j] = i; }
      else
        { u[j] += u[i]; u[i] = j; }
    }
  }
}
```

The **`UnionFind`** class

```
class UnionFind {
   int[] u;

   UnionFind(int n) {
      u = new int[n];
      for (int i = 0; i < n; i++)
         u[i] = -1;
   }

   int find(int i) { ... }

   void union(int i,int j) { ... }
}
```

Trick 1: Iterative find

```
int find(int i) {
   int j, root;

   for (j = i; u[j] >= 0; j = u[j]) ;
   root = j;

   while (u[i] >= 0)
     { j = u[i]; u[i] = root; i = j; }

   return root;
 }
```

Trick 2: Union by size

```
void union(int i,int j) {
   i = find(i);
   j = find(j);

   if (i != j) {
      if (u[i] < u[j])
         { u[i] += u[j]; u[j] = i; }
      else
         { u[j] += u[i]; u[i] = j; }
   }
}
```

Disjointed Sets Performance

Performance

- In a nutshell
  - Running time complexity: *O(1)* for union
    - Using ONE pointer to connect from one root to another
  - Running time of find depends on implementation
    - Union by size: Find is *O(log(n))*
    - Union by height: Find is *O(log(n))*
- Union operations obviously take Θ(1) time
  - Code has no loops or recursion
    - Θ(f(n)) is when the ***worst case*** and ***best case*** are identical

Performance

- The ***average running*** time of any find and union operations in the quick-union data structure is so close to a constant that it's hardly worth mentioning that, in an asymptotic sense, it's *slightly* slower in real life

Performance

- A sequence of ***f*** find and ***u*** union operations (in any order and possibly interleaved) takes Theta(u + f α(f + u, u)) time in the worst case
- α is an extremely slowly-growing function
- Known as the inverse ***Ackermann function***.
  - This function is never larger than 4 for any values of **f** and **u** you could ever use (though it can get arbitrarily large—for unimaginably large values of f and u).
  - Hence, for all practical purposes think of quick-union as having find operations that run, ***on average***, in ***constant time***.