Disjoint Sets

Step 1

In the hit TV show *Chuck*, Sarah Walker is a secret agent working for the CIA, and she is assigned the task of spying on computer geek Chuck Bartowski. She wants to make sure Chuck isn't secretly working for some evil agency, so specifically, her task is to find out if there is some "connection" between Chuck and a known villain, either direct or indirect. To do so, she must look at Chuck's own social circle, and then look at the social circles of each individual in Chuck's social circle, and then look at the social circles of each individual in *those* people's social circles, etc. As one might expect, this task blows up in size quite quickly, so Sarah would be better off writing an efficient program to solve the problem for her. How might she formulate this CIA task as a formal computational problem?

We can represent Chuck's social network (and the social networks of those around him) using a *graph*: nodes (V) are individuals, and edges (E) represent individuals having a direct social connection to one another. Sarah's goal is to come up with a way of being able to query a person's name and determine if the person is connected to Chuck, either directly or indirectly. If she were to approach this as a naive graph traversal algorithm, she would need to use BFS each time she performs a query, which is $O(|V| + |E|)$. Luckily, Sarah did well in her university Data Structures course and, as a result, she had the Computer Science skills needed to solve the seemingly complex problem quite efficiently.

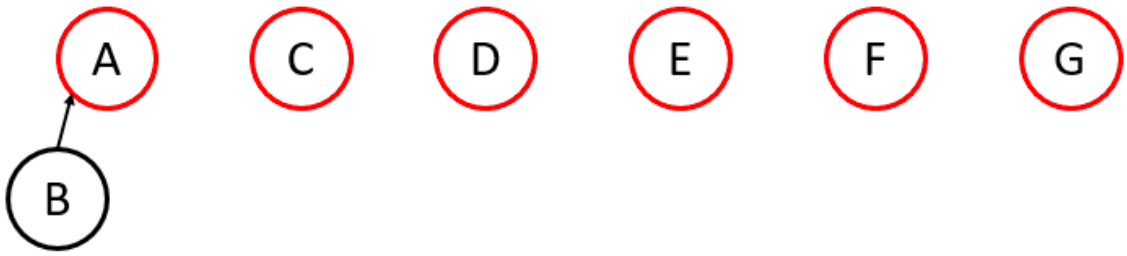
In this section, we will discuss the **Disjoint Set** ADT, which will help Sarah greatly speed up her surveillance problem.

Step 2

The **Disjoint Set** is an Abstract Data Type that is optimized to support two particular operations:

- **Union:** Given two elements u and v, merge the sets to which they belong
- **Find:** Given an element e, return the set to which it belongs

As a result, a Disjoint Set is also commonly referred to as a "Union-Find" data type. Disjoint Sets can be very efficiently implemented using the **Up-Tree** data structure, which in essence is simply a graph in which all nodes have a single "parent" pointer. Nodes that do not have a parent (i.e., the roots of their respective trees) are called **sentinel nodes**, and each sentinel node represents a single set. In the example below, there are 6 sets, each represented by its respective sentinel node (shown in red): A, C, D, E, F, and G.



A disjoint set often starts out as a forest of single-node sets. We begin connecting vertices together by calling the union operation on two vertices at a time, such as what you see in the figure above. By doing so, we slowly begin creating a structure that looks like an upside-down tree (i.e., a tree where all pointers point up *to the root* instead of the usual down *from the root*), hence the name "Up-Tree". We will explore the union operation in more depth later in this lesson.

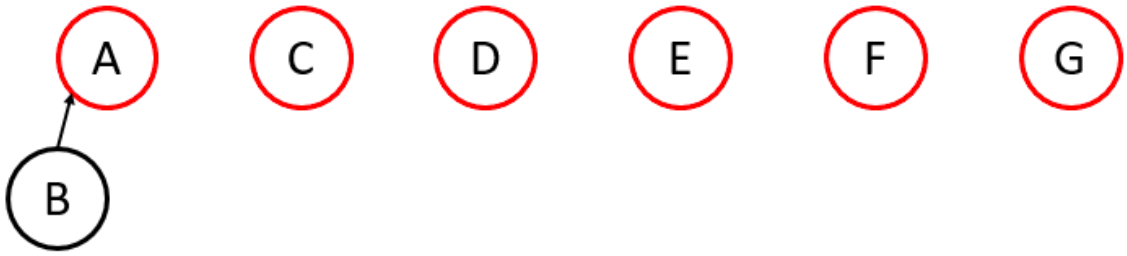
The algorithm behind the "find" operation of a Disjoint Set implemented as an Up-Tree should hopefully be intuitive. Recall that the sentinel nodes represent (or really name) the sets. As a result, given an element, finding the set to which the element belongs is actually the exact same problem as finding the sentinel node that is the ancestor of the element. As a result, to find an element's set, you can simply start at the element and traverse "parent" pointers up the tree until you reach the element's sentinel node (i.e., the ancestor that has no parent). Then, simply return the sentinel node, because the sentinel node "is" the element's set.

Because the "find" algorithm traverses up the tree of the given element, it should hopefully be intuitive that the larger the height of the tree an element is in, the slower the "find" operation is on that element. As a result, our goal should always be to try to minimize our tree heights as much as possible, as this will translate into faster "find" operations.

Step 3

In the meantime, how do we go about implementing this "forest of nodes"? One option would be to implement it as an adjacency list or an adjacency matrix, just like we went about implementing the graphs in the previous lessons. Note, however, that nodes in an Up-Tree will always only need to store an edge to a single other node, its parent, as opposed to multiple adjacent vertices. As a result, we can take advantage of a simple array to store the information that we need.

To understand how this array representation works, we will reproduce the example Up-Tree from the previous step below:



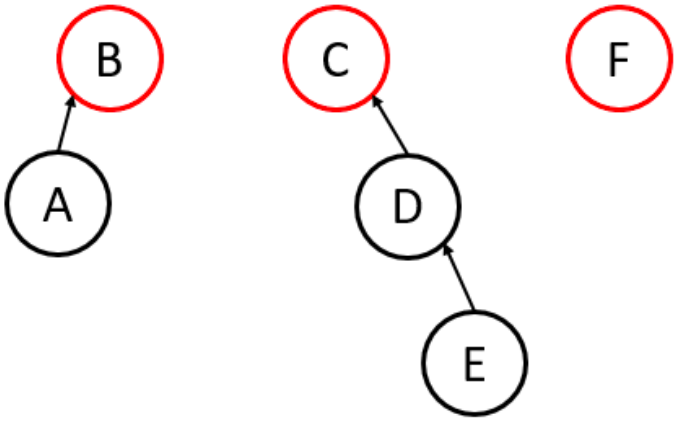
Below is an array that represents the Up-Tree in the example above:

A	B	C	D	E	F	G
-1	0	-1	-1	-1	-1	-1

Every index of the array corresponds to a particular vertex (e.g. index 0 corresponds to vertex A, index 3 corresponds to vertex D, etc.). The content of each cell of the array corresponds to the corresponding vertex's parent. For example, the index corresponding to vertex B — index 1 — has a 0 in it because index 0 corresponds to vertex A and vertex A is the parent of vertex B. Also, the index corresponding to vertex F — index 5 — has a -1 in it because an index of -1 corresponds to NULL, and since vertex F doesn't have a parent, its parent is NULL.

Step 4

EXERCISE BREAK: Fill in the missing elements in the array for the corresponding Up-Tree shown below.



A	B	C	D	E	F
?	?	-1	?	3	-1

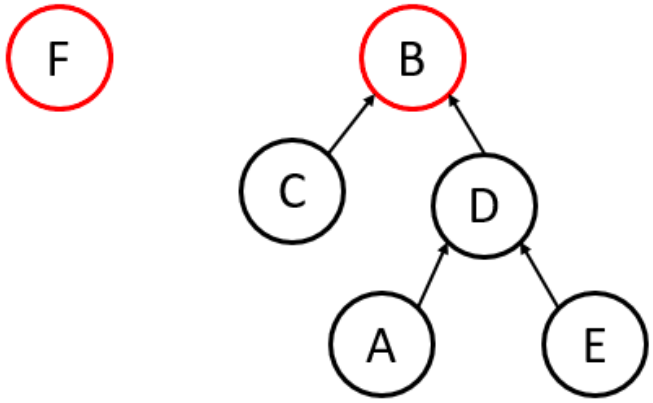
To solve this problem please visit https://stepik.org/lesson/30027/step/4

Step 5

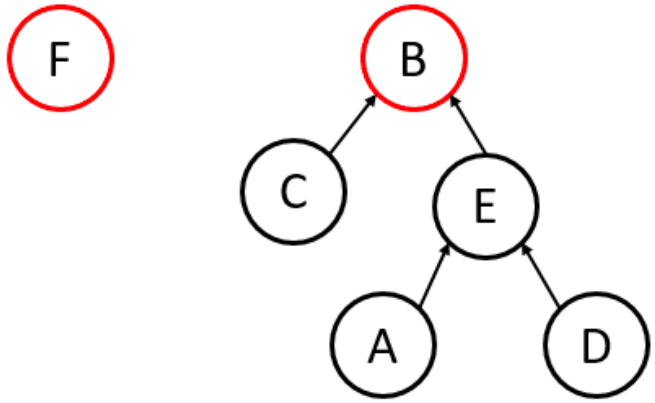
EXERCISE BREAK: Select the corresponding disjoint set for the following array:

A	B	C	D	E	F
5	-1	4	4	5	-1

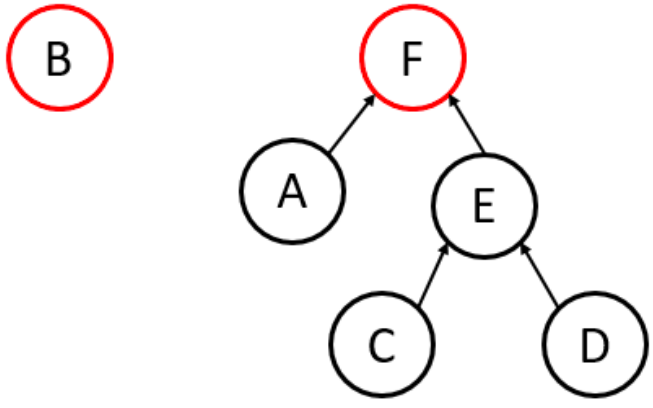
Choice A:



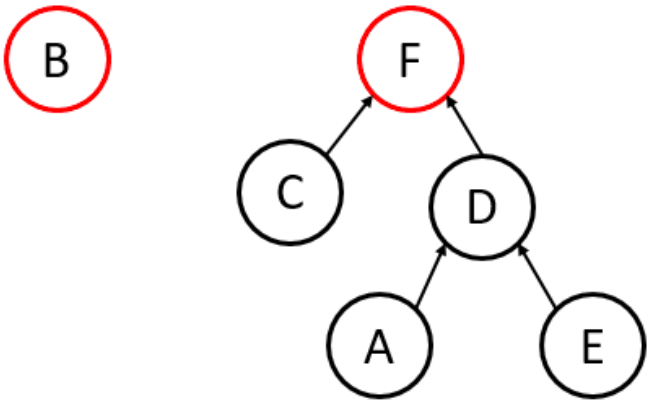
Choice B:



Choice C:



Choice D:



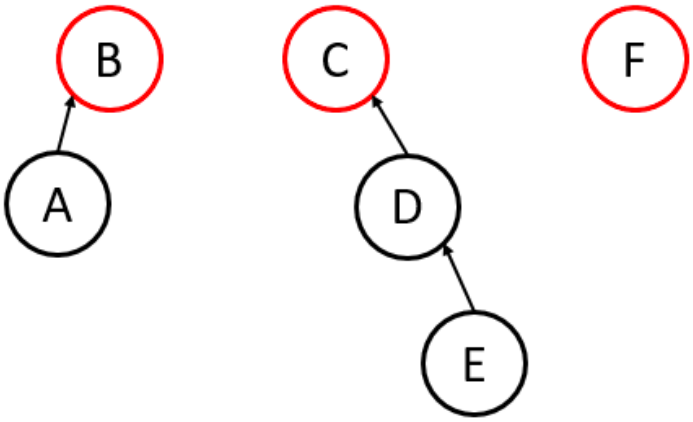
To solve this problem please visit https://stepik.org/lesson/30027/step/5

Step 6

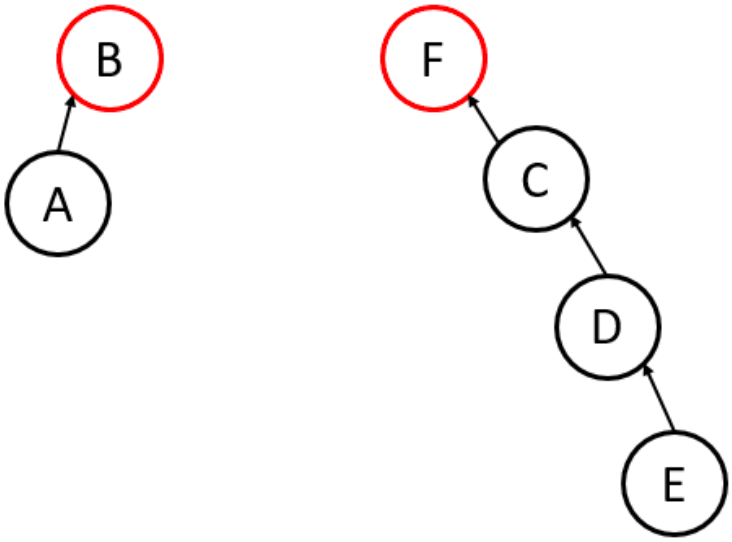
We mentioned earlier that elements in the disjoint set are connected through the **union** operation (we care about connecting elements in our set in order to be able to show a relationship between the elements, just like in a general graph).

How exactly does the **union** operation work? Formally, the purpose of the **union** operation is to join two sets. This implies that, after we have unioned two different sets, the two sets should end up with the **same** sentinel node. Consequently, a lot of freedom actually surrounds the concept behind the **union** operation because there are so many ways to merge two sets of vertices.

For example, suppose we start with the disjoint set below:



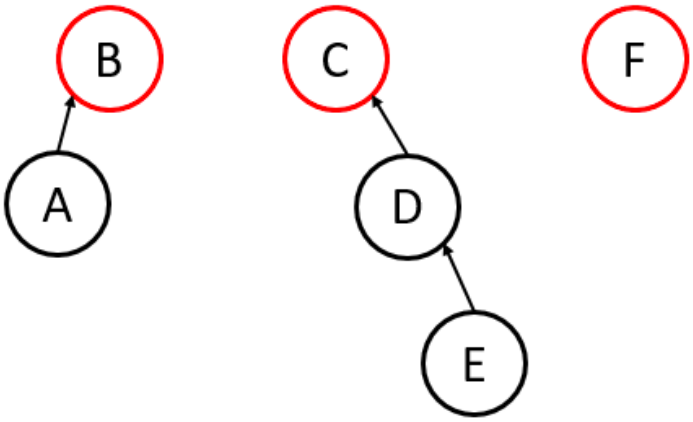
Let's say we want to union(F, E) (i.e., merge the set vertex F is in with the set vertex E is in). First we must find the set that contains vertex F by calling "find" on vertex F. This would return vertex F since the sentinel node of vertex F is vertex F. Second we must find the set that contains vertex E by by calling "find" on vertex E. This would return vertex C since the sentinel node of vertex E is vertex C. Lastly we must find a way to merge both sets. An intuitive way might be to union the sets by connecting vertex C to vertex F like so:



In the example above, we basically just stacked the vertices and literally connected the sentinel node of vertex E (vertex C) to the sentinel node of vertex F (vertex F). Thus vertex F becomes the parent and vertex C becomes the child. Notice that we specifically connected the **sentinel nodes** of the two elements on which we called union, not necessarily the elements themselves! This is extremely important because, otherwise, we might be disrupting the structure of our Up-Tree. Now, the important question to ask is: Do vertex F and vertex E end up returning the **same** sentinel node? The answer in this case is obviously yes; if we traverse up from both vertex E and vertex F, the root is vertex F.

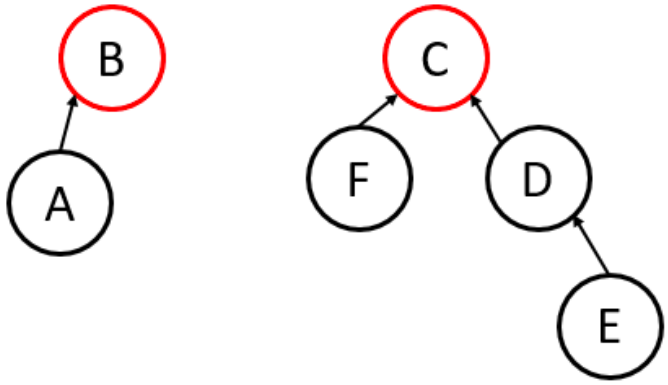
Step 7

Below is the same example as in the previous step:



Recall that we arbitrarily chose which sentinel nodes we wanted to use as the parent and child when merging. However, as mentioned before, our overarching goal while building an Up-Tree is to minimize its height because a smaller tree height corresponds to a faster "find" operation. Consequently, instead of making the choice arbitrarily, we can perform a "**Union-by-Size**." In this insertion tactic, the sentinel node of the *smaller* set (i.e., the set with less elements) gets attached to the sentinel node of the *larger* set. Thus, the sentinel node of the smaller set becomes the child and the sentinel node of the larger set becomes the parent. Ties can be broken arbitrarily.

Below is the result of performing the exact same operation, **union**(F, E), by using the **Union-by-Size** approach. Note that in the original disjoint set above, the sentinel node of F (node F) has 1 element in its set (node F), and the sentinel node of E (node C) has 3 elements in its set (node C, node D, and node E). As a result, the sentinel node of the *larger* set (node C) is made the *parent*, and the sentinel node of the *smaller* set (F) is made the *child*.

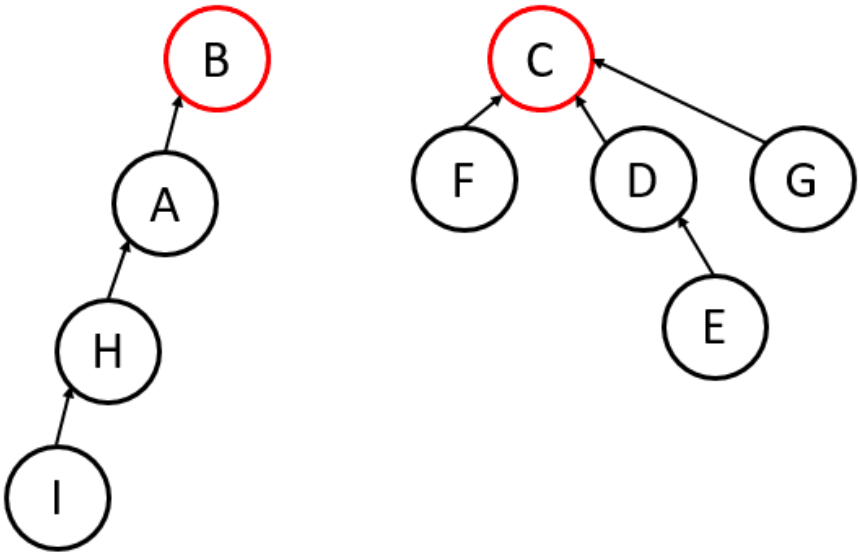


Under **Union-by-Size**, the only way for the height of a tree resulting from the union of two sets to be *larger* than the heights of both of the two initial sets is if the sets have exactly the same height. As a result, the "worst case" for **Union-by-Size** is considered to be the situation in which each merge had to merge sets with equal size. For example, say we have a dataset of *n* elements, each in their own set initially. To invoke the worst case, we can perform $n/2$ union operations to form $n/2$ sets, each with exactly 2 elements. Then we can perform $n/4$ union operations to form $n/4$ sets, each with exactly 4 elements. If we keep doing this until we are left with one set with *n* elements, that one set will have the shape of a balanced tree. As a result, the worst-case time complexity of a "find" operation in **Union-by-Size** is the same as the worst-case height of a balanced tree: **O(log *n*)**.

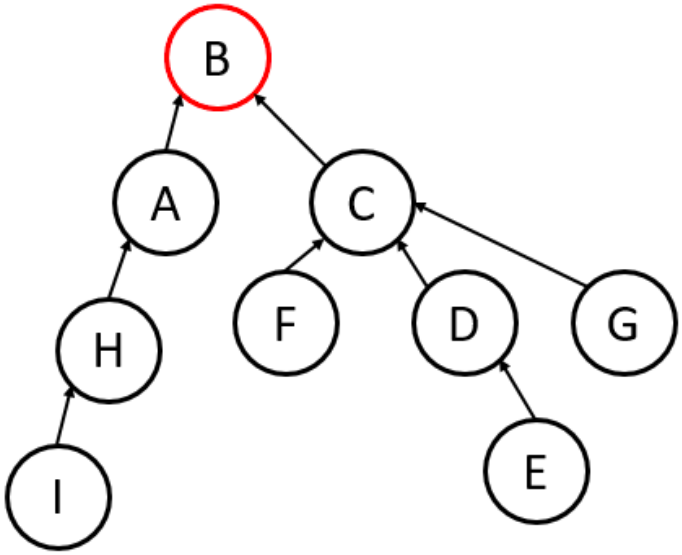
Step 8

Along side the **Union-by-Size** method, there is also a "**Union-by-Height**" method. In the "**Union-by-Height**" method, the sentinel node of the *shorter* set (i.e., smaller height) gets attached to the sentinel node of the *taller* set. Again, ties can be broken arbitrarily.

Let's say we start with the disjoint set below:



Below is the result of performing **union**(A, C) by using the **Union-by-Height** approach. The sentinel node of node A (node B) has a height of 3 and the sentinel node of node C (node C) has a height of 2. As a result, the sentinel node of the *taller* set (node B) is made the *parent* and the sentinel node of the *shorter* set (node C) is made the *child*.

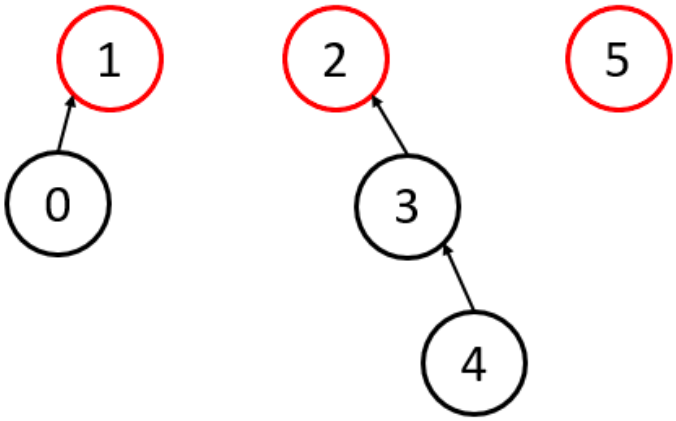


Just like **Union-by-Size**, the **Union-by-Height** method results in a worst-case time complexity of **O(log *n*)** for the "find" operation.

STOP and Think: If we used **Union-by-Size** instead of **Union-by-Height** on the example above, would the resulting tree be better, worse, or just as good as the one produced by the **Union-by-Height** method?

Step 9

EXERCISE BREAK: Fill in the array that would result in performing **union**(0, 3) using the **Union-by-Height** method on the disjoint set below. Just as before, use -1 to represent NULL.



0	1	2	3	4	5
?	?	?	?	?	?

To solve this problem please visit https://stepik.org/lesson/30027/step/9

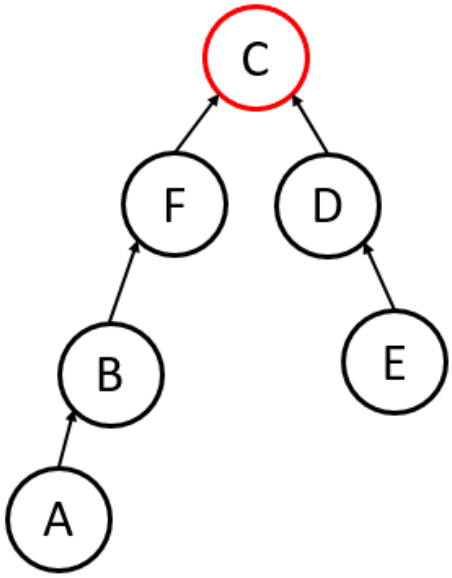
Step 10

As we've seen, choosing the "smarter" union operations (**Union-by-Height** or **Union-by-Size**) can lead to big improvements when it comes to finding particular vertices. Furthermore, we can actually do *even better* by making a slight optimization to the "find" operation: adding **path compression**.

As mentioned previously, the original implementation of the "find" operation involves starting at the queried node, traversing its tree all the way up to the sentinel node, and then returning the sentinel node. This algorithm, assuming we used **Union-by-Height** or **Union-by-Size** to construct our Up-Tree, has a worst-case time complexity of **O(log *n*)**. If we want to use the "find" operation again to search for *another* vertex's sentinel node, the operation will again take **O(log *n*)** time (as expected).

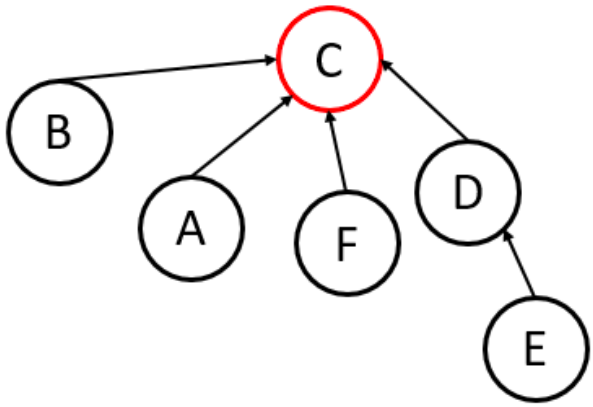
So how can we go about improving the time complexity of the "find" operation *even more*? The answer lies in taking advantage of *all* of the information we obtain during the "find" algorithm: not just the sentinel node of interest.

Take a look at the example disjoint set below:



Suppose we want to perform find(A). We know that the goal of "find" is to return the sentinel node of vertex A (which is vertex C, in this case). Notice, however, that *every* vertex on our exploration path (vertex A, vertex B, and vertex F) have vertex C as their sentinel nodes, not just vertex A! As a result, it is unfortunate that, in the process of finding vertex A, we have gained the information that vertex B and vertex F also share the same root (which would save us time later if we wanted to find(B) or find(F)), yet we have no way of easily and efficiently memorizing this information... Or do we?

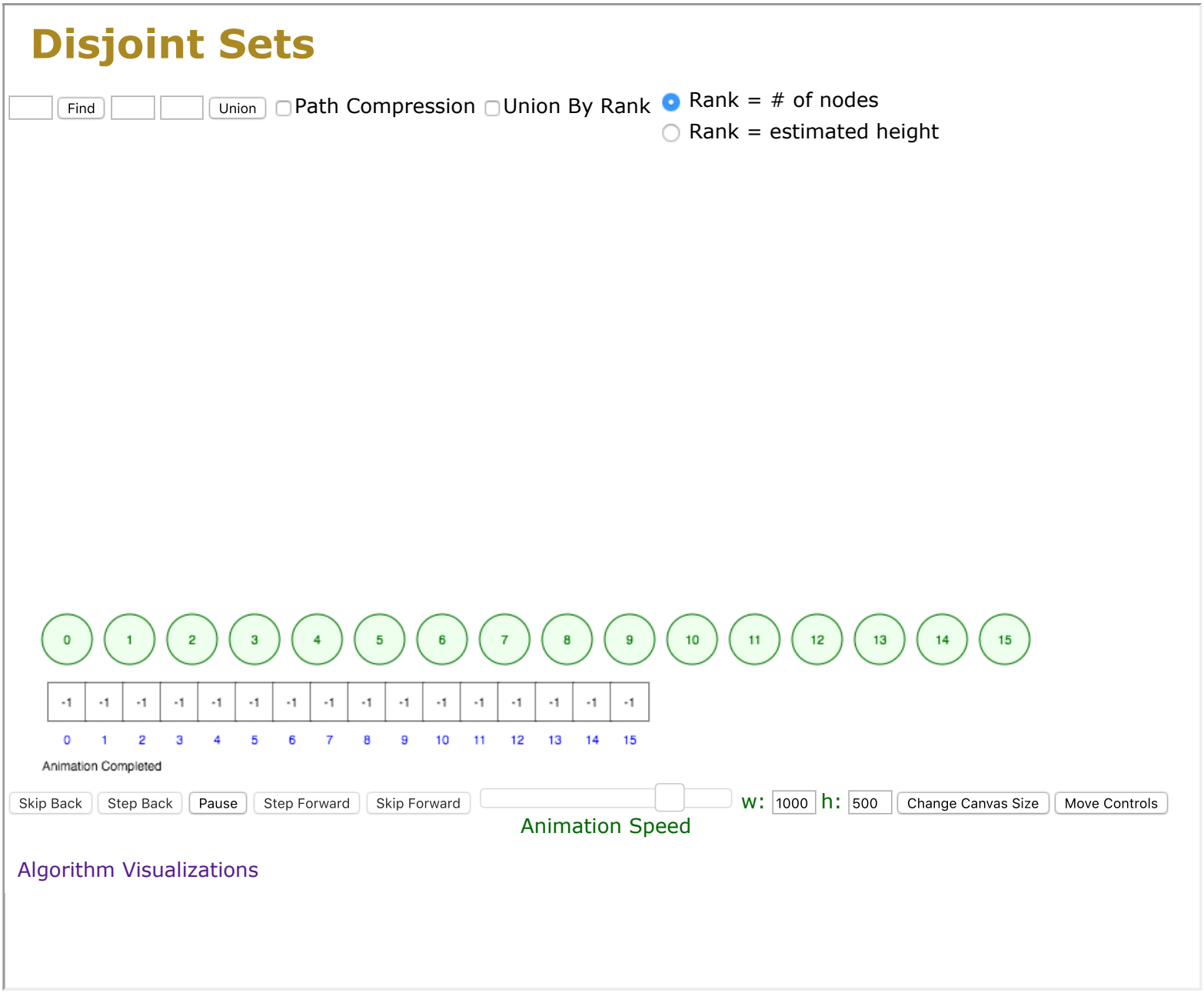
It turns out that we actually do! The solution is as follows: As we traverse the tree to return the sentinel node, we re-attach each vertex along the way directly to the root. This *extremely* simple, yet quite powerful, technique is called **path compression**. For example, using **path compression** on the disjoint set above, calling find(A) would yield the following disjoint set once the operation finishes:



Now, if I call find(B) on the new disjoint set above, this operation will be **constant** time! This is because we guarantee that since the vertex is attached directly to the root, we will only need one operation to return the root. You are now allowed to be mind-blown.

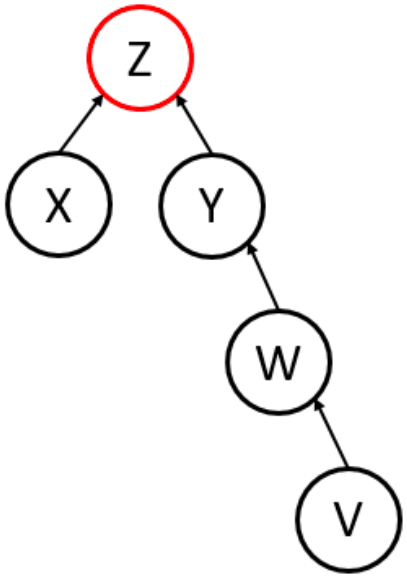
Step 11

Here is an interactive visualization, created by David Galles at the University of San Francisco, to help you further explore the union and find operations, as well as **path compression**.



Step 12

EXERCISE BREAK: Select which vertices will be connected directly to the sentinel node (vertex Z) as a result of performing **find(V)** using **path compression** on the disjoint set below:



To solve this problem please visit https://stepik.org/lesson/30027/step/12

Step 13

An Up-Tree with an implementation of **path compression** is an example of a **self-adjusting structure**. In a self-adjusting structure, an operation like the "find" operation occasionally incurs a high cost because it does extra work to modify the structure of the data (such as the reattachment of vertices directly to the root in the example of **path compression** we saw previously). However, the hope is that subsequent similar operations will be made *much* more efficient. Examples of other self-adjusting structures are splay trees, self-adjusting lists, skew heaps, etc. (just in case you're curious to see what else is out there).

So how do we figure out if the strategy of self-adjustment is actually worth it in the long run? The answer lies in **amortized cost analysis**.

In regular algorithmic analysis, we usually look at the time or space cost of doing a single operation in either the best case, average case, or worst case. However, if we were to do so for an operation that involved something like **path compression**, we would not get the tightest complexity because we already know that, based on how **path compression** was designed, the first time we use it will guaranteed take longer than subsequent times. **Amortized cost analysis** considers the time or space cost of doing a *sequence of operations* (as opposed to a single operation) because the total cost of the entire sequence of operations might be less with the intial extra initial work than without!

We will not go into the details of how to actually perform amortized cost analysis because the math can get quite intense. Nonetheless, a mathematical function that is common in amortized cost analysis is log*(*N*) (read: "log star of *N*" and also known as the "single variable inverse Ackerman function"), which is equal to the number of times you can take the log base-2 of *N*, until you get a number less than or equal to 1. For example,

- log*2 = log* 2^1 = 1
- log*4 = log * 2^2 = 2 (note that 4 = 2^2)
- log*16 = log * 2^4 = 3 (note that 16 = 2^(2^2))
- log*65536 = log*2^16 = 4 (note that 65536 = 2^(2^(2^2)))
- log*2^65536 = 5 (note that 2^65536 = 2^(2^(2^(2^2))) is a huge number)

The reason we use this particular function is because it is able to take into consideration both aspects of a self-adjusting operation:

- The first non-constant time part (log base-2 in this case) in which the structure is re-adjusting itself

- The second constant time part (less than or equal to 1 in this case) in which the structure has finished re-adjusting and is now reaping the benefits of the self-adjustment

It can be shown that, with **Union-by-Size** or **Union-by-Height**, using **path compression** when implementing "find" performs any combination of up to *N*-1 "union" operations and *M* "find" operations and therefore yields a worst-case time complexity of **O(*N* + *M* log* *N*)** *over all operations*. This is much better than old-fashioned logarithmic time because log**N* grows *much* slower than log *N*, and for all practical purposes, log**N* is never more than 5 (so it can be considered practically constant time for reasonable values of *N*). Therefore, *for each individual find or union operation*, the worst-case time complexity becomes constant (i.e., **O(1)**).

Step 14

We have now introduced the **Disjoint Set** ADT, which allows us to create sets of elements and to very efficiently perform union and find operations on these sets.

You may have noticed (hopefully triggered by the **STOP and Think** question) that **Union-by-Height** always produces an Up-Tree that is always either just as good, or many times even better, than the tree produced by **Union-by-Size**. However, it turns out that, in practice, people typically choose to use **Union-by-Size** when they implement their Up-Trees. The reason why they choose **Union-by-Size** over **Union-by-Height** is that there is a *huge* savings when it comes to **path compression**; one side effect of **path compression** is that the height of the tree and the subtrees within it change very frequently. As a result, maintaining accurate information about the *heights* of sets (which is required for **Union-by-Height**) becomes quite difficult because of **path compression**, whereas maintaining accurate information about the *sizes* of sets (which is required for **Union-by-Size**) is extremely easy (constant-time, actually). As a result, though **Union-by-Size** may get *slightly* worse results than **Union-by-Height**, the difference is so small in comparison to the savings from **path compression** that it is considered negligible.

Going back to the example that we had initially introduced in the lesson, by using a **Disjoint Set** implemented using an **Up-Tree**, Sarah can very efficiently solve her problem! In her case, elements of the **Disjoint Set** would be the people she notices, and as she sees individuals *u* and *v* "connect", she can simply perform union(*u*, *v*). When she wants to check if some suspect *w* is connected in some way to Chuck, she can perform find("Chuck") and find(*w*) and compare the sets they return for equality.