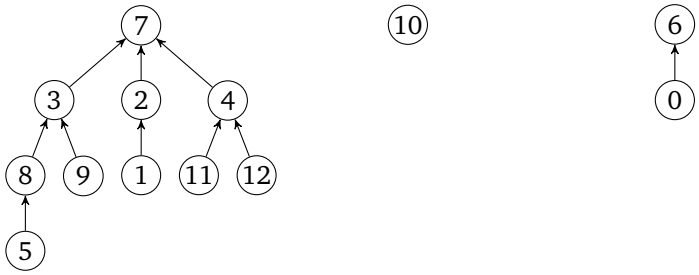
Section 08: Solutions

1. Disjoint sets

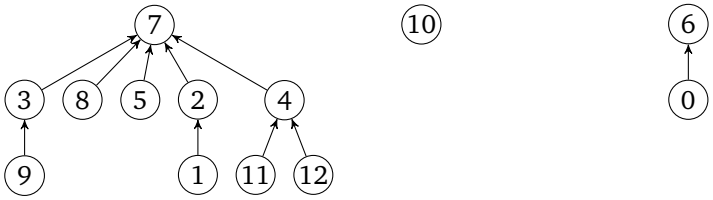
(a) Consider the following trees, which are a part of a disjoint set data-structure:



For these problems, use both the **weighted quick union by size** and **path compression** optimizations.

(i) Draw the resulting tree(s) after calling `find(5)` on the above. What value does the method return?

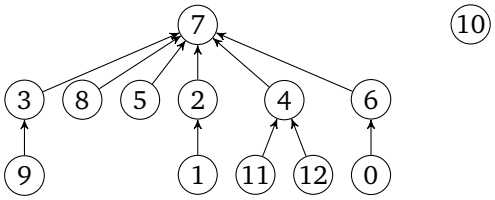
Solution:



The method returns 7, which is the root of the tree.

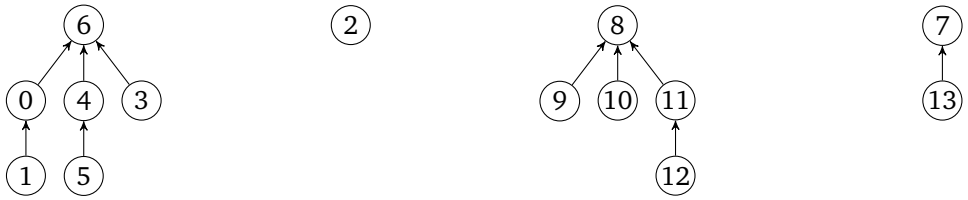
(ii) Draw the final result of calling `union(2,6)` on the result of part a.

Solution:



We merge the second tree to the first since the size of the first tree is larger.

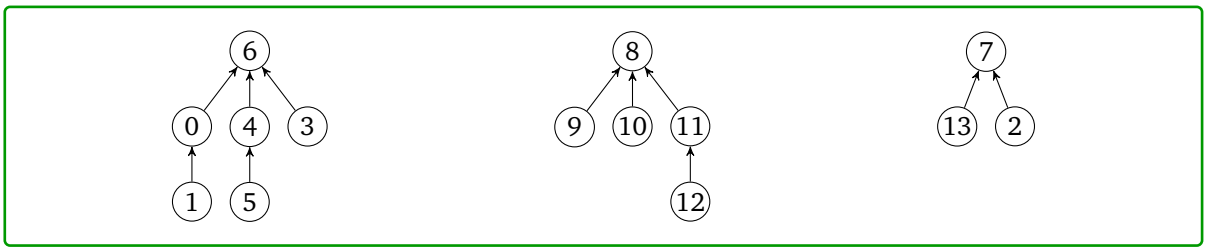
(b) Consider the disjoint-set shown below



What would be the result of the following calls on union if we add the **weighted quick union by size** and **path compression** optimizations.

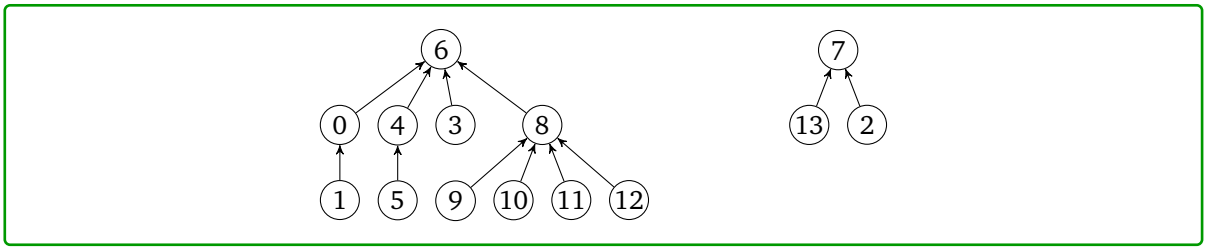
(i) `union(2, 13)`

Solution:



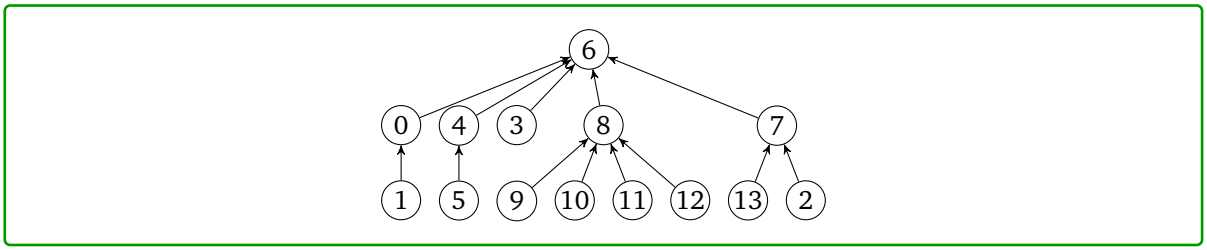
(ii) `union(4, 12)`

Solution:



(iii) `union(2, 8)`

Solution:

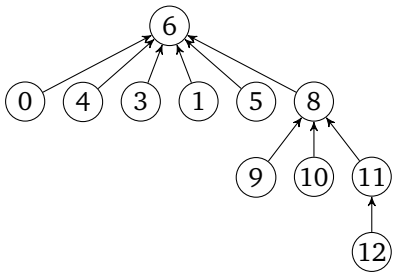


(c) Consider the disjoint-set shown below

What would be the result of the following calls on union if we add the **weighted quick union by size** and **path compression** optimizations.

(i) `union(10, 0)`

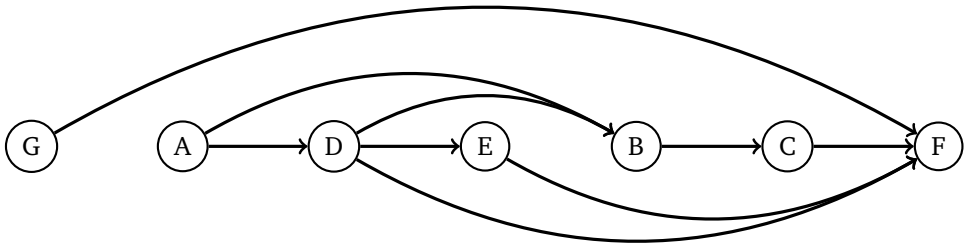
Solution:



Note in particular here that if we do the weighted quick union by size we are always choosing the tree with the larger number of nodes to adopt the tree with the smaller number of nodes. Note that we want to keep the height as small as possible, so in this case, the "best solution" would have been to connect the 6 to the 8, since the resulting height would be 2 instead of 3. However, we can't know the exact height of these trees when we are performing the path compression optimization, so we accept sub-par results like this in some cases. It's easier to always connect the tree with the smaller number of nodes to the tree with the larger number of nodes than it is to calculate the exact height of both trees (note that it would take $O(n)$ time to calculate the height of both trees, but we need this operation to be fast).

2. Topological sort

(a) Give a valid topological sort of the graph below. For your reference, some orderings of the graph are provided below the graph.




```
DFS preorder: ABCFDE (G)
DFS postorder: FCBEDA (G)
BFS: ABDCEF (G)
```

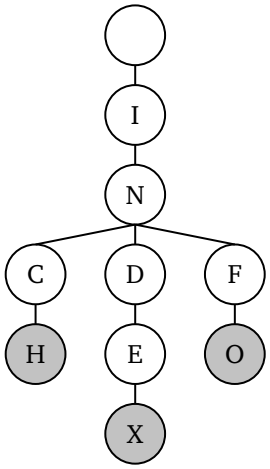
Solution:

A valid topological sorting can be obtained by reversing the DFS postorder.

One valid topological sort is $G - A - D - E - B - C - F$. There are many others. In particular, G can go anywhere except after F, since it has no incoming edges and only one outgoing edge (to F).

3. Tries

(a) Consider the trie shown below:



(i) What strings are stored in the trie?

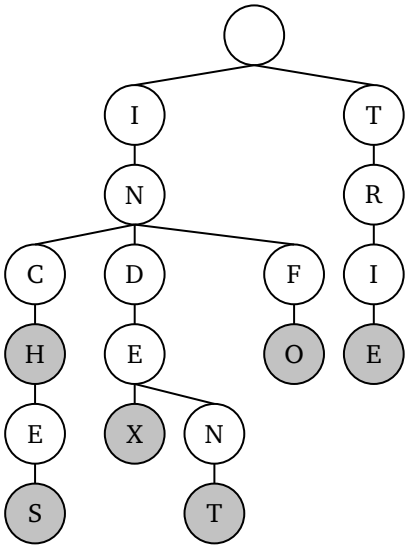
Solution:

The strings originally contained in the trie are *inch, index, and info*.

(ii) Insert the strings *indent*, *inches*, and *trie* into the trie.

Solution:

The trie after inserting *indent, inches, and trie*.



(b) How could you modify a trie so that you can efficiently determine the number of words with a specific prefix in the trie?

Solution:

We can add a `numWordsBelow` variable to each of the nodes in our trie. When we insert we will increment

this variable for all nodes on the path to insertion. In order to determine the number of words that start with a specific prefix, we can traverse the trie following the letters in the prefix. Once we reach the end of the prefix, we return `numWordsBelow` of the last character in the prefix, or 0 if the entrie prefix is not contained in the tree. If the length of the prefix is k then this code will run in $\Theta(k)$ in the worst case. If we have the case that the lengths of the strings will be assumed to be a constant, then this runtime of $\Theta(k)$ will actually be $\Theta(1)$ as we drop the constant coefficients.