Code Task 1

Chica Gomes, Christian Tuttle, Russell Cannon, Stephen Bruner III

February 13th, 2025

Abstract

Sedgewick, Robert. Algorithms in C, Parts 1-4. Available from: Colorado Mesa University, (3rd Edition). Pearson Technology Group, 1997.

Zivanovi, Saso. Forest: a pgf/TikZ-based package for drawing linguistic trees. Available from: University of Utah.

Geeks for Geeks. Templates in c++ with Examples.

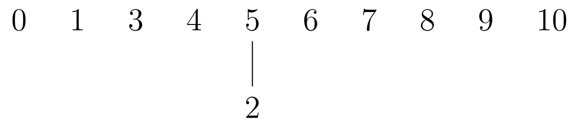
Stack Overflow. C++ Overloading Array Operator.

1 Union-Find Data Structure

Using the weighted Quick-Union with path compression algorithm, the following 11 union operations result in the accompanying tree representations.

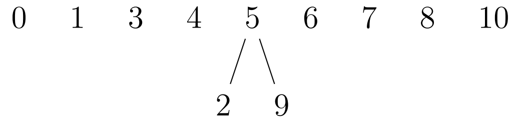
0 1 2 3 4 5 6 7 8 9 10

Operation 1. $Union(2, 5)$

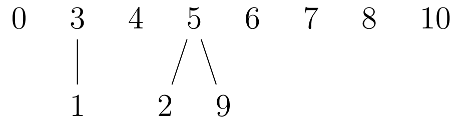


Node 2 is parented to node 5 to union the two.

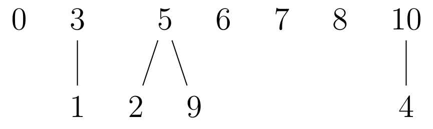
Operation 2. $Union(5, 9)$



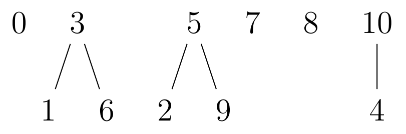
Operation 3. $Union(1, 3)$



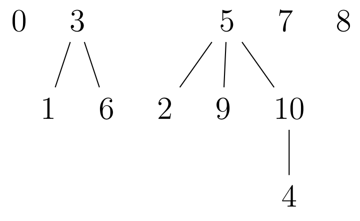
Operation 4. $Union(4, 10)$



Operation 5. $Union(3, 6)$

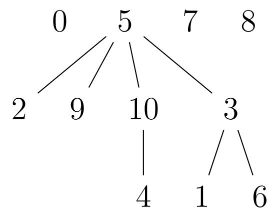


Operation 6. $Union(5, 10)$



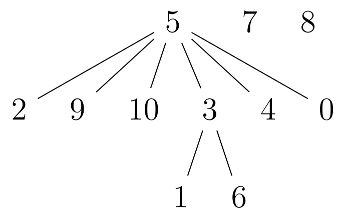
Node 10 is parented to node 5 to union the two. Node 10 also has a child node (4) and is parented alongside.

Operation 7. $Union(1, 2)$



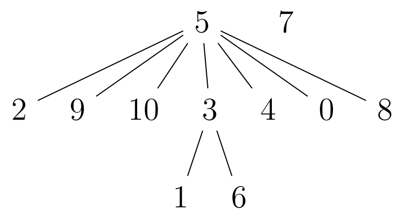
Node 1's root node is node 3 and node 2's root node is 5. To connect node 1 and 2, we have to merge both of their roots. Because node 5 has a much larger tree, we choose to parent node 3 to node 5. The children of node 3 all have longer path lengths after this merge. This is unavoidable, so we chose to parent the smaller tree onto the larger so the least number of nodes increase in path length.

Operation 8. $Union(4, 0)$

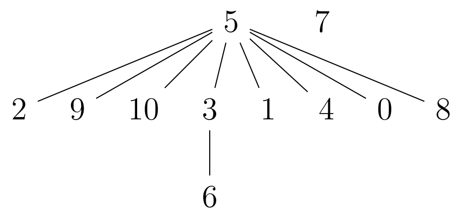


Node 0 is parented to node 4's root: node 5. Because we accessed 4 to find its root node, we are able to compress the path between the node and its root to make it faster to access later.

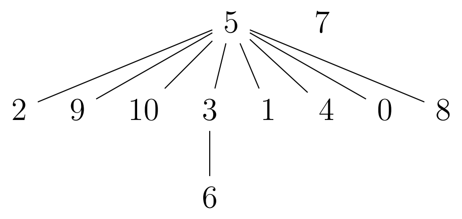
Operation 9. $Union(4, 8)$



Operation 10. $Union(1, 8)$



Operation 11. $Union(2, 0)$



2 Algorithm Analysis

In the worst case scenario, the time complexity is M + N lg(N). For example, if we had N = 10 to the power of 6 and M = 10 to the power of 6, then the time complexity would be close to 20 million. Comparing this to Quick Find, the time complexity would be around 47000 times longer (MN). Below, we created a graph that compares Quick union with path compression to quick find. We set M = 11 and N as the variable X. As you can see, quick-find becomes much slower very quickly.

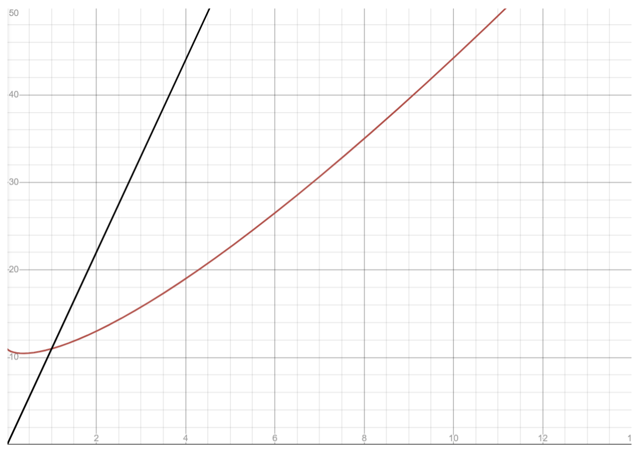


Figure 1: $O(N * M)$ in black vs $M + N * log(N)$ in red. Time: y, N: x, M: 11.

As far as some possible optimizations, weighted quick union with path compression is already a near-linear algorithm. We could possibly find/ implement a better path compression system which may allow the algorithm to get even closer to linear complexity.

3 Additional Task

For the additional task, we implemented a Queue and a stack using a resizing array. To accomplish this, we had a resizing array class that included push and pop functions. These helped us to add or remove an element from the array. In order to be able to resize the array, we had two private member functions called double and quarter both of which would create a new array either twice or a fourth of the size of the previous array, then move all the elements from the old array to the new array.

Then we had a class for both the stack and queue. The main difference between the two is stack is last in - first out and a queue is first in - first out. This basically means that we need to be careful about which elements we are popping for each of the classes. Finally, in the main file, we first used the stack and queue for adding and removing numbers. Then we used a stack to reverse a string and used two stacks to simulate a queue.

Possible optimizations:
For the Resizing Array queue implementation, pushing and popping is $O(N^2)$. Every time an element is pushed onto or popped off of the queue, the array shifts one element backwards or forwards, respectively. A common workaround for this is to keep track of a two indices in the array for the head and tail of the queue. This allows us to make pushing and popping elements $O(N)$ but also makes resizing the array more complicated. This occurs because if the head moves far enough away from index 0, the tail can hit the end of the array without the actual data being larger than the array. This is solved by looping around the end of the array back to the start. Unfortunately, this also means the tail can be behind the head, complicating the array further. All this complexity means when we resize the array, we must unravel the queue and move the head back to index 0 in the new array.