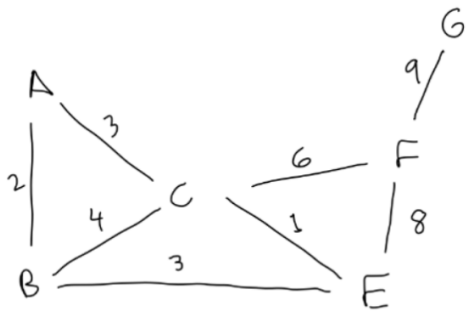
1: Union-Find Data Structure

1. For the graph below, use Kruskal's algorithm, paired with the array-based Union-Find Data Structure, to find the MST. Walk through the intermediate stages of the Union-Find data structure——you should start by initializing the Union Find Data Structure with 6 singleton sets, and then slowly merge them together while updating set membership as Kruskal's algorithm proceeds.



2. Repeat the above problem but use the tree-based Union-Find Data Structure instead.

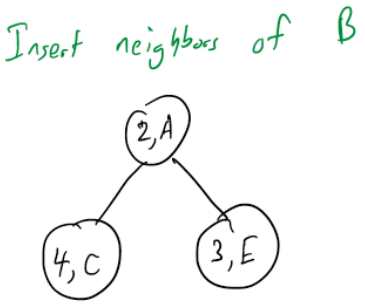
3. When merging two sets of the same size, we always merge the smaller set into the larger set. What happens if you instead always merge the larger set into the smaller set?

   (a) How slow can it be when you merge the larger set into the smaller set for Kruskal's with an array-based Union-Find? Give an example of a graph where the runtime is maximally bad with the reversed set merging rule.

   (b) What if we used the tree-based Union Find instead with the reversed union merging rule? Where do things go wrong? Give an example of a graph that is maximally bad.

2: Heaps

1. For the same graph above, walk through the intermediate states of a min-heap when implementing Prim's algorithm, where we choose B as our starting node. The first step after inserting neighbors of B is below:



2. When using the makeheap operation to construct a heap, we normally start from the bottom, and sift down, which gives us a linear runtime. True of False: if we instead start from the top, and sift up instead, it will be the same speed as if we just repeatedly called insertkey on every item.

3. Normally, we use a complete binary tree for heaps, where every node has two children, but any tree can be heap-ordered. Suppose you have a tall and thin heap-ordered tree, where every node has one child instead of two. How does this change the runtime of findmin, insert, delete, and makeheap?

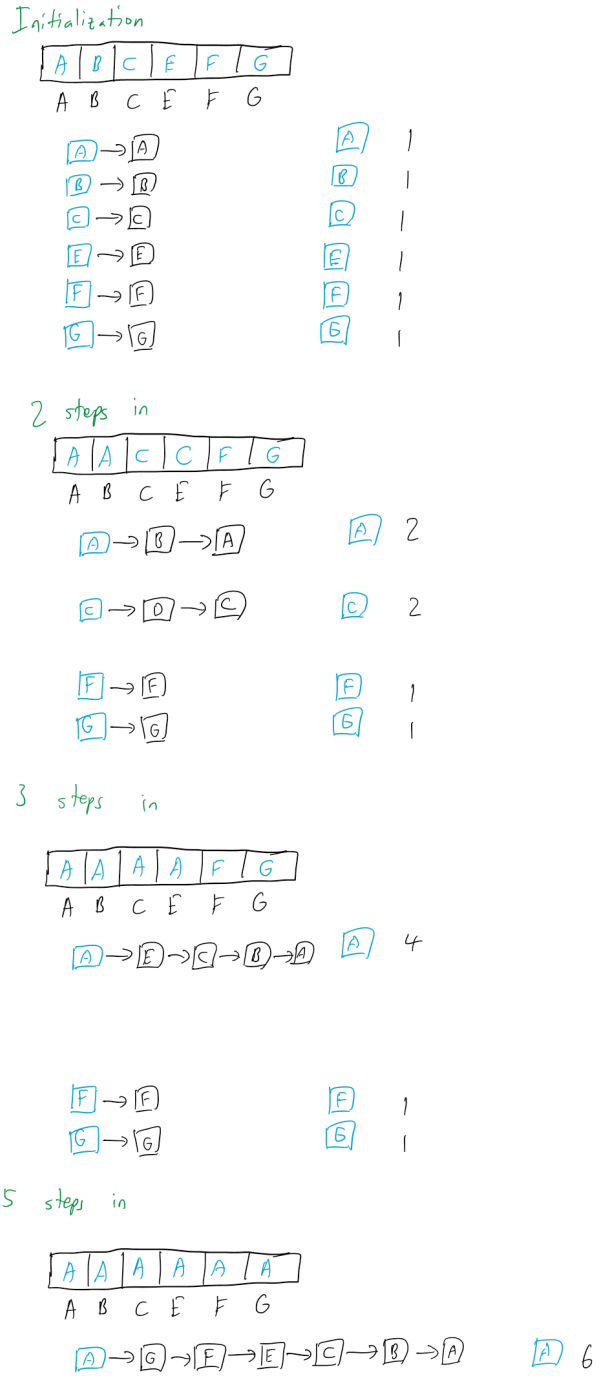
Helpful Videos:

- (Tree-based) Union Find in 5 minutes, Potato Coders, `https://www.youtube.com/watch?v=ayW5B2W9hfo`
- Heaps algorithm in 3 minutes, Micheal Sambol, `https://youtu.be/0wPlzMU-k00?si=jmm0M2P3k4gutFCs`

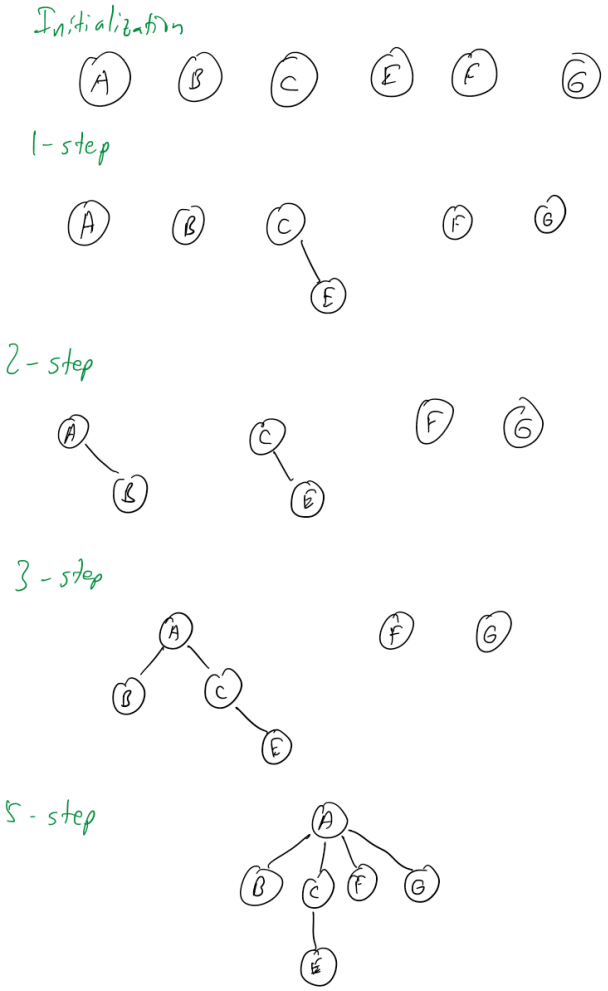
3: Solutions

1 Union Find Data Structure

Intermediate steps for array-based Union-Find



2. Tree-based Union-Find Data Structure Intermediate steps

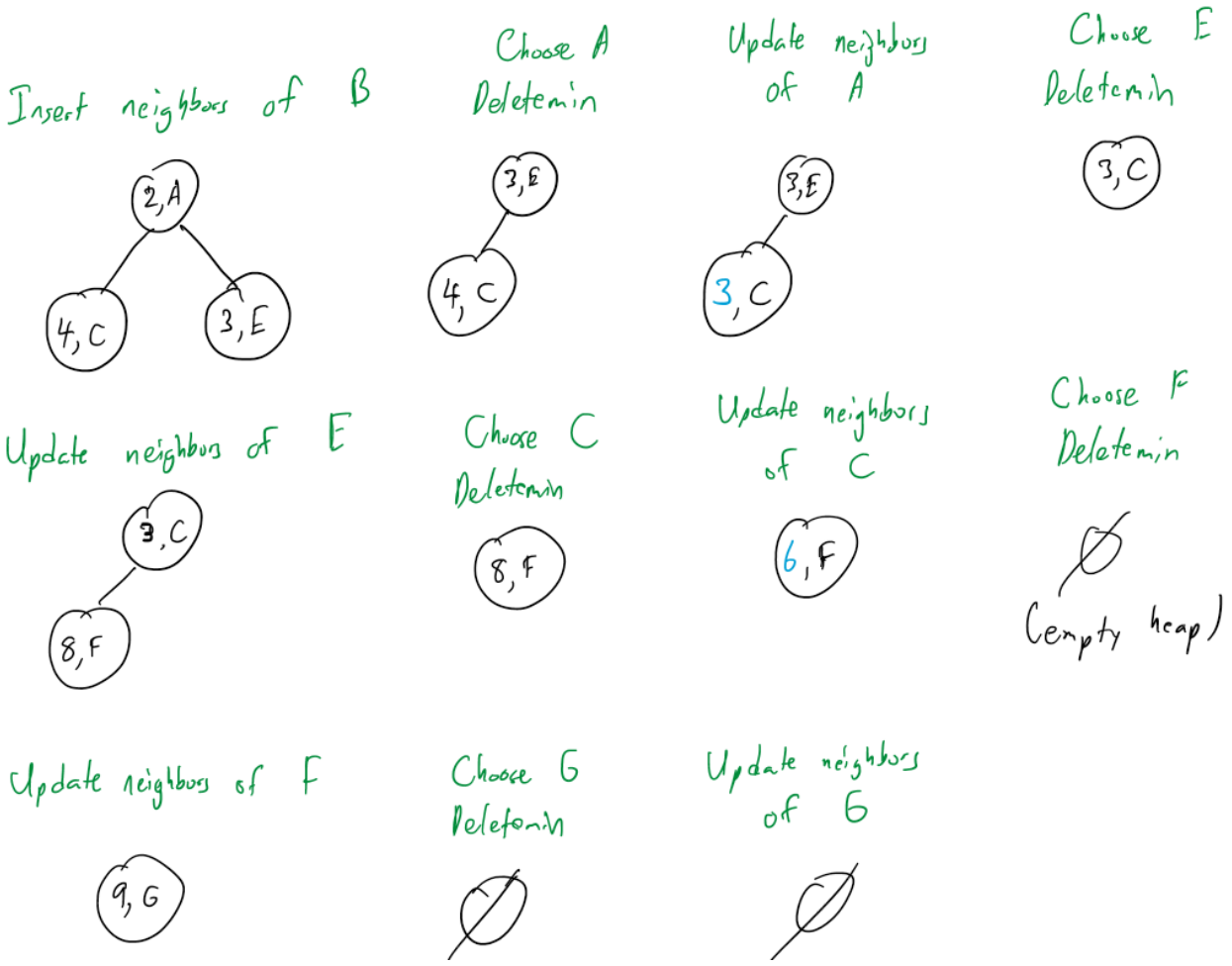


3a. The runtime can be as bad as quadratic $O(n^2)$. Consider a line graph, with the edges in increasing order of weight along the line. Then Kruskal will obviously need to include every edge, but each time, you will need to relabel every node that's already been touched before, since we are relabelling the bigger set according to the smaller set. Thus, there will be a quadratic number of relabellings.

3b. The runtime can also be as bad as quadratic $O(n^2)$. Consider a star graph, where all nodes are connected to a single central node. Checking whether two nodes are in the same connected component takes time proportional to tree depth. But if we point the larger set at the smaller set, the tree-based UF will end up as a very tall tree with depth equal to the number of nodes, with the central node at the bottom. Every time we add a node, we have to check the central node against the new node, but that takes time proportional to the tree height. Thus, the checking time in total will be quadratic.

2 Heaps

1. Full walkthrough is below. Note that you often have to update the weight on a node. Luckily, we didn't have to do any sifting when updating weights.



2. True. Indeed, starting from the top and sifting up is exactly the same process as calling insertkey, because insertkey just adds in the next open tree position, and sifts up.

3. findmin: $O(1)$

insert/delete: $O(n)$ (because might have to sift all the way up the tree, and it's not really a tree since it doesn't branch

makeheap: $O(n^2)$. This is basically bubble sort.