Homework 3

Question 1

The algorithm will be based on the following logic:

- If an edge (u, v) is deleted from the spanning tree, let T_u and T_v be the trees that were created, rooted at u and v respectively
- Adding any edge between T_u and T_v would not create a cycle, and would create a spanning tree
- In order to preserve this as a spanning tree, we must add the lightest such edge!

We have used both of the last two claims in previous algorithms and therefore their proof is omitted.

Rephrasing the logic above, we can state that the edge to be added as a replacement for a deleted edge, is the lightest among all the edges going out of some $w \in T_v$ and into T_u. So, what we'd like to do is to be able to pick out the lightest edge going out of every edge to the other tree (once the tree is split due to a deletion). To do so, we'll use the following data structure:

- For every $v \in V$ maintain a copy of the current spanning forest (denote this as F_v)
- The copies of the current spanning forest (F_v for $v \in V$) will be represented as a Euler-Tour trees
- For each node $w \in F_v$, it's weight will be the weight of the edge (v, w) in the graph G
  - Note that we are talking about the original graph, not the MST
  - If the edge (v, w) doesn't exist, the weight of w in F_v would be $+\infty$

Beginning with G as an empty graph (only nodes without edges) is trivial. To maintain G and it's MST, here is the exact logic we are going to apply:

Insert((u, v), Δ)

```
1. Update the weight of $u$ (in $F_v$) and $v$ (in $F_u$) to $\Delta$ - $O(\log n)$ work
2. Check if $u$ and $v$ are connected by running a BFS (from $v$) on the existing MST - $O(n)$ work
3. If $u$ and $v$ are not connected, add the edge $(u, v)$ to the MST with the weight $\Delta$
    a. Update each copy of the spanning forest (in the different nodes) - $n \cdot O(\log n)$ work
4. If $u$ and $v$ are connected, then the BFS found a cycle containing $(u, v)$
    a. The cycle can be tracked by following the edges that were taken by the BFS
    b. Sort the edges along the cycle by weight - $n \cdot O(\log n)$ work
    c. If $(u, v)$ is the heaviest edge on that cycle, do nothing
    d. Otherwise, let $(w, x)$ be the heaviest edge on the cycle
        i. Remove $(w, x)$ from the MST and insert $(u, v)$ instead
            1. Update all the copies of the spanning forest - $n \cdot O(\log n)$ work
            2. Update the weights of $w$ in $F_x$ and $x$ in $F_w$ to $+\infty$ - $O(\log n)$ work
```

Delete((u, v))

1. Update the weight of u (in F_v) and v (in F_u) to $+\infty$ - $O(\log n)$ work
2. Delete the edge (u, v) in all of the copies of the spanning forest - $n \cdot O(\log n)$ work
3. For each node $w \in V$, using T_u and T_v (as defined previously), do:
    a. Find whether w is in T_u or T_v by checking if w is connected to u or v
        i. Do this using **find-tree** on u, v and w - $O(\log n)$ work
    b. Without loss of generality, *assume that* $w \in u$
    c. Find the lightest node in T_v - $O(\log n)$ work
        i. Denote the node as $best(w)$
4. Pick the lightest edge e among the edges $\big(w, best(w)\big)$ that actually exist in G
    a. Once we have all the edges, $O(n)$ work
5. If e does not exist, there is no replacement edge. Stop.
6. Otherwise, add e to the MST
    a. Update all the copies of the spanning forest - $n \cdot O(\log n)$ work

Explanation

At every moment, the following invariants are kept:

- Insertions maintain the MST as in other algorithms (we haven't made any change)
- The weight of u in F_w is $x \Leftrightarrow$ The edge (u, w) exists and has a weight w
    - This is trivially satisfied by our definition of the algorithm
- When we pick $best(w)$, then $\big(w, best(w)\big)$ is the lightest edge going from w to the "other" tree that was created by the deletion of the edge (T_v by our assumption)
- Picking the lightest edge of the form $\big(w, best(w)\big)$ results in picking the lightest edge between T_u and T_v at the moment after (u, v) was deleted
- Adding that edge, preserves our MST under deletions!

Complexity

The time cost of each step is $O(n \cdot \log n)$ where $O(\log n)$ comes from the implementation of the Euler-Tour trees. Implementing them using the right kind of balanced binary trees (red-black trees for example) will result in this time bound being a worst-case time bound.

The space complexity of the entire data structure is made of maintaining the graph with weights ($O(n^2)$) and n copies of the spanning forest with weights (n times $O(n)$ space), resulting in an overall cost of $O(n^2)$.

Question 2

Note: The solution for this question is based on the suggestion from the article of Dietz and Sleator ('87)

We have already seen (in class) a data structure which supports insertion and deletion in $O(\log n)$ amortized time, and query in $O(1)$ worst-case time. Our solution is going to be based on that data structure, which will store "lists" in its leaves. So, let's begin by describing the "lists".

The idea behind the lists is to create a data structure for order-maintenance of a "fixed" number of elements (preventing the problem we had with the amount of bits in labels), that operates in $O(1)$ amortized time per operation (insertion, deletion and query). Using that construction, we will be able to reduce the overall amortized cost of the algorithm.

Fixed-Size Order Maintenance using Lists

Here is the brief of the idea:

- The elements will be stored in linked lists
- The lists will be limited to $\log N$ **elements**
  - N is the amount of elements with which we started the phase, as we described for the order maintenance algorithm
    - A phase lasts as long as $N/2 \leq \#(\text{current number of elements}) \leq 2N$
  - Elements in the list will be labeled $(1,2,\dots,\log N)$
- To insert x after y, append x to the end of the list
  - The label assigned to x will be $1 + (\text{label of last element in the list})$
  - If the list exceeds its maximal size, split it to too smaller lists
    - Each list now contains $1/2 \cdot \log N = O(\log N)$ elements
    - Relabel the elements in both lists to $1,2,\dots,1/2 \cdot \log N$
- To delete x, simply remove it from the list
  - If the list becomes empty, delete it (we'll soon explain the meaning)

Running time analysis:

- Order queries can trivially be done in $O(1)$ time by comparing their labels
- Insertion costs $O(1)$ for the simple case, and $O(\log N)$ for the complex case which happens only once every $O(\log N)$ insertions. The amortized cost is $O(1)$ per insertion
- Deletions cost $O(1)$ per deletion

The entire analysis is assuming that we can find the node containing an element in $O(1)$ amortized time; this can be achieved using a hash-map which is maintained as we insert and delete elements (including creating another hash-map when splitting the list).

Improving the Base Structure

In each phase, we will now hold a complete binary tree of $M/\log N$ leaves (instead of M leaves). Each "taken" leaf will be associated with a list (as described previously). The operations would then work in the following way:

- To insert an element x after y, find the list containing x and try inserting it there
- To delete an element x, delete it from its list
- To compare the order of two elements, compare the labels of their lists (if the labels are equal, they are in the same list – so compare their order in the list using the in-list labels)
  - This costs $O(1)$ under the following modification:

- Instead of having one element-to-node hash-map per list, have one such map from all the elements to their nodes in the list
- Each list node should also have a pointer to the "leaf" representing the list in the tree

And now for more details and complexity analysis:

- Every time we split a list, we do it by inserting the list of the second half as a new element in the tree (after the list of the first half)
    - Only once every $O(\log N)$ insertions we create a new list, so the $O(\log N)$ cost of inserting into the tree is now reduced to amortized $O(1)$ per insertion!
- Every time we delete a list, we do it by deleting it from the tree
    - Only once every $O(\log N)$ deletion we delete a new list, so in a similar fashion to insertions, it is now reduced to amortized $O(1)$ per insertion!

Final considerations:

- When increasing the tree size, we can remain with the small lists (and simply increase their size limit)
- When decreasing the size of the tree, it means we had $O\big((M/\log n)\cdot(\log n)\big) = O(M) > O(N)$ deletions and we now need to recreate the lists for $O(N)$ elements – we can still have the deletions pay for it
    - Note that even though M and N change in each phase, this applies every time again to the M and N of the current phase – so it's not as if we are charging the same deletions over and over…

Question 3

When using the order-maintenance structure, we could have inserted, deleted and queried the order of elements in $O(1)$ amortized time, which lead to a time bound of $O(mn)$ amortized time. In the solution to this question, we'd like to replace the order-maintenance structure (for holding the vertices in topological order) by an array, in order to achieve the same time bound (this time, without the amortization on the final bound).

Returning to the algorithm, let's formalize the exact usage we did with the structure:

- We queried the order of elements (to see when we passed the end of our search range)
- Afterwards, if we inserted an edge (v, w), and the search traversed through k more nodes (denote them as $s_1, \dots, s_k$) then we did the following operations (assuming w is before v):
    - Remove the k nodes and w from the order maintenance structure
    - Add them again (in the same order!) directly after v
- The entire cost of moving the elements around was $O(k)$ (amortized) time

The implementation we are going to provide is not going to work in $O(k)$ time for moving the elements around; instead, it is going to take $O(n)$ time. Although this increases the overall running time of the algorithm, the asymptotic time bound is going to stay the same! This is because we are increasing the running time by at $\cdot\, O(n-k) < O(nm)$, and this does match the required time bound!

So once we agreed to use $O(n)$ time for re-ordering the elements after each insertion, the implementation becomes trivial; here is the pseudo-code:

```
function Update-Topological-Order(current_array, n, v, w, {s_1,s_2,...,s_k})
new_array := allocate array of size n
last := 1

v_index := index of v in current_array

for i in 1,2,…,v_index, do:
    if current_array[i] not in {w,v,s_1,s_2,...,s_k}, then:
        new_array[last] := current_array[i]
        last := last + 1
    end if
end for

new_array[last] := v
last := last + 1
new_array[last] := w
last := last + 1

for i in 1,2,…,n, do:
    if current_array[i] in {s_1,s_2,...,s_k} or i > v_index, then:
        new_array[last] := current_array[i]
        last := last + 1
    end if
end for

current_array := new_array
```

The algorithm appends all the elements that were not involved in the current update (were not scanned from v, and are not v and w) and come before v, to the beginning of a new array. Afterwards, it adds the elements involved in the scan from v (including v and w) preserving their order. Finally, it appends all the elements that come after v to the end of the array. This behavior is equivalent to the one of the original algorithm and is therefore correct! The only remaining part is to analyze the running time:

- We had 2 loops of $O(n)$ elements that cost $O(n)$ time
    - Checking if the current element is in the set $\{s_1, \dots, s_k\}$ can be done in $O(1)$ time by simply marking those elements when they are encountered ub the scan with some bit flag, and clearing the flag at the end of the running of the update
- We had one allocation of an array of size $O(n)$, which costs $O(n)$ time

We can see that the overall running time for maintaining the topological order after each edge insertion is $O(n)$, and so we meet the required running time of $O(mn)$!