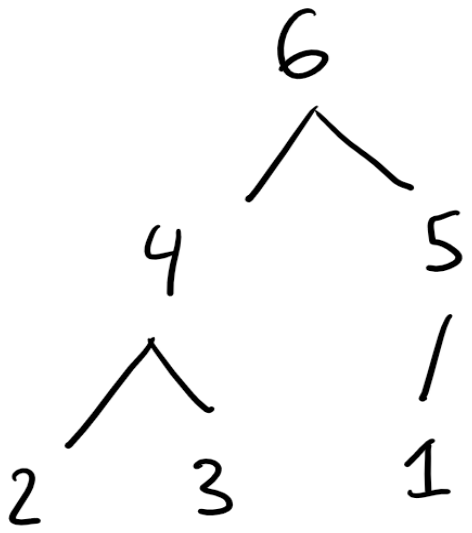
Question 1

Note: On Piazza, we were told that we can hand-draw our heaps, and insert them into LaTeX.

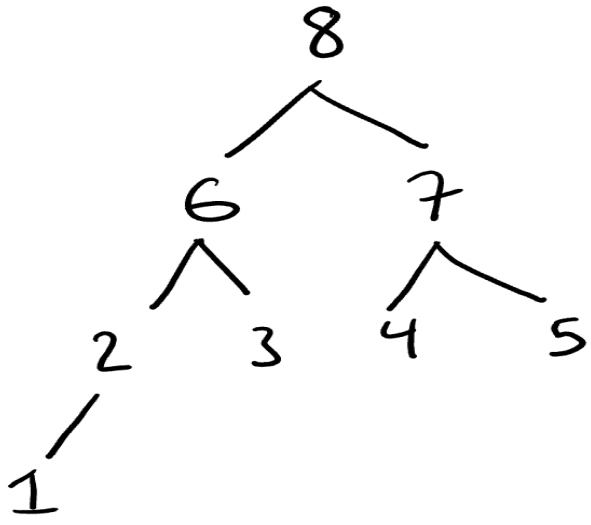
Part (a)

The following is an EIS max-heap with 6 elements.



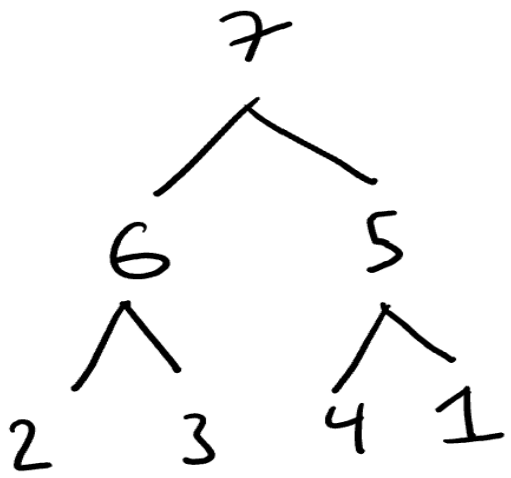
Part (b)

The following is a right-dominant max-heap with 8 elements.

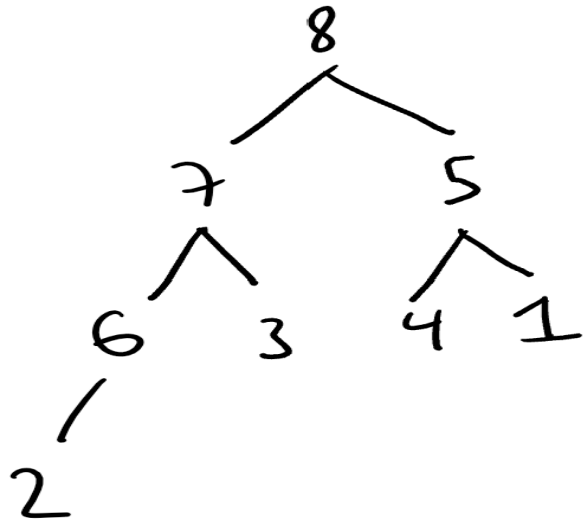


Please see the next page.

Below is the heap after ExtractMax.



Below is the heap after re-inserting the root. Notice it is not the initial heap. Hence, our initial heap was a right-dominant max heap with at least 6 nodes, but was not EIS.



Part (c)

Statement: Any right-dominant heap H is an EIS heap if and only if it has $2^i - 1$ or $2^i - 2$ nodes, for some natural number i.

Proof. Let H be an arbitrary right-dominant heap.

Definition: The right-most path of H is the path P of nodes of H starting from the root, and moving rightwards (whenever possible) until reaching a leaf node. Note, we only possibly move left on P once when possibly reaching a leaf node.

($\Leftarrow$): Assume H has either $2^i - 1$ or $2^i - 2$ nodes, for some $i \in \mathbb{N}$. We will prove each case separately.

Case 1: Assume H has $2^i - 1$ nodes. Hence, H is a full binary tree where every level of H is completely filled with nodes. Let P be the right-most path of H as defined earlier. Let y be the rightmost node on the last level of H. Let x be the parent of y. Since H is completely filled with nodes on every level, we have that y is on the path P and y has some sibling z where $y > z$ and z is the left child of x and y is the right child of x since H is right-dominant. Now apply ExtractMax. We know that the root will first be removed and replaced with y. When bubbling down y, since H is right-dominant, we know that y will be swapped with nodes that are right children on the path P until we reach the second-to-last level of H where y is now a parent of z. Since H is right-dominant and y and z were initially siblings, we know that $y > z$. Hence, y will not be swapped with z when bubbling down. Now when reinserting the root, the root will first be placed in y's original position in H as the sibling of z. Now the root will be bubbled up, first swapping with y. Now y is back to its original position. And the root will continue to be bubbled up on our path P until it reaches its original position. Since all swaps occur on the node positions on the path P, we have that H remains unchanged after ExtractMax and Insert. Hence, H is EIS.

Case 2: Assume H has $2^i - 2$ nodes. Hence, H is a nearly full binary tree where the last level of H has 1 node position (the rightmost position of the last level) that is not filled. Let P be the right-most path of H. Let y be the rightmost node on the last level of H. Note, y is still on the path P. Let x be the parent of y. Since H is not completely filled, we know y is the left child of x and that y has no sibling. Now apply ExtractMax. We know that the root will first be removed and replaced with y. When bubbling down y, since H is right-dominant, we know that y will be swapped with nodes that are right children on the path P until y reaches the second-to-last level of H where y is now in x's original position. Now, when reinserting the root, the root will first be placed as a left-child of y. The root is now in y's original position. When bubbling up the root, we first swap the root with y. Now y is in its original position. We continue to bubble up the root on the path P until it reaches its original position. Since all swaps occur on the node positions on the path P, we have that H remains unchanged after ExtractMax and Insert. Hence, H is EIS.

Therefore, in either case we have that H is EIS.

($\Rightarrow$): We will prove the contrapositive. Assume H does not have $2^i - 1$ or $2^i - 2$ nodes for any $i \in \mathbb{N}$. Hence, we know that H has at least 4 nodes and that the last level of H must have at least 2 node positions that are not filled. Let P be the right-most path of H. Let y be the right-most node on the last level of H. Since H must have at least 2 nodes on the last level that are not filled, we have that y is **NOT** on the path P. Now apply ExtractMax. The root will be removed and replaced with y. Since H is right-dominant, y will be bubbled down with nodes on the path P. Now we reinsert the root into y's original position which is a node position not on the path P. The root will then be bubbled up but will not make any swaps with y which is currently on the path P. Hence, after the root is bubbled up to its original root position, we have that y is not in its original position anymore. Hence, H has changed after ExtractMax and Insert. Hence, H is not EIS which completes this direction of the proof (using the contrapositive).

This completes the proof of the statement, as required. $\square$

Question 2

Part a)

First consider the following implications for distinct commuters c_i and c_j. Note, $d > 1$.

$$
\begin{aligned}
|c_i - c_j| \leq d &\Rightarrow -d \leq c_i - c_j \leq d \\
&\Rightarrow -1 \leq \frac{c_i}{d} - \frac{c_j}{d} \leq 1 && \text{Since } d > 1 \\
&\Rightarrow -1 + \frac{c_j}{d} \leq \frac{c_i}{d} \leq 1 + \frac{c_j}{d} \\
&\Rightarrow \left\lfloor -1 + \frac{c_j}{d} \right\rfloor \leq \left\lfloor \frac{c_i}{d} \right\rfloor \leq \left\lfloor 1 + \frac{c_j}{d} \right\rfloor && \text{By properties of floor} \\
&\Rightarrow -1 + \left\lfloor \frac{c_j}{d} \right\rfloor \leq \left\lfloor \frac{c_i}{d} \right\rfloor \leq 1 + \left\lfloor \frac{c_j}{d} \right\rfloor && \text{By properties of floor} \\
&\Rightarrow \left\lfloor \frac{c_i}{d} \right\rfloor \in \left\{ -1 + \left\lfloor \frac{c_j}{d} \right\rfloor, \left\lfloor \frac{c_j}{d} \right\rfloor, 1 + \left\lfloor \frac{c_j}{d} \right\rfloor \right\}
\end{aligned}
$$

Hence, if c_i and c_j are d-near-standers, then $\left\lfloor \frac{c_i}{d} \right\rfloor$ and $\left\lfloor \frac{c_j}{d} \right\rfloor$ differ by at most 1.

We will use a hash table to send index i of commuter c_i to hash bucket $\left\lfloor \frac{c_i}{d} \right\rfloor$ if it does not already contain some index. If it does contain some index j already, then we will check to see if c_i and c_j are d-near-standers. We will also similarly check buckets that are 1 more and 1 less than $\left\lfloor \frac{c_i}{d} \right\rfloor$ as in our implications above.

On Piazza, the instructor indicated that for this question we can assume that hashing takes constant time. Hence, we will assume each hashing operation (insert, access, etc.) take constant time.

Please see the next page to see our algorithm NearStanders.

Consider the following algorithm.

NearStanders($[c_1, ..., c_n], d$)

```
1: Let T be a new hash table with n (or more) buckets. Each bucket will contain at most
   1 index from {1, ..., n} at any given time.
2:
3: for each i from 1 to n do
4:     We will check each bucket in {−1 + ⌊c_i/d⌋, ⌊c_i/d⌋, ⌊c_i/d⌋ + 1} first.
5:
6:     j ← −1 + ⌊c_i/d⌋
7:     if T[j] is a nonempty bucket containing some index k then
8:         if |c_i − c_k| ≤ d then
9:             return i and k
10:        end if
11:    end if
12:
13:    j ← ⌊c_i/d⌋
14:    if T[j] is a nonempty bucket containing some index k then
15:        if |c_i − c_k| ≤ d then
16:            return i and k
17:        end if
18:    end if
19:
20:    j ← ⌊c_i/d⌋ + 1
21:    if T[j] is a nonempty bucket containing some index k then
22:        if |c_i − c_k| ≤ d then
23:            return i and k
24:        end if
25:    end if
26:
27:    key ← ⌊c_i/d⌋
28:    Insert index i into T[key]                    ▷ If we had not returned yet
29: end for
30: return None                                      ▷ If there are no d-near-standers
```

Please turn to the next page.

Part b)

Required: Prove a tight asymptotic bound on the worst-case runtime of your algorithm.

Want to Show: The worst-case runtime of NearStanders is $\Theta(n)$.

Let n be the number of distinct commuters. We know that our for loop has at most n iterations. Each iteration of the for loop takes at most $\mathcal{O}(1)$ time given our assumption that hashing takes constant time. Hence, NearStanders runs in at most $\mathcal{O}(n)$ time in the worst case.

Now we will show an omega bound. Consider a family of inputs, one for each $n \in \mathbb{N}$ defined as follows. Let $c_i = i(d+1)$ for $i \in \{1, ..., n\}$ and $d > 1$. Hence, our list of commuters is $[c_1, ..., c_n] = [d+1, 2(d+1), 3(d+1), ..., n(d+1)]$. Hence, there are no near-d standers amongst our commuters as every pair of distinct commuters c_i and c_j are at least $d+1$ distance apart. Hence, our algorithm will return **None**. Hence, our algorithm will never return earlier than line 30. Hence, we will have at least n iterations of the for loop. Each iteration of the for loop takes constant time given our assumption that hashing takes constant time. Hence, NearStanders runs in at least $\Omega(n)$ time.

Since we have $\mathcal{O}(n)$ and $\Omega(n)$, we get a tight worst-case runtime of $\Theta(n)$.

Question 3

Part a)

We will use a min heap (implementing the priority queue ADT) as well as an AVL tree (implementing the dictionary ADT).

The min heap will have keys/priorities representing star powers, and items representing team names. There may be multiple nodes with the same star power.

Each node in the AVL tree will have a key representing a team name, and value that is a pointer to the corresponding node in the min-heap representing this team. Note that the BST property will compare keys based on alphabetical ordering.

The min heap will allow for `FindMin, ExtractMin, Insert` operations, and the AVL tree will allow for `Search, Insert, Delete` operations. Note that we will refer to any `Insert` operations with **respect to** either the min heap or AVL tree to avoid ambiguity in our discussion. And `Delete` in the AVL tree will **only** be called on a team name that was an item from a node returned from a call to `ExtractMin` on the min heap.

Assume that our data structure is initialized with the n teams. i.e. for each team, we have a node in the min heap with key/priority representing the team's star power, and item representing the team's name. Furthermore, there is a node in the AVL tree with key that represents this team's name, and whose value is a pointer to the corresponding node in the min heap. The min heap and AVL tree each have n nodes thus far.

Part b)

Assume that our data structure is initialized with the n teams as discussed in part a).

Consider the following implementation of `Trade`(S_1, S_2, x).

Assume that there exists teams in our data structure with names S_1 and S_2. First use `Search` on the AVL tree to find the node with key S_1 and use `Search` on the AVL tree to find the node with key S_2. Consider the values that keys S_1 and S_2 correspond to. These values are pointers to corresponding nodes in the min heap. Call these pointers p_1 and p_2 for S_1 and S_2 respectively.

Use p_1 and p_2 to access the corresponding nodes in the min-heap. These nodes have keys/priorities representing the respective teams' star powers. Let k_1 and k_2 be the keys in the min-heap whose nodes correspond to the pointers p_1 and p_2 respectively. Decrease k_1 by x and then bubble up this node with key $k_1 - x$ in the min heap. Increase k_2 by x and then bubble down this node with key $k_2 + x$ in the min heap.

Since we are only using `Search` twice in the AVL tree as well as bubble-up and bubble-down

in the min-heap, we know that all of these four sub-procedures take $\mathcal{O}(\log n)$ time each in the worst-case as seen in lecture. And accessing pointers take constant time.

Hence, `Trade`(S_1, S_2, x) takes $\mathcal{O}(\log n)$ time in the worst-case.

Part c)

Assume that our data structure is initialized with the n teams as discussed in part a).

Consider the following implementation of `Eliminate_and_replace(T)`.

Use `ExtractMin` on the min-heap to remove a node in the min heap with the lowest star power (minimum key). Look at the item (team name) corresponding to this minimum key. Call this team name $name_0$. We will eventually return $name_0$.

Use `Delete` on the AVL tree to delete the node in the AVL tree with key $name_0$.

Consider the team T. Let t be its star power, and let $name_1$ be its team name. Use `Insert` on the min heap to insert a node with priority t and item $name_1$. Create a pointer for this node in the min heap. Call this pointer p. Use `Insert` on the AVL tree to insert the node with key $name_1$ and value p which is the pointer to the corresponding node in the min-heap.

Finally, return $name_0$.

We know `ExtractMin` and `Insert` in the min heap each takes $\mathcal{O}(\log n)$ time in the worst-case from lecture. We know `Delete` and `Insert` in the AVL each takes $\mathcal{O}(\log n)$ time in the worst-case from lecture.

Hence, overall we have that `Eliminate_and_replace(T)` takes $\mathcal{O}(\log n)$ time in the worst-case.

Possible Alternative Solution using Hashing

We could replace our use of an AVL tree with a hash table. We decided to use an AVL tree as it is sufficient to meet the asymptotic $\mathcal{O}(\log n)$ runtime requirements in this question. The runtime of AVL trees is much more straightforward. We do not need to make any hashing assumptions, etc.