ESC190 Midterm 1

Sample Solution

And Grading Scheme

Question 1a

Bob is correct, an enqueue operation has worst-case time complexity in $O(\log n)$ (tight upper bound). This is because in the worst case, the number of swaps after an insertion is-upper bounded by the height of the heap plus 1. The height of the heap is given by ceil(log(n+1) - 1), so the number of swaps is upper bounded by ceil(log(n+1)). This is on the order of log(n), so the worst-case time complexity is in $O(\log n)$.

Alice is also correct, an enqueue operation has worst-case time complexity in $O(n \cdot \log(n))$ (weaker upper-bound). Since $O(n \cdot \log(n))$ represents an upper-bound, we have that $O(\log(n))$ is a subset of $O(n \cdot \log(n))$, so an enqueue operation also has worst-case time complexity in $O(n \cdot \log(n))$.

+0.5 for saying **both** Bob and Alice are correct

+0.5 for a reasonable explanation (for **either** Alice or Bob, or both) involving the height of the heap (or something about how worst-case involves doing comparisons/swaps up to the root).

Question 1b

Best-case runtime complexity is in **Ω**(1) & lower and in O(1) & higher. Since the actual best-case runtime is 0, the best-case complexity will never pass the constant function nor n, asymptotically. Thus, the best-case time complexity is in O(1) **and** in O(n) as they act as upper bounds of the constant function. A similar argument holds for **Ω**.

Worst-case runtime complexity is in **Ω**(n^2) & lower and in O(n^2) & higher. Since the actual worst-case runtime is n^2, the worst-case complexity will never be lower than n^2 nor n, asymptotically. Thus, the worst-case time complexity is in **Ω**(n^2) **and** in **Ω**(n) as they act as lower bounds of n^2. A similar argument holds for O.

Correct answers: C, D, E, F

If 4 or less answers circled: +0.5 for every correct answer

If more than 4 answers circled: -0.5 for every incorrect answer

Question 2a

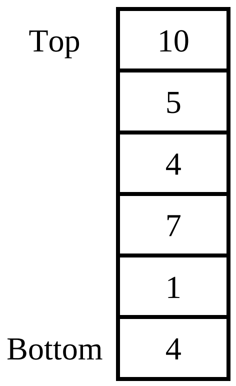
-0.5 for 1 misplaced number

-1 for 2-3 misplaced numbers

-1.5 for 4 misplaced numbers

-2 for 5+ misplaced numbers

+0.5 if the order is correct but bottom and top are flipped



Question 2b

```
stack = [root]
curr = root
while stack != [] :
        curr = stack[-1]
        if   curr.left:
               curr = curr.left
               stack.append(curr)
        elif  curr. right:
                curr = curr.right
                stack.append( curr)

         else :
                curr = stack.pop()
                self.delete_node(curr)
                # deleted node, parent
                # may still have child
                # cannot delete it yet
```

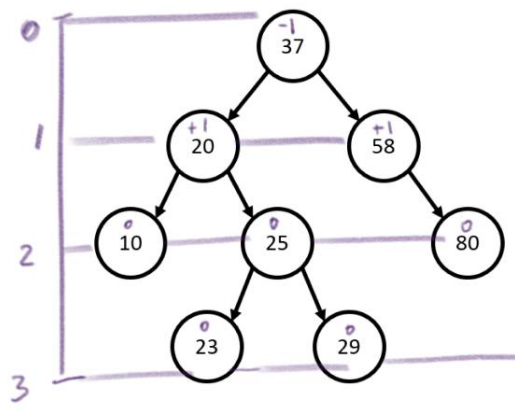
0/3: Code makes no sense whatsoever, or question was left blank.

1/3: Code didn't make much sense, but knew to use post-order traversal, tried to check if children existed before deleting, or knew the loop should terminate when the tree is empty.

2/3: Knew what to do, made small mistakes. E.g., assumed nodes had parent attributes.

3/3: Solution works, even if inefficient. Overlooked minor mistakes that don't affect the algorithm (e.g., calling delete instead of delete_node).

Question 3a



+1 if all balance factors are correct

+0.5 if 1 or 2 are incorrect

+0 if at least 3 are incorrect

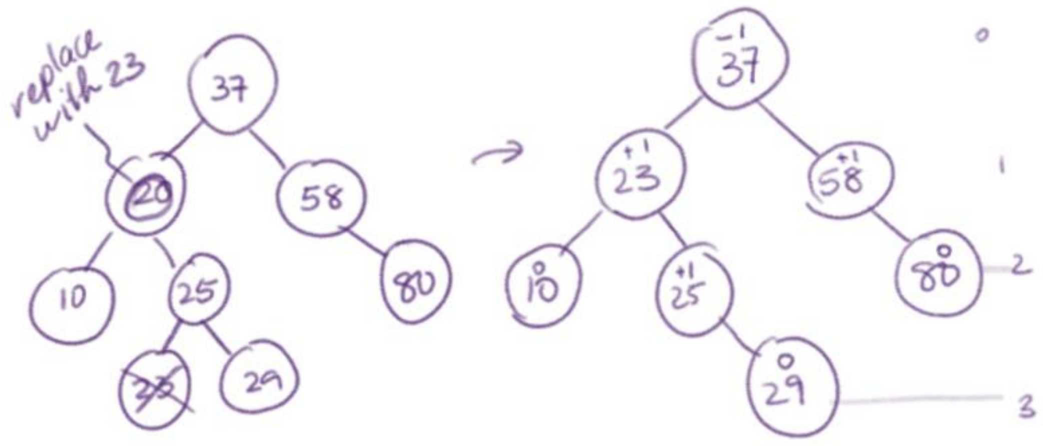
If a student uses bf=h(left)-h(right)

+0.5 if all of them are correct

+0 otherwise

Do not subtract marks for parts b and c for the same mistake

Question 3b



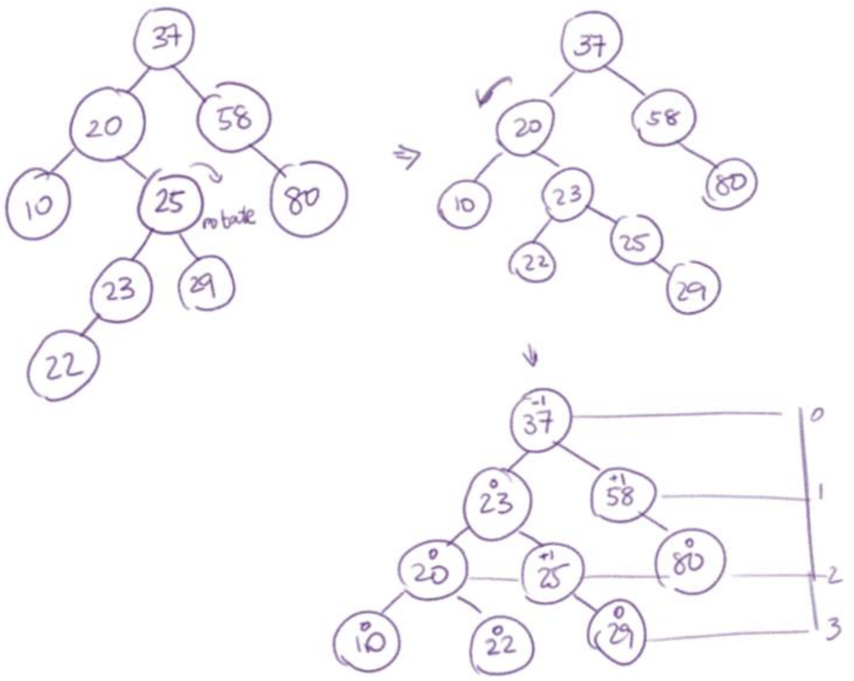
+1 for correct tree

+1 if all balance factors are correct

+0.5 if 1 or 2 are incorrect

+0 if at least 3 are incorrect regardless of the correctness of the final tree

Question 3c



+0.5 for correct insertion

+1 for correct steps 2 AND 3 (if the insertion step is incorrect, can still get +1 if the tree is adjusted properly)

If at least one of step 2 or 3 is incorrect, (e.g. by left rotate(20) then right rotate(37)), +0.5 for getting an AVL tree as the end result regardless of the correctness of the final tree

+0.5 if all balance factors are correct

+0 otherwise regardless of the correctness of the final tree

If a student inserts the node into the tree from part b,

+0.5 for correct insertion

+0.5 for all balance factors are correct

Question 4a

```
def print_non_increasing(root):
     '''
     BST -> None
     Prints the data of each node in the BST indicated
     by root in non-increasing order
     '''
     if root:
          print_non_increasing(root.right)
          print(root.data)
          print_non_increasing(root.left)
```

[0.5 points] Correct base case
[0.5 points] Making two recursive calls and a print statement
[1.5 points] Placing the above in the correct order

Question 4b

$O(n)$: there are at worst n recursive calls on the call stack at a time since it is a regular BST, not an AVL tree.

+1 Correct answer

Question 5a

```
def union(S1, S2):
     flat_s1 = BST_inorder(S1)
     flat_s2 = BST_inorder(S2)
     merged_s = merge_lists(flat_s1, flat_s2)
     root = Node(merged_s.pop())
     bst = BST(root)
     for item in merged_s:
          balanced_bst_insert(bst, item)
```

+ 1: code contains some notion of reading from S1 and/or S2

  -0.5: traversal of tree does not explore all nodes or incorrectly treats trees as iterable

+ 1: code contains some notion of merging the elements of S1 and S2

  -0.5: input of merge not sorted or incorrect merging function implemented

+ 1: code contains some notion of building a final return value

  -0.5: does not return an AVL tree

-0.25: minor syntax errors

Question 5b

The solution for 5b depends on your implementation in 5a.

Alternative Solutions:

1. Get an inorder list of one BST (say S1). Insert all elements of S1 into S2. This would take worst-case time complexity in $\Theta(n_1 \log(n_1 + n_2))$. (Swap n1 and n2 if you insert S2 into S1).
2. Create a new BST, create two inorder lists from S1 and S2, and insert S1 elements into the new BST followed by S2 elements. This would take worst-case time complexity in $\Theta(n_1 \log(n_1) + n_2 \log(n_1 + n_2))$. (Swap n1 and n2 if you insert S2 elements before S1 elements).
3. Create two sorted lists, merge them, then insert the merged list entries into a new BST from the beginning of the list. This would take worst-case time complexity in $\Theta((n_1 + n_2) \log(n_1 + n_2))$.

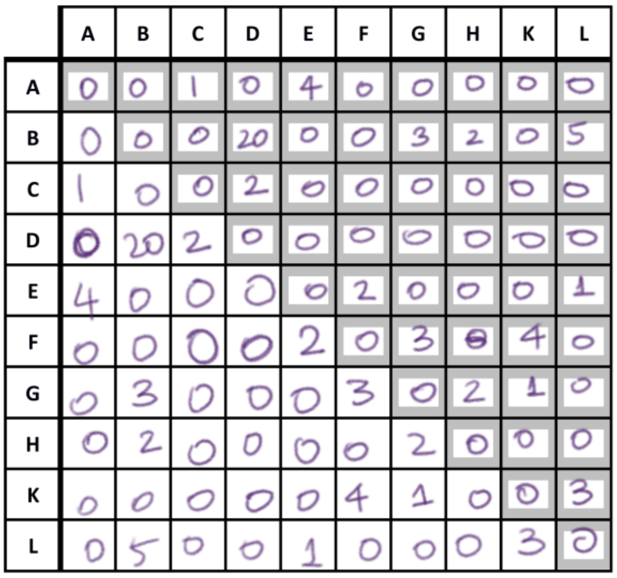
4. Create two sorted lists, merge them, then insert the merged list into the BST by starting at the middle of the array, making this the root, and repeating for the left and right halves of the list recursively. This would take worst-case time complexity in $\Theta(n_1 + n_2)$.

   +0.5: demonstrates balanced_insert $\Theta(\log n)$ and bst_inorder $\Theta(n)$

   +0.25: notion of combining complexity

   +0.25: correctly concludes what the complexity of their algorithm is

Question 6a



	A	B	C	D	E	F	G	H	K	L
A	0	0	1	0	4	0	0	0	0	0
B	0	0	0	20	0	0	3	2	0	5
C	1	0	0	2	0	0	0	0	0	0
D	0	20	2	0	0	0	0	0	0	0
E	4	0	0	0	0	2	0	0	0	1
F	0	0	0	0	2	0	3	0	4	0
G	0	3	0	0	0	3	0	2	1	0
H	0	2	0	0	0	0	2	0	0	0
K	0	0	0	0	0	4	1	0	0	3
L	0	5	0	0	1	0	0	0	3	0

-1 Missing subdiagonal or superdiagonal (With notes like “symmetric” or “mirrored” the mark was still deducted)

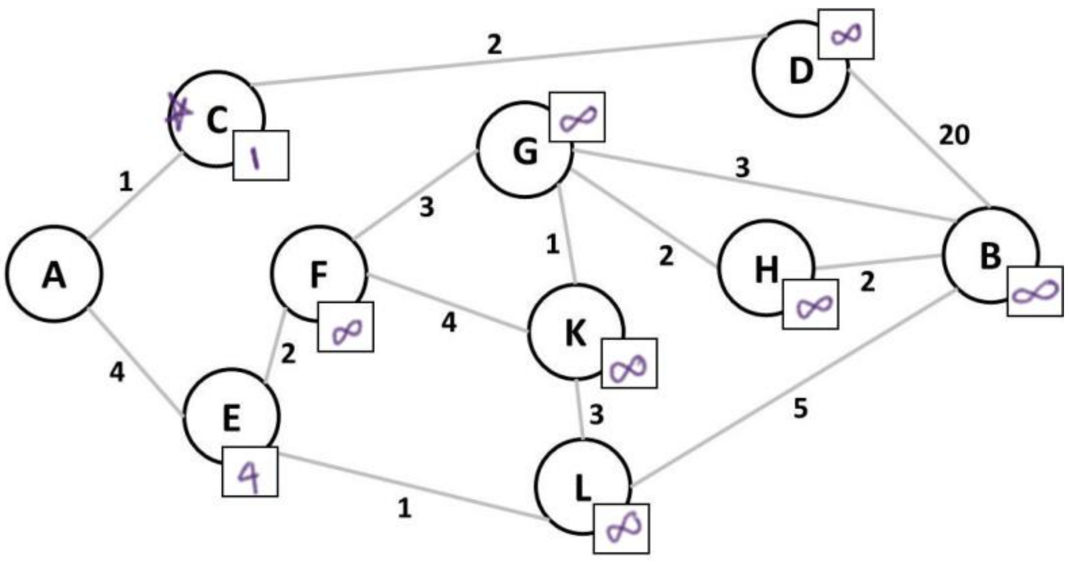
-0.5 Blanks instead of 0s or inf (unless put a note)

-1.5 Missing diagonal values (Note: “x” on the diagonal counts as “missing”)

-1 Nonzeros on the diagonal

-2 did not consider the weights

Question 6b

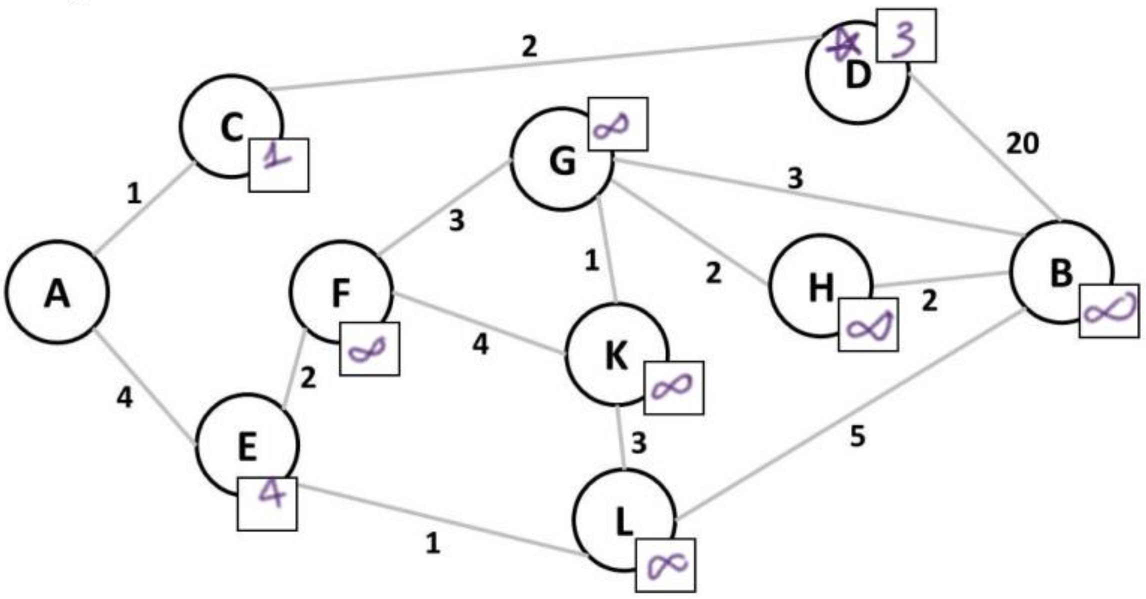


+0.5 for all distances correct

+0.5 for star on node C

-0.5 marks if infinity distances blank

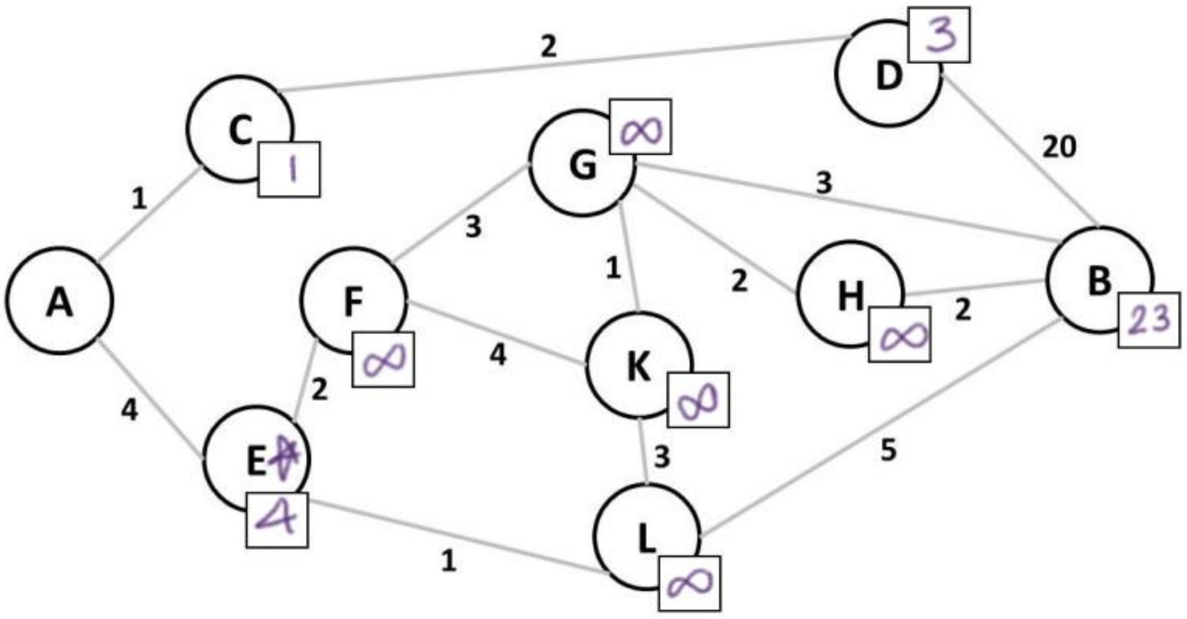
Question 6c



+0.5 for setting node D to 3

+0.5 for star on node D

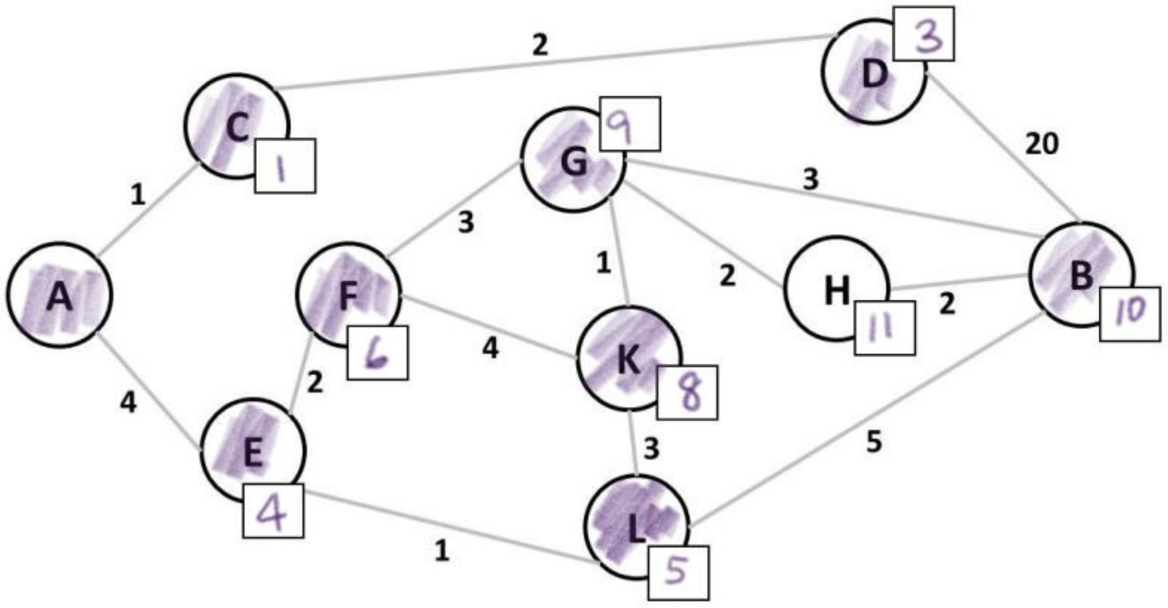
Question 6d



+0.5 for setting node B to 23

+0.5 for star on node E

Question 6e



-0.5 per incorrect node out of the set {B, F, G, H, K, L}