Q1

i)

Time complexity: Slightly improved search time for unsuccessful searches only when searching for a key that existed and was deleted. Instead of continuing probing past a deleted slot marked as AVAILABLE, you stop as soon as you see -key, since key assumed to be positive.
Best improvement: If -k is encountered, we immediately know that k existed and was removed (this saves time in negative lookups).

Worst time: Same as with AVAILABLE, you may still have to probe a long chain of occupied or previously deleted slots.

Space: No extra space is required beyond the original table; negative values are encoded in-place.

Possible issues/misbehaviours: Assume key 16 is deleted, so mark it as -16. If 16 is reinserted, will it overwrite -16? If not, the search for 16 may stop at -16, failing to find the reinserted value

ii)

Time complexity: Potentially reduces the average search time, since displaced keys are moved closer to their ideal hashed positions. Improves successful search times for those reinserted elements.

Insertion/deletion time increases. After removing an element, you now need to look forward through the cluster, find entries whose original hashed position is the deleted slot (or earlier), and shift them back. Can possibly cascade and become expensive as you may need to do this for the whole table. (Basically shift by putting element that should have been in hashed position (but was probed) into hashed position so that searching for this moved element would just have the element be found in the hashed position since it was shifted (shifting this way would probably also require shifting of all elements that required probing due to this element taking up a position before it was shifted).)

Space: No extra space used; entries are relocated in-place.

Possible issues/misbehaviours: Hash chain may be broken and entries may be inaccessible. Example, you delete item at index i. You find item at index i+1 that hashes to i+2, but you move it to i anyway. Now it’s in the wrong place, and may not be found during a search that expects it further along.

Q2.
a)

39	29		38		18	45		29	35	88		25

b)

Longest cluster: 3

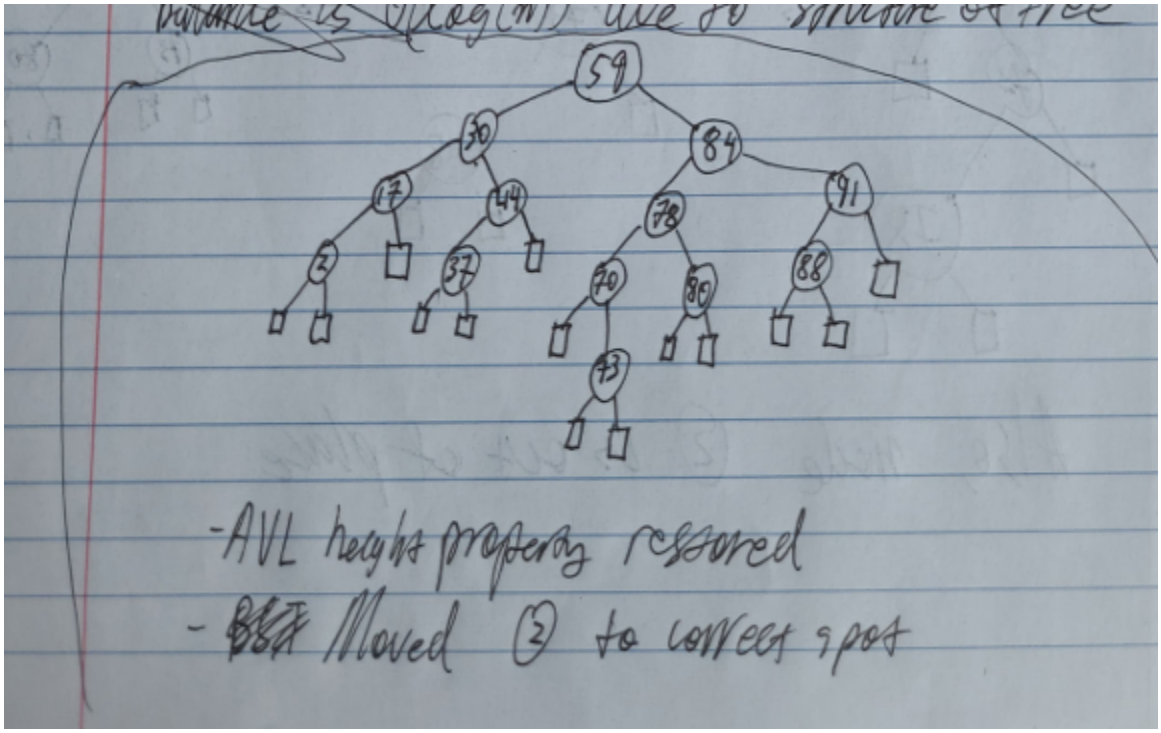
c)
Collisions: 4

d)
Load factor: 9/13

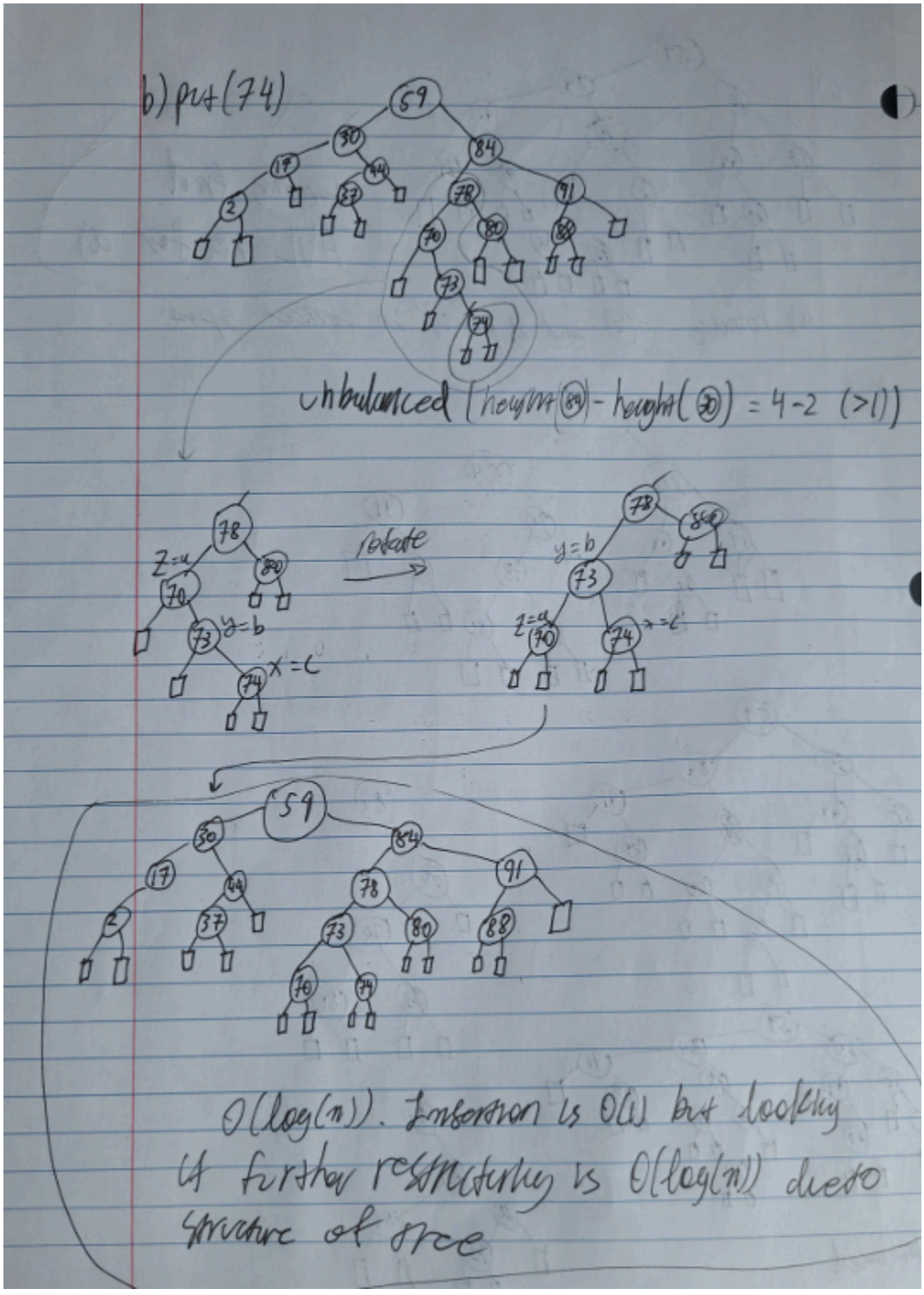
Q3.

a)

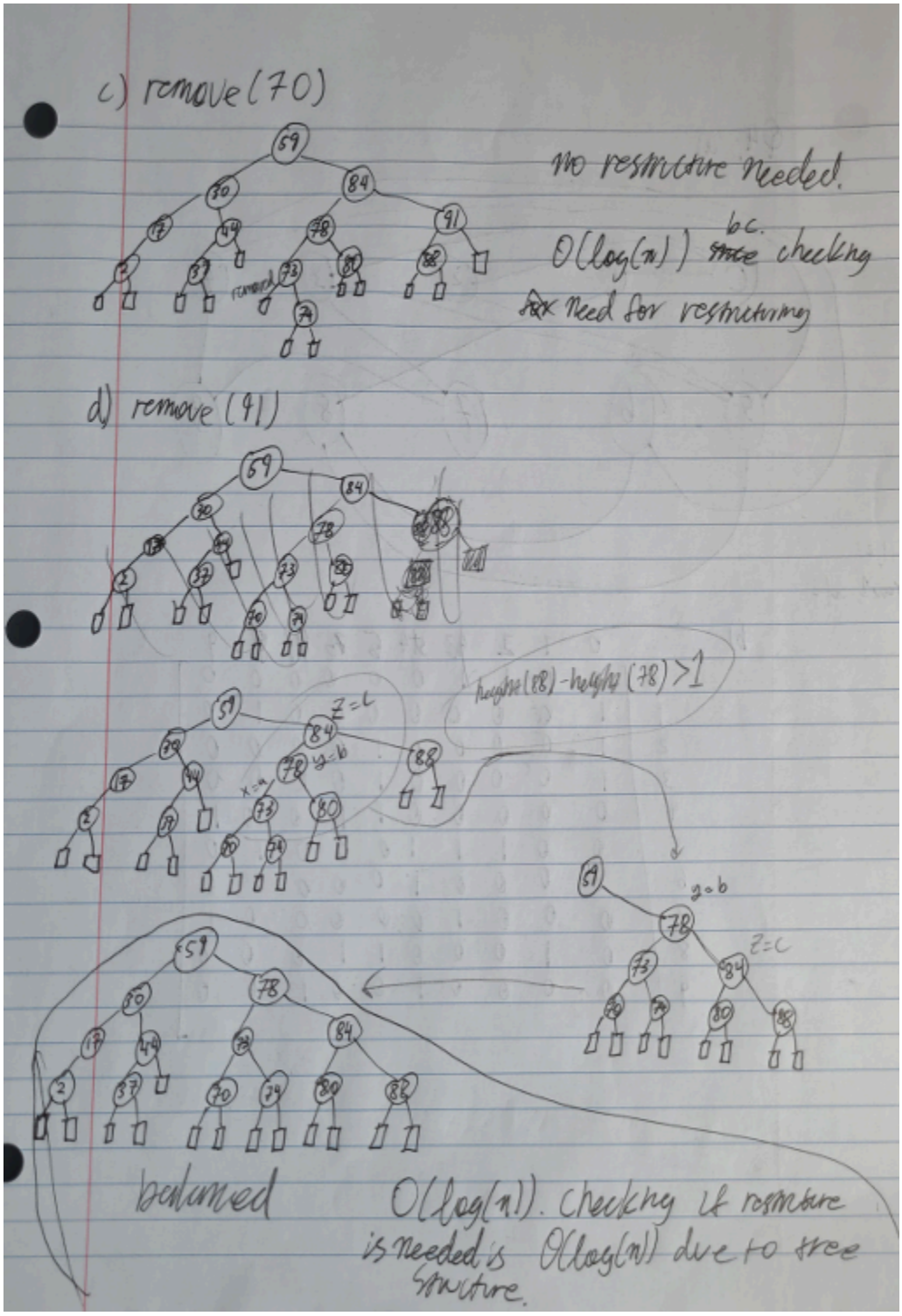
Not an AVL tree. Node “70” has a child “2” with height of 1 and a child “80” with height of 3. height(80) - height(2) = 2 and 2 > 1, so this violates the AVL height balance property. Also, node “2” is out of place and does not satisfy the BST property of all elements of right subtree being greater than root. This violates AVL properties because an AVL tree is a BST.



b)

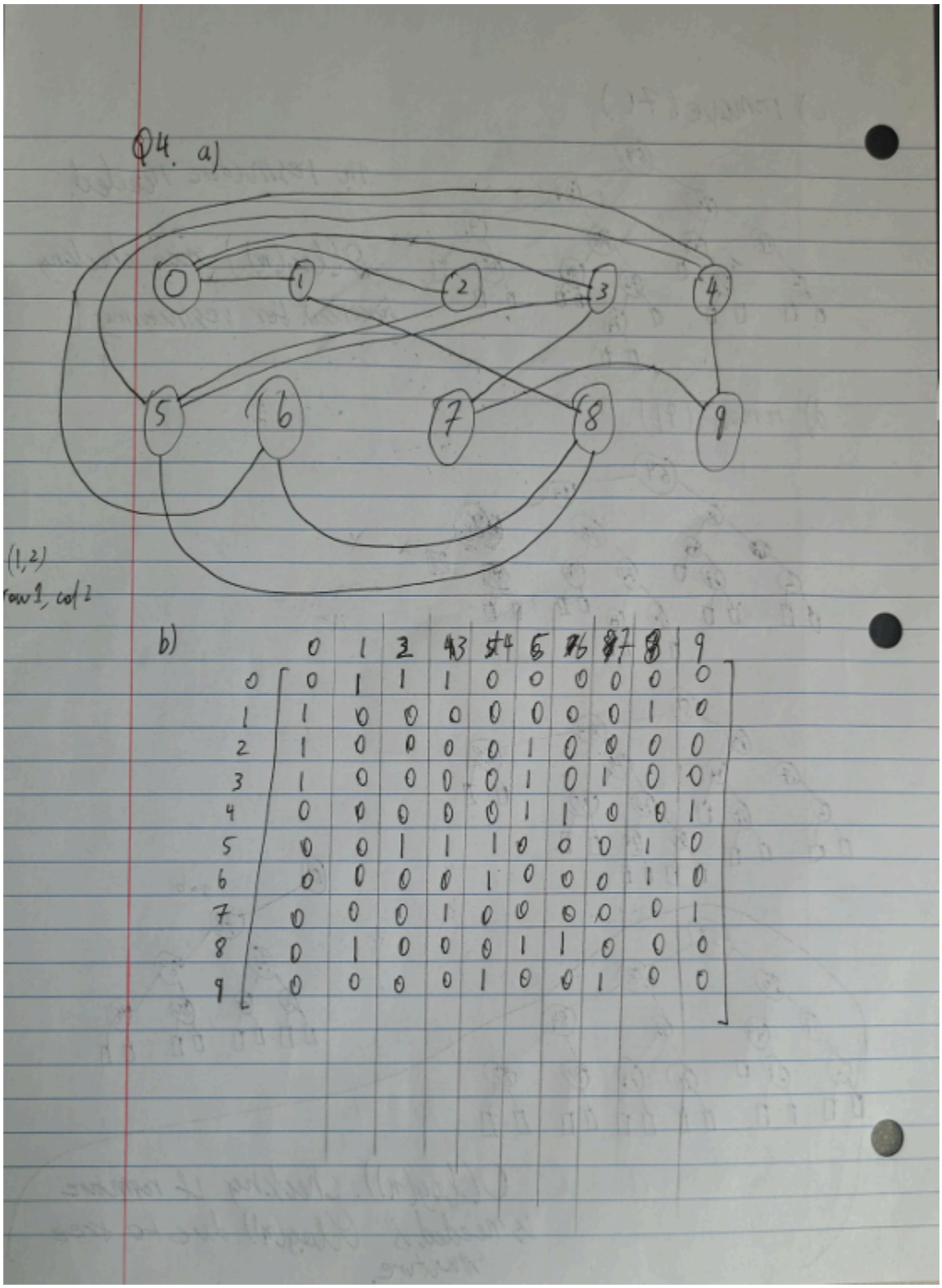

b) put(74)
59
30
84
17
44
78
91
2
37
73
80
88
70
74
unbalanced (height(84) - height(30) = 4-2 (>1))
78
80
Z=a
70
73
y=b
x=c
74
rotate
78
84
y=b
73
Z=a
70
74
x=c
59
30
84
17
44
78
91
2
37
73
80
88
70
74
O(log(n)). Insertion is O(1) but looking
& further restructuring is O(log(n)) due to
structure of tree

c) + d)


c) remove(70)
no restructure needed.
O(log(n)) bc. checking
Need for restructuring
d) remove(91)
height(88) - height(78) > 1
z=c
y=b
x=a
y=b
z=c
balanced
O(log(n)). checking if restructure is needed is O(log(n)) due to tree structure.

Q4.

a) + b)



	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	1	0
2	1	0	0	0	0	1	0	0	0	0
3	1	0	0	0	0	1	0	1	0	0
4	0	0	0	0	0	1	1	0	0	1
5	0	0	1	1	1	0	0	0	1	0
6	0	0	0	0	1	0	0	0	1	0
7	0	0	0	1	0	0	0	0	0	1
8	0	1	0	0	0	1	1	0	0	0
9	0	0	0	0	1	0	0	1	0	0

c)
i)
Visit order: 0, 1, 8, 5, 2, 3, 7, 9, 4, 6

ii)
Edge label order: (0,1)D, (1,8)D, (8,5)D, (5,2)D, (2,0)B, (5,3)D, (3,0)B, (3,7)D, (7,9)D, (9,4)D, (4,5)B, (4,6)D, (6,8)B

d)
i)
No.

ii)
Recursive algorithm also makes use of stack – call stack for recursive calls. The iterative method mimics this order by manually pushing/popping from explicit stack. Both algorithms follow the same order given by G.incidentEdges(v), and both use stacks that only push/recur on unexplored nodes.

Q5.

N → W:	N,W (3)
N → D:	N, D (1)
N → T:	N, D, M, T (7)
N → A:	N, W, A (5)
N → M:	N, D, M (5)
N → E:	N, D, M, E (8)
N → K:	N, D, M, K (14)