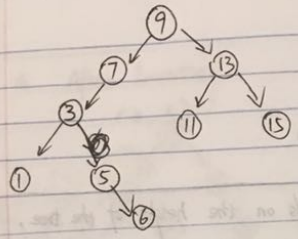
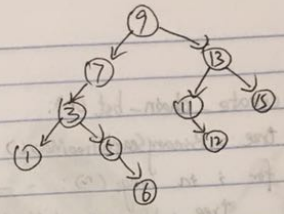
Q1

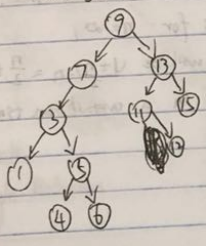
bst[6] = None.



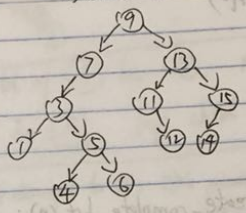
bst[12] = None



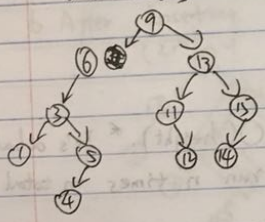
bst[4] = None



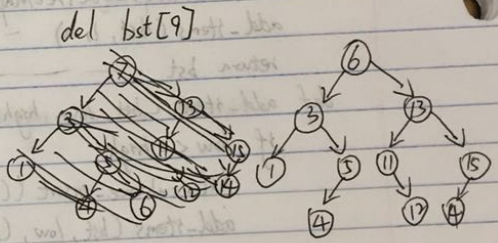
bst[14] = None



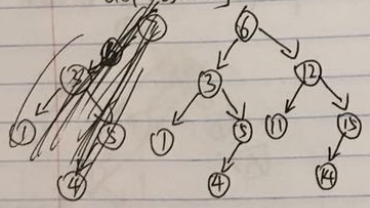
del bst[7]



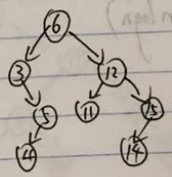
del bst[9]



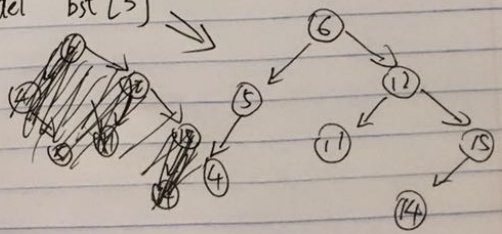
del bst[13]



del bst[1]



del bst[3]



Q2 C

Q2 C .

(a) def create_chain_bst(n):
    tree = BinarySearchTreeMap() ---- O(1)
    for i in range(n): ------
        tree.subtree_insert(i+1) ----
    return tree.

O(n²)

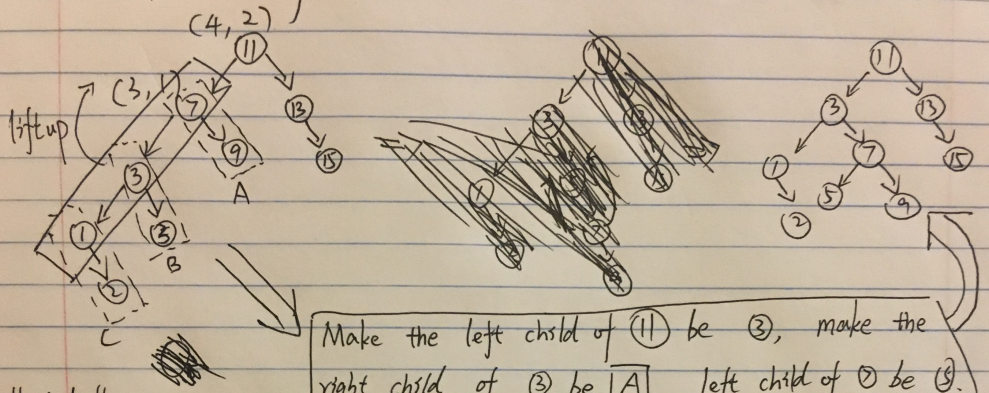
depends on the height of the tree,
in this case, the insert method will always go through the whole chain tree,
which means for $n = \infty$,
the run time will be $\frac{(1+n)}{2} \cdot n = \frac{n}{2} + \frac{n^2}{2}$,
which result in a overall run time $O(n^2)$

(b)
def create_complete_bst(n):
    bst = BinarySearchTreeMap() ---- O(1)
    add_items(bst, 1, n) ---- O(n log n)
    return bst ---------- O(1)
def add_items(bst, low, high):
    if low <= high:
        bst.subtree_insert((low+high)//2) --- O(height). * It's a balanced tree.
        add_items(bst, low, (low+high)//2 - 1)
        add_items(bst, (low+high)//2 + 1, high)
    } run n times in total

O(n log n)

Q6

Q6 a After Inserting

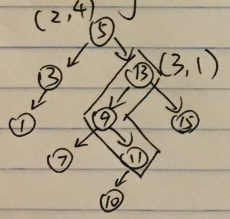


Make the left child of (11) be (3), make the right child of (3) be [A], left child of (7) be (5).

(Also the parent of (3) changed to (11), (7)'s parent was changed to (3), and parent of (5) changed to (7).)

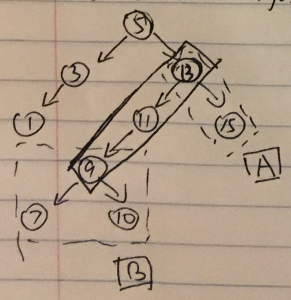
Left → Left, Single Rotation.

b After Inserting



Left → Right: Double Rotation.

① Make (13) → (9) → (11) to a left, left structure



② Lift up (11), repeat the process of part a.