1. Consider the function `my_function(L)` where `L` is assumed to be a Python list:

```
1 def my_function(L):
2     for k in range(1, len(L)):
3         current = L[k]
4         j = k
5         while j>0 and L[j-1]>current:
6             L[j] = L[j-1]
7             j -= 1
8         L[j] = current
```

(a) What is this function doing?

(b) What is its worst case asymptotic running time as a function of `n = len(L)`?

(c) What is its best case asymptotic running time? Describe a list `L` for which the best case running time is achieved.

(5+5+5)

(a) Insertion-sort, sorting L in ascending order

(b) $O(n^2)$, realized, e.g., when L is reverse-sorted.
In that case, the inner loop will always run from k down to 1.

(c) $O(n)$, realized when L is already sorted in ascending order.
In that case, the inner loop will be skipped on first check,
so running time is proportional to outer loop of length $n-1$.

2. Sketch, using Python or Python-like pseudocode, the implementation of a function that checks whether a graph $G = (V, E)$ has a cycle. (10)

Note: In the following, we remove vertices from V as they are discovered.

```
def has_cycle(v, p):
    # returns true if a cycle is found when visiting v via p
    for u in neighbors(v):
        if u is not p:
            if u is in V:
                remove u from V      # mark as visited
                if has_cycle(u, v):
                    return True
            else:
                return True          # we have already been at u!
    return False

def graph_has_cycle:
    while V is not empty:
        Take arbitrary vertex u from V
        if has_cycle(v, None):
            return True
    return False
```

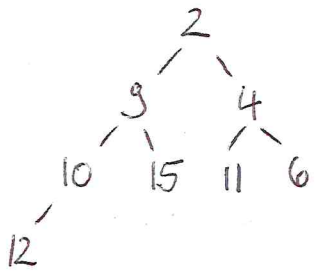
3. Consider the array [2, 9, 4, 10, 15, 11, 6, 12] representing a binary tree in the standard array-representation.

   (a) Draw the corresponding binary tree. Is it a min-heap? If not, restore the min-heap property in a bottom-up pass through the tree.

   (b) Insert the element 1 into the min-heap from part (a) and show all "bubble-up" operations required to restore heap order.

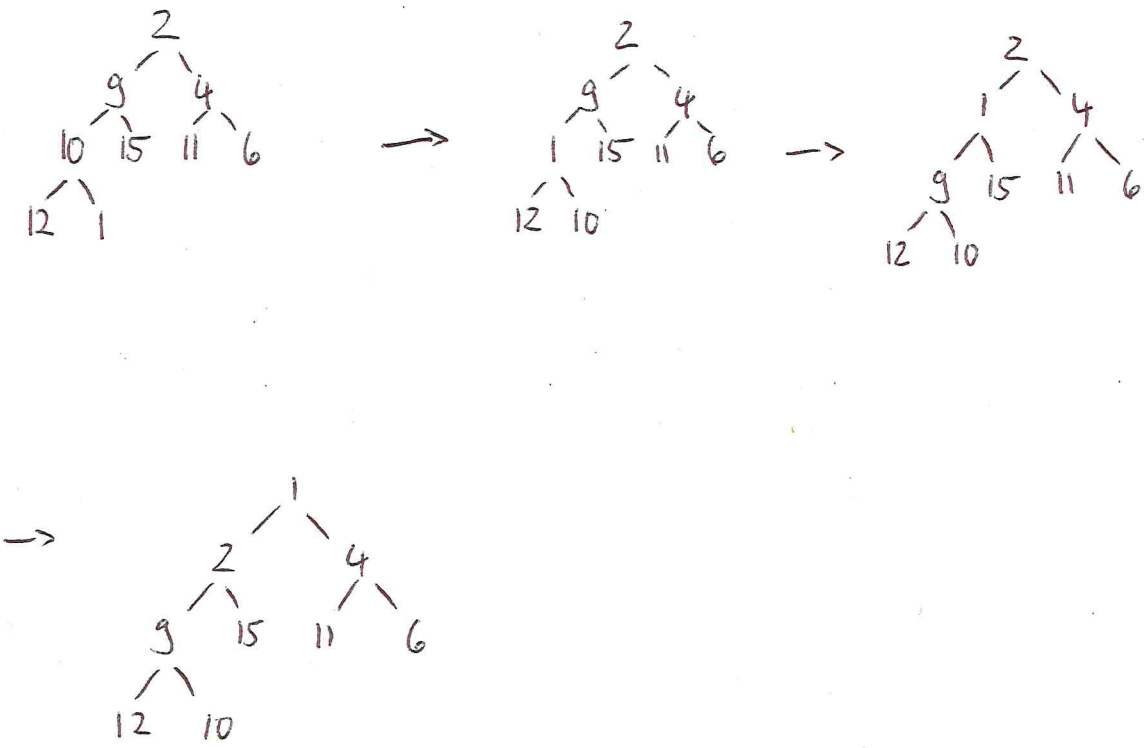
(5+5)

(a)

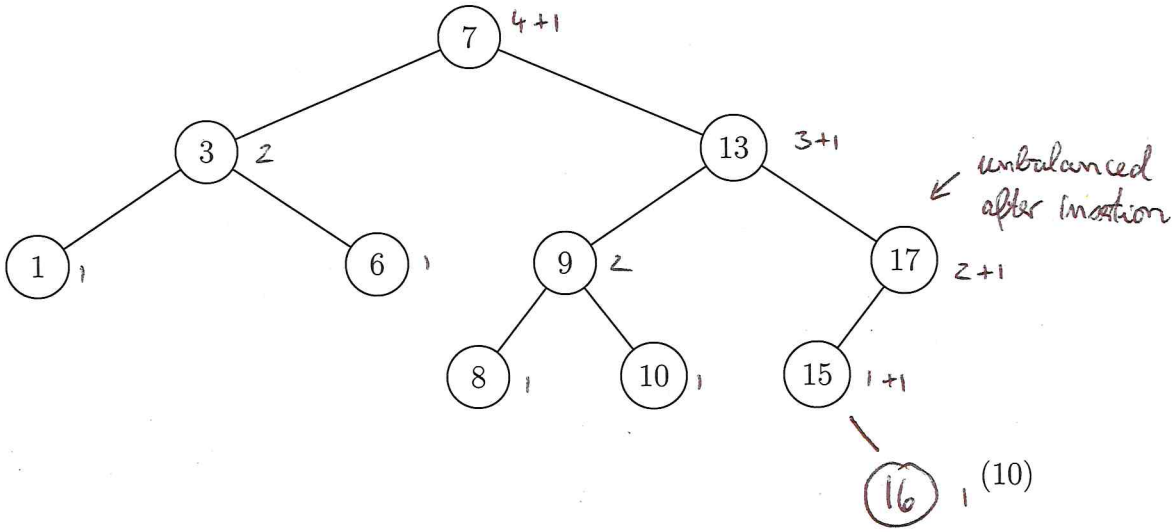


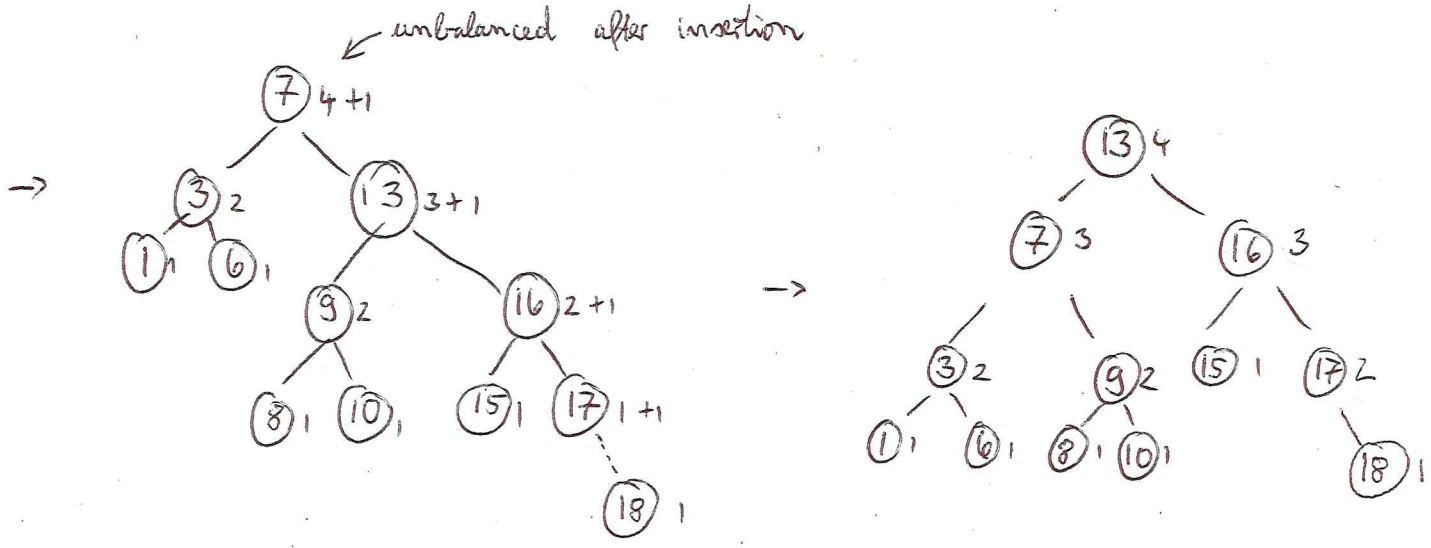
This is a min-heap

(b)



4. Insert the element 16, then the element 18 into the following AVL tree. Indicate all subtree heights before each insertion, and show the changes introduced by the insertions and the subsequent rebalancing step-by-step. (You may consult the summary of tree rotations on the front page.)





5. We noted in class that we can use the splay tree data structure to sort an array L of length n as follows: Take an initially empty splay tree S and move the elements of L one-by-one into S. Then traverse S in-order, moving the elements back into L. This algorithm is known as "splay-sort".

   (a) Prove that splay-sort takes $O(n \log n)$ time.
   Note: You can use, without proof, the result from class that m splay tree operations starting from an initially empty splay tree take $O(m \log n)$ time, where n is the maximal size of the tree.

   (b) Prove that splay-sort takes $O(n)$ time when L is already sorted.

(5+5)

(a) Inserting the n elements from L into S takes $O(n \log n)$ time (Result from class), as clearly the maximum size of S is n.

Traversing S takes $O(n)$ time.

=> Total cost is $O(n \log n)$

(b) Inserting the current largest element into S will always attach it as the right child of the root before splaying. The splay operation then moves it to root, so that all the existing smaller elements go into the left subtree, and the right subtree remains empty.

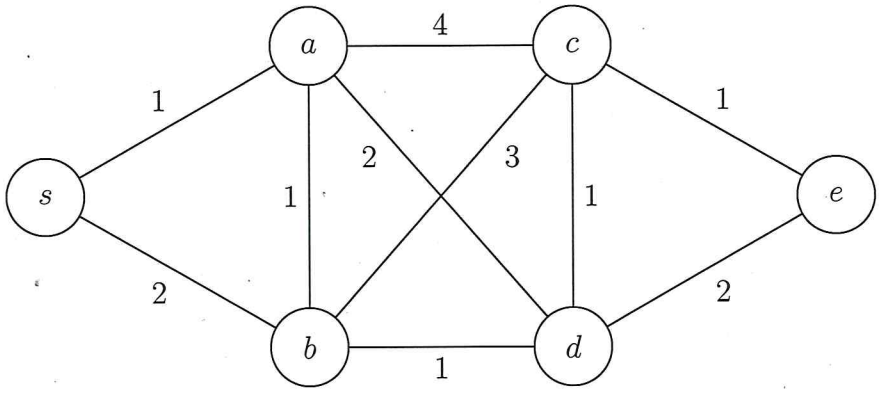
Every such operation therefore only takes $O(1)$ time, inserting all n elements thus takes $O(n)$ time.

Traversing the tree takes $O(n)$ time.

=> Total cost is $O(n)$.

6. Let $G = (V, E)$ be a connected weighted graph with non-negative weights. A *shortest-path tree* of G is a tree that contains all the vertices of G such that the path from its root s to every vertex $v \in V$ is a shortest path in G.

(a) Draw a shortest-path tree for the graph

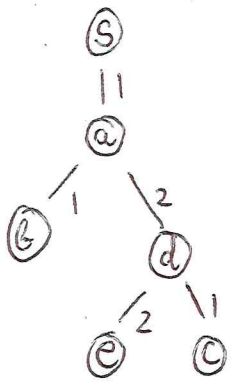


(b) Show that a shortest-path tree always exists under the conditions stated.

(c) Give a small example that shows that a shortest-path tree is not necessarily a minimum spanning tree.

(5+5+5)

(a)



(there are many other correct solutions!)

(b) We construct a shortest-path tree as follows:

Start with root s as the only vertex in the tree T.

Iterate for as long as there is a vertex $v \in V$ that is not in T:

Take a shortest path from v to s, $P = (v \equiv v_0, v_1, v_2, \ldots, v_n \equiv s)$ (*)

Let v_i be the first vertex on P that is already in T.

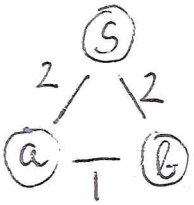
As T is already a shortest-path subtree, the path in T from v_i to s is a shortest path in G, thus the concatenation

$(V_j, \ldots, V_i)$ with the path from V_i to s in T is a shortest path from V_j to s in G, for $j = 0, \ldots, i$.

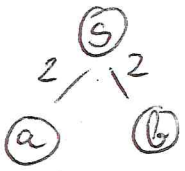
Thus, adding the path $(V_0 = v, \ldots, V_i)$ to T, we obtain an enlarged tree T that is a shortest-path subtree.

Eventually, T contains all vertices of G, so it is a shortest-path tree of G.

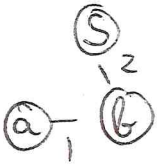
(c) G:



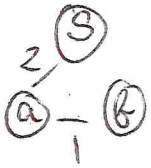
has shortest-path tree



which is not an MST. Minimal spanning trees are:



or



---

* Note: The condition that weights are non-negative guarantees the existence of shortest paths. Otherwise, it might be the case that a cycle of negative cost exists, so that paths with arbitrarily large negative cost exist.