1. (a) Order the following functions by their asymptotic growth rate:

$$n, \exp n, \log n, \exp(n^2), \log(\log n), (\log n)^2$$

(4) (5) (2) (6) (1) (3)

(b) An algorithm takes as input an n-element sequence of integers. It iterates through all of its elements, executing an $O(\log n)$-time computation if the value of the current element is even, resp. an $O(n)$-time computation if the value of the current element is odd. What are the best-case and worst-case running times of the algorithm?

(c) In the following, `T` is an instance of a standard binary tree class. It contains n nodes. What does the following code do? What is its running time as a function of n? Give a Big-Oh upper bound and a Big-Omega lower bound.

```
1 def do_something(T, p):
2     if p is not None:
3         do_something(T, T.left(p))
4         print(T.element(p))
5         do_someting(T, T.right(p))
6
7 do_something(T, T.root())
```

(5+5+5)

(b) Best case: all elements are even: $n \cdot O(\log n) = O(n \log n)$

Worst case: all elements are odd: $n \cdot O(n) = O(n^2)$

(c) do_something is an in-order traversal of T. As such, it performs $O(1)$ - work on every node of the tree, so completes in $O(n)$ and $\Omega(n)$ time.

2. Are the following statements true or false? Explain your answer in 1–2 sentences.

   (a) A singly linked list can be used to implement a stack such that "push" and "pop" execute in $O(1)$-time.

   (b) A singly linked list can be used to implement a queue such that "enqueue" and "dequeue" execute in $O(1)$-time.

   (c) An AVL tree can be used to sort a list $O(n \log n)$-time.

   (d) A heap can be used to sort a list in $O(n \log n)$-time.

   (e) A skip list can be used to sort a list in $O(n \log n)$-time.

(2+2+2+2+2)

(a) True. "push" needs to do insertion at the front of the list, which is $O(1)$
"pop" likewise removes from the front of the list.

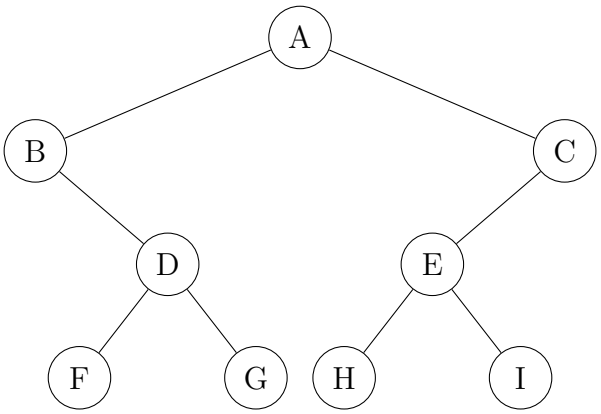
(b) True, provided an external reference to the tail of the list is kept. Then "enqueue" operates at the tail of the list, "dequeue" operates at the head.

(c) True. Do n insertions at $O(\log n)$ cost into an empty tree : $O(n \log n)$ time
Then do an in-order traversal in $O(n)$ time.

(d) True. This is standard heap-sort.

(e) Almost true: Skip lists only guarantee expected performance, individual instances can be worse. However, on average, skip lists perform similarly to AVL trees.

3. Give the pre-order, in-order, and post-order traversals of the following binary tree:



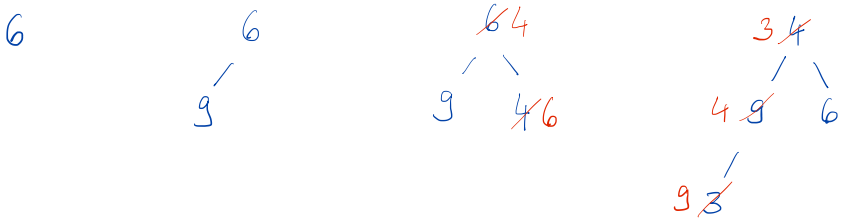
(5)

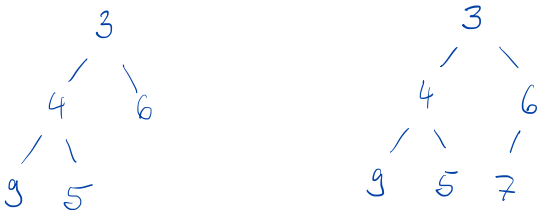
pre-order: A - B - D - F - G - C - E - H - I

in-order: B - F - D - G - A - H - E - I - C

post-order: F - G - D - B - H - I - E - C - A

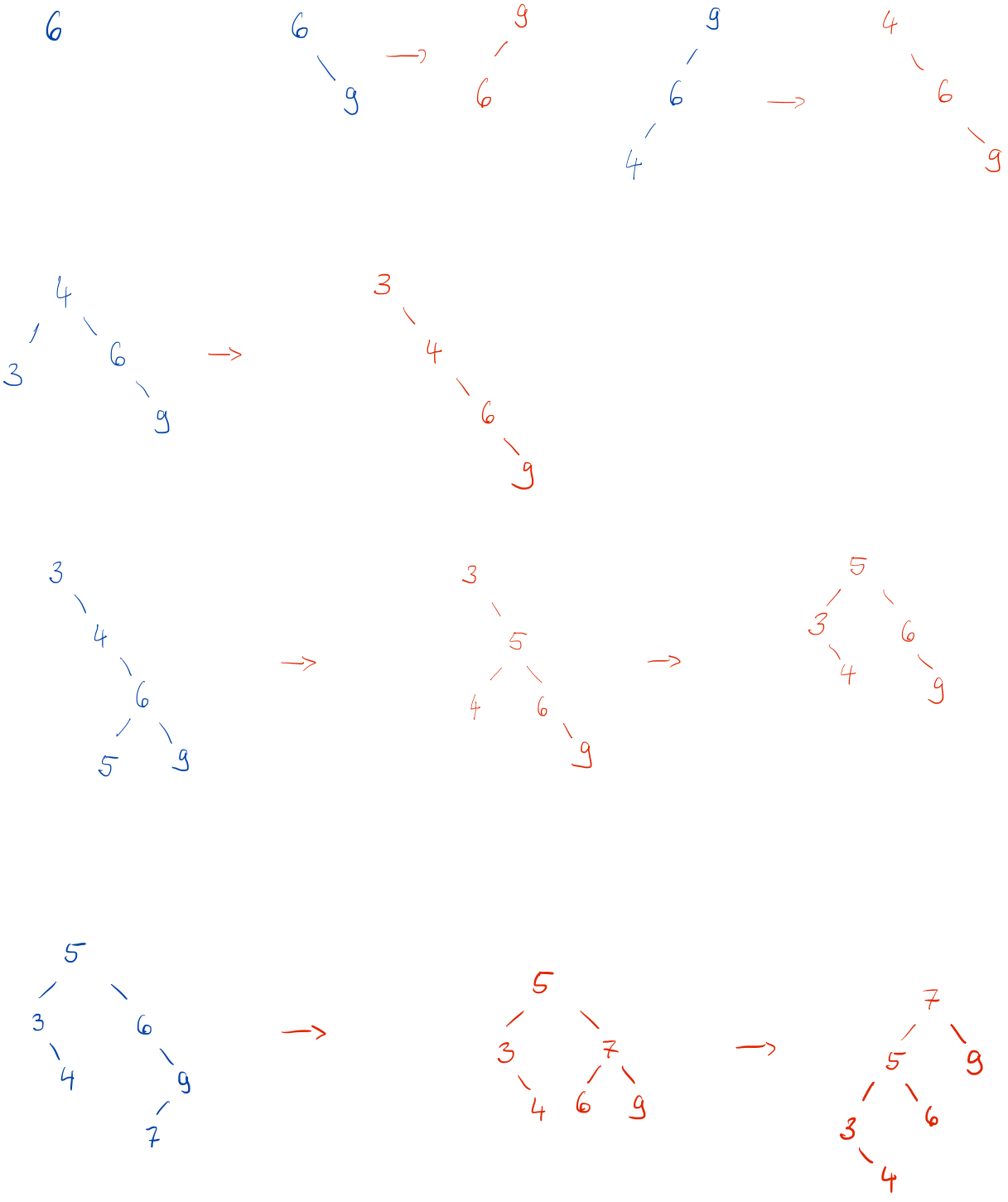
4. Insert the keys 6, 9, 4, 3, 5, 7 into an initially empty heap. Show the heap at each step of insertion. (5)





5. Insert the keys 6, 9, 4, 3, 5, 7 into an initially empty splay tree. Show the splay tree at each step of insertion.

   (You may consult the attached splay tree "cheat sheet".) (5)



6. Describe an algorithm that takes an undirected, connected graph $G = (V, E)$ as input and returns `True` if the graph is a tree and `False` otherwise. (5)

```
V = {}

def is_tree (e):
    if e[1] is in V:          # edge e leads to a vertex we have already visited
        return False
    else:
        V[e[1]] = e[0]        # mark e[1] as visited, e.g. via e[0]
        for p in neighbors(e[1]):
            if p is not e[0] and not (is_tree((e[1], p))):
                return False
        return True
```

Now take any vertex, r say, and call the above function: is_tree(r)

7. Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

   (a) Initialize path to *start*.

   (b) Initialize set *visited* to {*start*}.

   (c) If *start*=*goal*, return path and exit. Otherwise, continue.

   (d) Find the edge (*start*, v) of minimum weight such that v is adjacent to *start* and v is not in *visited*.

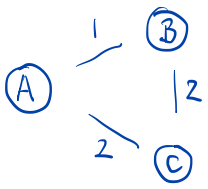
   (e) Add v to path.

   (f) Add v to *visited*.

   (g) Set *start* equal to v and go to step (c).

   Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counterexample. (5)

No, e.g.:



Apply the above with start = A, goal = C

(a) path = (A)

(b) visited = {A}

(c) continue

(d) v = B

(e) path = (A, B)

(f) visited = {A, B}

(g) start = B

(c) continue

(d) v = C

(e) path = (A, B, C)

(f) visited = {A, B, C}

(g) start = C

(c) exit

So we get the non-optimal path A-B-C.