1. (a) Simplify the following Big-Oh resp. Big-Omega expressions as much as possible:

   (i) $O(\log n) + O(n \log n) + O((\log n)^4)$ $= O(n \log n)$

   (ii) $\Omega(\log n) + \Omega(\log \log n) + \Omega(\log \log \log n)$ $= \Omega(\log n)$

   (iii) $O(1 + 2n + 3n^2 + 4n^3)$ $= O(n^3)$

   (iv) $O\left(\sum_{i=1}^{n} i^2\right)$ $= O(n^3)$

   (v) $O(n) + \Omega(n)$ $= \Omega(n)$

   (b) What is the running time of the following code as a function of $n$? Give a Big-Oh upper bound and a Big-Omega lower bound.

```
1 def idle(n):
2     r = 0
3     i = n
4     while i>0:
5         r += i
6         i = i//2
7     return r
```

(10+5)

(b) Suppose $2^k \leq n < 2^{k+1}$.

Repeated integer division by 2 can be done exactly $k$ times before the result is zero, where

$$k \leq \log n < k+1$$

So the loop body is executed $\lfloor \log n \rfloor + 1$ times, i.e. $O(n)$ and $\Omega(n)$.

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

   (a) Python lists (i.e., dynamic arrays) can be used to efficiently implement a stack, using `L.append(e)` and `L.pop()` for push and pop, respectively.

   (b) Python lists (i.e., dynamic arrays) can be used to efficiently implement a queue, using `L.append(e)` and `L.pop(0)` for enqueue and dequeue, respectively.

   (c) A heap can be searched more efficiently than an unsorted array.

   (d) A heap can be searched more efficiently than a sorted array.

   (e) Breadth-first traversal of a binary tree takes worst-case $O(1)$ time for each node.

(2+2+2+2+2)

(a) True, because append/pop take $O(1)$ amortized time, which is asymptotically optimal. But not that there is no gurantee that individual operations finish in $O(1)$-time.

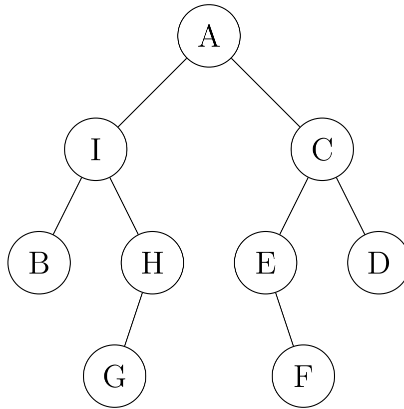
(b) False, as pop(0) requires copying elements in range $[1, n)$ to $[0, n-1)$, which takes $\Omega(n)$ time. Not asymptotically optimal.

(c) Not true in general, because for most elements, large parts of the heap need to be searched. So the asymptotic upper bound is $O(n)$ in both cases. However, when searching for an element that is NOT in the list, all elements of an unsorted array need to be looked at, so it's $\Omega(n)$. For a heap, the best case is $O(1)$, namely when the search key is smaller than the minimum key on the heap.

(d) Definitely false. Binary search on a sorted array is $O(\log_2 n)$, whereas worst-case searching in a heap is $O(n)$.

(e) True, if the queue data structure 3 which needs to be used is optimal (e.g. linked list with $O(1)$ enqueue and dequeue). There are at most one dequeue and two enqueues per node visited.

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



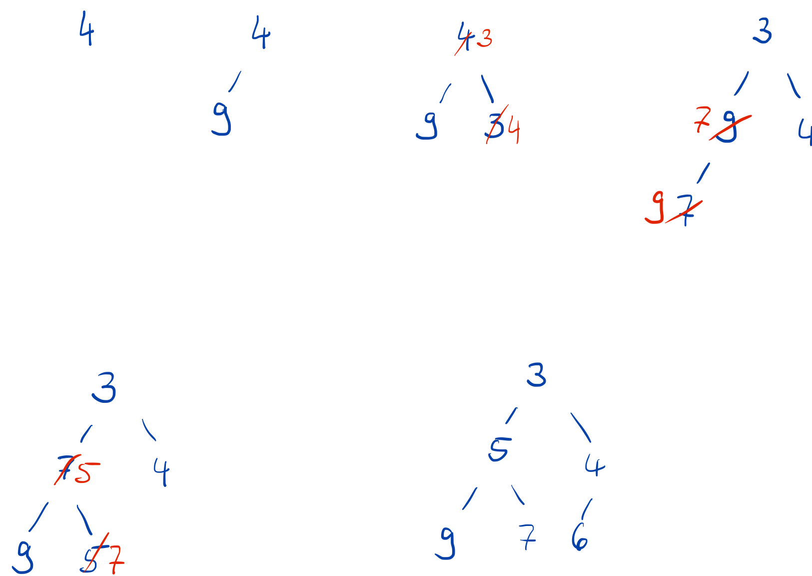

(5)

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

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

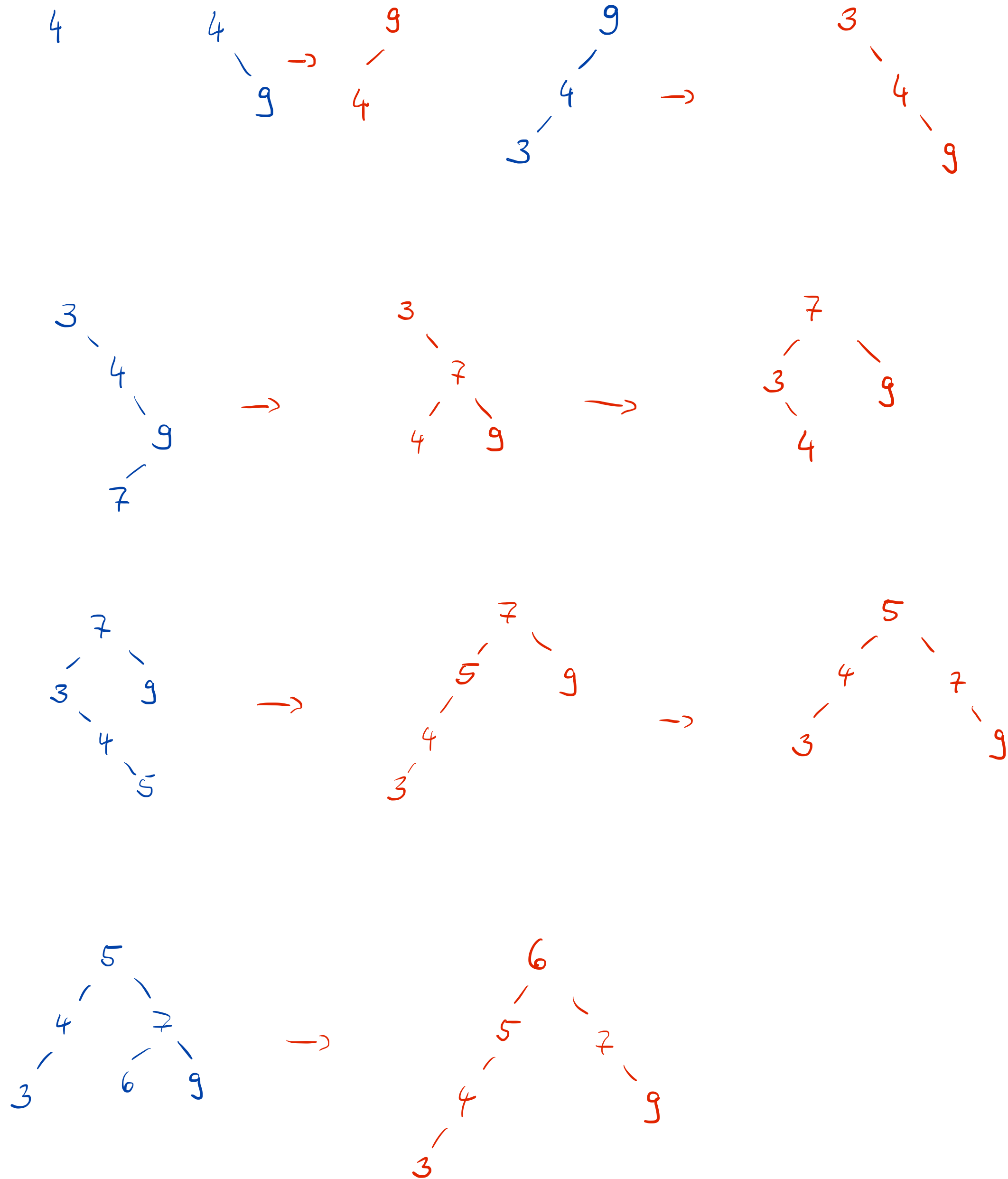
post-order : B - G - H - I - F - E - D - C - A

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



5. Insert the keys 4, 9, 3, 7, 5, 6 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. A function takes as input a list of integers `L` of length $n$ and an integer value `total`. It returns a tuple of elements from the list with sum `total`, if possible, and `None` otherwise. Describe an algorithm that can do this in $O(n \log n)$ time. (5)

Step 1: Sort the list, e.g. using heap sort or merge sort, at O(n log n) cost.

Step 2: Initialize two pointers, one to first element, one to last element of list.

Step 3: If L[first] + L[last] == total, return the tuple (L[first], L[last]).

If first + 1 == last, we have exhausted the list, return None.

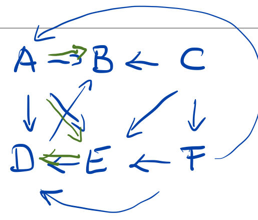
If L[first] + L[last] < total, increment first and repeat Step 3

If L[first] + L[last] > total, decrement last and repeat Step 3.

Step 3 effectively iterates over the list, so it's O(n).

7. In the following, $G$ is a graph in a "map of maps" representation. An example is the following:

```
1 G = {'A' : {'B', 'E', 'D'},
2      'B' : {},
3      'C' : {'B', 'E', 'F'},
4      'D' : {'B'},
5      'E' : {'D'},
6      'F' : {'A', 'D', 'E'}}
```




For (G): DFS-tree is marked in green (not unique as dictionary semantics does not fix order of iterator!)

(a) Draw a representation of this graph in the plane.

(b) What does the following function do if it is executed on a graph G above?

```
1 def mystery_function(p):
2     for n in G[p]:
3         if n not in v:
4             v[n] = p
5             mystery_function(n)
6
7 v = {}
8 v['A'] = None
9 mystery_function('A')
```

It generates a DFS traversal of the graph, storing the last node on the discovery path for every node in v. (Effectively, this stores the discovery edge!)

(c) What data structure is encoded in v if this code is run on a general graph G?

(d) If n=len(G), what is the running time of this algorithm? Give a Big-Oh upper bound and a Big-Omega lower bound. Justify your answer using your knowledge about the running time of the native Python dictionary operations.

(e) Sketch, using Python or Python-like pseudocode, a "breadth-first search" (BFS) traversal of a graph of this type.

(2+3+2+3+5)

(c) v represents a tree, where the parent of every node is explicitly stored. (So node A is the root, as its parent is None.)

This tree is called a spanning tree of the nodes reachable from A.

(d) Every node and every edge is visited exactly once. So if m denotes the total number of edges, cost is $O(n+m)$.

(Solution ctd. or scratch.)

(e)

```
v = {}
v['A'] = None
Q = Queue()
Q.enqueue('A')
while Q.not_empty():
    p = Q.dequeue()
    for n in G[p]:
        if n not in v:
            v[n] = p    # mark as visited via node p
            Q.enqueue(n)
```