Write your anonymous code (*not* your name):

Basic question 1: Sorting

Perform in-place insertion sort *and* in-place selection sort on the array [77, 63, 57, 24, 41, 38].

Show the array after each iteration of the outer loop in the algorithm.

Insertion sort

0	1	2	3	4	5
77	63	57	24	41	38

0	1	2	3	4	5
63	77	57	24	41	38

0	1	2	3	4	5
57	63	77	24	41	38

0	1	2	3	4	5
24	57	63	77	41	38

0	1	2	3	4	5
24	41	57	63	77	38

0	1	2	3	4	5
24	38	41	57	63	77

Selection sort

0	1	2	3	4	5
77	63	57	24	41	38

0	1	2	3	4	5
24	63	57	77	41	38

0	1	2	3	4	5
24	38	57	77	41	63

0	1	2	3	4	5
24	38	41	77	57	63

0	1	2	3	4	5
24	38	41	57	77	63

0	1	2	3	4	5
24	38	41	57	63	77

Getting the names mixed up is fine, as long as one is Insertion and the other is Selection.

Basic question 2: Hash tables

You are given the following five strings:

cat bird horse dog salamander

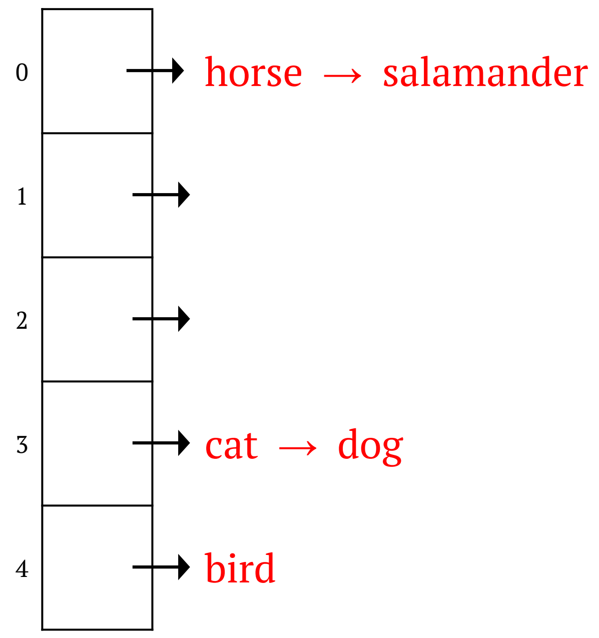
The hash function is $h(s)$ = the number of letters = the length of the string.

Insert the elements *in the given order* into the following two empty hash tables. The left one is an open addressing table (using +1 linear probing, the way it is always done in this course), and the right one is a separate chaining table (using linked lists). Both use normal modular hashing.

Open addressing

0	salamander
1	
2	
3	cat
4	bird
5	horse
6	dog
7	
8	
9	

Separate chaining

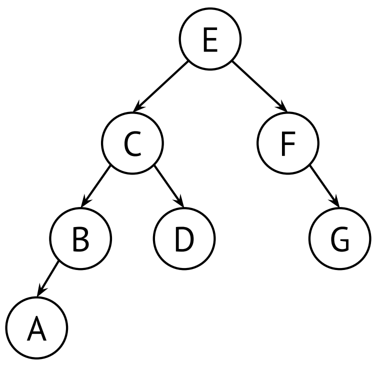


Here we add last to the linked list

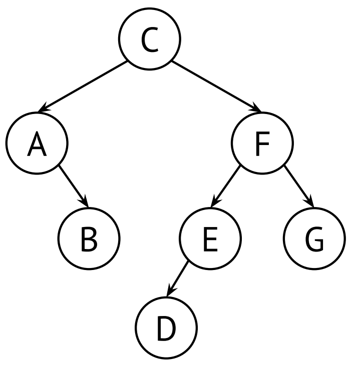
Consistently adding first would also be fine (reversing the first two lists)

Basic question 3: Search trees

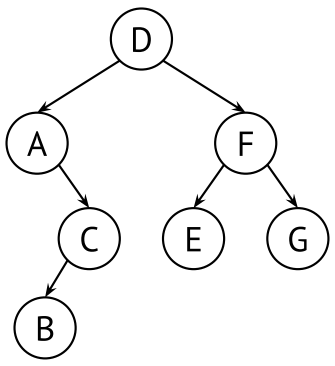
You are given the following three binary search trees:



Left



Middle



Right

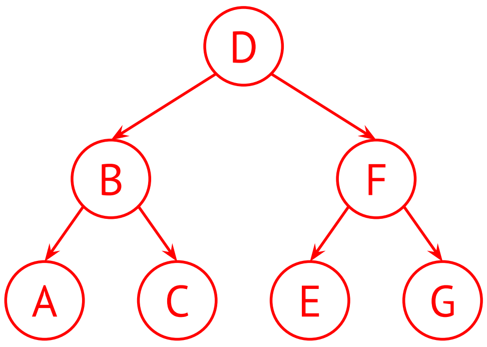
One of the trees is not a correctly balanced AVL tree, which one? Right

Which node is imbalanced? A

Perform AVL rotations to rebalance the tree.
(Recall that you need at most two rotations.)

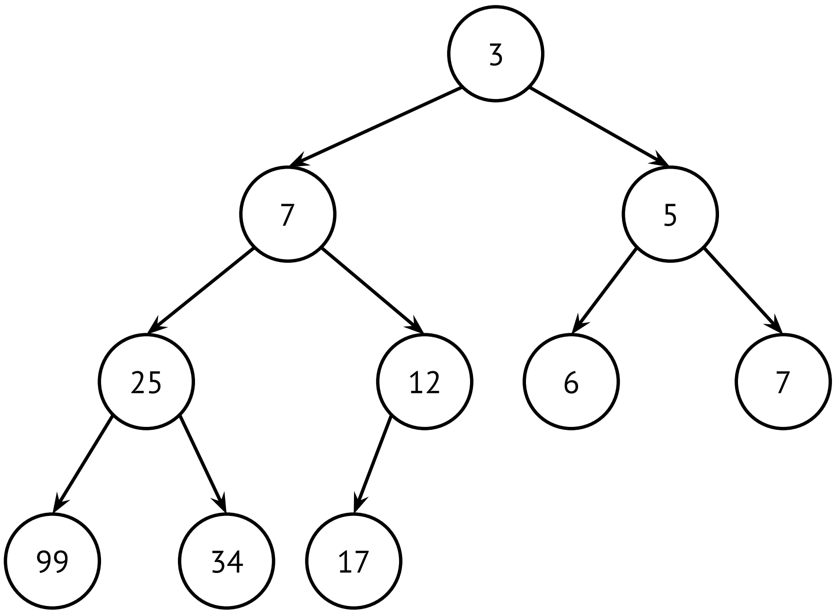
Which rotation(s) did you perform? Right rotate C, then left rotate A

How does the final tree look? Draw it below:



Basic question 4: Priority queues

You are given a minimum priority queue implemented as the following binary heap:



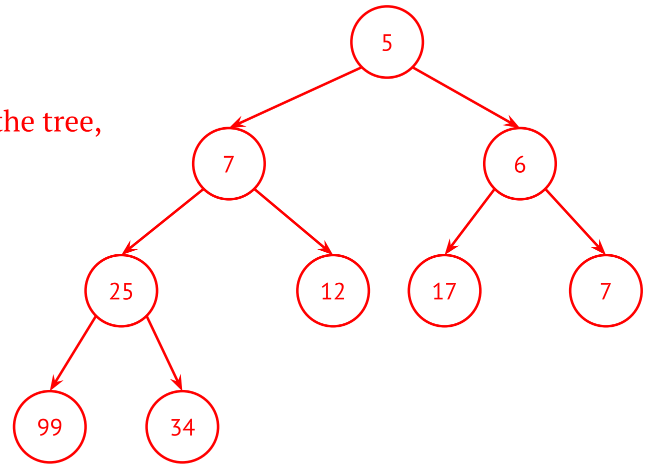
How does the above heap look when represented as an array?

0	1	2	3	4	5	6	7	8	9	10	11
3	7	5	25	12	6	7	99	34	17		

Now remove the minimum element from the priority queue (using the standard heap removal algorithm). How does the heap look afterwards?

0	1	2	3	4	5	6	7	8	9	10	11
5	7	6	25	12	17	7	99	34			

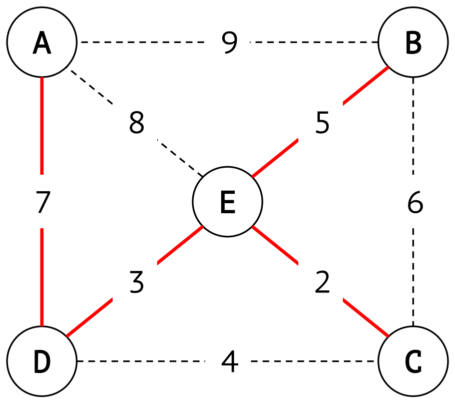
You didn't have to draw the tree, but here it is anyway:



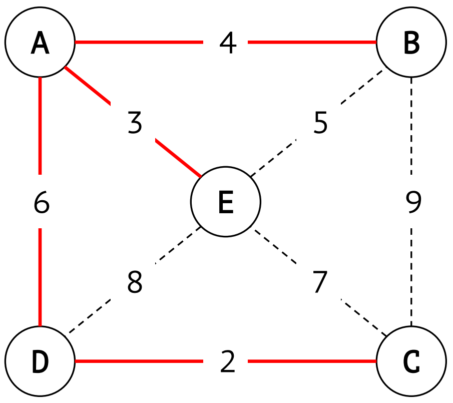
Basic question 5: Graphs

Run Prim's or Kruskal's algorithm on each of the following two graphs.
(If you choose Prim you should start from node A).

Left graph



Right graph



Mark the edges in each minimum spanning tree (MST) directly in the graphs above.

In what order are the edges added to each of the MSTs?
(Write XY for the edge between X and Y.)

Left graph:	*Prim*:	AD DE EC BE
	Kruskal:	EC DE BE AD
Right graph:	*Prim*:	AE AB AD CD
	Kruskal:	CD AE AB AD

Basic question 6: Complexity

Here are implementations of two different sorting algorithms. Both of them are not-in-place, meaning that they don't change the original array, but instead return a new array with the sorted elements. What is their asymptotic complexity in the size of the array, and which one is asymptotically more efficient?

The input array is the variable A, and N is the size of the array. The result is stored in R.

Which of Algorithm 1 or 2 is more efficient (when the input size N is large)? 2

Algorithm 1

Complexity: O(N^2)

Explanation:

```
function sort1(A):                                
   N = size of A                             O(1)
   R = new array of size N                   O(1)

   for i = 0, 1, ..., N-1:                       O(N) ×
      j = i                                          O(1)

      while j > 0 and R[j-1] > A[i]:                 O(N) ×
         R[j] = R[j-1]                                   O(1)
         j = j - 1                                       O(1)

      R[j] = A[i]                                    O(1)

   return R
```

Algorithm 2

Complexity: O(N log N)

Explanation:

```
function sort2(A):
   N = size of A                             O(1)
   H = new empty binary heap                 O(1)

   for i = 0 to N-1:                             O(N) ×
      H.add(A[i])                                    O(log N)

   R = new array of size N                       O(1)

   for j in 0, 1, ..., N-1:                      O(N) ×
      R[j] = H.removeMin()                           O(log N)

   return R
```

Advanced question 7: Complete binary trees

In this question, your task is to write an algorithm for checking if a given binary tree is *complete*.

Assume the following standard definitions. A (binary) *tree* is either null (None in Python) or a (binary) *node* with left and right child trees:

```
class TreeNode:
   left: TreeNode (or null)
   right: TreeNode (or null)
   value: Anything   // the value type is not relevant for this problem
```

A binary tree is *complete* if:

- every level is completely filled with nodes,
- except possibly the last level, which from the left is filled with nodes up to some point and empty afterwards (that is, no null tree occurs to the left of a node in the level).

Design an algorithm for testing if a given tree is complete or not. You may answer using pseudocode or your favourite programming language.

Hint: Recall that a complete binary tree can be efficiently represented using an array (which is how binary heaps are implemented, as you already know).

Answer:

We can first transform the tree into an array, and then check that the array doesn't contain holes:

```
function isComplete(root):
    A = new dynamic array
    toArray(root, 0, A)
    return all(cell is true for each cell in A)

function toArray(node, i, A):
    if node is not null:
        A[i] = true   // Note: this assumes that the dynamic array auto-resizes
        toArray(node.left, 2*i+1, A)
        toArray(node.right, 2*i+2, A)
```

Another solution is to do BFS traversal like here: https://algo.monster/liteproblems/958

```
function isComplete(root):
    Q = new queue
    Q.enqueue(tree.root)

    while Q is not empty:
        node = Q.dequeue()
        if node is null:
            break
        Q.enqueue(node.left); Q.enqueue(node.right)

    return all(node is null for each node in Q)
```

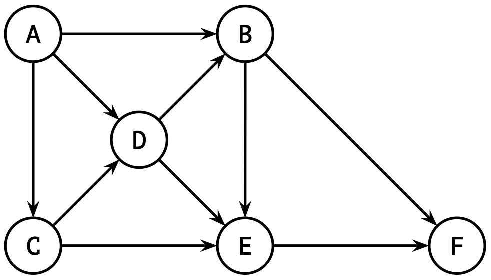
Note: For 1 point you need a good solution idea. For 2 points the implementation must be correct.

Advanced question 8: *k*-length paths in a graph

In the following, your task is to provide an algorithm that returns the number of k-length paths between two given vertices in a DAG (directed acyclic graph).

More formally, you are given an unweighted DAG $G = (V, E)$, two vertices v_{start} and v_{goal} in V, and a natural number k. Your algorithm should return the number of paths of length k from v_{start} to v_{goal}. You may answer using pseudocode or your favourite programming language, and you can make use of any additional data structure that we covered in the course.

Here, the length of a path between two nodes is the number of its edges. For example, in the graph below, there are 3 paths of length 4 from vertex A to vertex F, namely: ADBEF, ACDEF, ACDBF.



You can assume the following data type for graphs and extend it as needed for your solution:

```
class DirectedGraph<Vertex>:
    successors: Map<Vertex, List<Vertex>>
    predecessors: Map<Vertex, List<Vertex>>
    vertices: Set<Vertex>
```

Answer:

```
function Count(G, start, goal, k):
    // This solution searches forward from the start node.
    // M is a map from iteration i to a counter for each vertex.
    // An alternative is to use two M-counters: one for the previous i and one for the current.

    M = new empty Map<Int, Map<Vertex,Int>>
    M[0] = new empty map
    M[0][start] = 1                  // searching backwards: M[0][goal]

    for i in 1, 2, ..., k:
        M[i] = new empty map

        for v in keys of M[i-1]:
            for v' in G.successors[v]:  // searching backwards: predecessors
                M[i][v'] += M[i-1][v]

    return M[k][goal]                // searching backwards: M[k][start]

    // To search backwards from the goal node we just have to change the commented lines.
```

Note: For 1 point you need a correct solution. For 2 points it must be efficient as well.

Advanced question 9: Amortized complexity

Consider the following implementation of a queue:

1. The queue is composed of two stacks: STACK-1 and STACK-2
2. ENQUEUE(x) adds an element x to the queue by applying PUSH(x) on STACK-1
3. DEQUEUE(), removes an element from the queue, as follows:
   a. If STACK-2 is not empty, then simply POP from STACK-2 and return the element
   b. If STACK-2 is empty, then:
      i. while STACK-1 is not empty, POP it and PUSH onto STACK-2
      ii. POP from STACK-2 and return the result

Answer the following questions:

A. What is the worst-case complexity in the size of the queue for the operations ENQUEUE and DEQUEUE? Explain your solution!

   ENQUEUE: O(1)

   DEQUEUE: O(n) where n is the size of the queue

   The worst case is when STACK-2 is empty, then 3b(i) exhausts the whole STACK-1 which can be O(n) in the worst case

B. What is the amortized cost for DEQUEUE?

   Justify your answer! We recommend using the *aggregate method*; however, you are free to use any other known methods for analyzing the amortized complexity (such as the *potential method* or the *accounting method*).

   O(1) amortized cost.

   Explanation: Consider a sequence of n operations. The sequence of operations will involve at most n elements. The cost associated with each element will in the worst-case be at most 3 (popped from STACK-1, pushed to STACK-2, and popped from STACK-2, is the most expensive case of the DEQUEUE operation). Hence, the actual cost of n operations will be bounded from above by $T(n) = 3n$. Hence, using the aggregate method, the amortized cost of each operation can be $O(T(n)/n) = O(3n/n) = O(3) = O(1)$.

Note: For 1 point you need to get the costs for both A and B correct.
For 2 points the explanations must also be correct.