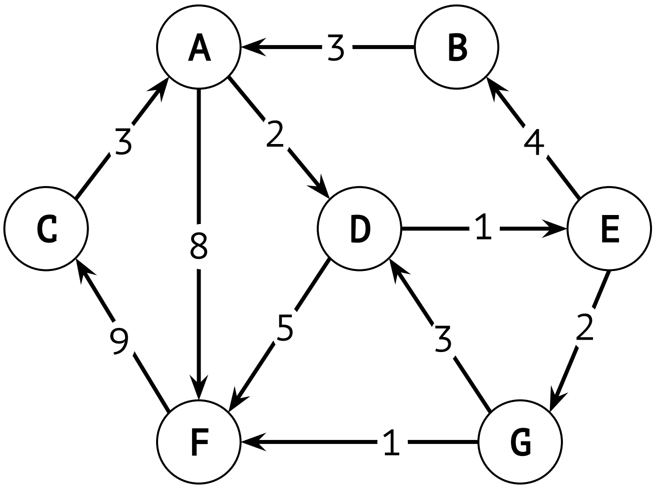
Write your anonymous code (*not* your name):

DAT038/DAT525, solution suggestions

Basic question 1: Graphs



You are given the directed weighted graph to the right.

Perform uniform-cost search (also known as Dijkstra's algorithm). In which order does the algorithm visit the vertices, and what is the computed distance to each of them?

If you start searching from node A?

	first visited					*last visited*
vertex	A	D	E	G	F	B	C
distance from A	0	2	3	5	6	7	15

If you start searching from node G?

	first visited					*last visited*
vertex	G	F	D	E	B	C	A
distance from G	0	1	3	4	8	10	11

Basic question 2: Sorting complexity

The following algorithm should sort an array A of n distinct numbers in descending order, in-place. However, the data structure to use for X remains to be specified:

```
X = new  ???  (initially empty)
for j in A:
    X.add(j)

for i in 0 to A.size()-1:
     x = get largest element from X
    remove largest element from X
    A[i] = x
```

What data structure(s) can we use to get a sorting algorithm with runtime O(n log(n))?

linked list	dynamic array	binary search tree	[✓ AVL tree]
hash table	[✓ red-black tree]	[✓ max-heap]	min-heap

Circle **all** correct answers – there may be several.

Brief explanation of why the data structure(s) you chose are suitable:
(You do not need to explain why the other ones are unsuitable.)

In a binary search tree, the three important operations (adding an element, deleting an element, and finding the largest element) are all O(h), where h is the height of the tree.

- For normal BSTs h = O(n), so that's not good enough
- AVL trees and red/black trees are self-balancing meaning that h = O(log n), so both of them work fine

In a max-heap, adding an element, and finding and removing the largest, are all O(log n).

Basic question 3: Ordered collections

Here is a program interacting with a collection data type:

```
S = new empty collection
S.add(3)
S.add(5)
print(S.remove())
S.add(4)
S.add(2)
S.add(8)
print(S.remove())
S.add(6)
print(S.remove())
```

Which numbers will be printed, and in which order, assuming that:

a) add/remove are enqueue/dequeue for a queue?

3
5
4

b) add/remove are push/pop for a stack?

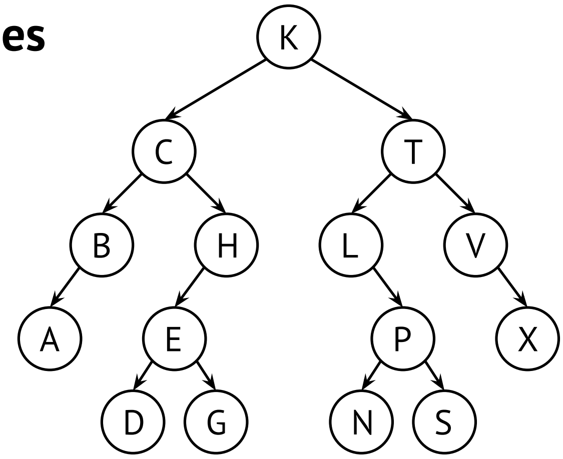
5
8
6

c) add/remove are add/removeMin for a priority queue?

3
2
4

Basic question 4: Binary search trees

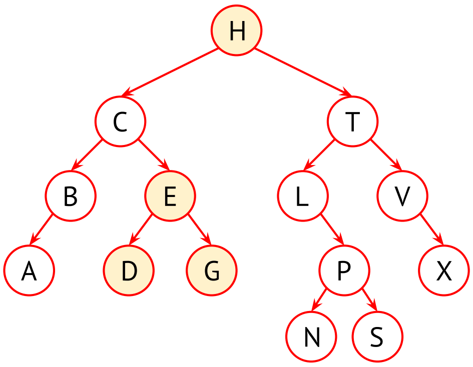
Given the following BST (binary search tree):



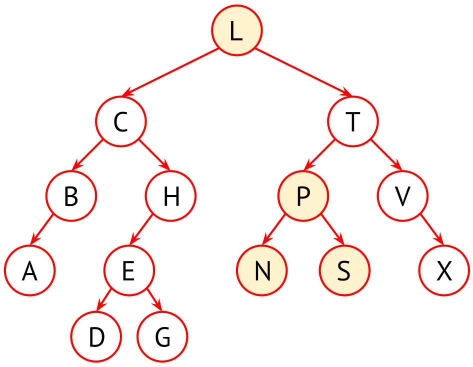
First, delete the node **K** from the tree (using the standard BST deletion procedure). What does it look like after that?

There are two ways of deleting a BST node – replacing it with the predecessor or successor.

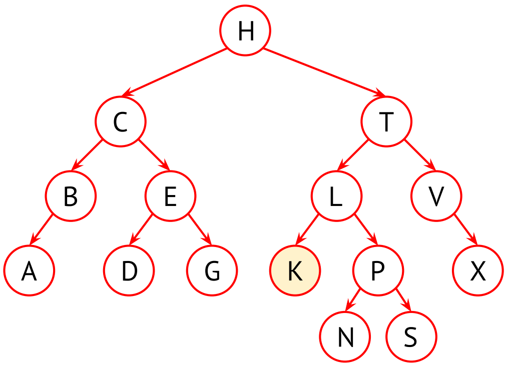
Replace K with the predecessor (H):

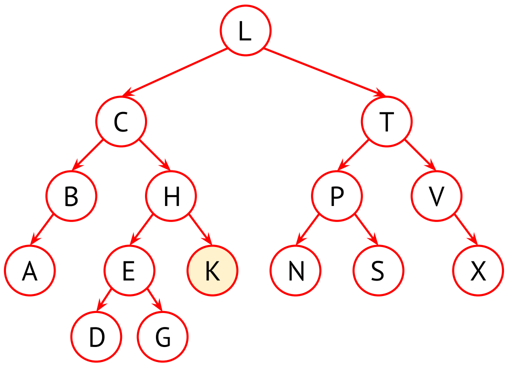


Replace K with the successor (L):



Afterwards, reinsert **K** into the tree (using the standard BST insertion procedure). What does it look like now?

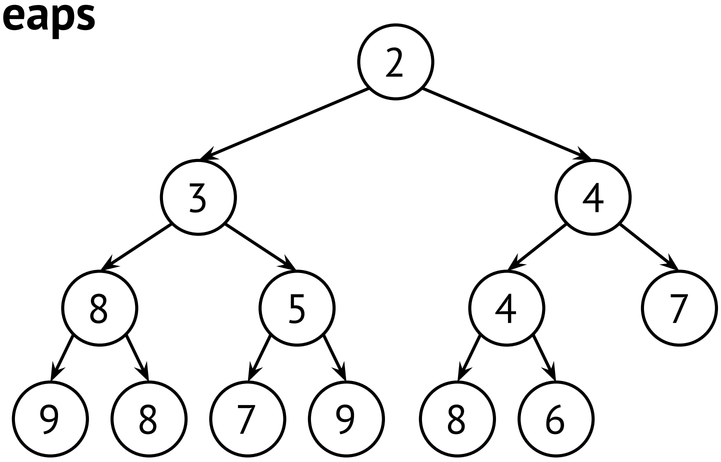




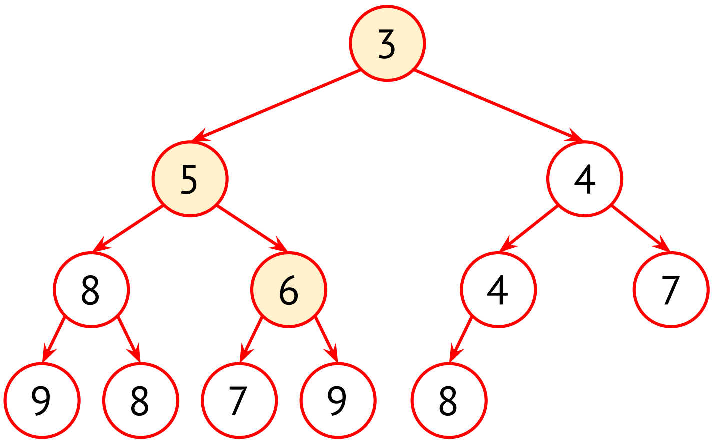
Note: the coloured nodes show the parts of the tree that were changed.

Basic question 5: Binary heaps

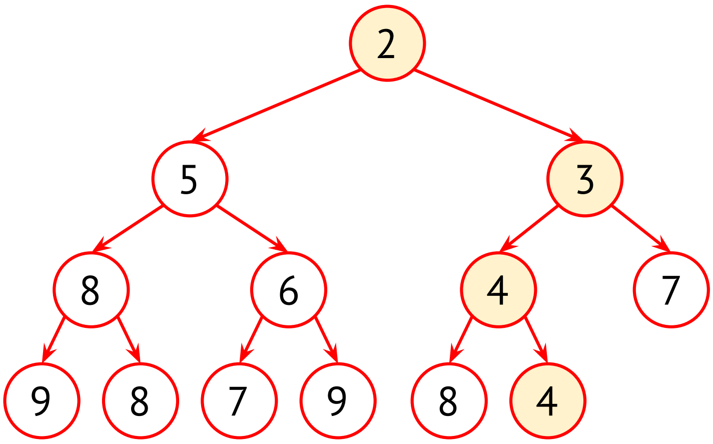
Given the following binary heap:



First, delete the minimum element from the heap – what does it look like after that?



Afterwards, reinsert the deleted number into the heap – how does it look like now?



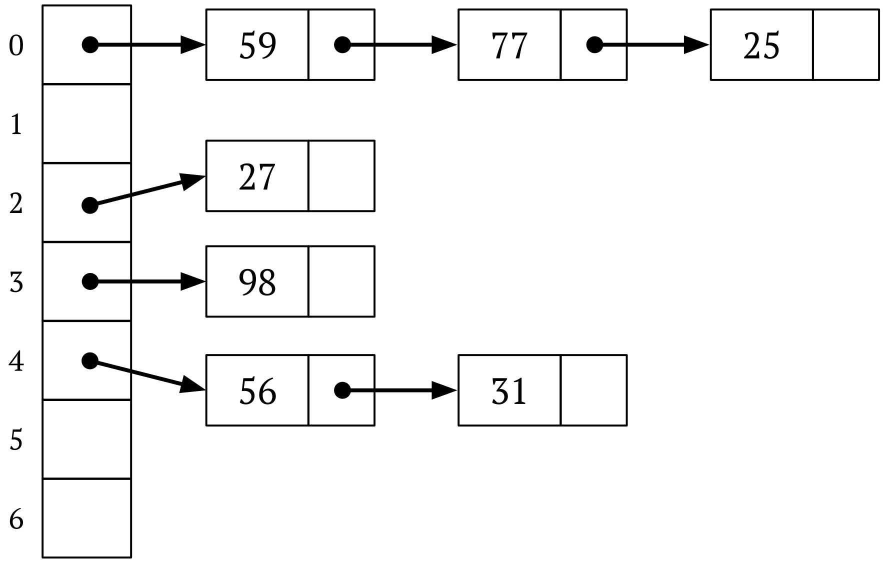
Note: the coloured nodes show the parts of the tree that were changed.

Basic question 6: Hash tables

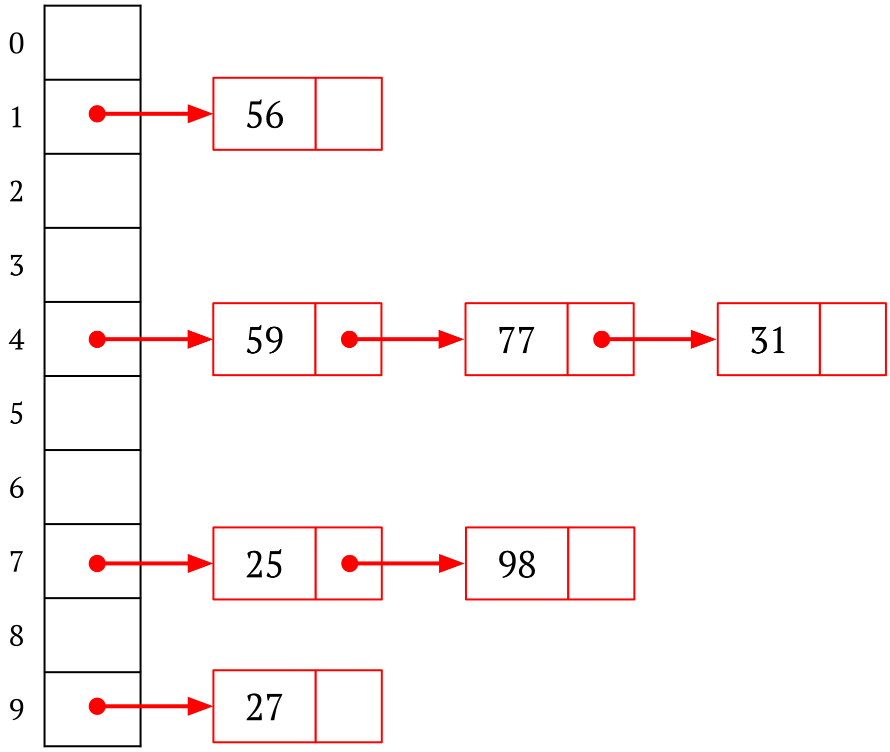
Here is a separate chaining hash table:

The hash function is the digit sum, so that e.g.:

$h(358) = 3+5+8 = 16$



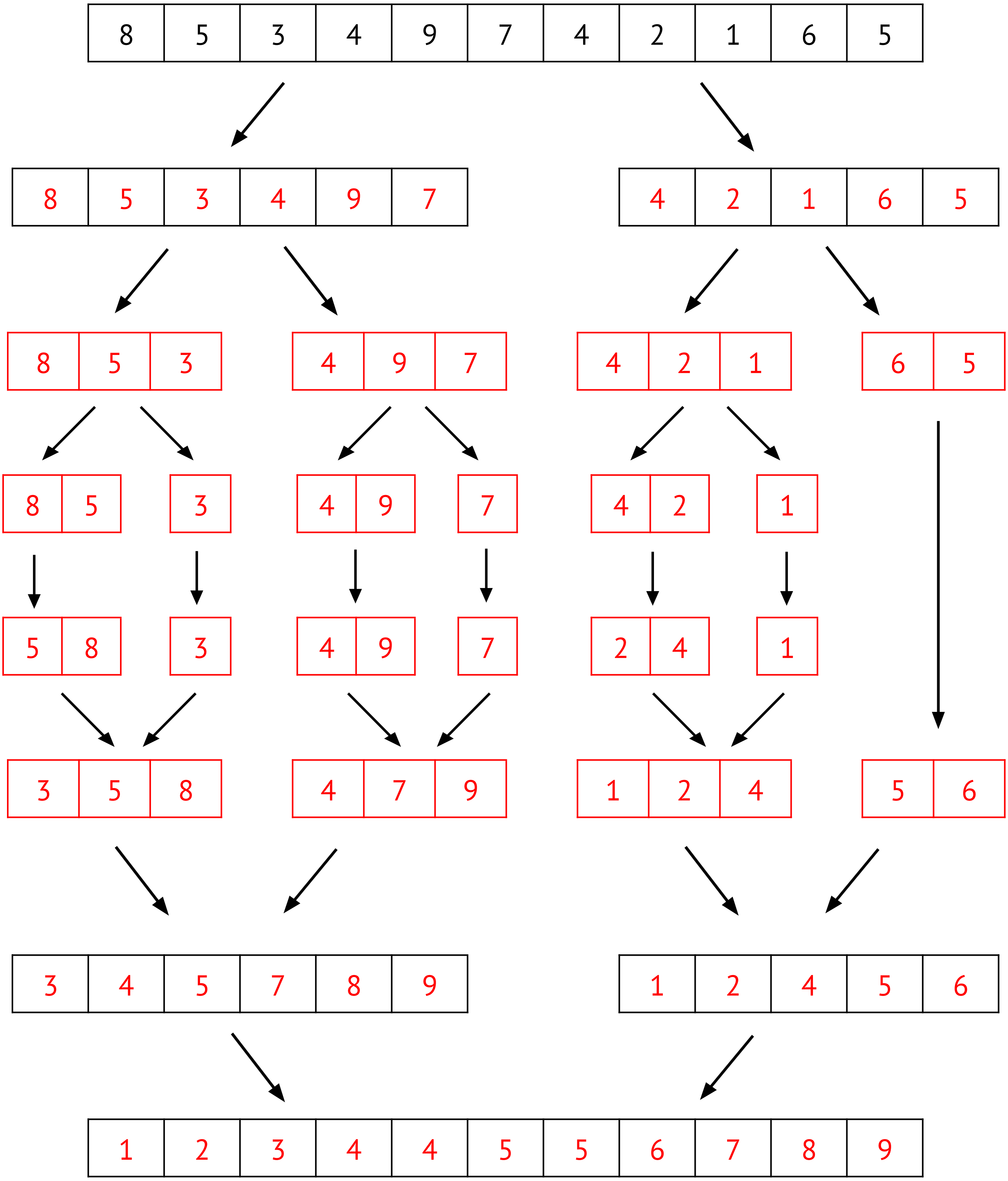
Resize the table to an array of 10 elements. What does the resulting hash table look like?
Make sure you use a consistent algorithm for deciding in which order elements should be inserted.



Note: There are other possible orders between 59/77/31, and 25/98 – but not all orders are ok. In this case we iterated through the old table from index 0 upwards and always added new nodes *at the end* of the linked list.

Basic question 7: Merge sort

Perform merge sort on the following list. Show each step of splitting and merging.



Basic question 8: Complexity of graph search

Here is an implementation of the general graph search algorithm.

```
graph_search(start, goal):
    visited = new set of vertices                  ____________ $O(1)$ ______
    agenda = new priority queue of vertices        ____________ $O(1)$ ______
    add start to agenda                            ____________ $O(1)$ ______
    while agenda is not empty:                     ______ $O(N)$ iterations ____
        current = remove a vertex from agenda      __________ $O(\log N)$ ______
        if current is not in visited:              __________ $O(\log N)$ ______
            add current to visited                 __________ $O(\log N)$ ______
            if current == goal:                    ____________ $O(1)$ ______
                return true                        ____________ $O(1)$ ______
            for next in neighbourVertices(current): ____ $O(1)$ iterations ____
                add next to agenda                 __________ $O(\log N)$ ______
    return false                                   ____________ $O(1)$ ______
```

Answer the following questions asymptotically in terms of *N*, the number of vertices in the graph:

- How many times will the while-loop be iterated? ____________ $O(N)$ iterations ______

- What is the asymptotic complexity of the algorithm? ____________ $O(N \log N)$ ______

Write your answers in O-notation, and be as exact and simple as possible.
Justify your answer by annotating each line in the code. Some lines are already done for you.

You can assume the following:

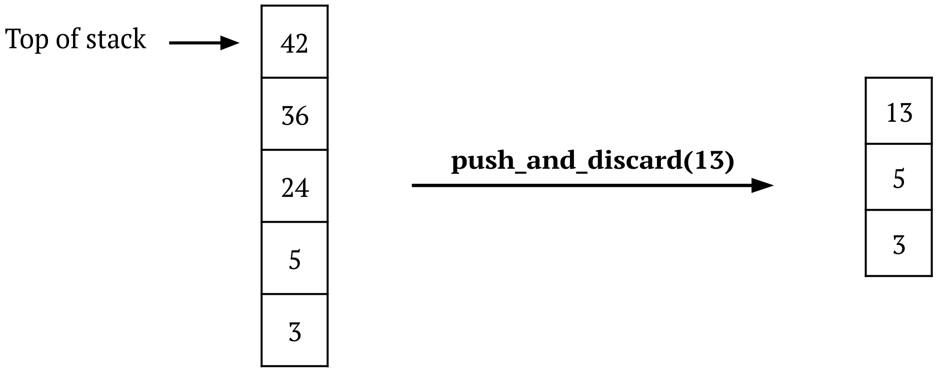
- the set `visited` is implemented as a *self-balancing binary search tree*
- the agenda is implemented as a *binary heap*
- there are at most a *constant number* of `neighbourVertices` per vertex
  (in other words, the graph is sparse)

Advanced question 9: Complexity of ordered stacks

An ordered stack is a stack where the elements appear in increasing order (from bottom to top). We implement the ordered stack as a linked list, maintaining a pointer to the top element. An ordered stack supports the operations:

- **init()**: creates an empty stack
- **pop()**: returns and deletes element on top of the stack
- **push_and_discard(*x*)**: pushes elements on top of the stack, maintaining the increasing order, i.e., elements larger than *x* are popped until *x* is the largest element in the stack

Below you see an example of an ordered stack and the resulting stack after performing a **push_and_discard** operation for the new element 13:



Answer the following questions on a separate sheet of paper.

A. Starting with an empty ordered stack, perform the following operations in the given order. Show the resulting stack after every operation.

> **push_and_discard(4)**
> **push_and_discard(9)**
> **push_and_discard(15)**
> **push_and_discard(5)**
> **push_and_discard(23)**
> **push_and_discard(3)**

B. What is the worst-case time complexity of **init**, **pop**, and **push_and_discard** in the size of the stack *N*? Write your answers in O-notation, and be as exact as possible.

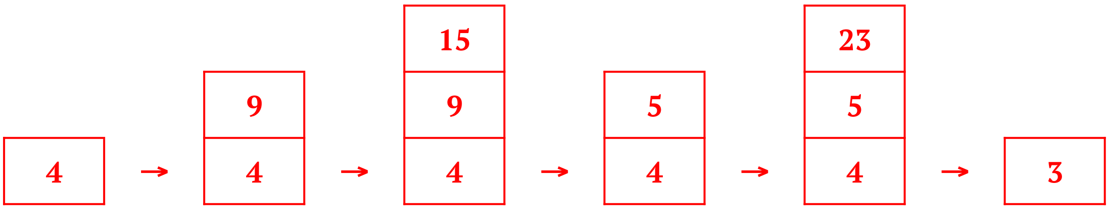
C. If we start with an empty ordered stack, what is the amortized complexity of **push_and_discard**? Write your answer in O-notation, and be as exact as possible! Provide an explanation for your answer!

   Hint: What is the worst-case complexity of performing *N* operations in sequence?

Complexity of ordered stacks: Solution

A. Starting with an empty ordered stack, perform the following operations in the given order. Show the resulting stack after every operation.

> **push_and_discard(4)**
> **push_and_discard(9)**
> **push_and_discard(15)**
> **push_and_discard(5)**
> **push_and_discard(23)**
> **push_and_discard(3)**



B. What is the worst-case time complexity of **init**, **pop**, and **push_and_discard** in the size of the stack *N*? Write your answers in O-notation, and be as exact as possible.

> **init**: O(1)
> **pop**: O(1)
> **push_and_discard**: O(*N*)

C. If we start with an empty ordered stack, what is the amortized complexity of **push_and_discard**? Write your answer in O-notation, and be as exact as possible! Provide an explanation for your answer!

> It's constant, O(1).
>
> The elements that **push_and_discard** pops are already elements that have been pushed before. So the worst case is that every element is pushed once and then popped some time later. So the worst case for performing *N* operations is that we get *N* pushes and *N* pops, which is O(*N*) "basic" operations.
>
> O(*N*) for performing *N* operations gives O(1) *amortised* complexity.

Advanced question 10: Diameter in a graph

Here are three definitions related to distance measures, copied from Wikipedia (https://en.wikipedia.org/wiki/Distance_(graph_theory)#Related_concepts):

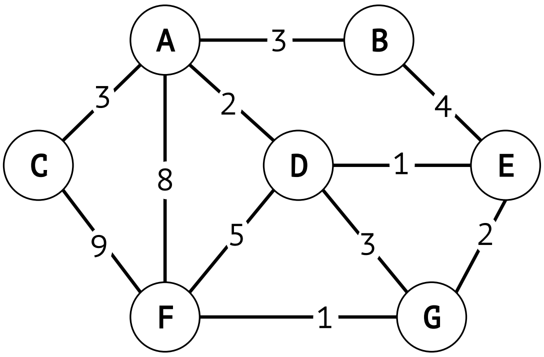
- The *distance* $d(v, u)$ between two vertices v and u is the length of a shortest path.
- The *eccentricity* $\epsilon(v)$ of a vertex v is the greatest distance between v and any other vertex:

  $$\epsilon(v) = \max_{u \in V} d(v, u)$$

  It can be thought of as how far a node is from the node most distant from it in the graph.
- The *diameter* d of a graph is the maximum eccentricity of any vertex in the graph.
  That is, d is the greatest distance between any pair of vertices or, alternatively:

  $$d = \max_{u \in V} \epsilon(u) = \max_{v \in V} \max_{u \in V} d(v, u)$$

A: What is the diameter of the graph on the right?



B: How can you find out the diameter of a graph?

Describe your algorithm as pseudocode, or as a detailed description. If you use data structures or algorithms from the course you don't have to describe how they work.

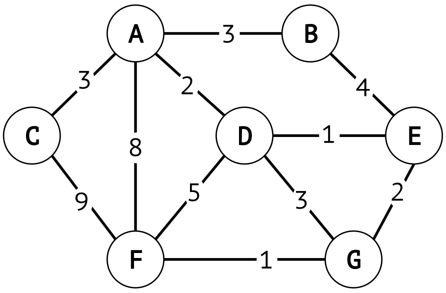
Write your answer on a separate sheet of paper.

C: What is the asymptotic complexity of your algorithm?

Describe the complexity in terms of the number of vertices $N = |V|$.
You can assume that the graph is sparse, meaning that the number of edges is proportional to the number of vertices, or $|E| \in O(|V|)$.

Justify your answer.

Diameter in a graph: Solution



A: What is the diameter of the graph on the right?

$d = d(C,F) = 9$ is the largest distance

(ε(A)=6, ε(B)=7, ε(C)=9, ε(D)=5, ε(E)=6, ε(F)=9, ε(G)=7)

B: How can you find out the diameter of a graph?

One very simple solution is to iterate over all pairs (*v,w*) of vertices and run Dijkstra/UCS to find *d*(*v,w*). Using Python comprehensions it would look like this:

```
d = max(searchUCS(v,w) for v in Vertices for w in Vertices)
```

A more efficient variant is to iterate over each vertex *v* and run Dijkstra/UCS from v to find the shortest path tree. Then you iterate over each vertex *w* to find the eccentricity of *v*, $\varepsilon(v)$. The code becomes just slightly more complicated:

```
d = max(eccentricity(v) for v in Vertices)

def eccentricity(v):
    spt = searchUCS(v)
    return max(spt.distance[w] for w in Vertices)
```

C: What is the asymptotic complexity of your algorithm?

The very simple algorithm uses two nested loops (O(N) each) and then runs UCS (O(N log N)). So the total complexity becomes:

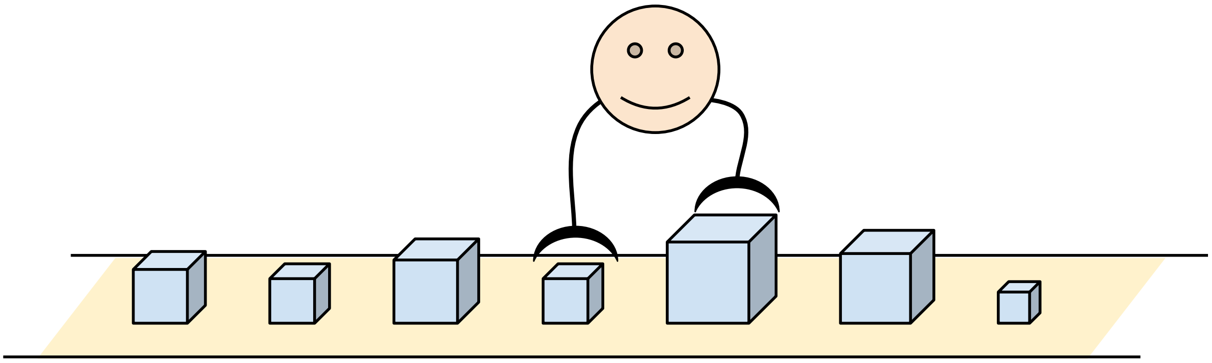
$$O(N) \cdot O(N) \cdot O(N \log N) \;=\; \mathbf{O(N^3 \log N)}$$

The more efficient algorithm uses one outer loop (O(N)) for calculating the eccentricity. This in turn first runs UCS (O(N log N)), followed by a loop over all vertices (O(N) or O(N log N) depending on how you store the SPT). The total complexity then becomes:

$$O(N) \cdot [O(N \log N) + O(N)] \;=\; \mathbf{O(N^2 \log N)}$$

Advanced question 11: Sorting robot

Assume you have a long line of *N* objects of different sizes, and a robot:



You want to instruct the robot to sort the long line (with the smaller objects to the left), but it has just a limited instruction set. The robot is always in front of two adjacent objects (as in the picture). The number of objects is known (called N) and the robot always starts in the leftmost position. The robot understands the following instructions:

- LEFT: move one step left – return true if it succeeded, otherwise false
- RIGHT: move one step right – return true if it succeeded, otherwise false
- COMPARE: compare the two objects in front – return true if the left object is smaller than the right object
- SWAP: swap places of the two objects in front – this always succeeds

A: Implement an algorithm that makes the robot sort the objects.

Write your answer on a separate sheet of paper.
You can use pseudocode, Java, Python, or some other programming language.

B: What is the asymptotic complexity of your algorithm in terms of *N*?

Justify your answer.

Sorting robot: Solution

A: Implement an algorithm that makes the robot sort the objects.

There are many possible solutions. Here is one based on bubble sort:

```
in_order = False
while not in_order:

    // Move to the left end.
    while LEFT:
        pass

    // Go from left to right and swap whenever
    // adjacent elements are in the wrong order.
    in_order = True
    loop:
        if not COMPARE:
            in_order = False
            SWAP

        if not RIGHT:
            break
```

Here is a very compact insertion sort variant:

```
while true:
    while not COMPARE:
        SWAP
        LEFT
    if not RIGHT:
        break
```

And some 100 more versions…

B: What is the asymptotic complexity of your algorithm in terms of *N*?

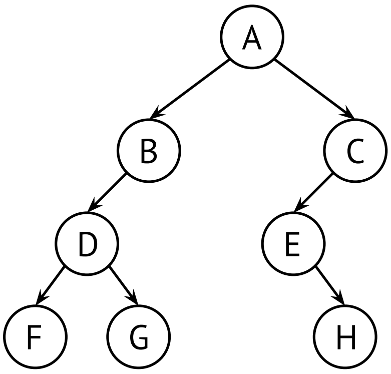
After k iterations, the k largest elements will be in the correct place. So the outer loop iterates at most N times.

The outer loop has $O(N)$ iterations. All inner loops have $O(N)$ iterations. Everything else is constant time. So the total complexity is $\mathbf{O(N^2)}$.

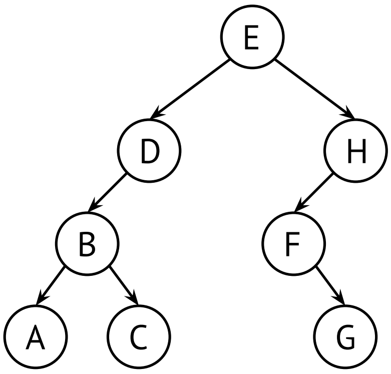
Advanced question 12: Convert a binary tree to a BST

Implement an algorithm that converts an unordered binary tree into a binary search tree (BST), without changing its skeleton, i.e., by retaining the structure of the tree. This means that the "skeleton" of the tree should remain the same, but the values of the nodes should move around so that the tree becomes a BST. Here's an example of how it should work:

The following unordered binary tree...



...should be transformed into this BST:



You can assume the following class definition for trees and nodes, where the values are strings. You can also assume that the node values can be compared in constant time.

```
class Node:
    left, right : Node
    value : String

class BinaryTree:
    root : Node
    size : int

    method convertToBST():
        ...this should be implemented...

    (...possible auxiliary private methods that you want to use...)
```

You can use any standard data structures and algorithms from the course without explaining how they work. You are allowed to create any kind of intermediate structure, but the final tree should maintain the same structure as the original tree.

Your solution must have worst-case complexity O($N \log(N)$) in the number of tree nodes N.

Hint: you can, e.g., use an array with all node values as an intermediate structure.

Write your answer on a separate sheet of paper.
You can use pseudocode, Java, Python, or some other programming language.

Convert a binary tree to a BST: Solution

The idea is quite simple:

1. collect all nodes into an array
2. sort the array
3. do an inorder traversal of the tree, assigning each node the next value in the array

Collecting the nodes, and reassigning them, are very similar processes – both can be done using inorder traversal of the tree, and the only difference is if we assign the array cell the node value, or the other way around.

Here is a version using recursive tree traversal:

```
convertToBST():
    array = new Array of size: self.size
    collectNodes(self.root, 0, array)
    sort(array)
    fillNodesFromArray(self.root, 0, array)

collectNodes(node, i, array) -> int:
    if node is NULL:
        return i

    j = collectNodes(node.left, i, array)
    array[j] = node.value
    k = collectNodes(node.right, j+1, array)
    return k

fillNodesFromArray(node, i, array) -> int:
    if node is NULL:
        return i

    j = fillNodesFromArray(node.left, i, array)
    node.value = array[j]
    k = fillNodesFromArray(node.right, j+1, array)
    return k
```

Note that both `collectNodes` and `fillNodesFromArray` take an integer as input and returns an integer – this is a counter of how many nodes they have visited, and is the corresponding index in the array.

In-order traversal is linear in the size of the tree, so the dominating part is sorting the nodes. Therefore, the complexity is $O(N \log N)$.

Convert a binary tree to a BST: Even simpler solution

An even simpler solution is to collect the nodes in a priority queue instead of an array. Then you don't need the counter i, and you don't have to call `sort` in the middle.

```
convertToBST():
    pq = new MinPriorityQueue
    collectNodes(self.root, pq)
    fillNodes(self.root, pq)

collectNodes(node, pq):
    if node is not NULL:
        collectNodes(node.left, pq)
        collectNodes(node.right, pq)

fillNodes(node, pq):
    if node is not NULL:
        fillNodesFromArray(node.left, pq)
        node.value = pq.removeMin()
        fillNodesFromArray(node.right, pq)
```