Basic question 1: Sorting

Here is a sketch of quicksort:

```
function qs(array: array of integers, low: integer, high: integer):
        if low < high:
                pivotPosition = partition(array, low, high)
                qs(array, low, pivotPosition-1)
                qs(array, pivotPosition+1, high)
```

The function call partition(array, low, high) chooses a pivot in an unknown way and partitions the array range [low, high] before returning the new position of the pivot element.

Consider the following array:

0	1	2	3	4	5	6	7	8
10	5	7	3	9	2	8	3	11

Calling partition(array, 0, 8) returns 4.

- What was the chosen pivot value?
- Which values do the parts of the partition now contain?

Note: the order of values in your answers does not matter.

Pivot value: 7

Values in left part: 2, 3, 3, 5

Values in right part: 8, 9, 10, 11

Basic question 2: Hash tables

The following open addressing hash table models a set S of animals. It uses modular compression (using the modulo operator) and linear probing (with probing constant 1, as usual).

0	1	2	3	4	5	6	7	8	9
Fly	Cat			Bee				Gnu	Ant

For each of the following animals x, state how many array cells (possibly empty) the call

$$S.contains(x)$$

accesses.

Animal	Hash code	Number of array accesses
Bee	7114	1
Cat	3650	2
Dog	9374	2
Elk	1509	4

Basic question 3: Search trees

Draw an AVL tree representing the set of integers {1, 2, 3, 4, 5, 6, 7} that is as **unbalanced** as possible (that is, has maximum height without breaking the AVL invariants).

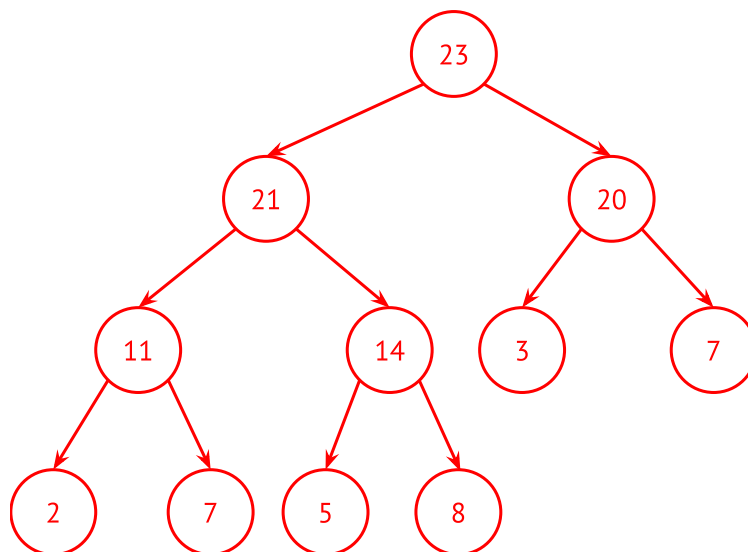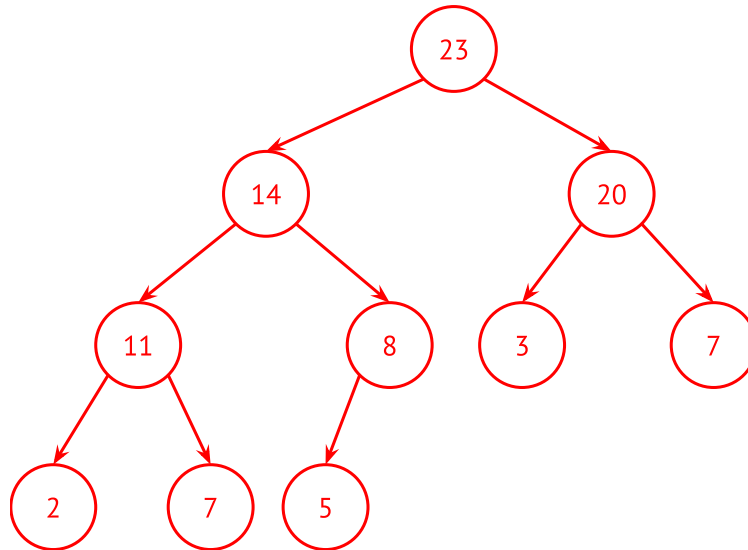There are many possibilities (all of the same height). This one leans to the left as much as possible:
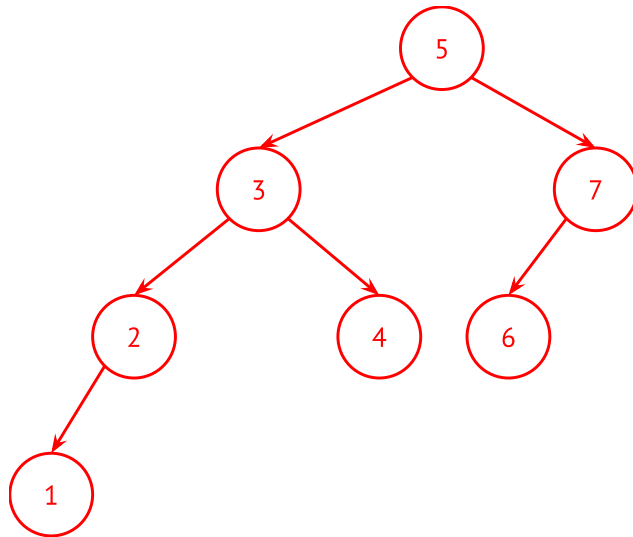
Basic question 4: Priority queues

The following array represents a binary max-heap:

| 0	1	2	3	4	5	6	7	8	9
23	14	20	11	8	3	7	2	7	5

a) Draw the tree representation of the heap.

b) Draw the tree representation of the heap after adding the value 21 to it.

Basic question 5: Graphs

We run this familiar piece of code on the graph to the right:

```
visited = new map from nodes to distances
agenda = new min-priority queue of pairs of cost and node
agenda.add( (0, A) )
while agenda is not empty:
    (cost, node) = agenda.removeMin()
    if not visited.containsKey(node):
        visited.put(node, cost)
        for e in outgoingEdges(node):
            agenda.add( (cost + e.weight, e.target) )
```
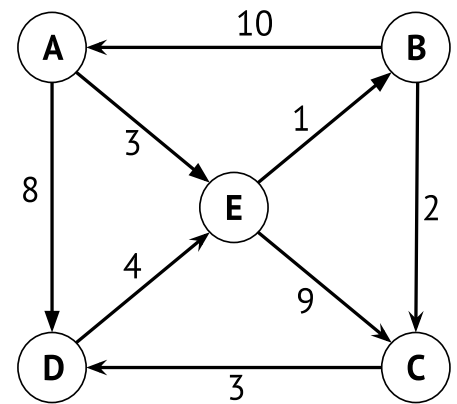
We run three iterations of the while-loop and then stop. **At that point**:

- What are the three entries of the *visited* map (you get one for free)?

$$A \mapsto 0$$

$$E \mapsto 3$$

$$B \mapsto 4$$

- What are the four entries of the *agenda*, in ascending order?

Cost	Node
6	C
8	D
12	C
14	A

Basic question 6: Complexity

Your boss wants you to calculate the product a · b for large integers a, b ⩾ 0. Just after your multiplication operator broke down last week! Here are two workarounds you came up with:

```
function mulA(a, b):
    if a == 0:
        return 0
    return b + mulA(a - 1, b)
```

```
function mulB(a, b):
    result = 0
    while a > 0:
        if a % 2 == 1:
            result = result + b
        result = result + result
        a = a // 2
    return result
```

State the asymptotic time complexity of each algorithm **in terms of a**.

In each case, **briefly state** how you concluded this (you may do this by annotating the programs).

Notes:

- Assume that all arithmetic operations take O(1) time.
- If you answer in O-notation, be as exact and simple as possible.
- We write a // d for the *quotient* and a % d for the *remainder* in integer division of a by d.

Asymptotic complexity of mulA**:** $O(a)$

Short justification: The first argument decreases by one for every recursive call until it reaches 0, So mulA(a, b) makes a many function calls overall. The work in each call is O(1).

Asymptotic complexity of mulB**:** $O(\log(a))$

Short justification: The while-loop has $\log_2(a)$ iterations. Its body takes O(1) time.

[mulB does not actually implement multiplication. It goes over the bits of a in the wrong order.]

Advanced question 7: Heapification

Recall the helper functions that we use to implement a binary heap:

- *swim* (also called *swimming up* or *sifting up*),
- *sink* (also called *sinking down* or *sifting down*).

Here are two algorithms for turning an array of integers into a binary heap:

(1) Go over the array in *forward* order and call *swim* at every cell.
(2) Go over the array in *backward* order and call *sink* at every cell.

For each algorithm, determine and justify the asymptotic time complexity in the array size n. Based on your analysis, conclude which algorithm has better scaling behaviour.

Hint: In a complete binary tree, the average node height is $O(1)$. You can use this without proof.

The representation of the array as a complete binary tree has $O(\log(n))$ levels. Most nodes are close to the bottom level, with level close to $\log_2(n)$ and height close to 1. We observe:

- *Swimming* a node reduces its level by one with every swap, so takes time linear in its **level**. Most nodes are near the bottom, with level close to $\log_2(n)$. So the average time is $O(\log(n))$.
- *Sinking* a node reduces its height by one with every swap, so takes time linear in its **height**. By the hint, this averages over all nodes to $O(1)$.

Summing over all n nodes, we get time cost $O(n \log(n))$ for algorithm (1) and $O(n)$ for algorithm (2). So algorithm (2) has better scaling behaviour.


Not required: Here is a proof that the nodes in a complete binary tree have average height $O(1)$. Because order of growth allows for constant factors, it suffices to show that the average node height in a perfectly balanced binary tree is less than 2. We induct on the overall height of the tree.

- If there is only one level, then the average node height is 1.
- Adding a new level of nodes increases the height of all old nodes by one. By induction hypothesis, the old nodes had average height less than 2 before, so now their average height is less than 3. The number of new nodes (with height 1) is larger than the number of old nodes (by one). Therefore, the average node height is less than (3 + 1) / 2 = 2.

Advanced question 8: Water world

In the game of water world, a player must navigate a hexagonal grid of map tiles without drowning. The grid is implemented using this class:

> **class** Tile: // can be compared/hashed in O(1) time
>   elevation: number
>   neighbours: list of Tile   // at most six neighbours
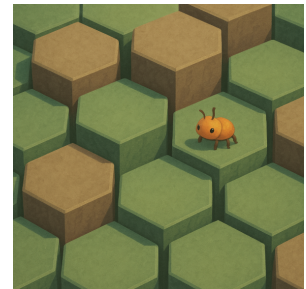
The player can jump from a tile to a neighboring tile if their elevations differ by at most 2.

The water level of the world starts at 0 and increases by 1 after making a jump. The player drowns if the water level exceeds the elevation of their tile.

Specify an algorithm

> **function** solvable(start: Tile, goal: Tile) → boolean

that returns true if the player can get from the start tile to the goal tile without drowning. Your algorithm should run in average O(N) time where N is the number of grid tiles.

Notes:
- You do not have to justify the complexity of your algorithm.
- You can use **unambiguous** natural language or pseudocode to describe your algorithm.
- You can freely use data structures and algorithms from the course – you do not have to explain how they work. In particular, hash sets of tiles have average-case O(1) operations.

High-level answer: We modify breadth-first search by adding a maximum cost to each node. We keep track of the cost (length of path so far) in agenda entries using a queue. We only visit a node if the cost does not exceed the maximum cost. We run this on the following graph:

- The nodes are tiles.
- The outgoing edges are given by neighbours with elevation difference at most 2.
- The maximum cost of a node is its elevation.

Detailed answer: We run a breadth-first graph search with added limitations on elevation and water level. The agenda contains pairs of a water level and a tile. Before visiting a tile, we check if it is flooded. (This also catches the case that the starting tile has negative elevation.)

```
visited = new set of visited tiles                    // uses a hash set
agenda = new queue of pair of Tile and water level    // for example, a circular dynamic array
agenda.add( (0, start) )
while agenda is not empty:
    (level, tile) = agenda.dequeue()
    if not visited.contains(tile) and level ⩽ tile.elevation [and level ⩽ goal.elevation]:
        visited.add(tile)                              ↑ possible optimization
        if tile = goal: return true
        for next in tile.neighbors:
            if |next.elevation − tile.elevation| ⩽ 2:
                agenda.enqueue( (level+1, next) )
return false
```

Advanced Question 9: Ordered Set

Recall that a **set** of values of type *T* has the following operations:

- contains(x: T) → boolean: check if *x* is in the collection.
- add(x: T): add *x* to the collection unless it is already an element.

An **ordered set** additionally remembers the order in which its elements were added.
For our purposes, it should support the following additional operations:

- pop() → T: remove and return the **newest** element of the ordered set.
  This method assumes that the ordered set is not empty.
- replace(x: T, y: T): replace *x* with *y* without changing its position in the order of elements.
  This method assumes that *x* is an element and *y* is not currently an element.

We assume that T has a good, constant-time hash function. Design a data structure implementing an ordered set such that all of the above methods run in **average $O(1)$ time**. Your answer should:

- define the data structure,
- state the implementations of *pop* and *replace*.

Note: you can freely use data structures and algorithms from the course – you do not have to explain how they work.

Hint: A standard hash table implements a set with the desired complexity, but not an ordered set. A hash table can still be a useful ingredient of your solution (together with additional structure).

We want to combine the behaviour of a stack and a set. For the *replace* method, we need to be able to find elements in the stack. So instead of just storing values in a hash set, we store references to where those values appear in the stack. Here is an example implementation:

class HashedLinkedList **implements** OrderedSet:
    stack: linked list stack
    map: hash map from T to nodes in stack

    method replace(x: T, y: T):
        node = map.get(x)
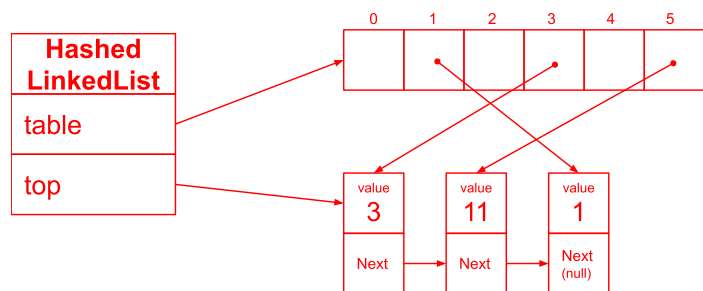   node.value = y
        map.remove(x)
        map.put(y, node)

    method pop() -> T:
        x = stack.pop()
        map.remove(x)
        return x

The picture depicts an optimized implementation. In a standard hash map, the hash table stores pairs of a value and a node reference. Here, the node reference suffices as it already has the value.