Question 1: Complexity

Consider the following (hybrid) in-place sorting function:

```
sizeThreshold = 10000     // array size threshold
recursionThreshold = 10   // recursion depth threshold

sort(int[] array) {
  hybridSort(array, 0, array.length-1, 0)
}

hybridSort(int[] array, int lo, int hi, int depth) {
  if (hi-lo+1 < sizeThreshold) {
    insertionSort(array, lo, hi)
  } else if (depth ≥ recursionThreshold) {
    mergeSort(array, lo, hi)
  } else {
    i = partition(array, lo, hi)
    d = depth + 1
    hybridSort(array, lo, i - 1, d)
    hybridSort(array, i + 1, hi, d)
  }
}
```

The function `partition(array, lo, hi)` partitions the part of the array between index `lo` and index `hi`: it selects the first element (`array[lo]`) as pivot and rearranges `array[lo..hi]` such that smaller elements are placed before the pivot and larger elements after the pivot. It returns the new index of the pivot.

Your task is to describe the asymptotic (worst-case) complexity of `sort` in terms of the size of its input. Express your answer in the simplest form possible. You may assume that the functions `partition`, `insertionSort` and `mergeSort` have the expected asymptotic complexities. The answer needs to be well-motivated!

Write n for the length of `array`. We have to find the asymptotic complexity in n.

`hybridSort` is a recursive function. The branching factor of the recursion is two. However, the depth of the recursion is limited to a constant number. So in total, there will only be a constant number of calls to `hybridSort` (1 at depth 0, ≤2 at depth 1, ≤4 at depth 2, …, ≤2^i at depth i for i ≤ 10). Since asymptotic complexity ignores constant factors, the asymptotic complexity of sort will be that of a call the `hybridSort`, ignoring the recursive subcalls (the calls to `hybridSort`) since we already account for those.

The call to `insertionSort` is quadratic in the length `hi-lo+1` of the part of the array under consideration, but it is only called when this length is below the constant `sizeThreshold`, so has constant complexity.

The call to mergeSort is O(k log(k)) where k = `hi-lo+1`. Since k ≤ n, it is also O(n log(n)).

The call to partition is linear in k, hence also O(n).

All other operations are arithmetic, hence O(1).

In total, the complexity is O(n log(n)).

(Note. The complexity is actually Θ(n log(n)), i.e. not better than O(n log(n)). This can be seen as follows. For large enough n, at a recursion depth of 10, there will be at least one call to mergeSort with k ≥ n/2^10 (by the pigeonhole principle). This call is Θ(k log(k)) = Θ(n log(n).)

Question 2: Quicksort partitioning

We consider the following array with an unknown entry X:

[X, 21, 15, 27, 5, 1, 10, 12, 19]

The elements are integers in the range 0–29 (inclusive) and are unique (no repetitions).

Part A

We use the partitioning algorithm of quicksort, using the **first element** as pivot. Call the left partition L and the right partition R. The pivot is not part of either partition.

For each of the below statements, state **all** possible values for X that make it true.

1. L has size 4 and R has size 4.
2. L has size 2 and R has size 6.
3. L has size 6 and R has size 2.

Note: You do not have to justify your answer. You may use ranges (e.g. 3–7), intervals, or other common set notation in your answer.

L consists of all elements smaller than the pivot and R consists of all elements larger than the pivot. In this part, the pivot is X.

To help us, we list the other elements in sorted order:

$1 < 5 < 10 < 12 < 15 < 19 < 21 < 27$

We can now read off which range X needs to be in to make each statement true. Recall that X needs to be different from the other elements.

1. 13–14

2. 6–9

3. 20

Part B

We now select the pivot using **median-of-three** (using the first, middle, and last element of the array). Solve the problem from Part A with this change.

In this part, the pivot is the median of X, 5, 19. There are three cases, depending on where X lies relative to the other two numbers.

- If X is between 5 and 19, then X is the pivot and L and R are as in Part A.
- If X is less than 5, then 5 is the pivot and R consists of the given elements larger than 5, so has size 6 (and then L has size 2).

- If X is larger than 19, then 19 is the pivot and L consists of the given elements smaller than 19, so has size 5 (and then R has size 3).

WIth this information, we can once again read off from

1 < 5 < 10 < 12 < 15 < 19 < 21 < 27

the ranges that X needs to be in to make each statement true.

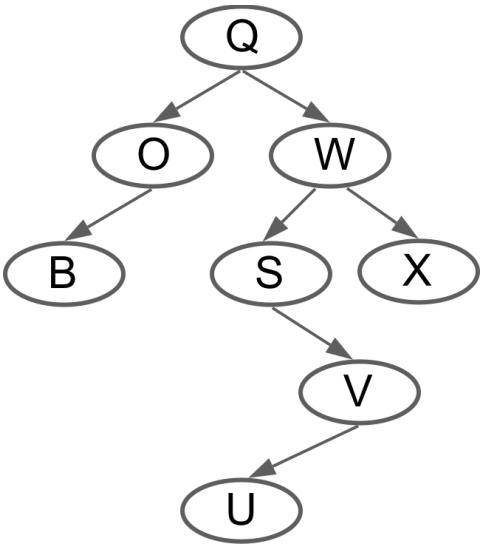
1. 13–14

2. 0, 2–4, 6–9

3. no possible values

Question 3: Trees

Do an insertion and removal in the displayed binary search tree (BST). The key ordering is alphabetical.



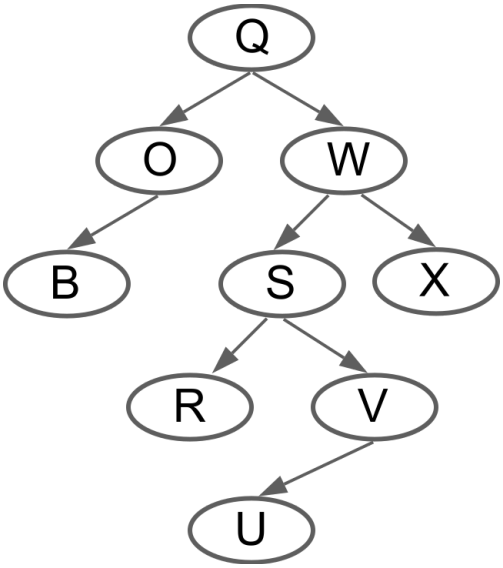
Part A

Insert **R** using the standard insertion algorithm. Show the new tree. Explain what comparisons the algorithm makes during the insertion.

Starting from the root node (at the top), we compare R with the node value and go to the left child if it is smaller and to the right child if it is larger:

- larger than Q, going right
- smaller than W, going left
- smaller than S, going left

We have reached an empty subtree (null). That's where we put the new node.



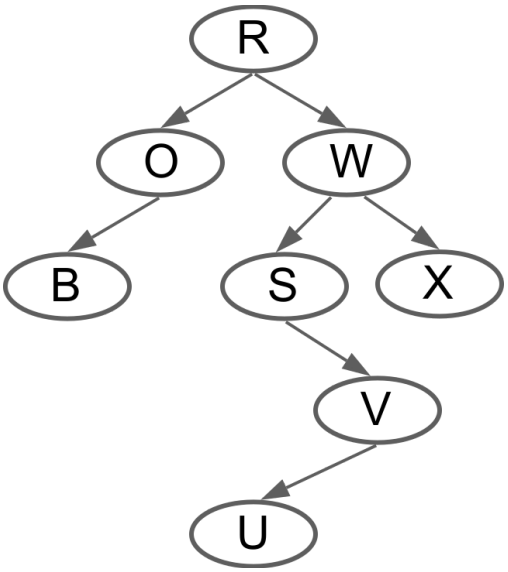
Part B

From the BST resulting from Part A, remove **Q** using the standard deletion algorithm. Show the final tree and explain how the algorithm has rearranged it.

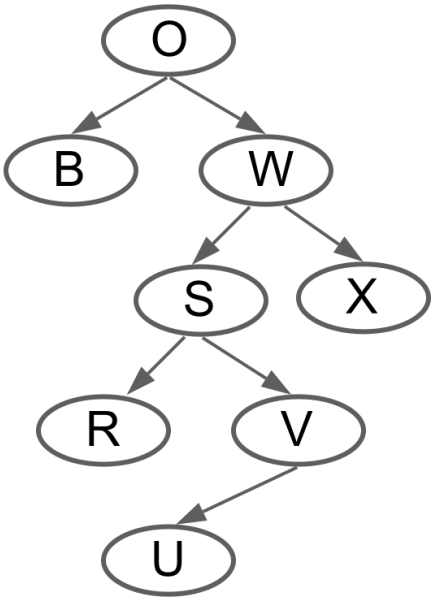
The node to be deleted has two children. In this case, we have to change the tree structure at another node. For this, we have two options, both equally valid: either the largest node in

the left subtree (option L, this is what is done in the lectures) or the smallest node in the right subtree (option R).

Option R. The smallest node in the right subtree of Q is R. Since it has no children, we replace it by an empty subtree (null) and overwrite the value of node Q with R.



Option L. The largest node in the left subtree of Q is O. By its choice, its right subtree is empty (null). We replace it by its left subtree and overwrite the value of node Q with O.



Question 4: Priority queues

This question is about **binary max-heaps** and their representations using arrays and trees.

Part A

Exactly one of the following arrays represents a max-heap:

a. [19, 18, 17, 15, 1, 12, 11, 10, 4, 13]
b. [0, 4, 5, 7, 8, 12, 15, 16, 18, 19]
c. [17, 15, 14, 12, 15, 2, 0, 1, 11, 1]

Find it. For the other ones, say why they are not max-heaps by concretely pointing out what is wrong with some elements.

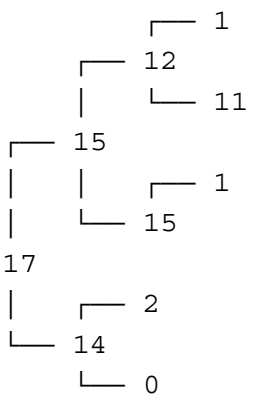
a. This is not a max-heap: the element 13 is larger than its parent 1.
b. This is not a max-heap: the root 0 is not the maximum. (Note that it is a min-heap instead.)
c. This is a max-heap.

Part B

Remove the maximum from the max-heap you have identified in Part A. Your answer must state:

- the sequence of swaps performed,
- the resulting **tree representation** of the max-heap, and
- the resulting **array representation** of the max-heap.

It helps to first visualize the heap as a tree:



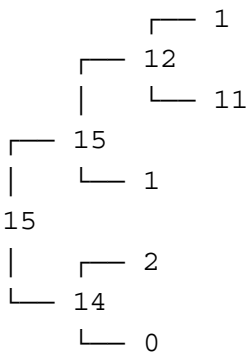
(Note that the tree representation is diagonally flipped compared to the usual way. This is just because it is easier to write this way using text.)

We swap the root 17 with the last element 1, delete the new last element 17, and then sink down the root 1:

We swap 1 with its left child 15.

We swap 1 with its right child 15.

Final state represented as a tree:



Final state represented as an array: [15, 15, 14, 12, 1, 2, 0, 1, 11]

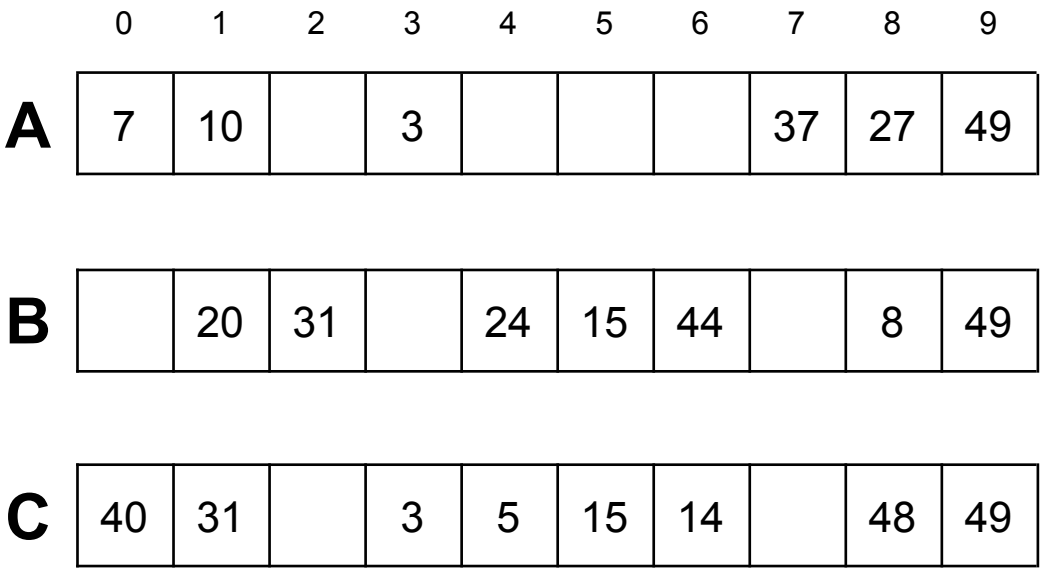
Question 5: Hash tables

Part A

Which of the following three arrays is a valid *hash table using linear probing*?
Assume that the hash table represents a set of integers, and that the hash function used is *h(n) = n* (modulo the array size). The array size is 10.

Write down:

- which array (A, B or C) is the valid hash table
- why the other two arrays are not valid hash tables - be specific, e.g. by pointing out which array elements cause a problem



A is a valid hash table. *[Note: 7 might look strange, but it will end up at that position if inserted after 37, 27 and 49.]*

B is not valid because e.g. looking up 20 will fail.

C is not valid because e.g. looking up 5 will fail.

Part B

Now delete 49 from the hash table, using lazy deletion. How does the array look afterwards?

Finally, after deleting 49, suppose that you look up 38 (which is not present) in the hash table. Which positions in the array does the lookup algorithm check?

Write down:

- how the array looks after deleting 49

- which positions in the array the lookup algorithm checks

After deleting 49, the hash table will look like this (where null represents an empty space and XXX a deleted value):

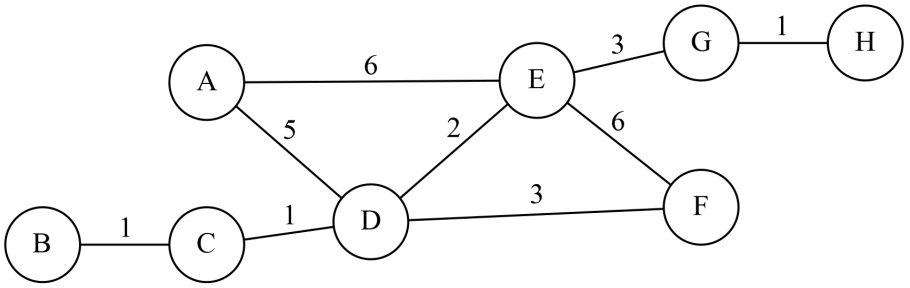
{7, 10, null, 3, null, null, null, 37, 27, XXX}

(Just replace 49 with XXX.)

Looking up 38 will check positions 8, 9, 0, 1, and 2 of the array.

Question 6: Graphs

You are given the following weighted, undirected graph:



Now, let's define the *eccentricity* of a node v – it's the maximum shortest distance between v and any other node (definition from Wolfram Alpha):

$$\mathrm{ecc}_G(v) = \max_{w \in G} \mathrm{dist}(v, w)$$

Part A

- What is the eccentricity of node B?
- Which node is the furthest away from B?
    - i.e., give a node with the same distance from B as the eccentricity
    - note that there might be several matching nodes, but give only one
- Give a shortest path from B to the furthest node

Eccentricity of B: **8**

Node furthest away from B: **H**

Shortest path from B to the furthest node: **B-C-D-E-G-H**

Part B

- What is the minimum eccentricity, and which node has it?
- What is the maximum eccentricity, and which node has it?

Note that there might be several matching nodes, but you should only give one.

Minimum eccentricity: **6**

Node with minimum eccentricity: **D or E**

Maximum eccentricity: **10**

Node with maximum eccentricity: **A or H**

Question 7 (advanced): Autocorrect [3 points]

Note: this question is worth **3 advanced points.**

In this question you will design an algorithm for fixing common spelling mistakes, similar to the *autocorrect* feature which is built into most word processors and phones.

The task is as follows. You are given two inputs:

1. A list of possible spelling mistakes, along with the correct spelling for each one. (See the next page for information on what data structure this list uses.)
   For example:

Mistake	**Correct**
could of	could have
rigth	right
speled	spelled
teh	the
tihs	this

2. A string which might contain some of these mistakes, for example:

"tihs could of been speled rigth"

Your job is to fix the spelling mistakes. More precisely, your algorithm should:

- Go through the input string, finding all the *substrings* that are mistakes. A substring is a mistake if it is *equal* to some entry from the mistake list.
- Replace each such substring with its correct spelling as given in the mistake list.
- Return the resulting corrected string.

For the example above, the mistake substrings are "tihs", "could of", "speled", and "rigth", so your algorithm should return:

"this could have been spelled right"

Here is another, more surprising example. Given the string:

"*tihs could of*fend someone"

and the same mistake list as before, your algorithm should produce:

"*this could have*fend someone"

Note that the substring "could of" is replaced by "could have", even though it occurs as part of "could offend". This is OK: your algorithm should not consider word boundaries—it should only check if each mistake string is present as a substring of the input.

Your task. Design and implement an algorithm for solving this task. Your algorithm should be implemented as a method

```
String autocorrect(String text, RedBlackMap<String, String> mistakes)
```

where:

- `text` is the text to be corrected.
- `mistakes` is a map, implemented as a red-black tree, containing common misspellings. In the red-black tree, the key is a misspelling and the value is the correct spelling. In the example above, `corrections.get("teh")` would return `"the"`.
- The return value is the corrected text.

To get full credit, the *number of string comparisons* that your algorithm makes should be worst-case **O(*n* log *r*)**, where:

- *n* is the length of the input text
- *r* is the size of the red-black tree (i.e. the number of possible misspellings)

Part A

Write down your algorithm. You can use any standard libraries and data structures, or the course API. You can write your answer in Java, Python or pseudocode.

Here is one way:

```
result = new dynamic array of characters
while text.length > 0:
  // Check if a prefix of text is present as a key in mistakes
  // (see floorKey examples below)
  maybeMistake = mistakes.floorKey(text)
  // Is the floor actually a prefix?
  if text.first(maybeMistake.length).equals(maybeMistake):
    // This is a mistake - correct it
    result.addAll(mistakes.get(maybeMistake))
    // Skip to after the mistake
    text = text.skip(maybeMistake.length)
  else:
    // No mistake here - output the first character and
    // skip to the next character
    result.add(text[0])
    text = text.skip(1)
return new String(result)
```

Part B

How many string comparisons does your algorithm make in the worst case (big-O, in terms of *n* and *r*)? Briefly justify your answer.

O(n log r). The loop runs at most n times. Each iteration makes one call to equals (1 string comparison) and two calls of red-black tree methods (O(log r) string comparisons).

Hints

Assumptions. You may assume that all strings are lowercase, so that you don't have to consider capitalisation.

You may assume that fixing a spelling mistake never introduces *more* spelling mistakes. So it's enough to go through the input string once.

You may assume that, if the red-black tree contains a string *s* as a key, it does not contain any substring of *s*. For example, if the tree contains "speled", it does not contain "spel" or "el" or "led".

Ordering. Remember that the ordering used when comparing strings is alphabetical, where e.g. "car" < "cart" < "cat" < "catamaran".

Red-black trees implement *ordered map* operations. You will probably need to use these! See the `OrderedMap` interface in the Course API for DAT038/TDA417 for details. In particular, the operation `map.floorKey(k)` looks up `k` in the map. If `k` is present, it returns `k`. Otherwise, it returns the closest (i.e. greatest) key which is less than `k`. If no key is less than `k`, it returns `null`. For example, using the mistake list on the previous page:

- `mistakes.floorKey("teh apple")` returns `"teh"` since `"teh"` is the greatest key in the map for which `"teh"` ≤ `"teh apple"` - there are no other keys between `"teh"` and `"teh apple"` in alphabetical order
  - in this case, your algorithm should replace `"teh"`
- in the same way, `mistakes.floorKey("test")` also returns `"teh"` since this is the greatest key smaller than `"test"`
  - in this case, `"test"` should not be replaced by the algorithm
- `mistakes.floorKey("apple")` returns `null` because no key comes before `"apple"` alphabetically

Strings. It's fine to treat a string as an array of characters, e.g. by writing `str[i]` to get the `i`th character of `str`.

You will need a way to create *substrings* of the input string. You can assume that a way to do this exists and that it takes **O(1)** time. For example, if you are familiar with the Java substring method or Python array slicing, you can use those. Otherwise, you can assume that the String class has the following two methods, both taking **O(1)** time:

- `String first(int n)`: return a string containing the first *n* characters of this string. For example, if `str = "example"`, then `str.first(4)` would return `"exam"`.

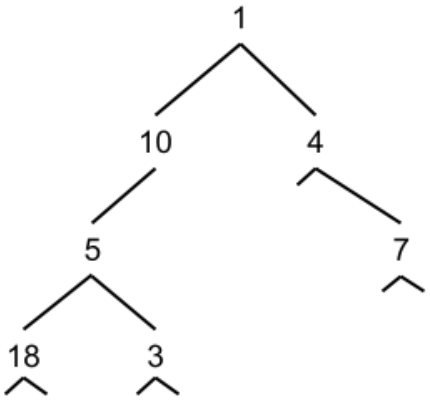
- `String skip(int n)`: return a string containing *all except* the first *n* characters of this string. For example, if `str = "example"`, then `str.skip(2)` would return `"ample"`.

Question 8 (advanced): Convert an unordered binary tree to a BST [3 points]

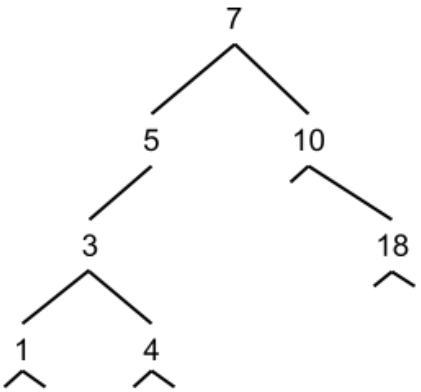
Note: this question is worth **3 advanced points.**

Implement an algorithm that converts an unordered binary tree into a binary search tree (BST), *in place*, and 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:



The node values are comparable (with constant time comparison function). You can assume the following class definition for trees and nodes (with class `Item` for node values):

```
class BinaryTree<Item>:
    class Node:
        Node left, right
        Item value
    Node root
    void convertToBST():
        ...this should be implemented…
    (...possible auxiliary private methods that you want to use...)
```

You can use any standard libraries, or the course API. ***You are allowed to create any kind of intermediate structure***, but the final tree should be changed in place (as can be seen by the signature of the method `convertToBST`).

To be awarded full credits, your solution must have optimal asymptotic time complexity in the number of tree nodes *n*. That is, there should not be another solution with better asymptotic complexity.

Part A

Show your implementation.

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:

```
class BinaryTree<Item>:

    void convertToBST():
        Item[] arr = new Item[treeSize()]
        collectNodes(root, 0, arr)
        Arrays.sort(arr)
        fillNodesFromArray(root, 0, arr)

    int collectNodes(Node node, int i, Item[] arr):
        if node is NULL:
            return i
        j = collectNodes(node.left, i, arr)
        arr[j] = node.value
        k = collectNodes(node.right, j+1, arr)
        return k

    int fillNodesFromArray(Node node, int i, Item[] arr):
        if node is NULL:
            return i
        j = fillNodesFromArray(node.left, i, arr)
        node.value = arr[j]
        k = fillNodesFromArray(node.right, j+1, arr)
        return k
```

Part B

What is the asymptotic complexity of your implementation in the number of tree nodes *n*? Justify your answer.

In-order traversal is linear in the size of the tree, so the dominating part is the sorting of the nodes.

Therefore, the complexity is O(n log n)