Algorithms and Data Structures — Graded Assignment n. 2
A. Carzaniga — May 4, 2021

Due date: *Thursday, May 13, 2021 at 22:00*

Instructions

- This is an individual assignment. You must write your code and documentation on your own. *Always acknowledge any and all sources you might use.*
- Write and submit source files with the exact names specified in each exercise. Do not submit any file, folder, or archive, other than what is required.
- You may only use the following, limited subset of the Python 3 language and libraries.
  - You may only use the built-in numeric types (e.g., `int`) and sequence types (e.g., arrays).
  - With arrays or other sequence types, you may only use the following operations:
    - direct access to an element by index, as in `return A[7]` or `A[i+1] = A[i]`
    - append an element, as in `A.append(10)`
    - delete the last element, as in `A.pop()` or `del A[len(A)-1]`
    - read the length, as in `n = len(A)`
  - You may use the `range` function, typically in a for-loop, as in `for i in range(10)`
  - You may not use any library or external function other than the ones listed above.

►**Exercise 1.** In a source file `ex1.py` write a Python function `bst_balance(t)` that takes the root of a binary search tree t and, using only rotations, balances t, returning the new root node. Balancing means returning a tree of minimal height. Again, you may not use any auxiliary data structure, and you must operate on the tree only by means of rotation operations. As a source-code comment, analyze the complexity of `bst_balance(t)`. *(40)*

Your implementation must use the following definition of a binary search tree node, and of the left and right rotation algorithms:

```
class node:
    def __init__(self,k):
        self.key = k
        self.left = None
        self.right = None

def right_rotate (t):
    assert t != None and t.left != None
    r = t.left
    t.left = r.right
    r.right = t
    return r

def left_rotate (t):
    assert t != None and t.right != None
    r = t.right
    t.right = r.left
    r.left = t
    return r
```

Hint: start by turning the input tree into a long single branch (left-to-right or right-to-left) and then turn that into a balanced BST.

►**Exercise 2.** In a source file `ex2.py` write a Python function `print_bst_by_level(t)` that takes the root of a binary search tree t and prints keys of the tree in a series of lines such that line ℓ contains the keys at depth ℓ, ordered from left (minimum) to right (maximum). *(30)*

Hint: use an auxiliary queue or list to explore the nodes of the tree level-by-level. It would in fact be easy to implement a queue using a linked list.

►**Exercise 3.** Consider the following algorithm Algo-X(A, B) that takes two arrays, A and B, of numbers.

```
Algo-X(A, B)
if Algo-Y(A, B) and Algo-Y(B, A)
      return TRUE
else return FALSE
```

```
Algo-Y(A, B)
X = array of A.length values all equal to 0
for i = 1 to B.length
      j = 1
      f = 0
      while j ≤ A.length and f == 0
            if X[j] == 0 and A[j] == −B[i]:
                  X[j] = 1
                  f = 1
            else j = j + 1
      if f == 0
            return FALSE
return TRUE
```

Answer the following questions in a PDF document called `ex3.pdf`:

Question 1: Explain what Algo-X does. Do not simply paraphrase the code. Instead, explain the high level semantics, independent of the code. *(10)*

Question 2: Analyze the complexity of Algo-X in the best and worst case. Justify your answer by clearly describing a best- and worst-case input of size n, as well as the behavior of the algorithm in each case. *(10)*

Question 3: Write an algorithm called Better-Algo-X that does exactly the same thing as Algo-X, but with a strictly better complexity (worst-case). Analyze the complexity of Better-Algo-X. *(10)*

Hint: you may use a sorting algorithm without detailing its implementation.