MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CENG 111

Fall 2020

Take-Home Exam 3

REGULATIONS

Due date: 02 February 2021, Tuesday, 23:59 *(Not subject to postpone)*

Submission: Electronically. You will be submitting your program source code through a text file which you will name as `the3.py` by means of the CENGCLASS system. Details will be announced on the forum.

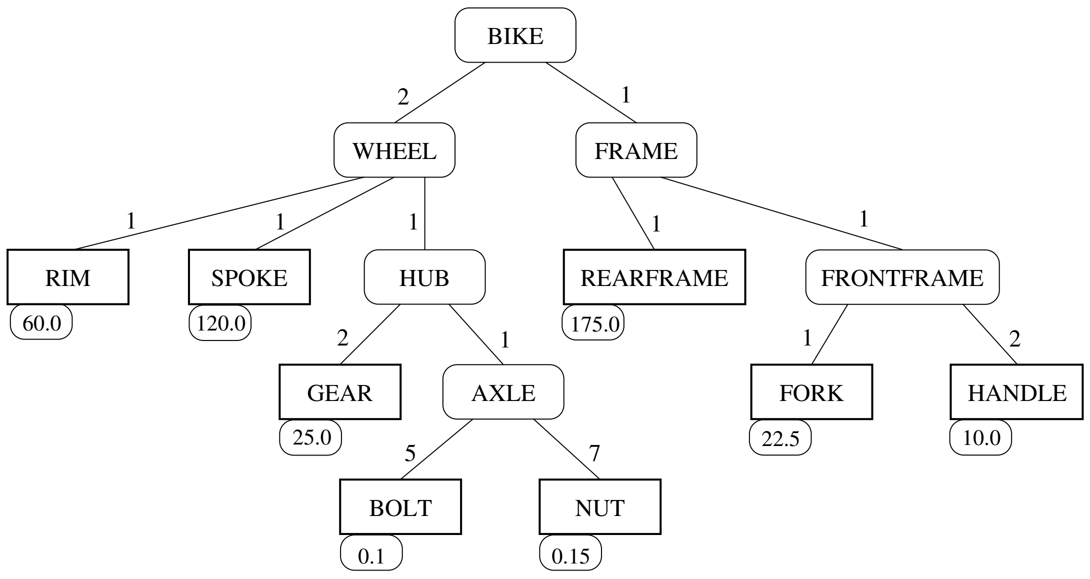
Team: There is **no** teaming up. The homework has to be done and turned in individually.

Cheating: Source(s) and Receiver(s) will receive zero and be subject to disciplinary action.

INTRODUCTION

In this exam, the purpose is to construct a parts inventory. Suppose we work in a bicycle factory, where it is necessary to keep an inventory of bicycle parts. If we want to build a bicycle, we need to know what is the total cost of the parts. Each part of the bicycle may have sub-parts; for example, each wheel has some spokes, a rim and a hub. Furthermore, the hub can consist of an axle and some gears.

Let us consider a tree-structured database that will keep the information about which parts are required to build a bicycle (or any other composite object). There are two kinds of parts that we use to build our bicycle (or generally any such composite object): assemblies and basic parts. Each assembly consists of a quantity of basic parts and (may be) a quantity of other assemblies. Since it is possible to calculate the price of any composite part, only the unit price for the basic parts are provided. Below you see such a tree:



In this case, for example, a gear has a unit price of 25.0 and you will need 2 such gears to make a hub (shown by a number on the edge).

PROBLEM

You are expected to write three functions:

(1) First function: `calculate_price(part_list)`.

- This function is for calculating the total price of the composite object.
- `part_list` is a list of lists where each list is defining a node and its children. For the example above, a possible form of the list is:

```
[
  ["bike", (2, "wheel"), (1, "frame")],
  ["wheel", (1, "rim"), (1, "spoke"), (1, "hub")],
  ["rim", 60.],
  ["spoke", 120.],
  ["hub", (2, "gear"), (1, "axle")],
  ["gear", 25.],
  ["axle", (5, "bolt"), (7, "nut")],
  ["bolt", 0.1],
  ["nut", 0.15],
  ["frame", (1, "rearframe"), (1, "frontframe")],
  ["rearframe", 175.],
  ["frontframe", (1, "fork"), (2, "handle")],
  ["fork", 22.5],
  ["handle", 10.]
]
```

- The ordering of the elements of the outermost list can vary. Also the order of the tuples in a sublist can vary. For example:

  ```
  ["wheel",  (1, "hub"), (1, "rim"), (1, "spoke")]
  ```

  would also be legitimate.
- This function should return a floating point number that is the total price of the assembled entity. For the example `part_list`, the output would be `680.6`.

(2) Second function: `required_parts(part_list)`.

- This function is for obtaining the required parts.
- The argument is a `part_list`, which is the same as the argument of the `calculate_price` function.
- This function should return a parts list as a list of tuples, where each tuple corresponds to one basic (not composite) part$_i$ as:
  (total_quantity$_i$, basic_part$_i$)
  There is no predefined order of the parts list.
- For the example above, the expected return value for `required_parts(part_list)` would be (the ordering can be different):

```
[(2, "rim"), (2, "spoke"), (4, "gear"), (10, "bolt"), (14, "nut"), \
        (1, "rearframe"), (1, "fork"), (2, "handle")]
```

(3) The third function: `stock_check(part_list, stock_list)`.

- This function is about stock control.
- It takes two parameters: `part_list`, which is the same as the inputs of `calculate_price` and `required_parts` functions; and, a `stock_list` as the second argument.
- `stock_list` contains the quantity of each basic_part$_k$ in stock. It is a list of 2-tuples each of which is of the form:

  (stock_quantity$_k$, basic_part$_k$)

  Here is an example:

  ```
  stock_list = [(2, "rim"), (2, "spoke"), (4, "gear"), (8, "bolt"), (12, "nut"),
                (1, "rearframe"), (1, "fork"), (1, "handle")]
  ```

- There is no predefined order for the stock list. Note that it contains all parts in the stock (not only the parts used in the assembly).
- `stock_check` is supposed to compute the quantity of basic parts which are short in stock (e.g. assembly needs a quantity of N for a basic_part$_i$ and stock can provide only M (where $M < N$): We are $N - M$ short in that basic_part$_i$).
- `stock_check` returns a short parts list, a list of tuples, where each tuple corresponds to one basic_part$_i$ that is **short** (not available) in stock:

  (basic_part$_i$, shortness_amount$_i$),

  where **shortness_amount**$_i$ is $N - M$, as explained above. There is no predefined order of the short parts list.
- If there is no shortage, the function should return an empty list.
- For the example `part_list` and `stock_list` above, the expected output is as follows:

  ```
  [("bolt", 2), ("nut", 2), ("handle", 1)]
  ```

EXAMPLE RUN

Note: Some lines are manually wrapped onto the following line for the sake of visibility.

```
>>> part_list = [["bike", (2, "wheel"), (1, "frame")],["wheel", (1, "rim"), (1, "spoke"),
                (1, "hub")],["rim", 60.],["spoke", 120.],["hub", (2, "gear"), (1, "axle")],
                ["gear", 25.],["axle", (5, "bolt"), (7, "nut")],["bolt", 0.1],["nut", 0.15],
                ["frame", (1, "rearframe"), (1, "frontframe")],["rearframe", 175.],
                ["frontframe", (1, "fork"), (2, "handle")],["fork", 22.5],["handle", 10.]]
>>> calculate_price(part_list)
680.6
>>> part_list = [["rearframe", 175.0], ["bike", (2, "wheel"), (1, "frame")],
                ["frontframe", (1, "fork"), (2, "handle")], ["hub", (2, "gear"), (1, "axle")],
                ["fork", 22.5], ["nut", 0.15], ["rim", 60.0], ["axle", (5, "bolt"), (7, "nut")],
                ["handle", 10.0], ["frame", (1, "rearframe"), (1, "frontframe")], ["bolt", 0.1],
                ["spoke", 120.0], ["gear", 25.0], ["wheel", (1, "rim"), (1, "spoke"), (1, "hub")]]
```

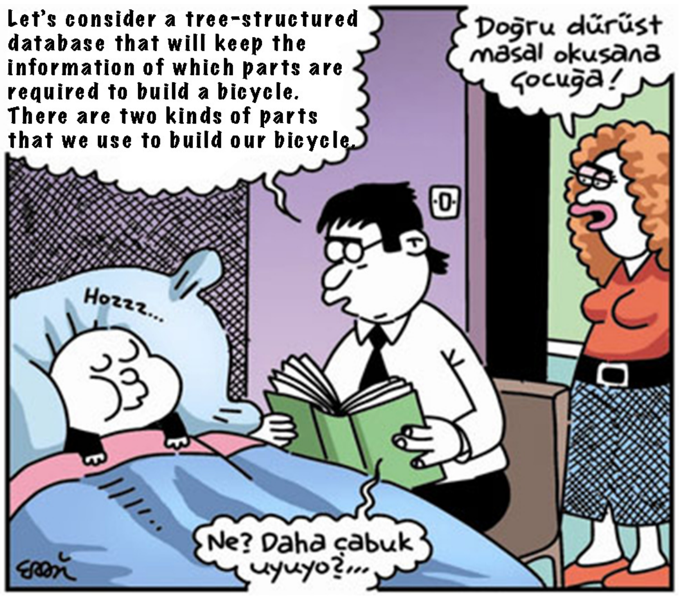
```
>>> calculate_price(part_list)
680.6
>>> required_parts(part_list)
[(2, 'rim'), (2, 'spoke'), (4, 'gear'), (10, 'bolt'), (14, 'nut'), (1, 'rearframe'), (1, 'fork'),
     (2, 'handle')]
>>> stock_list = [(2, "rim"), (2, "spoke"), (4, "gear"), (10, "bolt"), (14, "nut"), (1, "rearframe"),
      (1, "fork"), (2, "handle")]
>>> stock_check(part_list, stock_list)
[]
>>> stock_list = [(2, "rim"), (2, "spoke"), (4, "gear"), (8, "bolt"), (12, "nut"), (1, "rearframe"),
        (1, "fork"), (1, "handle")]
>>> stock_check(part_list, stock_list)
[('bolt', 2), ('nut', 2), ('handle', 1)]
>>> stock_list = [(2, "rim"), (2, "spoke"), (8, "bolt"), (12, "nut"), (1, "rearframe"),
        (1, "fork"), (5, "horn"), (1, "handle")]
>>> stock_check(part_list, stock_list)
[('gear', 4), ('bolt', 2), ('nut', 2), ('handle', 1)]
```

SPECIFICATIONS

- You can assume that the inputs are error free and comply with the input definitions above. It is guaranteed that no part is used in two different composite parts – i.e. each node can have at most one parent. Furthermore, it is guaranteed that no part X is making use of a part that uses X.
- No numerical information is provided about the depth of the tree. Your program should not be making numerical assumptions on these. Of course, due to the nature of the computer infrastructure, there are some practical limits.

HINTS and INSTRUCTIONS

- Use trees and recursion! The representation for the tree is up to you. However, you may NOT use object-oriented programming (ie. you cannot define classes).


Let's consider a tree-structured database that will keep the information of which parts are required to build a bicycle. There are two kinds of parts that we use to build our bicycle.
Doğru dürüst masal okusana çocuğa!
Hozzz...
Ne? Daha çabuk uyuyo?...