Algolab Exam

Scientific Programming Module 2
Algorithms and Data Structures

Monday 4th, September 2017

Introduction

- **Taking part to this exam erases any vote you had before, both lab and theory**
- **If you don't ship or don't pass this lab part, you lose also the theory part.**

Allowed material

There won't be any internet access. You will only be able to access:

- Sciprog Algolab worksheets (index.html)
- Alberto Montresor slides (../montresor/Montresor%20sciprog/cricca.disi.unitn.it/montresor/teaching/scientific-programming/slides/index.html)
- Stefano Teso docs (../teso/disi.unitn.it/_teso/courses/sciprog/index.html)
- Python 2.7 documentation : html (../python-docs/html/index.html) pdf (../python-docs/pdf)
  - In particular, Unittest docs (../python-docs/html/library/unittest.html)
- The course book *Problem Solving with Algorithms and Data Structures using Python* html (../pythonds/index.html) pdf (../pythonds/ProblemSolvingwithAlgorithmsandDataStructures.pdf)

Grading

- **Lab grade:** The grade of this lab part will range from 0 to 30. Total grade for the module will be given by the average with the theory part of Alberto Montresor.
- **Correct implementations**: Correct implementations with the required complexity grant you full grade.
- **Partial implementations**: Partial implementations *might* still give you a few points. If you just can't solve an exercise, try to solve it at least for some subcase (i.e. array of fixed size 2) commenting why you did so.
- **Bonus point**: One bonus point can be earned by writing stylish code. You got style if you:
  - do not infringe the Commandments (../algolab/index.html#Commandments)
  - write pythonic code (http://docs.python-guide.org/en/latest/writing/style)
  - avoid convoluted code like i.e.

    ```
    if x > 5:
        return True
    else:
        return False
    ```

    when you could write just

    ```
    return x > 5
    ```

Valid code

> **WARNING: MAKE SURE ALL EXERCISE FILES AT LEAST COMPILE !!!**
> **10 MINS BEFORE THE END OF THE EXAM I WILL ASK YOU TO DO A FINAL CLEAN UP OF THE CODE**

> **!!!!!!!!! WARNING !!!!!!!!!**
>
> **!!!!!!!!! **ONLY** IMPLEMENTATIONS OF THE PROVIDED FUNCTION SIGNATURES WILL BE EVALUATED !!!!!!!!!**

For example, if you are given to implement:

```
def f(x):
        raise Exception("TODO implement me")
```

and you ship this code:

```
def my_f(x):
    # a super fast, correct and stylish implementation

def f(x):
    raise Exception("TODO implement me")
```

We will assess only the latter one `f(x)`, and conclude it doesn't work at all :P !!!!!!!

Helper functions

Still, you are allowed to define any extra helper function you might need. If your `f(x)` implementation calls some other function you defined like `my_f` here, it is ok:

```
# Not called by f, will get ignored:
def my_g(x):
    # bla

# Called by f, will be graded:
def my_f(y,z):
    # bla

def f(x):
    my_f(x,5)
```

How to edit

To edit the files, you can use any editor of your choice:

- **Editra editor** is easy to use, you can find it under *Applications->Programming->Editra*.**
- **The Terminal** to run python can be found in *Accessories -> Terminal*
- Others could be *GEdit* (simpler), or *PyCharm* (more complex).

> **IMPORTANT: Pay close attention to the comments of the functions.**

> **WARNING: *DON'T* modify function signatures! Just provide the implementation.**

> **WARNING: *DON'T* change the existing test methods, just add new ones !!! You can add as many as you want.**

> **WARNING: *DON'T* create other files. If you still do it, they won't be evaluated.**

Debugging

If you need to print some debugging information, you are allowed to put extra `print` statements in the function bodies.

> **WARNING: even if `print` statements are allowed, be careful with prints that might break your function!**
>
> **For example, avoid stuff like this:**
> ```
> x = 0
> print 1/x
> ```

What to do

1) Download algolab-2017-09-04.zip (../algolab-2017-09-04.zip) and extract it **on your desktop**. Folder content should be like this:

```
algolab-2017-09-04
    |- FIRSTNAME-LASTNAME-ID
        |- exercise1.py
        |- exercise2.py
        |- exercise3.py
```

2) Rename `FIRSTNAME-LASTNAME-ID` folder: put your name, lastname an id number, like `john-doe-432432`

From now on, you will be editing the files in that folder. At the end of the exam, that is what will be evaluated.

3) Edit the files following the instructions in this worksheet for each exercise. Every exercise should take max 25 mins. If it takes longer, leave it and try another exercise.

1) MultiSet

You are going to implement a class called `MultiSet`, where you are only given the class skeleton, and you will need to determine which Python basic datastructures like list, set, dict (or combinations thereof) is best suited to actually hold the data.

In math a multiset (or bag) generalizes a set by allowing multiple instances of the multiset's elements.

The multiplicity of an element is the number of instances of the element in a specific multiset.

For example:

- The multiset `a, b` contains only elements `a` and `b`, each having multiplicity 1
- In multiset `a, a, b`, `a` has multiplicity 2 and `b` has multiplicity 1
- In multiset `a, a, a, b, b, b`, `a` and `b` both have multiplicity 3

NOTE: order of insertion does not matter, so `a, a, b` and `a, b, a` are the same multiset, where `a` has multiplicity 2 and `b` has multiplicity 1.

1.0) run `EnvWorkingTest`

Now open the file `exercise1.py` and check your environment is working fine, by trying to run the test `EnvWorkingTest`: it should always pass, if it doesn't, tell your instructor.

Notice that `exercise1` is followed by a dot and test class name: `.EnvWorkingTest`

```
python -m unittest exercise1.EnvWorkingTest
```

1.1) `__init__`, `add` and `get`

Now implement *all* of the following methods: `__init__`, `add` and `get`:

```
def __init__(self):
        """ Initializes the MultiSet as empty."""
        raise Exception("TODO IMPLEMENT ME !!!")

    def add(self, el):
        """ Adds one instance of element el to the multiset

            NOTE: MUST work in O(1)
        """
        raise Exception("TODO IMPLEMENT ME !!!")

    def get(self, el):
        """ Returns the multiplicity of element el in the multiset.

            If no instance of el is present, return 0.

            NOTE: MUST work in O(1)
        """
        raise Exception("TODO IMPLEMENT ME !!!")
```

Testing

Once done, running this will run only the tests in `AddGetTest` class and hopefully they will pass.

Notice that `exercise1` is followed by a dot and test class name `.AddGetTest` :

```
python -m unittest exercise1.AddGetTest
```

1.2) `removen`

Implement the following `removen` method:

```
def removen(self, el, n):
        """ Removes n instances of element el from the multiset (that is, reduces el mul
tiplicity by n)

            If n is negative, raises ValueError.
            If n represents a multiplicity bigger than the current multiplicity, raises
LookupError

            NOTE: multiset multiplicities are never negative
            NOTE: MUST work in O(1)
        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise1.RemovenTest`

2) UnorderedList

Start editing file `exercise2.py`, which contains a simplified versioned of the `UnorderedList` we saw in the labs.

2.1) `find_couple`

Implement following `find_couple` method.

```
def find_couple(self,a,b):
        """ Search the list for the first two consecutive elements having data equal to
            provided a and b, respectively. If such elements are found, the position
            of the first one is returned, otherwise raises LookupError.

            - MUST run in O(n), where n is the size of the list.
            - Returned index start from 0 included

        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise2.FindCoupleTest`

2.2) `swap`

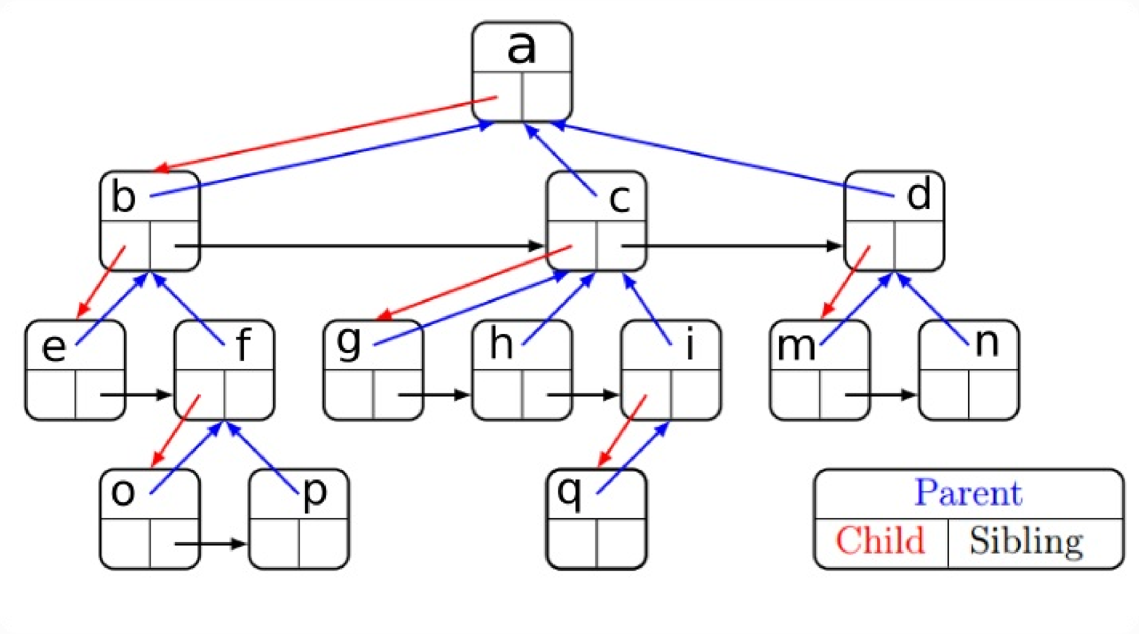
Implement the method `swap`:

```
def swap (self, i, j):
        """
            Swap the data of nodes at index i and j. Indeces start from 0 included.
            If any of the indeces is out of bounds, rises IndexError.

            NOTE: You MUST implement this function with a single scan of the list.

        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise2.NorepTest`

3) GenericTree

Start editing file `exercise3.py`, which contains a simplified versioned of the `GenericTree` we saw in the labs.



3.1) mirror

Implement the method `mirror`:

```
def mirror(self):
        """ Modifies this tree by mirroring it, that is, reverses the order
            of all children of this node and of all its descendants

            - MUST work in O(n) where n is the number of nodes
            - MUST change the order of nodes, NOT the data (so don't touch the data !)
            - DON'T create new nodes
            - It is acceptable to use a recursive method.


            Example:

            a     <-     Becomes:     a
            |-b                       |-i
            | |-c                     |-e
            | \-d                     | |-h
            |-e                       | |-g
            | |-f                     | \-f
            | |-g                     |-b
            | \-h                       |-d
            \-i                         \-c


        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise3.MirrorTest`

3.2) clone

Implement the method `clone`:

```
def clone(self):
        """ Clones this tree, by returning an *entirely* new tree which is an
            exact copy of this tree (so returned node and *all* its descendants must be
new).

            - MUST run in O(n) where n is the number of nodes
            - a recursive method is acceptable.
        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise3.CloneTest`