Lab Midterm

Scientific Programming Algolab

Friday 13th Jan 2017

Introduction

- This midterm makes sense only if you passed successfully the theory midterm by Alberto Montresor. If you didn't, you can still stay and do the midterm as exercise, but it won't be evaluated.
- If you don't ship or you don't pass this midterm, you lose also the theory midterm.
- Log into your computer in *exam mode*, it should start Ubuntu
- To edit the files, you can use any editor of your choice. *Editra* seems easy to use, you can find it under Applications->Programming->Editra. Other could be GEdit, or PyCharm (more complex).

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

- The grade of this midterm will range from 0 to 30. Total grade for the module will be given by the average with the theory midterm of Alberto Montresor.
- Correct implementations with the required complexity grant you full grade.
- 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. Doing so *might* still grant you a few points.
- One extra 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
```

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

For example, if you are given to implement:

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

and you ship this code:

```
def cool_fun_non_working_trial(x):
    # do some absurdity

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

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

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

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

```
def my_helper(y,z):
    # do something useful

def cool_fun(x):
    my_helper(x,5)

# this will get ignored:
def some_trial(x):
    # do some absurdity
```

What to do

In `/usr/local` you should find somewhere a file named `sciprog-midterm-17-01-13.zip`. Download it and extract it in a new folder on your desktop. The content should be like this:

```
|- docs
|- algolab-17-01-13
    |- exercise1.py
    |- exercise2.py
    |- exercise3.py
    |- exercise4.py
```

Under `docs/` folder you will find the slides and Python documentation.

2) Now, take the folder `algolab-17-01-13` and copy it to `/var/exam`.

Rename it to `algolab-17-01-13-midterm-FIRSTNAME-LASTNAME-IDNUMBER` like algolab-17-01-13-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.

> **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.**

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

> **IMPORTANT: 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, i.e. avoid stuff like this: `print 1/0`**

3) Every exercise should take max 25 mins. If it takes longer, leave it and try another exercise.

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

Exercises

1) quick sort

Quick sort is a widely used sorting algorithm and in this exercise you will implement it following the pseudo code.

> **IMPORTANT: Array `A` in the pseudo code has indexes starting from zero *included***

> **IMPORTANT: The functions `pivot` and `quicksort` operate an a subarray that goes from indeces `first` *included* and `last` *included***

Start editing the file `exercise1.py`:

1.1) swap

```
swap(ITEM[] A, int x, int y)
    int temp = A[x]
    A[x] = A[y]
    A[y] = temp
```

Implement `swap`:

```
def swap(A, x, y):
    """
        In the array A, swaps the elements at indeces x and y.
    """
    raise Exception("TODO IMPLEMENT ME!")
```

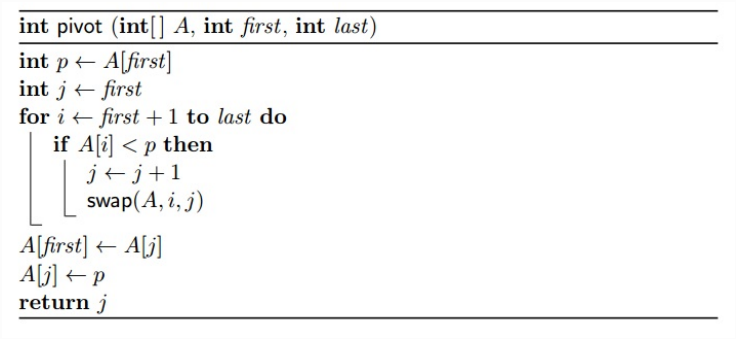
Once done, running this will run only the tests in `SwapTest` class and hopefully they will pass.

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

```
python -m unittest exercise1.SwapTest
```

1.2) pivot

Implement `pivot` method:



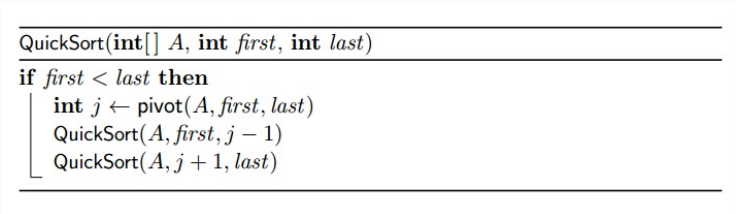

```
def pivot(A, first, last):
    """ Modifies in-place the slice of the array A with indeces between first included
        and last included. Returns the new pivot index.

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

You can run tests only for `pivot` with this command:

```
python -m unittest exercise1.PivotTest
```

1.3) implement quicksort and qs

Implement `quicksort` and `qs` method:




```
def quicksort(A, first, last):
    """
        Sorts in-place the slice of the array A with indeces between first included
        and last included.
    """
    raise Exception("TODO IMPLEMENT ME !")

def qs(A):
    """
        Sorts in-place the array A by calling quicksort function on the full array.
    """
    raise Exception("TODO IMPLEMENT ME !")
```

You can run tests only for both `quicksort` and `qs` with this command:

```
python -m unittest exercise1.QuicksortTest
```

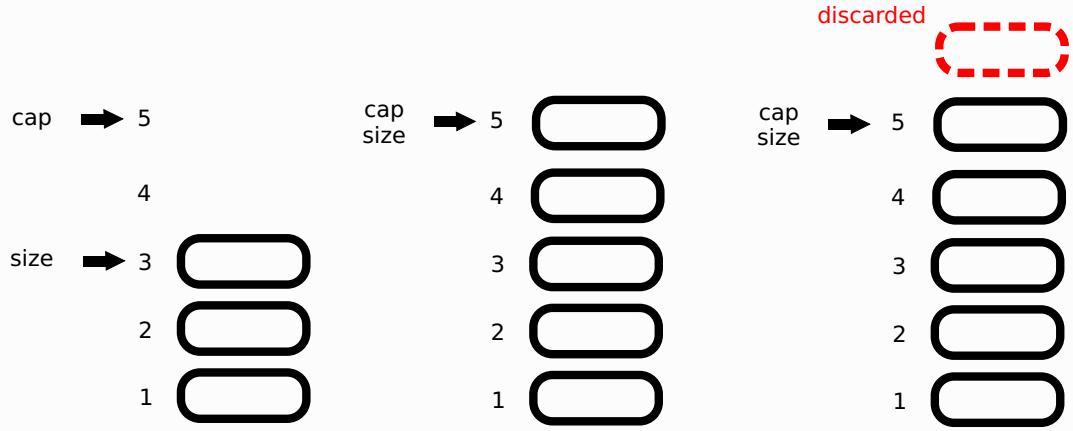
2) CappedStack

In class you implemented a `CappedStack` that had a fixed `cap`. In this exercise, you will implement a way to change the `cap` *after* that a stack instance has been created.

NOTE: On some condition, you will be requested to raise `IndexError`. To do so, just write:

```
raise IndexError("Error!")
```

Proceed with the following point and start editing the file `exercise2.py`



2.1) peekn

Implement the `peekn` method:

```
def peekn(self, n):
        """
            Returns a list with the n top elements, in the order in which they
            were pushed. For example, if the stack is the following:

                e
                d
                c
                b
                a

            peekn(3) will return the list ['c','d','e']

            If there aren't enough element to peek, raises IndexError
            If n is negative, raises an IndexError

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

2.2) popn

Implement the `popn` method:

```
    def popn(self, n):
            """ Pops the top n elements, and return them as a list, in the order in
                which they where pushed. For example, with the following stack:

                    e
                    d
                    c
                    b
                    a

                popn(3)

                will give back ['c','d','e'], and stack will become:

                    b
                    a

                If there aren't enough element to pop, raises an IndexError
                If n is negative, raises an IndexError
            """
```

2.3) set_cap

Implement the `set_cap` method:

```
    def set_cap(self, cap):
            """ Sets the cap value to the provided cap.

                If the cap is less then the stack size, all the elements above
                the cap are removed from the stack.

                If cap < 1, raises an IndexError
                Does *not* return anything!

                For example, with the following stack, and cap at position 7:

                cap ->  7
                        6
                        5  e
                        4  d
                        3  c
                        2  b
                        1  a


                calling method set_cap(3) will change the stack to this:

                cap ->  3  c
                        2  b
                        1  a

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

3) UnorderedList

Let's work on `UnorderedList`, which is a monodirectional linked list.

Start editing the file `exercise3.py`

3.1) occurrences

Implement this method:

```
    def occurrences(self, item):
            """
                Returns the number of occurrences of item in the list.
            """
```

Examples:

In [5]:

```
from exercise3_solution import *

ul = UnorderedList()
ul.add('a')
ul.add('c')
ul.add('b')
ul.add('a')
print ul
```

```
UnorderedList: a,b,c,a
```

In [6]:

```
print ul.occurrences('a')
```

```
2
```

In [7]:

```
print ul.occurrences('c')
```

```
1
```

In [8]:

```
print ul.occurrences('z')
```

```
0
```

3.2) shrink

Implement this method in `UnorderedList` class:

```
    def shrink(self):
            """
                Removes from this UnorderedList all nodes at odd indeces (1, 3, 5, ...),
                supposing that the first node has index zero, the second node
                has index one, and so on.

                So if the UnorderedList is
                    'a','b','c','d','e'
                a call to shrink will transform the UnorderedList into
                    'a','c','e'

                Must execute in O(n) where 'n' is the length of the list.
                Does *not* return anything.
            """
            raise Exception("TODO IMPLEMENT ME!")
```

In [9]:

```
ul = UnorderedList()
ul.add('e')
ul.add('d')
ul.add('c')
ul.add('b')
ul.add('a')
print ul
```

```
UnorderedList: a,b,c,d,e
```

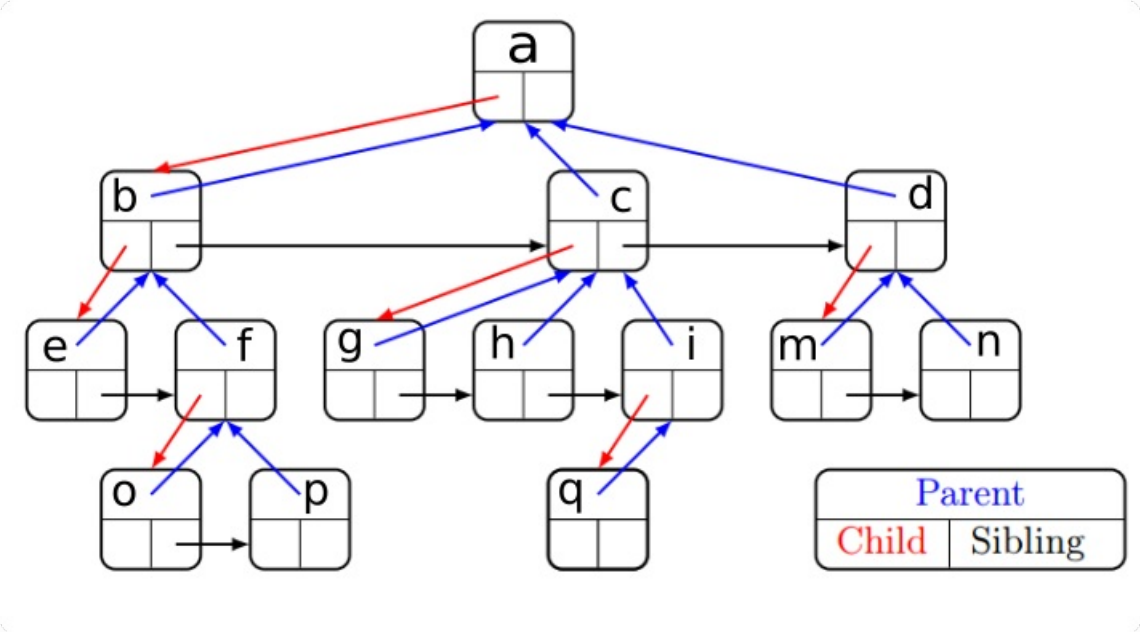
In [10]:

```
ul.shrink()
print ul
```

```
UnorderedList: a,c,e
```

4) GenericTree

In these exercises with a tree, you will be visiting a generic tree in various ways.



4.1) zig

The method `zig` must return as output a list of `data` of the root and all the nodes in the chain of `child` attributes. Basically, you just have to follow the red lines and gather `data` in a list, until there are no more red lines to follow. For example, in the labeled tree in the image, these would be the results of calling `zig` on various nodes:

- From a: `['a','b', 'e']`
- From b: `['b', 'e']`
- From c: `['c', 'g']`
- From h: `['h']`
- From q: `['h']`

4.2) zag

This function is quite similar to `zig`, but this time it gathers data going right, along the `sibling` arrows. For example, in the labeled tree in the image, these would be the results of calling `zag` on various nodes:

- From a : `['a']`
- From b : `['b', 'c', 'd']`
- From o : `['o', 'p']`

4.3) zigzag

If you arrived so far and some unit test for previous exercises (1,2,3,4) are still failing, it's probably more convenient to try to fix it. On the other hand, if everything is working fine, you should proceed with this slightly challanging exercise!

As you are surely thinking, `zig` and `zag` alone are boring. So let's mix the concepts, and go zigzaging. This time you will write a function `zigzag`, that first zigs collecting `data` along the child vertical red chain as much as it can. Then, if the last node links to at least a sibling, the method continues to collect data along the siblings horizontal chain as much as it can. At this point, if it finds a child, it goes zigging again along the child vertical red chain as much as it can, and then horizontal zaging, and so on. It continues zig-zaging like this until it reaches a node that has no child nor sibling: when this happens returns the list of data found so far. For example, these would be the results of calling `zigzag` on various nodes:

- From a: `['a', 'b', 'e', 'f', 'o']`
- From c: `['c', 'g', 'h', 'i', 'q']` NOTE: if node h had a child `z`, the process would still proceed to `i`
- From d: `['d', 'm', 'n']`
- From o: `['o', 'p']`
- From n: `['n']`