In[1]:

Algolab Exam

Scientific Programming Module 2
Algorithms and Data Structures

Thursday 8th, June 2017

Introduction

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

- 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*. Others could be *GEdit* (simpler), 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 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 with the required complexity grant you full grade.
- 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.
- 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
```

> **!!!!!!!!! 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/esame (/usr/local/esame) you should find a file named `algolab-17-06-08.zip`. Download it and extract it on your desktop. The content should be like this:

```
algolab-17-06-08
    |- FIRSTNAME-LASTNAME-ID
        |- exercise1.py
        |- exercise2.py
        |- exercise3.py
```

2) Check this folder also shows under `/var/exam`. TODO

3) 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.

4) 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 DO A FINAL CLEAN UP OF THE CODE**

1) SortedStack

You are given a class `SortedStack` that models a simple stack. This stack is similar to the `CappedStack` you already saw in class, the differences being:

- *it can only contain integers*, trying to put other type of values will raise a `ValueError`
- *integers must be inserted sorted in the stack*, either ascending or descending
- there is no cap

```
   Example:

        Ascending:       Descending

           8                  3
           5                  5
           3                  8
```

In [5]:

```
from exercise1_solution import *
```

To create a `SortedStack` sorted in ascending order, just call it passing `True`:

In [6]:

```
s = SortedStack(True)
print s
```

```
SortedStack (ascending):   elements=[]
```

In [7]:

```
s.push(5)
print s
```

```
SortedStack (ascending):   elements=[5]
```

In [8]:

```
s.push(7)
print s
```

```
SortedStack (ascending):   elements=[5, 7]
```

In [9]:

```
print s.pop()
```

```
7
```

In [10]:

```
print s
```

```
SortedStack (ascending):   elements=[5]
```

In [11]:

```
print s.pop()
```

```
5
```

In [12]:

```
print s
```

```
SortedStack (ascending):   elements=[]
```

For descending order, pass `False` when you create it:

In [13]:

```
sd = SortedStack(False)
sd.push(7)
sd.push(5)
sd.push(4)
print(sd)
```

```
SortedStack (descending):   elements=[7, 5, 4]
```

1.0) test `SortedStack`

Now open the file `exercise1.py` andcheck your environment is working fine, by trying to run the tests only for `SortedStackTest`, which tests already implemented methods like `pop`, `push`, etc ... : these tests should all pass, if they don't, tell your instructor.

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

```
python -m unittest exercise1.SortedStackTest
```

1.1) `transfer`

Now implement the `transfer` function. **NOTE**: function is external to class `SortedStack`.

```
def transfer(s):
    """ Takes as input a SortedStack s (either ascending or descending) and
        returns a new SortedStack with the same elements of s, but in reverse order.
        At the end of the call s will be empty.

        Example:

            s       result

            2         5
            3         3
            5         2
    """
    raise Exception("TODO IMPLEMENT ME !!")
```

Testing

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

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

```
python -m unittest exercise1.TransferTest
```

1.2) `merge`

Implement following `merge` function. **NOTE**: function is external to class `SortedStack`.

```
def merge(s1,s2):
    """ Takes as input two SortedStacks having both ascending order,
       and returns a new SortedStack sorted in descending order, which will be the sorte
d merge
       of the two input stacks. MUST run in O(n1 + n2) time, where n1 and n2 are s1 and
s2 sizes.

       If input stacks are not both ascending, raises ValueError.
       At the end of the call the input stacks will be empty.


       Example:

       s1 (asc)   s2 (asc)      result (desc)

          5          7            2
          4          3            3
          2                       4
                                  5
                                  7

    """

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

Testing: `python -m unittest exercise1.MergeTest`

2) UnorderedList

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

In [17]:

```
from exercise2_solution import *
```

2.1) `panino`

Implement following `panino` function. **NOTE**: the function is external to class `UnorderedList`.

```
def panino(lst):
    """ Returns a new UnorderedList having double the nodes of provided lst
        First nodes will have same elements of lst, following nodes will
        have the same elements but in reversed order.

        For example:

            >>> panino(['a'])
            UnorderedList: a,a

            >>> panino(['a','b'])
            UnorderedList: a,b,b,a

            >>> panino(['a','c','b'])
            UnorderedList: a,c,b,b,c,a

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

Testing: `python -m unittest exercise2.PaninoTest`

2.2) `norep`

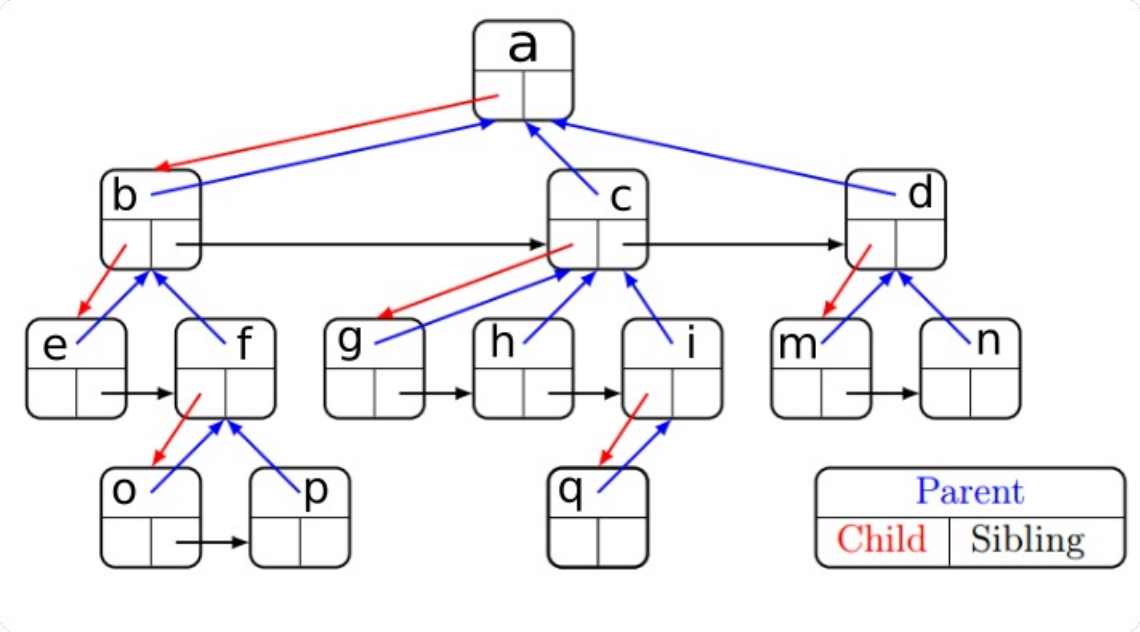
Implement the method `norep`:

```
def norep(self):
        """ Removes all the consecutive repetitions from the list.
            Must perform in O(n), where n is the list size.

            For example, after calling norep:

            'a','a','b','c','c','c'   will become  'a','b','c'

            'a','a','b','a'   will become   'a','b','a'

        """

        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.



Start editing file `exercise3.py`

In [20]:

```
from exercise3_solution import *
```

3.1) ancestors

Implement the method `ancestors`:

```
    def ancestors(self):
            """ Return the ancestors up until the root as a Python list.
                First item in the list will be the parent of this node.

                NOTE: this function return the *nodes*, not the data.
            """

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

Testing: `python -m unittest exercise3.AncestorsTest`

Examples:

```
- ancestors of p:  f, b, a
- ancestors of h:  c, a
- ancestors of a:  empty list
```

3.2) leftmost

Implement the method `leftmost`:

```
    def leftmost(self):
            """
                Return the leftmost node of the root of this node. To find it, from
                current node you need to reach the root of the tree and then from
                the root you need to follow the _child chain until a node with no children i
    s found.

                If self is already the root, or the root has no child, raises LookupError.

                NOTE: this function return a *node*, not the data.

            """

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

Testing: `python -m unittest exercise3.LeftmostTest`

Examples:

```
- leftmost of p:  e
- leftmost of h:  e
- leftmost of e:  raise LookupError
```

3.3) common_ancestor

Implement the method `common_ancestor`:

```
    def common_ancestor(self, gt2):
            """ Return the first common ancestor of current node and the provided gt2 node
                If gt2 is not a node of the same tree, raises LookupError

                NOTE: this function returns a *node*, not the data.

                Ideally, this method should perform in O(h) where h is the height of the tre
    e.
                (Hint: you should use a Python Set). If you can't figure out how to make it
                that fast, try to make it at worst O(h^2)

            """

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

Testing: `python -m unittest exercise3.CommonAncestorTest`

Examples:

```
- common ancestor of g and i: c
- common_ancestor of g and q: c
- common_ancestor of e and d: a
```