Chapter 3: Trees

Tree theory

See Alberto Montresor theory here: http://disi.unitn.it/~montreso/sp/slides/05-alberi.pdf (http://disi.unitn.it/~montreso/sp/slides/05-alberi.pdf)

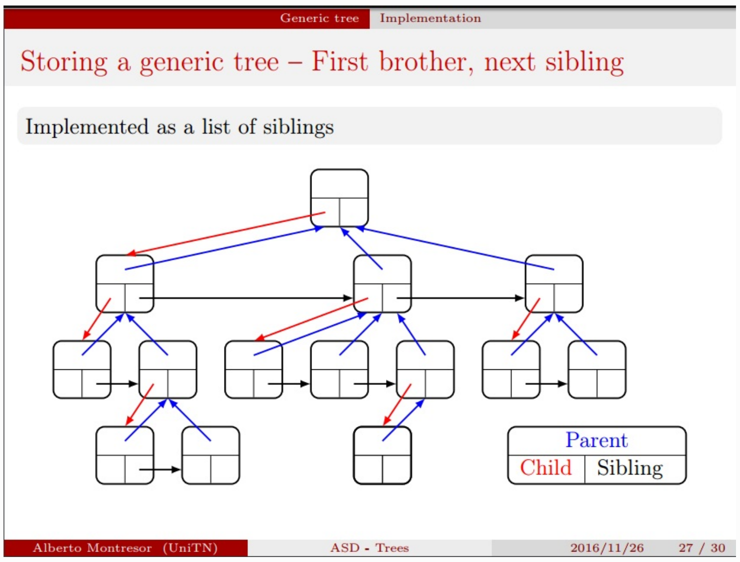
See Trees on the book (https://interactivepython.org/runestone/static/pythonds/Trees/toctree.html)

In particular, see :

- Vocabulary and definitions (https://interactivepython.org/runestone/static/pythonds/Trees/VocabularyandDefinitions.html)

`GenericTree` theory

See Alberto Montresor theory here (NOTE: currently they are being reworked): http://disi.unitn.it/~montreso/sp/slides/05-alberi.pdf (http://disi.unitn.it/~montreso/sp/slides/05-alberi.pdf) (slide 27 and following ones)


Generic tree | Implementation

Storing a generic tree – First brother, next sibling

Implemented as a list of siblings

Alberto Montresor (UniTN) | ASD - Trees | 2016/11/26 | 27 / 30

In this worksheet we are going to provide an implementation of a `GenericTree` class:

- Differently from the `UnorderedList`, which had actually two classes `Node` and `UnorderedList` that was pointing to the first node, in this case we just have one `GenericTree` class. So to grow a tree like the above one in the picture, for each of the boxes that you see we will need to create one instance of `GenericTree` and link it to the other instances.
- Ordinary simple trees just hold pointers to the children. In this case, we have an enriched tree which holds ponters also to up the *parent* and on the right to the *siblings*. Whenever we are going to manipulate the tree, we need to take good care of updating these pointers.

> **ROOT NODE: In this context, we call a node *root* if has no incoming edges *and* it has no parent nor sibling**

> **DETACHING A NODE: In this context, when we *detach* a node from a tree, the node becomes the *root* of a new tree, which means it will have no link anymore with the tree it was in.**

0) Code skeleton

You will implement the `GenericTree` class. First off, download the Python skeleton (trees.py) to modify. Solutions are in a separate file (trees_solution.py).

1) Building trees

Let's learn how to build `GenericTree`

> **IMPORTANT: All methods and functions in section 1) are already provided and you don't need to implement them !**

1.1) Pointers

A `GenericTree` class holds 3 pointers that link it to the other nodes: `_child`, `_sibling` and `_parent`. It also holds a value `data` which is provided by the user to store arbitrary data (could be ints, strings, lists, even other trees, we don't care):

```
class GenericTree:

    def __init__(self, data):
        self._data = data
        self._child = None
        self._sibling = None
        self._parent = None
```

To create a tree of one node, just call the constructor passing whatever you want like this:

```
tblah = GenericTree("blah")

tn = GenericTree(5)
```

Note that with the provided constructor you can't pass children.

1.2) Building with `insert_child`

To grow a `GenericTree`, as basic building block you will have to implement `insert_child`:

```
def insert_child(self, new_child):
        """ Inserts new_child at the beginning of the children sequence. """
```

You can call it like this:

```
>>> ta = GenericTree('a')
>>> print ta
a            # 'a' is the root

>>> tb = GenericTree('b')
>>> ta.insert_child(tb)
>>> print ta
a            # 'a' is the root
\-b          # 'b' is the child . The '\' means just that it is
             #  also the last child of the siblings sequence


>>> tc = GenericTree('c')
>>> ta.insert_child(tc)
>>> print ta
a           # 'a' is the root
|-c         # 'c' is inserted as the first child (would be shown on the left in the grap
h image)
\-b         # 'b' is now the next sibling of c  The '\' means just that it
            #  is also the last child of the siblings sequence

>>> td = GenericTree('d')
>>> tc.insert_child(td)
>>> print ta
a           # 'a' is the root
|-c         # 'c' is the first child of 'a'
| \-d       # 'd' is the first child of 'c'
\-b         # 'b' is the next sibling of c
```

1.3) Building with gt

If you need to test your data structure, we provide you with this handy function `gt` that allows to easily construct trees from other trees:

> **WARNING: DO NOT USE `gt` inside your implementation code !!!! `gt` is just meant for testing.**

```
def gt(data, children=[])
    """ Returns a GenericTree of which the root node is filled with provided data
        and children. Children must be instances of GenericTree.
    """
```

NOTE: this function is *not* a class method, you can directly invoke it like this:

```
>>> print gt('a')
    a

    >>> print gt('a', gt('b'), gt('c'))
    a
    |-b
    \-c
```

2) Implement missing methods

Start implementing `insert_child`, make sure the tests for it pass, and then implement the other methods. Don't worry if `insert_sibling` and `insert_siblings` test always fail, to fix them see next section.

3) Implement missing tests

3.1) Implement the missing tests `test_insert_sibling` and `test_insert_siblings`. To do it,feel free to use `gt`, `assertTreeEquals`, `assertRoot` and whatever other function you can find in the code. If possible, try to implement a test method for each case you might have

> **Is the function to test expected to raise an Exception in some circumstance?**

3.2) Once you're done and your new tests pass, save a copy of your work

3.3) Work in group and add to your test class the test implementation of somebody else, taking care of renaming test methods so to avoid name clashes. Run the tests and check if you agree with your .

3.4) Try to implement on you own tests for other methods, like `detach`. Check they pass and then exchange tests with your collegues.

GenericTree Solution

Solutions are in a separate file (trees_solution.py).

In [6]:

```
from trees import *
```

In [7]:

```
from trees_solution import *
algolab.run(GenericTreeTest)
```

```
...................
----------------------------------------------------------------------
Ran 19 tests in 0.019s

OK
```

In [8]:

```

```