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)

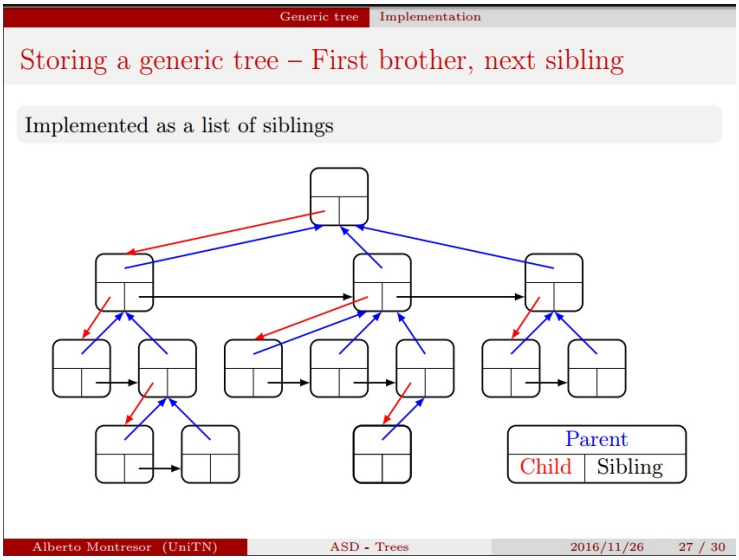
In particular, see :

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

GenericTree

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)



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.**

GenericTree exercises

You will implement the `GenericTree` class. You can start by copying the code skeleton and unit tests, then proceed reading the following.

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.

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
```

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
```

1) 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.

2) Implement missing tests

2.1) 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?**

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

2.4) 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 .

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

GenericTree Code Skeleton

In [5]:

```
import unittest

class GenericTree:
    """ A tree in which each node can have any number of children.

        Each node is linked to its parent and to its immediate sibling on the right
    """

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

```
        self._sibling = None
        self._parent = None

    def data(self):
        return self._data

    def child(self):
        return self._child

    def sibling(self):
        return self._sibling

    def parent(self):
        return self._parent

    def is_root(self):
        """ Return True if the node is a root of a tree, False otherwise

            A node is a root whenever it has no parent nor siblings.
        """
        return self._parent == None and self._sibling == None

    def is_subtree(self):
        """ Returns True if the node is a subtree of another tree

            A subtree always has a parent
        """
        return self._parent != None


    def children(self):
        """ Returns the children as a Python list """

        ret = []
        current = self._child
        while current != None:
            ret.append(current)
            current = current._sibling
        return ret

    def __str__(self):
        """ Returns a pretty string of the tree """

        def str_branches(node, branches):
            """ Returns a string with the tree pretty printed.

                branches: a list of characters representing the parent branches. Characters ca
n be either ` ` or '|'
            """
            strings = [str(node._data)]
            current = node._child
            while (current != None):
                if current._sibling == None:
                    # there are better end characters but let's not upset
                    # stupid Python with unicode problems
                    joint = '\-'
                else:
                    joint = '|-'


                strings.append('\n')
                for b in branches:
                     strings.append(b)
                strings.append(joint)
                if current._sibling == None:
                    branches.append('  ')
                else:
                    branches.append('| ')
                strings.append(str_branches(current, branches))
                branches.pop()
                current = current._sibling
```

```
            return "".join(strings)

        return str_branches(self, [])


    def insert_child(self, new_child):
        """ Inserts new_child at the beginning of the children sequence. """
        raise Exception("TODO Implement me !" )



    def insert_children(self, new_children):
        """ Takes a list of children and inserts them at the beginning of the current children
 sequence,

            NOTE: in the new sequence new_children appear in the order they are passed to the
function!


            For example:
                >>> t = gt('a', gt('b'), gt('c))
                >>> print t
                a
                |-b
                \-c

                >>>  t.insert_children([gt('d'), gt('e')])
                >>> print t
                a
                |-d
                |-e
                |-b
                \-c
        """
        raise Exception("TODO Implement me !" )

    def insert_sibling(self, new_sibling):
        """ Inserts new_sibling as the immediate next sibling """
        raise Exception("TODO Implement me !" )

    def insert_siblings(self, new_siblings):
        """ Inserts new_siblings at the beginning of the siblings sequence,
            in the same order as they are passed.

            For example:

                >>> bt =  gt('b')
                >>> t = gt('a', bt , gt('c))
                >>> print t
                a
                |-b
                \-c

                >>>  bt.insert_children([gt('d'), gt('e')])
                >>> print t
                a
                |-b
                |-d
                |-e
                \-c
        """
        raise Exception("TODO Implement me !" )

    def has_child(self):
        """ Returns True if this node has a child, False otherwise """
        return self._child != None

    def detach_child(self):
        """ Detaches the first child.

            if there is no child, raises an Exception
        """
```

```
        """

        raise Exception("TODO Implement me !" )


    def detach_sibling(self):
        """ Detaches the first sibling.

            If there is no sibling, raises an Exception
        """

        raise Exception("TODO Implement me !" )

    def detach(self, data):
        """ Detaches the first child that holds the provided data   """

        raise Exception("TODO Implement me !" )


def gt(*args):
    """ Shorthand function that returns a GenericTree containing the provided
        data and children. First parameter is the data, the following ones are the children.

        Usage examples:

        print gt('a')
        >>> a

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

    """
    if (len(args) == 0):
        raise Exception("You need to provide at least one argument for the data!")

    data = args[0]
    children = args[1:]

    r = GenericTree(data)
    for c in reversed(children):
        r.insert_child(c)
    return r

def str_trees(t1, t2, error_row=-1):
    """ Returns a string version of the two trees side by side

        If error_row is given, the line in error is marked.
        If error_row == -1 it is ignored
    """

    s1 = str(t1)
    s2 = str(t2)

    lines1 = s1.split("\n")
    lines2 = s2.split("\n")

    max_len1 = 0
    for line in lines1:
        max_len1 = max(len(line.rstrip().decode("utf-8")), max_len1)

    max_len2 = 0
    for line in lines2:
        max_len2 = max(len(line.rstrip().decode("utf-8")), max_len2)

    strings = []

    i = 0

    while i < len(lines1) or i < len(lines2):
```

```
        if i < len(lines1):

            strings.append(lines1[i].rstrip())
            len1 = len(lines1[i].rstrip().decode("utf-8"))
        else:
            len1 = 0

        if (i < len(lines2)):
            len2 = len(lines2[i].rstrip().decode("utf-8"))

            pad_len1 = 4 + max_len1 - len1
            strings.append((" " * pad_len1) + lines2[i].rstrip())
        else:
            len2 = 0

        if (error_row == i):
            pad_len2 = 2 + max_len1 + max_len2 - len1 - len2
            strings.append((" " * pad_len2) + "<--- DIFFERENT ! ")

        strings.append("\n")

        i += 1

    return "".join(strings)


class GenericTreeTest(unittest.TestCase):

    def assertReturnNone(self, ret, function_name):
        """ Asserts method result ret equals None """
        self.assertEquals(None, ret,
                          function_name
                          + " specs say nothing about returning objects! Instead you are retur
ning " + str(ret))


    def assertTreeEqual(self, t1, t2):
        """ Asserts the trees t1 and t2 are equal """

        def rec_assert(c1, c2, row):

            if c1.data() != c2.data():
                raise Exception("data() is different!\n\n "
                                + str_trees(t1,t2,row))

            self.assertTrue(c1 == t1 or c1.parent() != None,
                            "Left parent is None! "
                           + "\n\n" +  str_trees(t1,t2,row) )

            self.assertTrue(c2 == t2 or c2.parent() != None,
                            "Right parent is None!"
                             + "\n\n" +  str_trees(t1,t2,row) )

            self.assertTrue(c1.parent() == None or isinstance(c1.parent(), GenericTree),
                           "Left parent is not a GenericTree instance!"
                            + "\n\n" +  str_trees(t1,t2,row) )
            self.assertTrue(c2.parent() == None or isinstance(c2.parent(), GenericTree),
                           "Right parent is not a GenericTree instance! "
                            + "\n\n" +  str_trees(t1,t2,row) )

            if (c1.parent() == None):
                if (c2.parent() != None):
                    raise Exception("Different parents! "
                                    + "Left parent = None   Right parent.data() = " + str(c2.p
arent().data())
                                    + "\n\n" + str_trees(t1,t2,row) )

            else:
                if (c2.parent() == None):
                    raise Exception("Different parents! "
                                    + "Left parent.data() = " + str(c1.parent().data())
                                    + "   Right parent = None"
```

```
                                + "\n\n" + str_trees(t1,t2,row))
            else: # let's just check data for now
                self.assertEquals(c1.parent().data(), c2.parent().data(),
                                "Different parents ! "
                               + "Left parent.data() = " + str(c1.parent().data())
                                + "   Right parent.data() = " + str(c2.parent().data()
                                + "\n\n" + str_trees(t1,t2,row) ))
        i = 0

        cs1 = c1.children()
        cs2 = c2.children()
        if (len(cs1) != len(cs2)):
            raise Exception("Children sizes are different !\n\n"
                            + str_trees(t1, t2, row + min(len(cs1), len(cs2))) )
        while (i < len(cs1) ):
            rec_assert(cs1[i], cs2[i], row + 1)
            i += 1

    rec_assert(t1, t2, 0)

def assertRoot(self, t):
    """ Checks provided node t is a root, if not raises Exception """

    self.assertTrue(t.is_root(), "Detached node " + t.data() + " is not a root, does it ha
ve still the _parent or _sibling set to something ?")

def test_str_trees(self):
    self.assertTrue('a' in str_trees(gt('a'), gt('b')))
    self.assertTrue('b' in str_trees(gt('a'), gt('b')))

    self.assertTrue('a' in str_trees(gt('a', gt('b')), gt('b', gt('c'))))
    self.assertTrue('c' in str_trees(gt('a', gt('b')), gt('b', gt('c'))))


def test_assert_trees_equal(self):
    self.assertTreeEqual(gt('a'), gt('a'))
    self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('b')))

    with self.assertRaises(Exception):
        self.assertTreeEqual(gt('a'), gt('b'))
    with self.assertRaises(Exception):
        self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('c')))

    # different structure
    with self.assertRaises(Exception):
        self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('b',gt('c'))))

    with self.assertRaises(Exception):
        self.assertTreeEqual(gt('a', gt('b',gt('c'))), gt('a', gt('b')))


def test_insert_child(self):
    ta = GenericTree('a')
    self.assertEqual(ta.child(), None)
    tb = GenericTree('b')
    ret = ta.insert_child(tb)
    self.assertEqual(ret, None, self.assertReturnNone(ret, "insert_child"))
    self.assertEqual(ta.child(), tb)
    self.assertEqual(tb.parent(), ta)
    self.assertEqual(tb.sibling(), None)
    self.assertEqual(tb.child(), None)

    tc = GenericTree('c')
    ta.insert_child(tc)
    self.assertEqual(ta.child(), tc)
    self.assertEqual(tc.sibling(), tb)
    self.assertEqual(tc.parent(), ta)
    self.assertEqual(tb.sibling(), None)

def test_insert_children(self):

    t = gt('a')
```

```
        t.insert_children([gt('d'), gt('e')])

        self.assertTreeEqual(t, gt('a', gt('d'), gt('e')))
        t.insert_children([gt('b'), gt('c')])
        self.assertTreeEqual(t, gt('a', gt('b'), gt('c'), gt('d'), gt('e')))


    def test_detach_child(self):

        tb = gt('b')
        tc = gt('c')

        t = gt('a', tb, tc)

        ret = t.detach_child()
        self.assertReturnNone(ret, "detach_child")

        self.assertTreeEqual(t, gt('a', gt('c')))
        self.assertRoot(tb)

        ret = t.detach_child()
        self.assertTreeEqual(t, gt('a'))
        self.assertRoot(tc)


        with self.assertRaises(Exception):
            ret = t.detach_child()

    def test_detach_one_node(self):
        t = gt('a')

        with self.assertRaises(Exception):
            t.detach('a')

        self.assertTreeEqual(t, gt('a'))

    def test_detach_two_nodes(self):
        b1 = gt('b')
        t = gt('a', b1)
        t.detach('b')
        self.assertRoot(b1)

    def test_detach_duplicates(self):
        b1 = gt('b')
        t = gt('a', b1, gt('b'))
        t.detach('b')
        self.assertTreeEqual(t, gt('a', gt('b')))
        self.assertRoot(b1)


    def test_insert_sibling(self):
        raise Exception("TODO - Implement also the *TEST* :-) !"
                        + " If possible, try to add test methods for each case")

    def test_insert_siblings(self):
        raise Exception("TODO - Implement also the *TEST* :-) !"
                        + " If possible, try to add test methods for each case")
```

GenericTree Solution

In [6]:

```
import unittest

class GenericTree:
    """ A tree in which each node can have any number of children.

        Each node is linked to its parent and to its immediate sibling on the right
    """
```

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

    def data(self):
        return self._data

    def child(self):
        return self._child

    def sibling(self):
        return self._sibling

    def parent(self):
        return self._parent

    def is_root(self):
        """ Return True if the node is a root of a tree, False otherwise

            A node is a root whenever it has no parent nor siblings.
        """
        return self._parent == None and self._sibling == None

    def is_subtree(self):
        """ Returns True if the node is a subtree of another tree

            A subtree always has a parent
        """
        return self._parent != None


    def children(self):
        """ Returns the children as a Python list """

        ret = []
        current = self._child
        while current != None:
            ret.append(current)
            current = current._sibling
        return ret

    def __str__(self):
        """ Returns a pretty string of the tree """

        def str_branches(node, branches):
            """ Returns a string with the tree pretty printed.

                branches: a list of characters representing the parent branches. Characters ca
n be either ` ` or '|'
            """
            strings = [str(node._data)]
            current = node._child
            while (current != None):
                if current._sibling == None:
                    # there are better end characters but let's not upset
                    # stupid Python with unicode problems
                    joint = '\-'
                else:
                    joint = '|-'


                strings.append('\n')
                for b in branches:
                     strings.append(b)
                strings.append(joint)
                if current._sibling == None:
                    branches.append('  ')
                else:
                    branches.append('| ')
```

```
                strings.append(str_branches(current, branches))
                branches.pop()
                current = current._sibling

            return "".join(strings)

        return str_branches(self, [])


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

        new_child._sibling = self._child
        new_child._parent = self
        self._child = new_child


    def insert_children(self, new_children):
        """ Takes a list of children and inserts them at the beginning of the current children
 sequence,

            NOTE: in the new sequence new_children appear in the order they are passed to the
function!


            For example:
                >>> t = gt('a', gt('b'), gt('c))
                >>> print t
                a
                |-b
                \-c

                >>>  t.insert_children([gt('d'), gt('e')])
                >>> print t
                a
                |-d
                |-e
                |-b
                \-c
        """
        for c in reversed(new_children):
            self.insert_child(c)

    def insert_sibling(self, new_sibling):
        """ Inserts new_sibling as the immediate next sibling

            If self is a root, raises an Exception
        """
        if (self.is_root()):
            raise Exception("Can't add siblings to a root node !!")

        new_sibling.parent = self.parent
        new_sibling._sibling = self._sibling
        self._sibling = new_sibling

    def insert_siblings(self, new_siblings):
        """ Inserts new_siblings at the beginning of the siblings sequence.

            Nodes are inserted in the same order as they are passed.
            If self is a root, raises an Exception

            For example:

                >>> bt =  gt('b')
                >>> t = gt('a', bt , gt('c))
                >>> print t
                a
                |-b
                \-c

                >>>  bt.insert_children([gt('d'), gt('e')])
                >>> print t
```

```
                a

                |-b
                |-d
                |-e
                \-c
        """
        if (self.is_root()):
            raise Exception("Can't add siblings to a root node !!")

        for s in reversed(new_siblings):
            self.insert_sibling(s)

    def has_child(self):
        """ Returns True if this node has a child, False otherwise """

        return self._child != None

    def detach_child(self):
        """ Detaches the first child.

            if there is no child, raises an Exception
        """

        if (self._child == None):
            raise Exception("There is no child !")
        else:

            detached = self._child
            self._child = self._child._sibling
            detached._parent = None
            detached._sibling = None


    def detach_sibling(self):
        """ Detaches the first sibling.

            If there is no sibling, raises an Exception
        """

        if (self._sibling == None):
            raise Exception("There is no sibling !")
        else:
            self._sibling._parent = None
            self._sibling = self._sibling._sibling

    def detach(self, data):
        """ Detaches the first child that holds the provided data   """

        if (self._child != None):
            current = self._child
            prev = None
            while current != None:
                if (current._data == data):
                    if prev == None: # first element list
                        self.detach_child()
                    else:
                        current._parent = None
                        current._sibling = None
                        prev._sibling = current._sibling
                    return
                else:
                    prev = current
                    current = current._sibling
        raise Exception("Couldn't find any children holding this data:" + str(data))


def gt(*args):
    """ Shorthand function that returns a GenericTree containing the provided
        data and children. First parameter is the data, the following ones are the children.

        Usage examples:
```

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

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

    """
    if (len(args) == 0):
        raise Exception("You need to provide at least one argument for the data!")

    data = args[0]
    children = args[1:]

    r = GenericTree(data)
    for c in reversed(children):
        r.insert_child(c)
    return r

def str_trees(t1, t2, error_row=-1):
    """ Returns a string version of the two trees side by side

        If error_row is given, the line in error is marked.
        If error_row == -1 it is ignored
    """

    s1 = str(t1)
    s2 = str(t2)

    lines1 = s1.split("\n")
    lines2 = s2.split("\n")

    max_len1 = 0
    for line in lines1:
        max_len1 = max(len(line.rstrip().decode("utf-8")), max_len1)

    max_len2 = 0
    for line in lines2:
        max_len2 = max(len(line.rstrip().decode("utf-8")), max_len2)

    strings = []

    i = 0

    while i < len(lines1) or i < len(lines2):

        if i < len(lines1):
            strings.append(lines1[i].rstrip())
            len1 = len(lines1[i].rstrip().decode("utf-8"))
        else:
            len1 = 0

        if (i < len(lines2)):
            len2 = len(lines2[i].rstrip().decode("utf-8"))

            pad_len1 = 4 + max_len1 - len1
            strings.append((" " * pad_len1) + lines2[i].rstrip())
        else:
            len2 = 0

        if (error_row == i):
            pad_len2 = 2 + max_len1 + max_len2 - len1 - len2
            strings.append((" " * pad_len2) + "<--- DIFFERENT ! ")

        strings.append("\n")

        i += 1

    return "".join(strings)
```

```
class GenericTreeTest(unittest.TestCase):

    def assertReturnNone(self, ret, function_name):
        """ Asserts method result ret equals None """
        self.assertEquals(None, ret,
                          function_name
                          + " specs say nothing about returning objects! Instead you are retur
ning " + str(ret))


    def assertTreeEqual(self, t1, t2):
        """ Asserts the trees t1 and t2 are equal """

        def rec_assert(c1, c2, row):

            if c1.data() != c2.data():
                raise Exception("data() is different!\n\n "
                                + str_trees(t1,t2,row))

            self.assertTrue(c1 == t1 or c1.parent() != None,
                            "Left parent is None! "
                           + "\n\n" +  str_trees(t1,t2,row) )

            self.assertTrue(c2 == t2 or c2.parent() != None,
                            "Right parent is None!"
                             + "\n\n" +  str_trees(t1,t2,row) )

            self.assertTrue(c1.parent() == None or isinstance(c1.parent(), GenericTree),
                           "Left parent is not a GenericTree instance!"
                            + "\n\n" +  str_trees(t1,t2,row) )
            self.assertTrue(c2.parent() == None or isinstance(c2.parent(), GenericTree),
                           "Right parent is not a GenericTree instance! "
                            + "\n\n" +  str_trees(t1,t2,row) )

            if (c1.parent() == None):
                if (c2.parent() != None):
                    raise Exception("Different parents! "
                                    + "Left parent = None   Right parent.data() = " + str(c2.p
arent().data())
                                    + "\n\n" + str_trees(t1,t2,row) )

            else:
                if (c2.parent() == None):
                    raise Exception("Different parents! "
                                    + "Left parent.data() = " + str(c1.parent().data())
                                    + "   Right parent = None"
                                    + "\n\n" + str_trees(t1,t2,row))
                else: # let's just check data for now
                    self.assertEquals(c1.parent().data(), c2.parent().data(),
                                      "Different parents ! "
                                     + "Left parent.data() = " + str(c1.parent().data())
                                      + "   Right parent.data() = " + str(c2.parent().data()
                                      + "\n\n" + str_trees(t1,t2,row) ))
            i = 0

            cs1 = c1.children()
            cs2 = c2.children()
            if (len(cs1) != len(cs2)):
                raise Exception("Children sizes are different !\n\n"
                                + str_trees(t1, t2, row + min(len(cs1), len(cs2))) )
            while (i < len(cs1) ):
                rec_assert(cs1[i], cs2[i], row + 1)
                i += 1

        rec_assert(t1, t2, 0)

    def assertRoot(self, t):
        """ Checks provided node t is a root, if not raises Exception """
```

```
        self.assertTrue(t.is_root(), "Detached node " + t.data() + " is not a root, does it ha
ve still the _parent or _sibling set to something ?")

    def test_str_trees(self):
        self.assertTrue('a' in str_trees(gt('a'), gt('b')))
        self.assertTrue('b' in str_trees(gt('a'), gt('b')))

        self.assertTrue('a' in str_trees(gt('a', gt('b')), gt('b', gt('c'))))
        self.assertTrue('c' in str_trees(gt('a', gt('b')), gt('b', gt('c'))))


    def test_assert_tree_equal(self):
        self.assertTreeEqual(gt('a'), gt('a'))
        self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('b')))

        with self.assertRaises(Exception):
            self.assertTreeEqual(gt('a'), gt('b'))
        with self.assertRaises(Exception):
            self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('c')))

        # different structure
        with self.assertRaises(Exception):
            self.assertTreeEqual(gt('a', gt('b')), gt('a', gt('b',gt('c'))))

        with self.assertRaises(Exception):
            self.assertTreeEqual(gt('a', gt('b',gt('c'))), gt('a', gt('b')))


    def test_insert_child(self):
        ta = GenericTree('a')
        self.assertEqual(ta.child(), None)
        tb = GenericTree('b')
        ret = ta.insert_child(tb)
        self.assertEqual(ret, None, self.assertReturnNone(ret, "insert_child"))
        self.assertEqual(ta.child(), tb)
        self.assertEqual(tb.parent(), ta)
        self.assertEqual(tb.sibling(), None)
        self.assertEqual(tb.child(), None)

        tc = GenericTree('c')
        ta.insert_child(tc)
        self.assertEqual(ta.child(), tc)
        self.assertEqual(tc.sibling(), tb)
        self.assertEqual(tc.parent(), ta)
        self.assertEqual(tb.sibling(), None)

    def test_insert_children(self):

        t = gt('a')
        t.insert_children([gt('d'), gt('e')])
        self.assertTreeEqual(t, gt('a', gt('d'), gt('e')))
        t.insert_children([gt('b'), gt('c')])
        self.assertTreeEqual(t, gt('a', gt('b'), gt('c'), gt('d'), gt('e')))


    def test_detach_child(self):

        tb = gt('b')
        tc = gt('c')

        t = gt('a', tb, tc)

        ret = t.detach_child()
        self.assertReturnNone(ret, "detach_child")

        self.assertTreeEqual(t, gt('a', gt('c')))
        self.assertRoot(tb)

        ret = t.detach_child()
        self.assertTreeEqual(t, gt('a'))
        self.assertRoot(tc)
```

```
        with self.assertRaises(Exception):
            ret = t.detach_child()

    def test_detach_one_node(self):
        t = gt('a')

        with self.assertRaises(Exception):
            t.detach('a')

        self.assertTreeEqual(t, gt('a'))

    def test_detach_two_nodes(self):
        b1 = gt('b')
        t = gt('a', b1)
        t.detach('b')
        self.assertRoot(b1)


    def test_detach_duplicates(self):
        b1 = gt('b')
        t = gt('a', b1, gt('b'))
        t.detach('b')
        self.assertTreeEqual(t, gt('a', gt('b')))
        self.assertRoot(b1)

    def test_insert_right_sibling(self):
        ta = gt('a')
        tb = gt('b')
        ta.insert_child(tb)
        tb.insert_sibling(gt('c'))
        self.assertTreeEqual(ta, gt('a', gt('b'), gt('c')))

    def test_insert_middle_sibling(self):

        tb = gt('b')
        ta = gt('a', tb, gt('d'))
        tb.insert_sibling(gt('c'))

        self.assertTreeEqual(ta, gt('a', gt('b'), gt('c') , gt('d')))

    def test_insert_sibling_to_root(self):
        ta = gt('a')

        with self.assertRaises(Exception):
            ta.insert_sibling(gt('b'))

    def test_insert_siblings(self):
        tb = gt('b')
        ta = gt('a', tb, gt('e'))
        tb.insert_siblings([gt('c'), gt('d')])

        self.assertTreeEqual(ta, gt('a', gt('b'), gt('c') , gt('d'), gt('e')))

    def test_insert_siblings_to_root(self):
        ta = gt('a')

        with self.assertRaises(Exception):
            ta.insert_siblings([gt('b'), gt('c')])
```

In [7]:

```
algolab.run(GenericTreeTest)
```

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.041s

OK
```