Chapter 2.2: Data Structures - Stacks and linked lists

Stack

Stack theory

See theory here: http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf) (Slide 46)

See stack definition on the book (http://interactivepython.org/runestone/static/pythonds/BasicDS/WhatisaStack.html)

and following sections :

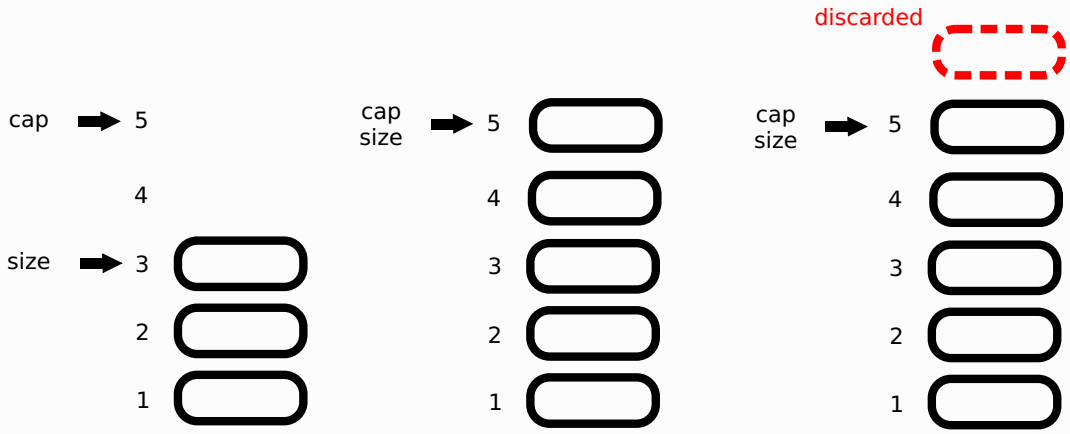
- Stack Abstract Data Type (http://interactivepython.org/runestone/static/pythonds/BasicDS/TheStackAbstractDataType.html)
- Implementing a Stack in Python (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html)
- Simple Balanced Parenthesis (http://interactivepython.org/runestone/static/pythonds/BasicDS/SimpleBalancedParentheses.html)
- Balanced Symbols - a General Case (http://interactivepython.org/runestone/static/pythonds/BasicDS/BalancedSymbols(AGeneralCase).html)

Stack exercises

On slide 46 of theory (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf (Slide 46) there is the pseudo code for a version of stack we will call `CappedStack`:

```
STACK
ITEM[] A                          % Elements
int size                          % Current size
int cap                           % Maximum size

STACK Stack(int dim)
    STACK t ← new STACK
    t.A ← new int[1...dim]
    t.cap ← dim
    t.size ← 0
    return t

boolean isEmpty()
    return size = 0

int size()
    return size

ITEM peek()
    if size > 0 then
        return A[size]

ITEM pop()
    if size > 0 then
        ITEM t ← A[size]
        size ← size − 1
        return t

push(ITEM v)
    if size < cap then
        size ← size + 1
        A[size] ← v
```

A capped stack has a limit called *cap* over which elements are discarded:



- Copy the following skeleton and unit tests, and then implement the pseudo code
- Name internal variables that you don't want to expose to class users by prepending them with one underscore `'_'`, like `_elements` or `_cap`.
  - The underscore is just a convention, class users will still be able to get internal variables by accessing them with field accessors like `mystack._elements`.
  - If users manipulate private fields and complain something is not working, you can tell them it's their fault!
- This time, we will try to write a little more robust code. In general, when implementing pseudocode you might need to think more about boundary cases. In this case, we add the additional constraint that if you pass to the stack a negative or zero cap, your class initalization is expected to fail and raise an `AssertionError`. Such error can be raised by commands like `assert my_condition` where `my_condition` is `False`
- For easier inspection of the stack, implement also an `__str__` method so that calls to `print` show text like `CappedStack: cap=4 elements=['a', 'b']`

> **IMPORTANT: The psudo code uses indexes to keep track the stack size. Since you are providing an actual implementation in Python, you can exploit any Python feature you deem correct to implement the data structure, and even depart a bit from the literal pseudo code. For example, internally you could represent the data as a list, and use its own methods to grow it.**

> **QUESTION: If we already have Python lists that can more or less do the job of the stack, why do we need to wrap them inside a Stack? Can't we just give our users a Python list?**

> **QUESTION: When would you *not* use a Python list to hold the data in the stack?**

Notice that:

- We tried to use more pythonic names (https://www.python.org/dev/peps/pep-0008/#id45) for methods, so for example `isEmpty` was renamed to `is_empty`
- In this case, when this stack reaches `cap` size, successive `push` requests silently exit without raising errors. Other implementations might raise an error and stop excecution when trying to push over on already filled stack.
- In this case, when this stack is required to `pop` or `peek`, if it is empty the functions will not return anything. During the Python translation, we might not return anything as well and relying on Python implicitly returning `None`.
- `pop` will both modify the stack *and* return a value

CappedStack Code Skeleton

```
In [2]
```

In [3]:

```
import unittest

class CappedStack:

    def __init__(self, cap):
        """ Creates a CappedStack capped at cap. Cap must be > 0, otherwise an AssertionError
is thrown
        """
        raise Exception("TODO Implement me!")

    def size(self):
        raise Exception("TODO Implement me!")

    def is_empty(self):
        raise Exception("TODO Implement me!")

    def pop(self):
        raise Exception("TODO Implement me!")

    def peek(self):
        raise Exception("TODO Implement me!")

    def push(self, item):
        raise Exception("TODO Implement me!")

    def cap(self):
        """ Returns the cap of the stack
        """
        raise Exception("TODO Implement me!")


class CappedStackTest(unittest.TestCase):
    """ Test cases for CappedStackTest

         Note this is a *completely* separated class from CappedStack and
         we declare it here just for testing purposes!
         The 'self' you see here have nothing to do with the selfs from the
         CappedStack methods!
    """

    def test_init_wrong_cap(self):
        """
            We use the special construct 'self.assertRaises(AssertionError)' to state
            we are expecting the calls to CappedStack(0) and CappedStack(-1) to raise
            an AssertionError.
        """
        with self.assertRaises(AssertionError):
            CappedStack(0)
        with self.assertRaises(AssertionError):
            CappedStack(-1)


    def test_cap(self):
        self.assertEqual(CappedStack(1).cap(), 1)
        self.assertEqual(CappedStack(2).cap(), 2)


    def test_size(self):
        s = CappedStack(5)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.pop()
        self.assertEqual(s.size(), 0)

    def test_is_empty(self):
        s = CappedStack(5)
        self.assertTrue(s.is_empty())
        s.push("a")
        self.assertFalse(s.is_empty())
```

```
    def test_pop(self):
        s = CappedStack(5)
        self.assertEqual(s.pop(), None)
        s.push("a")
        self.assertEqual(s.pop(), "a")
        self.assertEqual(s.pop(), None)

    def test_peek(self):
        s = CappedStack(5)
        self.assertEqual(s.peek(), None)
        s.push("a")
        self.assertEqual(s.peek(), "a")
        self.assertEqual(s.peek(), "a")  # testing peek is not changing the stack
        self.assertEqual(s.size(), 1)

    def test_push(self):
        s = CappedStack(2)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.push("b")
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")
        s.push("c")  # capped, pushing should do nothing now!
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")

    def test_str(self):
        s = CappedStack(4)
        s.push("a")
        s.push("b")
        print s
```

UnorderedList

UnorderedList Theory

An `UnorderedList` for us is a linked list starting with a pointer called *head* that points to the first `Node` (if the list is empty the pointer points to `None`). Think of the list as a chain where each `Node` can contain some data retriavable with `Node.get_data()` method and you can access one `Node` at a time by calling the method `Node.get_next()` on each node.

- See theory slides (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf) from slide 25 (Monodirectional list)
- See UnorderedList Abstract Data Type (http://interactivepython.org/runestone/static/pythonds/BasicDS/TheUnorderedListAbstractDataType.html) on the book
- See Implementing UnorderedListLinkedLists (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementinganUnorderedListLinkedLists.html on the book

UnorderedList Exercises

1) Understand the problem

Copy the following skeleton and unit tests, and then implement the missing methods, in the order they are presented in the skeleton.

- This time there is not much pseudo or Python code to find around, you should rely solely on theory from the slides and book, method definitions and your intuition
- Pay close attention to the comments below each method definition, especially for boundary cases

COMMANDMENT: You shall also draw lists on paper, helps a lot avoiding mistakes

WARNING: Do *not* use a Python list to hold data inside the data structure. Differently from the `CappedStack` exercise, here you can only use `Node` class. Each `Node` in the `_data` field can hold only one element which is provided by the user of the class, and we don't care about the type of the value the user gives us (so it can be an `int`, a `float`, a `string`, or even a Python list !)

WARNING: NEVER EVER WRITE code like this: `self = something`

Instead, assign values to *fields* of `self`, like `self._head = something`

WARNING: if there isn't the word "*Return*" in a method comment, then most probably that method does not return anything!

WARNING: Methods of the class `UnorderedList` are supposed to *never* return instances of `Node`. If you see them returned in the tests, then you are making some mistake. Users of `UnorderedList` are should only be able to get access to items inside the Node `data` fields.

Notice that there are a few differences with the book:

- We don't assume the list has all different values
- We used more pythonic names (https://www.python.org/dev/peps/pep-0008/#id45) for properties and methods, so for example private attribute `Node.data` was renamed to `Node._data` and accessor method `Node.getData()` was renamed to `Node.get_data()`. There are nicer ways to handle these kind of getters/setters pairs called 'properties' but we won't address them here.
- In boundary cases like removing a non-existing element we prefer to raise an exception with the command

```
raise Exception("Some error occurred!")
```

In general, this is the behaviour you also find in regular Python lists.

2) Save a copy of your work

You already wrote a lot of code, and you don't want to lose it, right? Since we are going to make many modifications, when you reach a point when the code does something useful, it is good practice to save a copy of what you have done somewhere, so if you later screw up something, you can always restore the copy.

- Add also somewhere in the copy (in a comment at the top of the `.py` file or in a separate `README.txt` file) the version (like `1.0`), the date, and a description of the main features (for example "Simple linked list, not particularly performant").
- Backing up the work is a form of the so-called *versioning* : there are much better ways to do it (like using git (https://git-scm.com)) but we don't address them here.

3) Faster size()

Implement a `size()` method that works in `O(1)`. To make this work without going through the whole list each time, we will need a new `_size` field that keeps track of the size. When the list is mutated with methods like add, append, etc you will also need to update the `_size` field accordingly. Proceed like this:

3.1) add a new field `_size` in the class constructor and initialize it to zero

3.2) modify the `size()` method to just return the `_size` field.

3.3) The data structure starts to be complex, and we need better testing. If you look at the tests, very often there are lines of code like `self.assertEquals(ul.to_python(), ['a', 'b'])` in the `test_add` method:

```
def test_add(self):
        ul = UnorderedList()
        self.myAssert(ul, [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])
```

Last line checks our unordered list `ul` contains a sequence of linked nodes that once transformed to a python list actually equals `['a', 'b']`. Since in the new implementation we are going to mutate `_size` field a lot, it could be smart to also check that `ul.size()` equals `len(["a", "b"])`. Repeating this check in every test method could be quite verbose. Instead, we can do a smarter thing, and develop in the `UnorderedListTest` class a new assertion method on our own:

3.3.1) Add this method to `UnorderedListTest` class:

```
def myAssert(self, unordered_list, python_list):
        """ Checks provided unordered_list can be represented as the given python_list.
Since v2.
        """
        self.assertEquals(unordered_list.to_python(), python_list)
        # check this new invariant about the size
        self.assertEquals(unordered_list.size(), len(python_list))
```

> **WARNING: method `myAssert` must *not* being with `test_`, otherwise `unittest` will run it as a test!**

3.3.2) Now, how to use this powerful new `myAssert` method? In the test class, just replace every occurence of

```
self.assertEquals(ul.to_python(), ['a', 'b'])
```

into calls like this:

```
self.myAssert(ul, ['a', 'b'])
```

> **WARNING: Notice the `.to_python()` after `ul` is gone.**

3.4) Actually update `_size` in the various methods where data is mutated, like `add`, `insert`, etc.

3.5) Run the tests and hope for the best ;-)

4) Faster append()

We are now better equipped to make further improvements. Once you're done implementing the above and made sure everything works, you can implement an append method that works in $O(1)$ by adding an additional pointer in the data structure that always point at the last node. To further exploit the pointer, you can also add a fast `last(self)` method that returns the last value in the list. Proceed like this:

4.1) Save a copy of your work somewhere, putting version (2.0), date, and comments on the improvements.

4.2) Add an additional pointer called `_last` in the constructor.

4.3) Copy this method into the class. Just copy it, don't implement it for now.

```
def last(self):
        """ Returns the last element in the list, in O(1).

            If list is empty, raises an Exception. Since v2.
        """
        raise Exception("TODO implement me!")
```

4.4) Let's do some so-called *test driven development*, that is, first we write the tests, then we write the implementation.

> **WARNING: During the exam you *will* be asked to write tests, so don't skip writing them now !!**

4.4.1) Implement a test for `last()` method, by adding this to `UnorderedListTest` class:

```
def test_last(self):
        raise Exception("TODO IMPLEMENT ME !")
```

In the method, create a list and add elements using only calls to `add` method and checks using the `myAssert` method. When done, ask your instructor if the test is correct (or look at the proposed solution at the bottom of the worksheet), it is important you get it right otherwise you won't be able to properly test your code.

4.4.2) You already have a test for the `append()` method, but, how can you be sure the `_last` pointer is updated correctly throughout the code? When you implemented the fast `size()` method you wrote some invariant in the `myAssert` method. We can do the same this time, too. Find the invariant and add the corresponding check to the `myAssert` method. When done, ask your instructor if the invariant is correct (or look at the proposed solution at the bottom of the worksheet): it is important you get it right otherwise you won't be able to properly test your code.

4.5) Update the methods that mutate the data structure (`add`, `insert`, `remove` ...) so they keep `_last` pointed to last element. If the list is empty, `_last` will point to `None`. Taking particular care of corner cases such as empty list and one element list.

4.6) Cross your fingers and run the tests!

5) Go bidirectional

Our list so far has links that allow us to traverse it fast in one direction. But what if we want fast traversal in the reverse direction, from last to first element? What if we want a `pop()` that works in $O(1)$? To speed up these operations we could add backward links to each `Node`. Proceed in the following way:

5.1) Save your work

Once you're done with previous points, save the version you have somewhere adding comments about the improvements done so far, the version number (like 2.0) and the date. Then start working on a new copy.

5.2) Node backlinks

In `Node` class, add backlinks by adding the attribute `_prev` and methods `get_prev(self)` and `set_prev(self, pointer)`.

5.3) Better `__str__`

Improve `__str__` method so it shows presence or absence of links, along with the size of the list. `next` pointers presence must be represented with `>` character , absence with `*` character. They must be put after the item representation. `prev` pointers presence must be represented with `<` character , absence with `*` character. They must be put befor the item representation.

For example, for the list ['a','b','c'], you would have the following representation:

```
UnorderedList(size=3):*a><b><c*
```

As a special case for empty list you should print the following:

```
UnorderedList(size=0):[]
```

Other examples of proper lists, with 3, 2, and 1 element can be:

```
UnorderedList(size=3):*a><b><c*
UnorderedList(size=2):*a><b*
UnorderedList(size=1):*a*
```

This new `__str__` method should help you to spot broken lists like the following, were some pointers are not correct:

```
Broken list, all prev pointers are missing:
UnorderedList(size=3):*a>*b>*c*

Broken list, size = 3 but shows only one element with next pointer set to None:
UnorderedList(size=3):*a*

Broken list, first backward pointer points to something other than None
UnorderedList(size=3):<a><b><c*
```

5.4) Modify add()

Update the `UnorderedList` `add` method to take into account you now have backlinks. Take particular care for the boundary cases when the list is empty, has one element, or for nodes at the head and at the tail of the list.

5.5) Add to_python_reversed()

Implement `to_python_reversed` method with a linear scan by using the newly added backlinks:

```
def to_python_reversed(self):
        """ Returns a regular Python list with the elements in reverse order,
            from last to first. Since v3. """
        raise Exception("TODO implement me")
```

Add also this test, and make sure it pass:

```
   def test_to_python_reversed(self):
           ul = UnorderedList()
           ul.add('c')
           ul.add('b')
           ul.add('a')
           pr = ul.to_python()
           pr.reverse()  # we are reversing pr with Python's 'reverse()' method
           self.assertEquals(pr, ul.to_python_reversed())
```

5.6) Add invariant

By using the method `to_python_reversed()`, add a new invariant to the `myAssert` method. If implemented correctly, this will surely spot a lot of possible errors in the code.

5.7) Modify other methods

Modify all other methods that mutate the data structure (`insert`, `remove`, etc) so that they update the backward links properly.

5.8) Run the tests

If you wrote meaningful tests and all pass, congrats!

UnorderedList Code Skeleton

In [8]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user. """

    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, raising except
ions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None

    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very handy for
 testing.
        """
        python_list = []
        current = self._head

        while (current != None):
```

```
            while (current != None):
                python_list.append(current.get_data())
                current = current.get_next()
            return python_list

        def __str__(self):
            """ For potentially complex data structures like this one, having a __str__ method is
essential to
                quickly inspect the data by printing it.
            """
            current = self._head
            strings = []

            while (current != None):
                strings.append(str(current.get_data()))
                current = current.get_next()

            return "UnorderedList: " + ",".join(strings)


        def is_empty(self):
            """ Returns True if the list has no nodes, True otherwise """
            raise Exception("TODO implement me!")


        def add(self,item):
            """ Adds item at the beginning of the list """
            raise Exception("TODO implement me!")


        def size(self):
            """ Returns the size of the list """
            raise Exception("TODO implement me!")


        def search(self,item):
            """ Returns True if item is present in list, False otherwise
            """
            raise Exception("TODO implement me!")

        def remove(self, item):
            """ Removes first occurrence of item from the list

                If item is not found, raises an Exception.
            """
            raise Exception("TODO implement me!")

        def append(self, e):
            """ Appends element e to the end of the list.

                For this exercise you can write the O(n) version
            """

            raise Exception("TODO implement me!")

        def insert(self, i, e):
            """ Insert an item at a given position.

                The first argument is the index of the element before which to insert, so list.ins
ert(0, e)
                inserts at the front of the list, and list.insert(list.size(), e) is equivalent to
 list.append(e).
                When i > list.size(), raises an Exception (default Python list appends instead to
the end :-/ )

            """
            raise Exception("TODO implement me!")

        def index(self, e):
            """ Return the index in the list of the first item whose value is x.
                If there is no such item, an exception is raised.
            """
```

```
        raise Exception("TODO implement me!")


    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        raise Exception("TODO implement me!")

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList

         Note this is a *completely* separated class from UnorderedList and
         we declare it here just for testing purposes!
         The 'self' you see here have nothing to do with the selfs from the
         UnorderedList methods!
    """

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))


    def test_is_empty(self):
        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.assertEquals(ul.to_python(), [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')
```

```
    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])

    def test_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.assertEquals(ul.to_python(),['a'])
        ul.append('b')
        self.assertEquals(ul.to_python(),['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
```

```
    def test_insert_two_element_list_insert_middle(self):

        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])


    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.pop()

    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.assertEquals(ul.to_python(), ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.assertEquals(ul.to_python(), [])
```

Solutions

CappedStack Solution

In [9]:

```
import unittest

class CappedStack:

    def __init__(self, cap):
        """ Creates a CappedStack capped at cap.

            Cap must be > 0, otherwise an AssertionError is thrown
        """
```

```
        assert cap > 0

        # notice we assign to variables with underscore to respect Python conventions
        self._cap = cap
        # notice with use _elements instead of the A in the pseudocode, because it is
        # clearer, starts with underscore, and capital letters are usual reserved
        # for classes or constants
        self._elements = []

    def size(self):
        return len(self._elements)

    def is_empty(self):
        return len(self._elements) == 0

    def pop(self):
        if (len(self._elements) > 0):
            return self._elements.pop()
        # else: implicitly, Python will return None

    def peek(self):
        if (len(self._elements) > 0):
            return self._elements[-1]
        # else: implicitly, Python will return None

    def push(self, item):
        if (len(self._elements) < self._cap):
            self._elements.append(item)
        # else fail silently

    def cap(self):
        """ Returns the cap of the stack
        """
        return self._cap

    def __str__(self):
        return "CappedStack: cap=" + str(self._cap) + " elements=" + str(self._elements)


class CappedStackTest(unittest.TestCase):

    """ Test cases for CappedStackTest

         Note this is a *completely* separated class from CappedStack and
         we declare it here just for testing purposes!
         The 'self' you see here have nothing to do with the selfs from the
         CappedStack methods!
    """

    def test_init_wrong_cap(self):
        """
            We use the special construct 'self.assertRaises(AssertionError)' to state
            we are expecting the calls to CappedStack(0) and CappedStack(-1) to raise
            an AssertionError.
        """
        with self.assertRaises(AssertionError):
            CappedStack(0)
        with self.assertRaises(AssertionError):
            CappedStack(-1)


    def test_cap(self):
        self.assertEqual(CappedStack(1).cap(), 1)
        self.assertEqual(CappedStack(2).cap(), 2)


    def test_size(self):
        s = CappedStack(5)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.pop()
```

```
        self.assertEqual(s.size(), 0)

    def test_is_empty(self):
        s = CappedStack(5)
        self.assertTrue(s.is_empty())
        s.push("a")
        self.assertFalse(s.is_empty())


    def test_pop(self):
        s = CappedStack(5)
        self.assertEqual(s.pop(), None)
        s.push("a")
        self.assertEqual(s.pop(), "a")
        self.assertEqual(s.pop(), None)

    def test_peek(self):
        s = CappedStack(5)
        self.assertEqual(s.peek(), None)
        s.push("a")
        self.assertEqual(s.peek(), "a")
        self.assertEqual(s.peek(), "a")  # testing peek is not changing the stack
        self.assertEqual(s.size(), 1)

    def test_push(self):
        s = CappedStack(2)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.push("b")
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")
        s.push("c")  # capped, pushing should do nothing now!
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")

    def test_str(self):
        s = CappedStack(4)
        s.push("a")
        s.push("b")
        print s
```

UnorderedList v1 Solution

In [11]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user. """

    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        UnorderedList v1
```

```
        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, raising except
ions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None

    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very handy for
 testing.
        """
        python_list = []
        current = self._head

        while (current != None):
            python_list.append(current.get_data())
            current = current.get_next()
        return python_list

    def __str__(self):
        """ For potentially complex data structures like this one, having a __str__ method is
essential to
            quickly inspect the data by printing it.
        """
        current = self._head
        strings = []

        while (current != None):
            strings.append(str(current.get_data()))
            current = current.get_next()

        return "UnorderedList: " + ",".join(strings)


    def is_empty(self):
        return self._head == None

    def add(self,item):
        """ Adds item at the beginning of the list """
        new_head = Node(item)
        new_head.set_next(self._head)
        self._head = new_head

    def size(self):
        """ Returns the size of the list """
        current = self._head
        count = 0

        while (current != None):
            current = current.get_next()
            count += 1

        return count

    def search(self,item):
        """ Returns True if item is present in list, False otherwise
        """
        current = self._head

        while (current != None):
            if (current.get_data() == item):
                return True
            else:
                current = current.get_next()

        return False
```

```
    def remove(self, item):

        """ Removes first occurrence of item from the list

            If item is not found, raises an Exception.
        """
        current = self._head
        prev = None

        while (current != None):
            if (current.get_data() == item):
                if prev == None:  # we need to remove the head
                    self._head = current.get_next()
                else:
                    prev.set_next(current.get_next())
                    current = current.get_next()
                return  # Found, exits the function
            else:
                prev = current
                current = current.get_next()

        raise Exception("Tried to remove a non existing item! Item was: " + str(item))

    def append(self, e):
        """ Appends element e to the end of the list.

            For this exercise you can write the O(n) version
        """

        if self._head == None:
            self.add(e)
        else:
            current = self._head
            while (current.get_next() != None):
                current = current.get_next()
            current.set_next(Node(e))

    def insert(self, i, e):
        """ Insert an item at a given position.

            The first argument is the index of the element before which to insert, so list.ins
ert(0, e)
            inserts at the front of the list, and list.insert(list.size(), e) is equivalent to
 list.append(e).
            When i > list.size(), raises an Exception (default Python list appends instead to
the end :-/ )

        """
        if (i < 0):
            raise Exception("Tried to insert at a negative index! Index was:" + str(i))

        count = 0
        current = self._head
        prev = None

        while (count < i and current != None):
            prev = current
            current = current.get_next()
            count += 1

        if (current == None):
            if (count == i):
                self.append(e)
            else:
                raise Exception("Tried to insert outside the list ! "
                                + "List size=" + str(count) + "  insert position=" + str(i))
        else:
            #0 1
            #  i
            if (prev == None):
                self.add(e)
            else:
```

```
                new_node = Node(e)

                prev.set_next(new_node)
                new_node.set_next(current)

    def index(self, e):
        """ Return the index in the list of the first item whose value is x.

            If item is not found, raises an Exception.
        """

        current = self._head
        count = 0

        while (current != None):
            if (current.get_data() == e):
                return count
            else:
                current = current.get_next()
                count += 1

        raise Exception("Couldn't find element " + str(e) )



    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        if (self._head == None):
            raise Exception("Tried to pop an empty list!")
        else:

            current = self._head

            if (current.get_next() == None): # one element list
                last_item = self._head.get_data()
                self._head = None
            else:    # we have more than one element
                prev = None
                while current.get_next() != None:  # current will reach last element
                    prev = current
                    current = current.get_next()

                last_item = current.get_data()
                prev.set_next(None)

            return last_item

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList

         Note this is a *completely* separated class from UnorderedList and
         we declare it here just for testing purposes!
         The 'self' you see here have nothing to do with the selfs from the
         UnorderedList methods!
    """

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))


    def test_insert(self):
```

```
    def test_is_empty(self):

        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.assertEquals(ul.to_python(), [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')

    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])


    def test_remove(self):
```

```
    def test_append(self):

        ul = UnorderedList()
        ul.append('a')
        self.assertEquals(ul.to_python(),['a'])
        ul.append('b')
        self.assertEquals(ul.to_python(),['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])


    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
```

```
        with self.assertRaises(Exception):

            ul.pop()

    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.assertEquals(ul.to_python(), ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.assertEquals(ul.to_python(), [])
```

UnorderedList v2 Solution

In [13]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user.

        Node v2 remains the same as Node v1
    """
    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        a linked list implementation, v2.

        Improvements upon UnorderedList v1:

        * calculates size() in O(1)
        * calculates append() in O(1)
        * adds last() method to retrieve last element in O(1)

        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, raising except
ions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None
        self._size = 0  # NEW attribute '_size'
```

```
        self._last = None # NEW attribute '_last'

    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very handy for
 testing.
        """
        python_list = []
        current = self._head

        while (current != None):
            python_list.append(current.get_data())
            current = current.get_next()
        return python_list


    def __str__(self):
        """ For potentially complex data structures like this one, having a __str__ method is
essential to
            quickly inspect the data by printing it.
        """
        current = self._head
        strings = []

        while (current != None):
            strings.append(str(current.get_data()))
            current = current.get_next()

        return "UnorderedList: " + ",".join(strings)


    def is_empty(self):
        return self._head == None

    def add(self,item):
        """ Adds item at the beginning of the list """
        new_head = Node(item)
        new_head.set_next(self._head)
        if self._head == None: # NEW
            self._last = new_head # NEW
        self._head = new_head
        self._size += 1 # NEW, we just return the field value. This is fast!


    def size(self):
        """ Returns the size of the list in O(1) """
        return self._size  # NEW, we just return the field value. This is fast!

    def search(self,item):
        """ Returns True if item is present in list, False otherwise
        """
        current = self._head

        while (current != None):
            if (current.get_data() == item):
                return True
            else:
                current = current.get_next()

        return False

    def remove(self, item):
        """ Removes first occurrence of item from the list

            If item is not found, raises an Exception.
        """
        current = self._head
        prev = None

        while (current != None):

            if (current.get_data() == item):
                if (self._last == current): # NEW
```

```
                if (self._last == current):  # NEW
                    self._last = prev        # NEW

                if prev == None:  # we need to remove the head
                    self._head = current.get_next()
                else:
                    prev.set_next(current.get_next())
                    current = current.get_next()
                self._size -= 1  # NEW, need to update _size
                return  # Found, exits the function
            else:
                prev = current
                current = current.get_next()

        raise Exception("Tried to remove a non existing item! Item was: " + str(item))

    def append(self, e):
        """ Appends element e to the end of the list, in O(1)
        """

        if self._head == None:
            self.add(e)
        else:
            new_node = Node(e)
            self._last.set_next(new_node) # NEW, we directly exploit _last pointer
            self._last = new_node # NEW, need to update _last
            self._size += 1  # NEW, need to update _size

    def insert(self, i, e):
        """ Insert an item at a given position.

            The first argument is the index of the element before which to insert, so list.ins
ert(0, e)
            inserts at the front of the list, and list.insert(list.size(), e) is equivalent to
 list.append(e).
            When i > list.size(), raises an Exception (default Python list appends instead to
the end :-/ )

        """
        if (i < 0):
            raise Exception("Tried to insert at a negative index! Index was:" + str(i))

        count = 0
        current = self._head
        prev = None

        while (count < i and current != None):
            prev = current
            current = current.get_next()
            count += 1

        if (current == None):
            if (count == i):
                self.append(e)
            else:
                raise Exception("Tried to insert outside the list ! "
                                + "List size=" + str(count) + "  insert position=" + str(i))
        else:
            #0 1
            #  i
            if (prev == None):
                self.add(e)
            else:
                new_node = Node(e)
                prev.set_next(new_node)
                new_node.set_next(current)

                self._size += 1 # NEW, need to update _size

    def index(self, e):
        """ Return the index in the list of the first item whose value is x.

            If it is not in the list, a ValueError is raised.
        """
        current = self._head
        count = 0
        while current != None:
            if current.get_data() == e:
                return count
            current = current.get_next()
            count += 1
        raise ValueError("{} is not in the list!".format(e))

    def pop(self, i=None):
        """ Remove the item at the given position in the list, and return it.
            If no index is specified, a.pop() removes and returns the last item in the list.
            Raises an Exception if the list is empty or index is outside the list.
        """
        if self._head == None:
            raise Exception("pop from empty list")

        if i == None:
            i = self.size() - 1

        current = self._head
        prev = None
        count = 0
        while current != None:
            if count == i:
                if prev == None:  # we need to remove the head
                    self._head = current.get_next()
                else:
                    prev.set_next(current.get_next())
                self._size -= 1  # NEW, need to update _size
                return current.get_data()  # Found, exits the function
            prev = current
            current = current.get_next()
            count += 1
        raise Exception("pop index out of range")

    def search(self, item):
        """ Search for item in the list.
            Return True if found, False otherwise.
        """
        current = self._head
        while current != None:
            if current.get_data() == item:
                return True
            current = current.get_next()
        return False

    def size(self):
        """ Return the size of the list, in O(1)
        """
        return self._size

    def __str__(self):
        """ Returns a human-readable string representation of the list
        """
        node = self._head
        s = "["
        while node != None:
            s += str(node.get_data())
            node = node.get_next()
            if node != None:
                s += ", "
        s += "]"
        return s

    def __repr__(self):
        """ Returns a human-readable string representation of the list
        """
        return str(self)
```

```
            If item is not found, raises an Exception.

        """

        current = self._head
        count = 0

        while (current != None):
            if (current.get_data() == e):
                return count
            else:
                current = current.get_next()
                count += 1

        raise Exception("Couldn't find element " + str(e) )

    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        if (self._head == None):
            raise Exception("Tried to pop an empty list!")
        else:

            current = self._head

            if (current.get_next() == None): # one element list
                popped = self._head
                self._head = None
                self._last = None  # NEW
            else:    # we have more than one element
                prev = None
                while current.get_next() != None:  # current will reach last element
                    prev = current
                    current = current.get_next()

                popped = current
                self._last = prev   # NEW
                prev.set_next(None)

            self._size -= 1  # NEW

            return popped.get_data()


    def last(self):
        """ Returns the last element in the list, in O(1).

            If list is empty, raises an Exception. Since v2.
        """

        if (self._head == None):
            raise Exception("Tried to get the last element of an empty list!")
        else:
            return self._last.get_data()

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList v2

        Test cases are improved by adding a new method myAssert(self, unordered_list, python_l
ist)

         Note this is a *completely* separated class from UnorderedList and
         we declare it here just for testing purposes!
         The 'self' you see here have nothing to do with the selfs from the
         UnorderedList methods!
    """

    def myAssert(self, unordered_list, python_list):
        """ Checks provided unordered_list can be represented as the given python_list. Since
v2
        """
```

```
        """
        self.assertEquals(unordered_list.to_python(), python_list)
        # we check this new invariant about the size
        self.assertEquals(unordered_list.size(), len(python_list))
        # we check this new invariant about the last element
        if len(python_list) != 0:
            self.assertEquals(unordered_list.last(), python_list[-1])

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))

    def test_is_empty(self):
        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.myAssert(ul, [])
        ul.add('b')
        self.myAssert(ul, ['b'])
        ul.add('a')
        self.myAssert(ul, ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')

    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
```

```
            ul.remove('c')

        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])

    def test_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.myAssert(ul,['a'])
        ul.append('b')
        self.myAssert(ul,['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.myAssert(ul, ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.myAssert(ul, ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.myAssert(ul, ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.myAssert(ul, ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.myAssert(ul, ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.myAssert(ul, ['a','b', 'c'])
```

```
    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.pop()

    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.myAssert(ul, ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.myAssert(ul, [])

    def test_last(self):
        """ This tests only simple cases. More in-depth testing will be provided by calls to m
yAssert """

        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.last()
        ul.add('b')
        self.assertEquals(ul.last(), 'b')
        ul.add('a')
        self.assertEquals(ul.last(), 'b')
```

In [15]:

```
```