Umeå Universitet
Department of Computing Science 12/02/2019

Datastrukturer och algoritmer (Python) 7.5p

Obligatory Assignment 2

Aladdin Persson (aladdin.persson@umu.se)

Version 1, Username: tfy17api
Supervisors: Thomas Johansson,
Henrik Rosén

Contents

1 Introduction

To be able to test if a given implementation has been done so correctly is an essential skill to have as a programmer. This way we are able to reassure ourselves and make sure that there are no obvious errors in our code. It is in practice usually impossible to test all possible combinations and here lies the reason in designing good test programs.

In this report we design a testing procedure for the data structure stack and give reasons to why the tests are chosen. To be able to understand the testing procedure it is good to have a understanding of what we wish to test and therefore a brief overview of the stack data structure is given in Section (2). Lastly we examine a testing procedure for stack and outline the testing functions for them in Section (3).

All necessary source code for this report is included in Appendix as well as in a separate zip file for this report.

2 Stack

A stack is a data structure which primarily gives us the ability to delete, add and read the latest added element and does so very quickly in constant time. Imagine a stack of plates and when taking your plate it is necessary to take the plate at the top of the stack. This works quite nicely as a visualization for the data structure stack. In a stack the element deleted from the set is the one most recently inserted and a stack implements a **LIFO** or **last-in, first-out** policy. Stacks can be described by the following informal specification

- Empty - Constructs an empty stack.
- Push(v,s) - Appends an element with value v on top of the stack s.
- Top(s) - Is the value on top of the stack (Assumed it isn't empty).
- Pop(s) - Deletes the element at the top of the stack (Assumed it isn't empty).
- Isempty(s) - Checks if the stack is empty.

A stack is most commonly implemented as either an array or a linked list. In this we look at the implementation of a stack using linked lists. I will not go into detail of the differences between implementation of a stack between these two data structures or the way these data structures work as we are only interested in the functionality behind a stack.

3 Testing implementation of a stack

We want to start with testing that the stack implementation has basic functionality and we also wish to construct tests that are specific and tests one thing at a time. This makes it easier for us to know what went wrong.

We start with testing what I denote as testing "axioms" or the very basic operations of a stack.

- **test_isempty()** - tests if a stack is empty when created
- **test_addelement()** - tests if a stack is empty after added an element
- **test_poppush()** - tests if a push followed by deletion results in the original stack
- **test_toppush()** - tests if added element is on the top of the stack
- **test_push_toppop()** - tests if we pop element and then if putting that element back results in the original stack

After passing these tests we are sure that it at least satisfies the very basic functionality of a stack. Recognize that these tests have used every function of a stack from the definition in section (2) and also if simple combinations of them results in what we expect the abstract data type to do. This does however not guarantee that the implementation has been correctly implemented, for example let us consider if the we wish to look at the top of an empty stack. This is not possible and a correctly constructed stack will raise an error when this occurs. This is why we construct what I call 'supplementary' tests.

- **test_popempty()** - tests if an error is raised when trying to pop empty stack
- **test_topempty()** - tests if an error is raised when trying to look at the top element of an empty stack
- **test_addelements(tot_elements)** - tests if given a total amount of elements to place to a stack results in the stack putting them all in the same order, i.e that the first element is put at the bottom and the next on top of that, etc.

After passing all of these we are certain that it shares many characteristics to the abstract data type of a stack. As stated in the introduction to this report we can

never guarantee that this encompasses all tests necessary to guarantee that this is a correctly implemented stack. For example consider if we test that it puts 10 elements in the correct order, does this guarantee that it will do so for 11, 12, etc?

Perhaps one could also imagine that a correctly implemented stack would no longer store the popped element in memory, in a high level language such as Python this is usually not of concern as it has in-built garbage collection but in other languages it might be necessary to have stricter memory management which is not tested by this procedure outlined. In this testing procedure we have tested the basic functionality or the axioms of a stack as well as cases where we expect a behavior such as raising errors. When doing this testing procedure on the implementation of a stack given in `Stack1Cell.py` we find no errors which implies that this is a robust implementation of a stack cell.

4 Appendix

4.1 StackTest.py

Source code for `stacktest.py`.

```
# This program tests an implementation of a stack.
# If any test was not passed then you will get an error message!

# Programmed by 2019-02-06 <aladdin.persson@umu.se>

from Stack1Cell import Stack1Cell

import random
import string
import copy

# Test: Check if stack is empty when created
def test_isempty():
    stack = Stack1Cell()

    if not stack.isempty():
        raise("Stack was created but wasn't empty!")

# Test: Check if stack is no longer empty after added element
def test_addelement():
    stack = Stack1Cell()
    stack.push('not_empty_now')

    if stack.isempty():
        raise("Element was added but stack is empty!")

# Test: Check if push followed by deletion results in the original stack
def test_poppush():
    stack = Stack1Cell()

    # Original stack
    stack.push('test')
    copy_stack = copy.copy(stack)

    # Do operations that give back original in the end
    random_character = random.choice(string.ascii_letters)
```

```
    stack.push(random_character)
    stack.pop()

    if stack.top() != copy_stack.top():
        raise("We added and deleted element, so it should be equal to original!")

# Test: Check if element added is put at the top of the stack
def test_toppush():
    stack = Stack1Cell()
    element_to_add = random.choice(string.ascii_uppercase)

    stack.push(element_to_add)

    if element_to_add != stack.top():
        raise("Element addded isn't equal the top of stack")

# Test: Check if push element that was popped results in original stack
def test_push_toppop():
    # We assume the stack isn't empty. If we pop the top element of a stack and pu
    # we popped then we should receive the original stack back.

    stack = Stack1Cell()
    stack.push(random.choice(string.ascii_uppercase))

    element_we_removed = stack.top()
    stack_copy = copy.copy(stack)

    stack.pop()
    stack.push(element_we_removed)

    if stack_copy.top() != stack.top():
        raise("We added the element we removed but it doesnt equal the original")

# The following tests are supplementary/more advanced than the previous ones. If i
# then we know it at least satisfies axioms from stacks.

# Test: Check if we can pop an empty stack (should result in error!)
def test_popempty():
    s = Stack1Cell()
```

```
    try:
        s.pop()
        raise("We was able to pop an empty stack. This should not be possible!")
    except:
        pass

# Test: Check if we can look at top element of empty stack
def test_topempty():
    s = Stack1Cell()

    try:
        s.top()
        raise("We tried to look at top element of an empty stack. This should not b
    except:
        pass

# Test: Check if added a bunch of elements is then put in the correct order in the
def test_addelements(tot_elements=10):
    # Generate random list
    L = [random.choice(string.ascii_letters + string.digits) for i in range(tot_ele

    stack = Stack1Cell()

    for each in L:
        stack.push(each)

    for i in range(len(L)-1,-1,-1):
        if stack.top() != L[i]:
            raise("Wrong order of stack!")
        stack.pop()


# Run all tests
def test():
    test_isempty()
    test_addelement()
    test_addelement()
    test_poppush()
    test_toppush()
    test_push_toppop()
```

```
    # If nothing has been raised at this point then we know it has passed all 'axi
    #print("--------------- All Axiom tests OK! ---------------")

    test_popempty()
    test_topempty()
    test_addelements(tot_elements=100)

    # If nothing has been raised here then it has passed all 'supplementary' tests
    #print("--------------- All Supplementary tests OK! ---------------")
    #print()
    #print("This implementation seems to be correct from the tests performed!")

if __name__ == "__main__":
    test()
```

.

4.2 Stack1Cell.py

Source code for `Stack1Cell.py`.

```
# -*- coding: latin-1 -*-

#Written by Lena Kallin Westin <kallin@cs.umu.se>.
#May be used in the course Datastrukturer och Algoritmer (Python) at Umeniversity.
#Usage exept those listed above requires permission by the author.

class EmptyStackError(Exception):
    pass

"""
Datatypen Stack enligt definitionen pidan 134 i Lars-Erik Janlert,
Torbjrn Wiberg Datatyper och algoritmer 2., [rev.] uppl.,Lund,
Studentlitteratur, 2000, x, 387 s. ISBN 91-44-01364-7

Variabler och funktioner som inleds med ett enkelt underscore "_" privata
fr klassen och ska inte anvas av de som anver denna klass.

Denna klass implementerar stacken med hj av en 1-cell
"""
```

```
from OneCell import OneCell

class Stack1Cell:

    def __init__(self):
        """
            Syfte: Skapar en tom stack med hj av en 1Cell
            Returve: -
            Kommentarer: I boken heter denna funktion Empty.
        """
        self._head = None

    def top(self):
        """
            Syfte: Ger vet av det versta elementet ptacken
            Returve: Vet verst ptacken
            Kommentarer: Ej definierad fr tom stack
        """
        if self.isempty():
            raise EmptyStackError("Error in top")
        return self._head.inspectValue()

    def push(self, obj):
        """
            Syfte: Ler ett element med vet v verst ptacken
            Returve: -
            Kommentarer:
        """
        temp = OneCell()
        temp.setValue(obj)
        temp.setLink(self._head);
        self._head=temp;

    def pop(self):
        """
            Syfte: Tar bort versta vet frstacken.
            Returve: -
            Kommentarer: Ej definierad fr tom stack
        """
        if self.isempty():
```

```
            raise EmptyStackError("Error in pop")
        self._head = self._head.inspectLink()

    def isempty(self):
        """
            Syfte: Testar om stacken tom
            Returve: Sant om stacken tom, annars falskt
            Kommentarer:
        """
        return self._head is None
```

.

4.3 OneCell.py

Source code for `OneCell.py`.

```
# -*- coding: utf-8 -*-

#Written by Lena Kallin Westin <kallin@cs.umu.se>.
#May be used in the course Datastrukturer och Algoritmer (Python) at Umeniversity.
#Usage exept those listed above requires permission by the author.

"""
Datatypen 1-Cell enligt definitionen på sidan 77 i Lars-Erik Janlert,
Torbjörn Wiberg Datatyper och algoritmer 2., [rev.] uppl.,Lund,
Studentlitteratur, 2000, x, 387 s. ISBN 91-44-01364-7

Variabler och funktioner som inleds med ett enkelt underscore "_" är privata
för klassen och ska inte användas av de som använder denna klass.

"""
class OneCell:
    def __init__(self):
        """
            Syfte: Skapar en ny cell utan definierat värde eller länk
            Parametrar: -
            Returvärde: -
            Kommentarer: I definitionen heter denna funktion Create
        """
        self._data = None
        self._link = None
```

```
    def setValue(self,data):
        """
            Syfte: Sätter cellens värde till data
            Parametrar:
            Returvärde:
            Kommentarer:
        """
        self._data = data

    def setLink(self,link):
        """
            Syfte: Sätter cellens länk till link
            Parametrar:
            Returvärde:
            Kommentarer:
        """
        self._link =link

    def inspectValue(self):
        """
            Syfte: Returnerar cellens värde
            Parametrar:
            Returvärde:
            Kommentarer:
        """
        return self._data

    def inspectLink(self):
        """
            Syfte: Returnerar cellens länk
            Parametrar:
            Returvärde:
            Kommentarer:
        """
        return self._link
```

.