DVA245 Lab Assignment 3
Linked List, Queue, Stack

Anna Friebe

January 2022

1 Introduction

In this lab, you will finalize the Linked List implementation from the Hubbard book `https://kentdlee.github.io/CS2Plus/build/html/index.html`, it is missing some features. Then you will use this linked list to implement a queue and a stack. Code skeletons and tests are in a zip-file on Canvas.

2 Linked List

With the book there is a linked list implementation that is not complete. The implementation can be found in the file `linkedlist_begin.py`, and tests in `linkedlist_test.py`. The Linked List implementation is described in more detail in chapter 4.10 of the book.

A linked list structure consists of nodes, where each node holds element data and a reference to the next node in the list. In the code of this lab, the list is defined as the `LinkedList` class. The `__Node` class is a *nested* class defined within the `LinkedList` scope. It has two data members, the `item` that holds the value of the node, and the `next` reference to the next node in the list. The linked list in the implementation holds references to the first and the last nodes in the list. The first node is a special dummy node, the one named head in the illustration in fig. 1. This never holds a data element, but is only there as a reference to the actual first node of the list, or `None` if the list is empty. This allows for simpler code with less special cases. The `LinkedList` class also stores the number of items in the list.

You will implement the remaining functionality of the linked list. The missing parts are:

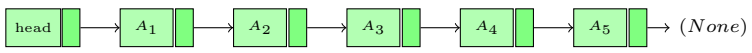

Figure 1: A singly linked list.

- `__iter__(self)`, that generates the items of each node in the list. An example of when this function is called on a `LinkedList` object instance `myList` is with the keyword `in` is:

  ```
  for i in myList:
  ```

  The implementation should be about 5 lines of code.

- `__contains__(self, item)`, that returns `True` if the input item equals the item of any node in the list, `False` otherwise. An example of when this function is called on a `LinkedList` object instance `myList` is with the keyword `in` is:

  ```
  if item in myList:
  ```

  The implementation should be about 5 lines of code.

- `__len__(self)`, that overloads the `len` operator and returns the number of items in the list. The implementation should be less than 5 lines of code.

- `__delitem__(self, index)`, used to overload `del` that removes the item at the list position given by index. When `__delitem__(self, index)` is implemented it can be used as:

  ```
  del myList[index]
  ```

  Implementation can be done starting from the pseudocode in algorithm 1.

You may want to implement `__iter__` before `__contains__`, as you can use the iteration in the implementation of `__contains__`. Recall the use of `yield` from the first lab.

The `__delitem__(self, index)` function is the most complicated. If you refer to fig. 1, when you remove the node A3 for example, you need to change the next reference of A2 to refer to A4. In addition our linked list holds a reference to the last node in the list ($self.last$), so if we delete A5, we need to update $self.last$ to refer to A4.

We provide pseudocode in algorithm 1. First, on lines 9-11, we check that the index is within the range of our list, and raise an IndexError exception if that is not the case. Then, on lines 12-13, we initialize cursor to the dummy head node, and delCursor to the actual first node of the list. On lines 14-17 we step forward to the node that should be deleted. The change of the next reference of the previous node (cursor) is done on line 19, and the update of the last reference on line 21. On line 23 the number of items in the list is decreased by 1.

Compared to the explanation in Chapter 4.10.7, and illustration in figure 4.12, the delCursor variable is added to the pseudocode. This makes the code for updating the next reference more readable, but if you want you can write the `__delitem__` function without the delCursor and next variables.

When you have implemented these four functions, the tests in `linkedlist_test.py` should pass.

Algorithm 1 Pseudocode for the `__delitem__(self, index)` function that deletes the list item at the position given by index.

```
1: function __DELITEM__(self, index)
2:     inputs:
3:         self, the self keyword referencing the linked list instance
4:         index indicates the position where a list item shall be deleted
5:     local variables:
6:         cursor, the previous list item
7:         delCursor, the list item to delete
8:         next, the next reference to update after deletion
9:     if index < 0 or index ≥ self.numItems then
10:        raise IndexError("Index out of range")
11:    end if
12:    cursor ← self.first                                  ▷ The dummy head node
13:    delCursor ← cursor.GETNEXT
14:    for i ← 0 to index do
15:        cursor ← delCursor
16:        delCursor ← cursor.GETNEXT
17:    end for
18:    next ← delCursor.GETNEXT
19:    cursor.SETNEXT(next)
20:    if delCursor = self.last then
21:        self.last ← cursor
22:    end if
23:    self.numItems ← self.numItems − 1
24: end function
```

3 Linked Queue and Stack

With the book there are implementations of a Stack and Queue that use python lists to store the elements. Here, you shall use the linked list implementation instead to implement a Stack and a Queue.

3.1 Linked Queue

In the file `queue_linked_begin.py`, you find a skeleton for your linked queue. You need to implement `enqueue`, `dequeue`, `front`, `isEmpty` and `clear`. By using the LinkedList functions, these can be short. Expect 1 line for `enqueue`, `isEmpty` and `clear`. The method `dequeue` and `front` can be up to 5 and 3 lines if you choose to raise exceptions with specific error messages if they are called when the queue is empty.

Since our linked list implementation stores a reference to the last element, it is efficient (constant time) to append an item at the end of the list. It is also efficient to remove the first item of the list, so the linked list is suitable for a queue implementation.

3.2 Linked Stack

In the file `stack_linked_begin.py`, you find a skeleton for your linked stack. You need to implement `pop`, `push`, `top`, `isEmpty` and `clear`.

To make the linked stack efficient we need to have the top of the stack at the first item of the linked list. We can efficiently (in constant time) add and remove items at the beginning of the linked list. We are storing a reference to the last item of the list, which makes adding items at the end of the list efficient. Removing items at the end of the list, however, requires that we traverse the entire list to find the new last reference. You need to implement the stack so that items are added and removed at the beginning of the linked list.

By using the LinkedList methods, the Stack implementation is short. Expect 1 line for `push`, `isEmpty` and `clear`. The methods `pop` and `top` can be up to 5 and 3 lines if you choose to raise exceptions with specific error messages if they are called when the stack is empty.

4 How to display

You need to run the tests in `linkedlist_test.py`, `linkedqueue_test.py` and `linkedstack_test.py`,and show that they all pass.