FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 8

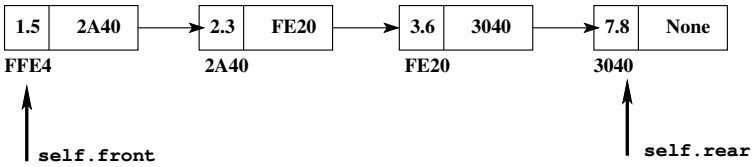
Semester 1, 2019

Objectives of this tutorial

- To understand programming with linked data structures.

Exercise 1 *

Consider the `Queue` class implemented with linked nodes provided in the lectures. Consider an object `my_queue = Queue()` of this class, to which we have appended four floats (1.5, 2.3, 3.6, and 7.8), and has the current form:



where the hexadecimal number under each node represents the address of the node. Without running the code (that is, by looking at it and tracing it by hand), write down the values of the following variables after executing `item = my_queue.serve()`:

- `my_queue.front`
- `my_queue.rear`
- `item`
- `my_queue.front.link`

Solution

Once the call `item = my_queue.serve()` has been executed:

- `my_queue.front`: would now point to the node object at address 2A40
- `my_queue.rear`: would have not changed (and thus point to the node at address 3040) since the `serve` method does not modify the rear.
- `item`: would have a reference to the float object with value 1.5
- `my_queue.front.next`: would now be a reference to the node object at address FE20

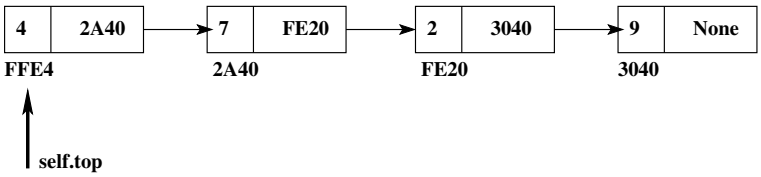
Exercise 2 *

Consider the `Stack` class implemented with linked nodes given in the lectures. Consider adding a method `sum_all` to the class whose function is to return the sum of all elements in the stack (zero if the stack is empty) without modifying the stack itself. The following shows three possible implementations of such method:

```
1  def sum_all(self):
2      sum = 0
3      while (not self.is_empty()):
4          sum += self.top.item
5          self.top = self.top.link
6      return sum
7
8  def sum_all(self):
9      current = self.top
10     sum = 0
11     while current.link is not None:
```

```
12          sum += current.item
13          current = current.link
14      return sum
15
16  def sum_all(self):
17      current = self.top
18      sum = 0
19      while current is not None:
20          current = current.link
21          sum += current.item
22      return sum
```

Consider an object `my_stack` with the form



Without running the code (that is, by looking at it and tracing it by hand), show the effect on the list of calling `result = my_stack.sum_all()` for each definition above. Show also the final value of `result`. Provide a correct definition for the above method.

Solution

The first implementation of `sum_all` does correctly compute the sum of all its elements (22), but it also modifies the stack by moving `self.top` along with each sum. Thus, at the end of the method the stack will be empty.

The second implementation fails to add all elements, in particular it will always miss the last node since it stops as soon as `(current.link == None)`. Further, if the list is empty the attempt to access `current.link` will give an exception. For our example stack, the second implementation returns 13 rather than 22.

The last implementation fails again to add all elements, in particular it always misses the first one since it keeps on moving `current` before accessing its data item. As a result, it will also give an exception when it reaches the end of the stack after adding 7 + 2 + 9, and it tries to access the data item of None.

A possible definition of `sum_all()` is as follows:

```
1  def sum_all(self):
2      current = self._top
3      sum = 0
4
5      while current is not None:
6          sum += current.item
7          current = current.link
8      return sum
```

Precondition: the stack elements should have and appropriate + method. Postcondition: the $sum = x_1 + ... + x_N$, where x_i is the i-th element in the stack and N is the length of the stack. The best and worst case for the above method are the same O(N)*O_{plus}, where N is the length of the stack and O_{plus} is the BigO time complexity of the + method, as the `sum_all` method as only one loop that is executed exactly N times, and all operations inside the loop, except possibly +, are constant. In many cases (e.g., integer elements, etc) O_{plus} will be O(1) and, then, the complexity of `sum_all` will be O(N).

Note that `sum_all` is a somewhat strange function for a `Stack` class, as users are only supposed to have access to the front element. It probably fits better in a `List` class.

Exercise 3 *

The following table compares the main advantages, for time efficiency, of using arrays (contiguous memory) and of using linked data structures for implementing data types. Some of these advantages are however only important for some, not all, data types.

Discuss which of these advantages (if any) are not important for implementing stacks.

Array (contiguous memory)	Linked nodes
Constant time access to any element	No shuffling required
Easy/fast traversal backwards	Easy/fast re-sizing (up and down)

Solution

The advantages that are not important for stacks are as follows.

- **Constant time access:** stacks only need operations (like `push`, `pop`, `peek`, `is_full`, `is_empty`) that do not require access to elements other than those at the top. Even if we want to implement things like calculating the length or converting to string, we will traverse all elements. Thus, there is no need for direct access to a specific element other than top.
- **Shuffling:** a stack will never need to shuffle its elements. Shuffling is only required when adding or deleting elements. The only time an element is added is when pushing a new top element. In this case, even if we have implemented the stack with an array, we will put the top element at the end of the array, so no shuffling required. The only time when an element is deleted is when popping the top element. Similarly, even if we have implemented the stack with an array, we will have the top element at the end of the array, so no shuffling required.
- **Easy/fast traversal backwards**: we only ever need to traverse the stack from top to bottom, and that is as easy to implement with linked lists as it is for arrays.

Therefore, the only advantage that is relevant for stacks is **Easy/fast resizing (up and down)**.

Exercise 4 *

Consider a queue data type provided by the `Queue` class which is implemented using some data structure (you do not need to know which one) and defines the following usual methods for queues:

```
serve()
append(new_item)
is_empty()
```

Write a Python function `def interleave(queue1, queue2)` that returns a new queue that interleaves the two queues in a fair way, i.e., the new queue has at the front the first element of `queue1` then the first element of `queue2`, then the second element of `queue1`, then that of `queue2`, and so on, until one of the queues is empty, in which case the rest of the elements in the other queue are appended at the end. For example, if `queue1` is (from front to rear) 20 -1 2 10 7 and `queue2` is 3 6 -4, then the new queue should be 20 3 -1 6 2 -4 10 7. If `queue1` is 20 -1 2 10 7 and `queue2` is empty, then the new queue should be 20 -1 2 10 7. And if both `queue1` and `queue2` are empty, the new queue is also empty.

Note that you are using the `Queue` class, not implementing it. Therefore, you have no idea (nor should you care) about how it has been implemented. Note also that you are only allowed to use the above methods to access/modify the queue (i.e., no other methods are allowed for queues). Also, you are allowed to modify the input queues `queue1` and `queue2`.

Solution

A possible solution is as follows:

```
1  def interleave(queue1, queue2):
2      out_queue = Queue()
3
4      while not queue1.is_empty() or not queue2.is_empty():
5          if not queue1.is_empty():
6              out_queue.append(queue1.serve())
7          if not queue2.is_empty():
8              out_queue.append(queue2.serve())
9
10     return out_queue
```

The above solution traverses both queues at the same time until both are empty. In each iteration it checks whether each queue is empty (as one of them might be empty already) and, if not, serves an element and appends it to the `out_queue`.

A possible alternative is:

```
def interleave(queue1, queue2):
    out_queue = Queue()

    while not queue1.is_empty() and not queue2.is_empty()):
        out_queue.append(queue1.serve())
        out_queue.append(queue2.serve())

    while not queue1.is_empty()):
        out_queue.append(queue1.serve())

    while not queue2.is_empty():
        out_queue.append(queue2.serve())

    return out_queue
```

The above solution traverses both queues at the same time until one of them is empty (first while loop). In each iteration of this loop it knows none of the queues is empty, so it can simply serve an element of each and append it to the `out_queue`. Once the first loop finishes one of the queues must be empty, but the other might still have elements in it. So it needs another two loops to serve the elements of the possibly non-empty queues and add them to the `out_queue`.

Both alternatives are OK. The code in the first one is shorter, while the second one is slightly more efficient (has less checks for `is_empty()`). Different people might find one easier to understand than the other one (I like the simplicity of the first version). They both have the same complexity: best and worse case is O(max(Q1,Q2)) where Q1 and Q2 are the lengths of the queues. This is because (a) the number of iterations performed by either the loop in the first version, or by the sum of the three loops in the second version, is max(Q1,Q2) no matter what the elements of the queues, and (b) in each iteration, the operations performed (`is_empty`, `append` and `serve`) are all constant in time.

Exercise 5

Show how to implement the Queue ADT using two stacks (and their corresponding ADT). What is the worst-case running time of the `serve` operation? What is the average running time?

Solution

The solution uses one stack in which to store appended items, and one to store the next items to be served.

```
class Queue:
    def __init__(self):
        incoming = Stack()
        outgoing = Stack()

    def is_empty(self):
        return self.incoming.is_empty() and self.outgoing.is_empty()

    def size(self):
        return self.incoming.size() + self.outgoing.size()

    def append(self, item):
        self.incoming.push(item)

    #we assume there is one item to serve
    def serve(self):
        if self.outgoing.is_empty():
            #we empty the "incoming" stack into the "outgoing" one
            while not self.incoming.is_empty():
                self.outgoing.push(self.incoming.pop())
        return self.outgoing.pop()
```

In the worst case, we call `serve` when `outgoing` is empty. Then we have n items to move from `ingoing` to `outgoing`, where n is the size of the queue. Hence the worst case is $O(n)$. However, each item ever served by the queue will only have been moved from `ingoing` to `outgoing` *once*. Indeed, if one serve operations moves n items from `ingoing` to `outgoing`, it costs $O(n)$ operations, but every subsequent $n-1$ serve will cost $O(1)$. Hence, n serve operations cost $\frac{O(n)+n*O(1)}{n} = O(1)$. This *amortised* analysis shows that the average complexity of the serve operation is $O(1)$.