Homework 6

Instructions: Please submit your solutions via Gradescope by **Friday, 4 March 2022, 10:00am**. Make sure your name, your class group number, and the name of your class teacher is put on **every page** of your submission. Your submission should, ideally, be a **PDF** file.

Exercise 6.1: Linked lists — 4 pts

Consider the implementation of a singly linked list LinkedList.py, discussed in the lecture and available from Moodle. This implementation just has a head pointer. It also comes with just two methods: `isEmpty(self)` and `add(self,item)`. The first one checks whether the linked list is empty; the latter one adds an item at the front of the linked list.

```
1   class Node:
2       def __init__(self,item,next):
3           self.item = item
4           self.next = next
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9
10      def isEmpty(self):
11          return self.head == None
12
13      def add(self,item):
14          temp = Node(item,self.head)
15          self.head = temp
```

(a) Explain in words which changes would be required to have a head and a tail pointer. Implement these changes in Python.

(b) Explain in words how to remove an element from the front of a singly linked list. Implement this method in Python. What is the time complexity of this method? Justify your answer.

(c) Explain how to remove the element at the end of the singly linked list. Implement this method in Python. What is the time complexity of this method? Justify your answer.

Exercise 6.2: Stacks and queues — 3 pts

(a) Explain how to implement a queue using two stacks. Analyse the running time of the queue operations enqueue and dequeue.

(b) Explain how to implement a stack using two queues. Analyse the running time of the stack operations pop and push.

Exercise 6.3: Hashing **3 pts**

Suppose you need to insert unique 3-character IDs into a hash table, where each ID is made up of some combination of two of the capital letters A-D, followed by one of the lower case letters x-z, such as: ABx, DCy, BBz, etc. Repeat letters are allowed in an ID.

(a) How many unique 3-character IDs are there?

(b) What is the smallest length of the `keys` list for which we can guarantee that no LinkedList in `keys` has length more than 2?

(c) Describe a hash function for this setting that guarantees that no key collides with more than one other key.

```
1    class Node:
2        def __init__(self,item,next):
3            self.item = item
4            self.next = next
5
6    class LinkedList:
7        def __init__(self):
8            self.head = None
9
10       def isEmpty(self):
11           return self.head == None
12
13       def add(self,item):
14           temp = Node(item,self.head)
15           self.head = temp
```

(a) Explain in words which changes would be required to have a head and a tail pointer. Implement these changes in Python.

(a) In the initialisation, create a pointer at tail

Also, update the add method to update the tail pointer if Linked List is empty.

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def empty(self):
        return self.head == None or self.tail == None

    def add(self, item):
        self.head = Node(item, self.head)
        if self.empty():
            self.tail = self.head
```

(b) Explain in words how to remove an element from the front of a singly linked list. Implement this method in Python. What is the time complexity of this method? Justify your answer.

(b) If the linked list is not empty, then

- get the current head (to return)
- move the head pointer 1 forward
- return the value to return
- This takes O(1) time as only constant time operations.

```
def removeFront(self):
    if not self.empty():
        front = self.head.item
        self.head = self.head.next

        # this works as my empty()
        # checks if head OR tail is None
        if self.empty():
            self.tail = None
        return front
```

(c) Explain how to remove the element at the end of the singly linked list. Implement this method in Python. What is the time complexity of this method? Justify your answer.

(c) ▷ If not empty

▷ If list has tail = head, return the head and adjust tail and head

▷ Else Creater an iterator variable that points at head

▷ Keep iterating pointer forward while its next is not the tail

▷ If next is tail, let node = tail and set pointer.next = None. Update the tail and return the node.

▷ Since the while loop must traverse the whole list this takes $\Theta(n)$ time always.

```
def removeEnd(self):
    if not self.empty():
        # one element list
        if self.head == self.tail:
            back = self.tail.item
            self.head, self.tail = None, None
        # if more than one element
        else:
            pointer = self.head
            while pointer.next != self.tail:
                pointer = pointer.next
            back = pointer.next.item
            pointer.next = None
            self.tail = pointer
        return back
```

Exercise 6.2: Stacks and queues **3 pts**

(a) Explain how to implement a queue using two stacks. Analyse the running time of the queue operations enqueue and dequeue.

(a) Enqueue: simply push onto stack 1

Dequeue: If stack 2 is empty, pop each element from stack 1 and push onto stack 2.

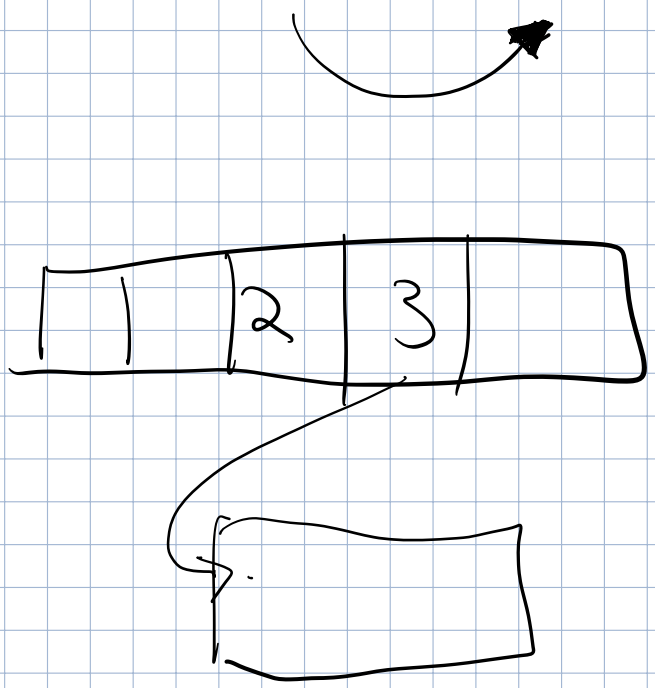
Now, pop from stack 2

In this case Enqueue is $\Theta(1)$ as pushing onto stack is constant time.

Dequeue in the worst case might require all the elements to be shifted from stack 1 to stack 2 requiring n pushes and n pops so $\Theta(n)$ worst case.

(b) Explain how to implement a stack using two queues. Analyse the running time of the stack operations pop and push.

Wasn't sure how to do this



Exercise 6.3: Hashing **3 pts**

Suppose you need to insert unique 3-character IDs into a hash table, where each ID is made up of some combination of two of the capital letters A-D, followed by one of the lower case letters x-z, such as: ABx, DCy, BBz, etc. Repeat letters are allowed in an ID.

(a) How many unique 3-character IDs are there?

(a) since order matters, 4 choices for first digit
4 choices for second and 3 choices for third

$\Rightarrow 4 \times 4 \times 3 = 48$ choices

(b) What is the smallest length of the keys list for which we can guarantee that no LinkedList in keys has length more than 2?

(b) In order for this to be possible we need load factor $\alpha \leq 2$

so $\frac{48}{m} \leq 2 \Rightarrow m \geq 24$. so at least 24 keys are required.

(c) Describe a hash function for this setting that guarantees that no key collides with more than one other key.

We could write out a specific hash function that maps

AAx → 1
AAy

and so on

AAz → 2
BAx

Not sure what the general rule is though