CSC148 Summer 2018: Final Exam Practice

This document contains practice questions covering topics that have appeared throughout the course. The focus of these questions are on material beyond the 2nd midterm, though the final exam is cumulative.

Class Design

A Food object has the following properties:

- A name
- A quantity (the number of items this Food object represents)
- An expiration date (in string format 'YYYY-MM-DD')
- The ___str___ method which should contain its name, quantity, and expiration date.
- The ___eq___ method which returns True if both items are Food objects that have the same name and expiration date, but not necessarily the same quantity.

A Storage object has the following properties:

- It has maximum a capacity (the number of items it can store)
- It can store items in it. The store() method should return True if the item was successfully stored, or False if the Storage was already at its maximum capacity.
- It has a remove() method which should be unimplemented.
- The ___str___ method should be able to print out the contents of the Storage.
- The ___eq___ method should return True if both items are Storages, and they both contain the same items (but not necessarily in the same order).

A Fridge is a type of Storage. However, it can only store Food items. Storing Foods with the same name and expiration date should add the quantity to the original Food in the fridge. Additionally, its remove() removes the Food with the earliest expiration date.

Implement the abstract Storage class and its Fridge subclass, as well as the class Food.

Stacks and Queues

A Priority Stack is a type of Stack where all items added into it are added with a priority. When removing from it, items are removed based on 1) whichever has the lowest priority and 2) if multiple items have the same priority, then the latest one added is the one removed.

A Priority Stack has the following methods:

- add(item, priority): Adds item to the priority stack with priority
- remove(): Removes and returns the newest item in the priority stack which has the highest priority (lowest priority number).
- is_empty(): Returns True iff the Priority Queue is empty.
- ___str___(): Prints out the contents of the Priority Stack with their priorities.

A Priority Stack uses the self._content attribute which is set to be an empty Stack initially. You may not assume anything about how the Stack stores its contents -- just that it has the add(), remove() and is_empty() methods that you should be familiar with.

Below is the class header and ___init___ for Priority Stack. Implement the rest of it. Make sure you include docstrings and type annotations. You do not need to write docstring examples.

```
class PriorityStack(Stack):
    """
    A class representing a PriorityStack.
    """

    def __init__(self) -> None:
        """
        Initialize this PriorityStack.
        >>> p = PriorityStack()
        >>> p.add("Yes", 2)
        >>> p.add("Maybe", 1)
        >>> p.add("No", 2)
        >>> p.remove()
        'Maybe'
        >>> p.remove()
        'No'
        """
        self._content = Stack()
```

Linked Lists

Implement a LinkedList method called remove_every() takes takes in a value and removes all occurrences of a value from it. For example, if a LinkedList looks like:
A -> B -> C -> B -> A -> |

And we wanted to remove all the As, it should look like:
B -> C -> B -> |

The front, size, and back attributes should all be adjusted accordingly.

Recursion

Read the documentation of the function **get_most_common** which takes in either a non-list element or a potentially nested list, and returns the element which occurs most often. You may want to write a helper function for this.

```
def get_most_common(lst: Any) -> Any:
    """
    Return the element that occurs in lst most often.

    >>> get_most_common(1)
    1
    >>> get_most_common([1, 2, 1])
    1
    >>> get_most_common([[1, 2], [[1], 2], [1]])
    1
    """
```

Trees

Read the documentation of the Tree method **replace_all_occurrences** and implement its body.

```
def replace_all_occurrences(self, to_replace: Any, replace_with: Any) -> None:
    """
    Replace all instances of the value to_replace in this Tree
    with the value replace_with.

    >>> t = Tree(5, [Tree(5), Tree(2), Tree(5)])
    >>> t.replace_all_occurrences(5, 1)
    >>> print(t)
        5
    1   2   1
    """
```

Binary Trees

Read the documentation of the function **switch_all_children** and implement its body.

```
def switch_all_children(t: Union[BinaryTree, None]) -> None:
    """
    Switch all of the left trees in t with the right trees.

    >>> t = Tree(1, Tree(2, Tree(3)), Tree(4, Tree(5), Tree(6, Tree(7))))
    >>> print(t)
         1
      2     4
    3     5    6
              7
    >>> switch_all_children(t)
    >>> print(t)
            1
        4       2
    6      5       3
      7
    """
```

Binary Search Trees

1. What is the difference between a Binary Tree and a Binary Search Tree?
2. If a Binary Search Tree has n elements in it, what's the worst its height could be? What's the best its height could be?
3. Give an example of an unbalanced Binary Search Tree.
4. Give an example of a balanced Binary Search tree.
5. Read the documentation of the function **delete_children_less_than** and implement its body. Assume the BinarySearchTree contains only int values.

```
def delete_children_less_than(t: Union[BinarySearchTree, None],
                              value: int) -> Union[BinarySearchTree, None]:
    """
    Remove all values less that value from t. All values in t that are > value
    should still be in t. Return the root of this BinarySearchTree or None
    if no such BinarySearchTree exists.

    >>> t = BinarySearchTree(5, BinarySearchTree(3, BinarySearchTree(1),
    ...                                             BinarySearchTree(4)),
    ...                         BinarySearchTree(6))
```

```
>>> print(t)
       5
  3       6
1   4
>>> delete_children_less_than(t, 4)
   5
4     6
"""
```

Runtime Complexity

1. Put the following runtimes in order from fastest to slowest:
   O(lgn) O(n) O(1) O(2^n) O(n^2) O(lgn)
2. Give an example of O(lgn) runtime. Why does it run in O(lgn) time?
3. To count the number of items in a list, we would have to visit each item once. What would the runtime of this be?
   a. Python's len() function gets the length of a list in O(1) time. How might this happen?

      Hint: Consider how LinkedLists work, and the runtime for us to return the size of a LinkedList.
4. If we were trying to create a copy of a list using a for-loop and append() (which takes O(1) time), what would the runtime be?
   a. What if we wanted to create a slice of a list of size k?
   b. If we were making a slice, what would the worst-case scenario be? What would be its runtime?

   As an aside: Python's [:] slicing operations also take the runtime specified and use the same approach.
5. Suppose we have the following runtimes:

Input size (n)	Runtime (seconds)
1	30.0
2	40.0
3	50.0
4	60.0

   What runtime complexity best fits the runtimes above?
6. Suppose we have the following runtimes:

Input size (n)	Runtime (seconds)
1	0.004
2	0.005
4	0.006
8	0.007

What runtime complexity best fits the runtimes above?

7. Suppose we have the following runtimes:

Input size (n)	Runtime (seconds)
1	7.0
2	11.0
4	23.0
8	55.0

What runtime complexity best fits the runtimes above?

8. Suppose we have the following runtimes:

Input size (n)	Runtime (seconds)
1	40.0
2	70.0
3	120.0
4	190.0

What runtime complexity best fits the runtimes above?

9. Suppose we have the following runtimes:

Input size (n)	Runtime (seconds)
1	8.0
2	14.0
3	26.0
4	50.0

What runtime complexity best fits the runtimes above?

10. Suppose we have the following code:

    ```
    def mystery(n):
        if n == 0:
            return 1
        return n + mystery(n - 1)
    ```

    What is the worst-case scenario for this code? What's the worst-case runtime relative to n? What about best-case?

11. Suppose we have the following code:

    ```
    def mystery(n):
        if n == 0:
            return 1
        return mystery(n - 1) + mystery(n - 2)
    ```

    What is the worst-case scenario for this code? What's the worst-case runtime relative to n? What about best-case?

12. Suppose we have the following code:

    ```
    def mystery(n):
        if n == 0:
            return 1
        return mystery(n // 2)
    ```

    What is the worst-case scenario for this code? What's the worst-case runtime relative to n? What about best-case?

13. Suppose we have the following code:

    ```
    def mystery(n):
        if n == 0:
            return 1
        return mystery(n // 2) + mystery(n // 2)
    ```

    What is the worst-case scenario for this code? What's the worst-case runtime relative to n? What about best-case?

Merge Sort and Quick Sort

1. What do we mean by "merge" for merge sort? What do we merge?
2. What do we mean by "pivot" for quick sort? How do we use our pivot?
3. What is the worst-case runtime for merge sort? Best case? When do these occur?
4. What is the worst-case runtime for quick sort? Best case? When do these occur?
5. Construct a list for quick sort which will take O(n) time to run if the pivot chosen is always the last item in the list.

6. Construct a list for quick sort which will take O(nlgn) time to run if the pivot chosen is always the last item in the list.

Hash Tables

1. For a hash table that's completely new, what's the runtime of inserting an item? Getting an item? Why?
2. If we used chaining, what would the worst-case runtime be for inserting an item to a hash table with n elements (if we don't have to worry about expanding the hash table)?
    a. What if we have to expand the hash-table and re-add everything?
3. If we used probing, what would be the worst-case runtime for inserting an item into a hash table with n elements?
4. Suppose we have the following hash values:
   hash(1) == 3
   hash(2) == 6
   hash(3) == 2
   hash(4) == 0
   hash(5) == 5

   Suppose we have a hash table of size 2 initially, which uses chaining. When a bucket has size == the size of our hash table, we double the size of the hash table and re-insert everything.

   Suppose we try to insert the following (key, value) pairs:
   (1, 'a')
   (2, 'b')
   (3, 'c')
   (4, 'd')
   (5, 'e')

   What does the hash table look like after each addition?
5. Suppose we have the following hash values:
   hash(1) == 3
   hash(2) == 6
   hash(3) == 2
   hash(4) == 0
   hash(5) == 5

   Suppose we have a hash table of size 2 initially, which uses a probing rule which looks at (the initial index from hashing % the size of our list) * 2. If that position is taken, then it multiplies that index by 2 until it goes beyond the list's index. If we reach the end of the list without finding a valid index to add to, we double the size of the hash table and re-insert everything.

   Suppose we try to insert the following (key, value) pairs:
   (1, 'a')
   (2, 'b')
   (3, 'c')
   (4, 'd')
   (5, 'e')

What does the hash table look like after each addition?