COMP_SCI 214 (Practice Exam for Winter 2024)
Exam 1

Your full name: ______________________________

NetID: ______________________________

- Write your name and NetID clearly on this page and write your netID on all other pages.
- Read the instructions for each question in full before attempting them. The questions and the appendix contain **all** the information you'll need.
- Write your answers in the box provided for each problem. Feel free to use any other empty space for doodling or as scratch paper, but your answer should be clearly indicated inside the box. If any part of your answer is contained outside the box (please try to avoid this), indicate it very clearly.
- This exam is closed book, no notes, no electronics, no nothing. Just you and your pencil.
- To maintain fairness and avoid disruptions, no questions will be taken during the exam; answer the questions based only on the information given (see bullet 2).

Problem 1 (4 points)	
Problem 2 (5 points)	
Problem 3 (6 points)	
Problem 4 (3 points)	
Problem 5 (2 points)	
Total (20 points)	

- A total of 17 points or above earns a **positive** modifier.
- A total between 11-16 earns a **neutral** modifier.
- A total between 8-10 points earns a single **negative** modifier.
- A total between 4-7 points earns two **negative** modifiers.
- A total between 0-3 points earns three **negative** modifiers.

Disclaimer: This number is **not** a measure of your worth as a human being; it is merely an (imperfect) measure of your mastery of the material in (the first half of) this class. Don't lose track of the big picture.

Problem 1 Analyzing tradeoffs (____ / 4)

In the real world of building programs, we will often have to make decisions about how to represent data and operate on this data. You will get practice with such decision-making in this problem. For each of the following scenarios, select which algorithm, data structure, or ADT is the best choice given the situation and constraints (if any) and why.

There is one correct answer for each question, however, the answer will only receive credit if it contains a clear and meaningful justification. This justification should include specific and accurate details about time and/or space complexity (always written in their tightest form), properties of algorithms/structures, and the purpose of the structure/algorithm where needed.

i. (1 pts) You need to sort a set of about one million elements and run this sort on a device with very little storage or memory space. You only need to run the sort infrequently (once every few months or so). Which sort would you use and why?

Choose between: (1) Selection sort (2) Merge sort

1) Selection sort because though it has more complexity O(n^2), it is in-place which is necessary for this low memory device. The high complexity is okay because this sort will be run very infrequently. Merge sort has lower complexity O(n log n) but needs a lot of extra memory which the device can't handle.

ii. (1 pts) You are implementing a dictionary ADT, which you can assume you need for three operations: (1) put entries in the dictionary; (2) look for the latest entry added to the dictionary (irrespective of key value); and (3) look up any key in the dictionary. You foresee using the first two operations very frequently while only rarely using the third operation. The key or value does not contain information about when the entry is added to the dictionary. Which concrete data structure would you use to implement the dictionary and why?

Choose between: (1) Array of (key, value) pairs sorted by key (2) Linked list of (key, value) pairs where new entries are added to the front

2) Linked list. Putting entries and looking for the latest entry added are the priority, so only focus on complexities for that operation. There is no way to find the latest entry in a sorted array as it is sorted by key and putting would be O(n) since you'd have to insert in right spot. In this linked list, it would be O(1) since it's at the front

iii. (1 pts) You plan to use an implementation of a graph ADT (any version) to represent a graph with 500 vertices and 20 edges. Your only concern is that you use space/memory as efficiently as possible, and not the runtime of specific operations. Which concrete representation would you use to implement the graph and why?

Choose between: (1) Adjacency matrix (2) Adjacency list

2) The graph is sparse, edges are in O(v). An adjacency list only stores space for what's part of the graph, so 520 spots. A matrix would store 250,000 spots which is way more than what's needed, so it's inefficient.

iv. (1 pts) You want to use an ADT to represent a collection of browser history visits. You have one specific purpose in mind for storing information about these browser visits: you want to be able to store and fetch the number of times a webpage was visited for a given URL (the address of a webpage). What ADT would you use to represent these browser visits and why?

Choose between: (1) Dictionary (2) Stack

1) Dictionary. The goal of the ADT seems to be about storing a browser link against a number, which resembles a dictionary that stores a value against a key. A stack would only allow getting the latest added element and not looking up by specific data.

Problem 2 Deluxe queues (____ / 5)

You've seen queues being used for a music player in your assignment. For this, you used a basic queue interface, which only allows going forward in the queue. However, most music players we use in real life allow us to go back and forth between tracks in the queue. To enable this, we extend the queue interface we saw in class to have an additional function:

```
def requeue(self)
```

This extended queue interface can be found in Appendix A.1.

The call `q.requeue()` moves the front of the queue `q` to be the element that was previously dequeued. For example, if initially `q` is the queue `< 1 2 3 4 5 <` where `1` is at the front and `5` is at the back, a `q.dequeue()` would result in the queue `< 2 3 4 5 <`. After a subsequent call to `q.requeue()`, `q` would then look like `< 1 2 3 4 5 <`. When the front is already the very first element that was enqueued, `q.requeue()` has no effect. When the queue is empty, `q.requeue()` triggers an error.

To implement this queue, we use a doubly-linked list (DLL) where each node has a pointer to the node before it and to the node after it. The following are important invariants of the DLL (in addition to the relevant linked list ones you saw in class) for queues: (1) each node has its `prev` field set to the node before it in the queue, or `None` only if it's the **original** front of the queue; (2) each node has its `next` field set to the node after it in the queue, or `None` if it's the end; (3) the `head` field of the DLL holds the current front of the queue; and (4) the `tail` field of the DLL holds the current back of the queue.

The DLL implementation for this queue in DSSL2 is similar to the singly-linked list you implemented in your assignment, with a few changes. One of the changes is that the `_cons` struct contains an additional pointer to the node before it. The other changes are to the `enqueue` and `dequeue` functions.

The change to the `_cons` struct and an incomplete version of the implementing class (`QueueDLL`) with the above changes are provided in Appendix A.3.

i. (2 pts) Complete the definition of the `requeue` function in the above implementation using the DLL and written in DSSL2. Your implementation must satisfy the above four invariants of the DLL at a minimum and have $O(1)$ complexity for it to be correct. This function should only "requeue" if the queue is not empty; the function should trigger an error otherwise.

```
def requeue(self):
    if self.empty?():
        error("empty queue")
    if self.head.prev != None:
        self.head = self.head.prev
```

ii. (3 pts) You are a client using the `QueueDLL` implementation for a music player. Your queue is stored in a variable `q`, an object of the `QueueDLL` class. The type `T` for your queue is a string data type just containing the name of the song. Your music queue `q` looks like this: `< "Everyday", "Hello", "Myself", "Hero", "Numb"<` where `"Everyday"` is at the front of the queue.

Once a song is dequeued and returned, you can simply call a function `play` to "play" the dequeued song by passing in the song name (e.g., `play("arcade")`); this function is standalone and not part of any class definition. Write a series of DSSL2 statements using only the `requeue` and `dequeue` operations on `q` and the `play(<song>)` function such that only the following three songs would "play" and in this exact order: `"Hero"`, `"Myself"`, and `"Hello"`.

```
q.dequeue()
q.dequeue()
q.dequeue()
play(q.dequeue())
q.requeue()
q.requeue()
play(q.dequeue())
q.requeue()
q.requeue()
play(q.dequeue())
```

Problem 3 Stacks from queues (____ / 6)

Consider the same interface for the deluxe queue from Problem 2 (provided in Appendix A.1). There you'll find another additional function:

```
def queue_reverse(self)   # O(1)
```

The call `q.queue_reverse()` reverses the queue `q` in constant time. For example, if initially `q` is the queue `< 1 2 3 4 5 <` where `1` is at the front and `5` is at the back, after a call to `q.queue_reverse()`, `q` will be the queue `< 5 4 3 2 1 <` whose front is `5` and back `1`. You do not need to worry about how this function is implemented, just the above expected behavior.

Reversing a queue enables us to insert or remove elements at either end. With this insight, we can use reversible queues to implement a *stack* ADT whose interface is recalled in Appendix A.2. Here's the start of this implementation:

```
class StackWithRevQueues[T] (STACK):
     # 'q' is an implementation of the QUEUE interface
     let q: QUEUE!

     def __init__(self):
        self.q = QueueDLL[T]()

     def empty?(self):
        return q.empty?()
```

i. (3 pts) Complete the implementation of `push` for the `StackWithRevQueues` class using only `q` (hint: keep in mind access rules for classes.). Before and after the execution of the function, `q` should be the original in-order queue (with the addition of the pushed element).
This operation should have $O(1)$ complexity.

```
def push(self, element: T) -> NoneC:

     self.q.enqueue(element)
```

ii. (3 pts) Complete the implementation of `pop` for the `StackWithRevQueues` class using only `q` (hint: keep in mind access rules for classes). Before and after the execution of the function, `q` should be the original in-order queue (without the popped element).
This operation should have $O(1)$ complexity.

```
def pop(self) -> T:
    self.q.reverse_queue()
    let x = self.q.dequeue()
    self.q.reverse_queue()
    return x
```

Note: functionality in pop and push can be flipped so that push does the reversing and pop just dequeues

Problem 4 Hash tables (___ / 3)

You have a hash table that has a capacity of 6, i.e., it can hold six (key, value) pairs. The table below depicts this hash table where the top column contains the indices of the buckets. This table resolves collisions via **linear probing**.

0	1	2	3	4	5

You would like to use the above hash table to implement a dictionary with names as the keys. You plan to enter the following keys in order into this hash table: `"bhagavatula"`, `"rasmussen"`, `"pineda"`, `"kapur"`, `"shin"`, `"sharaf"` (the values are irrelevant and hence, omitted).

You will be using the hash function `strlen()` which computes the number of characters in a string. For example, `strlen("sruti")` returns 5. You should determine the bucket from the hash code using the approach discussed in class and which you used in your homework.

i. (1 pts) How many of the keys can be inserted into the hash table **before** a collision occurs?

3

ii. (1 pts) What key is at index 3 after putting all keys in the hash table and resolving collisions?

Rasmussen

iii. (1 pts) What key is at index 2 after putting all keys in the hash table and resolving collisions?

Sharaf

Problem 5 Social network graphs (____ / 2)

You are building a basic social network, where users can follow users and in turn be followed by others. You plan to represent this as a **weighted directed graph** where each vertex represents one user, an edge from one user to another indicates the former following the latter, and the weight of each edge indicates the absolute difference in number of followers between the two users ($|\#followers_{from} - \#followers_{to}|$).

The following are users in the graph and their followers. Each item below is of the form: `{followers of <username>} <username>`.

- {ron, voldy, dumbly, hermy} harry
- {ron, harry} hermy
- {harry, hermy} ron
- {} voldy
- {harry, hermy, ron, voldy} dumbly

You have access to this dictionary mapping vertex #s to usernames:
`{0: "harry", 1: "dumbly", 2: "hermy", 3: "ron", 4: "voldy"}`

i. (2 pts) Represent the above users, their connections, and mutual followers with an **adjacency matrix**. The vertices should be natural numbers (0 or higher) as we saw in class; you should use the above dictionary to map vertex numbers to the users. Your answer should be a $|V|$-by-$|V|$ matrix where you clearly label the indices of each row and column and the contents of the matrix are filled according to the above details. If an edge does not exist in the graph, just leave the cell in the matrix blank.

On last page after appendix

This page intentionally left blank for scratch work, doodles, musings, or jokes.

You should detach this and the following pages for reference and to use as scratch paper. In doing so, please make sure not to destroy your exam.

A Appendix

A.1 Problems 2 and 3: Deluxe queue interface

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> NoneC
    def dequeue(self) -> T
    def empty?(self) -> bool?
    def requeue(self) -> NoneC  # For problem 2 only
    def reverse_queue(self) -> NoneC  # For problem 3 only
```

A.2 Problem 3: Stack interface

```
interface STACK[T]:
    def push(self, element: T) -> NoneC
    def pop(self) -> T
    def empty?(self) -> bool?
```

A.3 Problem 2: Incomplete DLL queue implementation

```
struct _cons:
     let data
     let prev: OrC(_cons?, NoneC)
     let next: OrC(_cons?, NoneC)

class QueueDLL[T] (QUEUE):  # implements queue from A.1
     let head
     let tail

     def __init__(self):
        self.head = None
        self.tail = None

     def enqueue(self, element: T):
        let new = _cons(element, None, None)
        if self.empty?():
            self.head = new
            self.tail = new
        else:
            new.prev = self.tail
            self.tail.next = new
            self.tail = new

     def dequeue(self):
        if self.empty?():
            error('empty queue')
        let res = self.head.data
        self.head = self.head.next
        # We don't overwrite 'prev' so that we can go back in 'requeue'
        if self.empty?():
            self.tail = None
        return res

     def empty?(self):
        return self.head is None

     def requeue(self):
        # TODO

     def reverse_queue(self):
        # Not shown; O(1) operation
```

Extra scratch paper

Extra scratch paper

	harry 0	dumbly 1	hermy 2	ron 3	voldy 4
harry 0		0	2	2	
dumbly 1	0				
hermy 2	2	2		0	
ron 3	2	2	0		
voldy 4	4	4			