COMP_SCI 214 (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 clarifying questions will be taken during the exam**; answer the questions based only on the information given (see second bullet).

Problem 1 (3 points)	
Problem 2 (4 points)	
Problem 3 (6 points)	
Problem 4 (7 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 Short-Answer Questions (____ / 3)

i. (2 pts) You need to do k searches in an array of n elements. This array is *not* sorted. You could either use linear search each time, or you could sort the array once (*using the fastest of the three sorting algorithms we saw in class*) and then use binary search for all k searches. For each of the following values of k, specify which approach(es) give(s) you the best worst-case asymptotic runtime overall (the total cost of all k searches). Check all that apply.

- $k = 42$ ☐ linear search ☐ first sort and then binary search
- $k = \log n$ ☐ linear search ☐ first sort and then binary search
- $k = n$ ☐ linear search ☐ first sort and then binary search
- $k = n \log n$ ☐ linear search ☐ first sort and then binary search

ii. (1 pts) You are planning to implement a dictionary ADT that allows duplicate keys with a concrete data structure. You hope to use this dictionary to maintain information about photo filenames (key) and when they were created (value), and foresee adding about 1000 photos in a year but only look for specific photos once every few months. Which of these two concrete data structures would you choose to implement it? (1) Unsorted array of (key, value) pairs; or (2) Array of (key, value) pairs sorted by key? Explain your answer by including specific details about the ADT, data structures, or complexities as applicable.

Problem 2 Analyzing an Algorithm (____ / 4)

Consider the DSSL2 implementation of an algorithm below that takes in an array as input.

```
1  def algorithm(array):
2      let swapped
3      for i in range(array.len()-1):
4          swapped = False
5          for j in range(array.len()-i-1):
6              if array[j] > array[j + 1]:
7                  # O(1) operation to swap values at array[j] and array[j+1]
8                  swap(array[j], array[j + 1])
9                  swapped = True
10
11         # If no two elements were swapped by inner loop
12         if not swapped:
13             break # Exit out of outer "for" loop on line 3
14     return array
```

i. **(1 pts)** What is the worst-case time complexity (in its tightest bound) of this algorithm?

ii. **(2 pts)** What is the best-case time complexity (in its tightest bound) of this algorithm? What is an example of an input array of size 4 that would exhibit this best-case runtime?

iii. **(1 pts)** Name an algorithm we saw in class that can achieve the same goal as the above algorithm.

Problem 3 Stacks and Duplicates (____ / 6)

In this problem, you'll use the stack interface provided in Appendix A.1 to implement functions that determine whether a stack contains duplicate elements.

i. (2 pts) Implement the *recursive* function `stack_contains`, which should return `True` only when the given stack contains the given element, and `False` otherwise. After this function returns, the stack should be in the same state (i.e., contain all the same elements in the same order) as it did before.

```
def stack_contains(s: STACK!, x) -> bool?:
    if s.empty?(): # Nothing to do
        return False
    let temp = s.pop()
    if temp == x: # Found!
        ___________________________ # Restore the stack
        return True
    # Recursive case
    let is_contained = stack_contains(______, ______)
    ___________________________ # Restore the stack
    return is_contained
```

ii. (2 pts) Having access to the `stack_contains` function lets us implement a function `stack_has_dupes`, which returns `True` when the stack contains duplicate elements and `False` otherwise. Again, it should leave the stack with the same elements in the same order.

```
def stack_has_dupes(s: STACK!) -> bool?:
    # Create a new temporary stack using
    # LL implementation
    let temp_s = ListStack()
    let dupes_exist = False
    # Go through the stack 's' and check for duplicates
    while ______________________:
        let temp = ______________________
        if ______________________:
            dupes_exist = True
        temp_s.push(temp)
    # Restore the original stack from the temp stack
    while not temp_s.empty?():
        ______________________
    return dupes_exist
```

iii. (1 pts) What is the time complexity of `stack_contains` for a stack that initially contains n elements?

iv. (1 pts) What is the time complexity of `stack_has_dupes` for a stack that initially contains n elements?

Problem 4 Least Recently Used Cache (____ / 7)

Computer systems maintain something called a "cache", which is simply a collection of data elements that can be quickly accessed (i.e., "looked up") for future use. In this problem, you will partially implement a type of cache called a **Least Recently Used (LRU)** cache, which can only hold a fixed number of data elements, which is it's "capacity".

If a data lookup into the cache is successful, a **"cache hit"** occurs and the data is accessed. If it is not successful, a **"cache miss"** occurs and the data is inserted into the cache. If the cache has already reached its capacity, the element that was accessed or inserted the longest time ago (i.e., the least recently used element) in the cache gets removed; the new data is then inserted in the cache. All operations are expected to have $O(1)$ **complexity**.

An LRU cache is represented by two concrete data structures: a **doubly-linked list (DLL)** and a **hash table**. Every piece of data in the cache is stored in the doubly-linked list (DLL), where each node contains the data and has a pointer to the node before it and to the node after it. A DLL is accessible by its head and tail nodes. In our implementation, each node is a `_cons` object, whose modified definition is given in Appendix A.2. This DLL allows easy access to the head and tail, however, if an element in the middle needs to be accessed, it would be an $O(n)$ operation. Therefore, a hash table is also used to map the data of each element in the cache (**key**) to the actual `_cons` node in the DLL which holds that data (**value**).

The combined data structure has five important invariants: (1) each node has its `prev` field set to the node before it in the list, or `None` only if it's the head of the list; (2) each node has its `next` field set to the node after it in the list, or `None` if it's the end; (3) the elements from the head to the tail are ordered from most recently used (either accessed or inserted) element to least recently used element; (4) the DLL should only have a number of elements less than or equal to its specified capacity; and (5) any key that exists in the hash table should be contained as data in one node of the DLL and vice-versa.

Below is an incomplete class that implements an LRU cache. The `_ht` field is an instance of a hash table implementing the `DICT` interface given in Appendix A.3; it uses **linear probing** to resolve collisions and should have enough buckets to hold all cache elements when at maximum capacity.

```
class LRUCache:
    let _ht: DICT! # Hash table object that implements DICT
    let _head: OrC(_cons?, NoneC) # Head node of the DLL (initially None)
    let _tail: OrC(_cons?, NoneC) # Tail node of the DLL (initially None)
    let _length: nat? # No. of elements actually in the cache
    let _cap: nat? # No. of allowed cache elements (> 0)

    def _remove_lru(self):
         ... TODO ...

    def cache_access(self, data):
        if (self._ht.mem?(data)): # Cache hit
            ... TODO ...
        else:                     # Cache miss
            self._remove_lru()
            # Not shown: code that inserts new data at head of the
            # list and inserts this new node into the hash table
```

i. (2 pts) Draw a boxes-and-arrows diagram (i.e., the diagrams we have been using in class) of an example LRU cache representing the class fields and the data structures. The cache should have exactly 3 elements with a capacity for 4 elements.

We will not be sticklers for object names and fields, as long as their purpose is clear.

ii. (3 pts) Complete the missing code inside the "Cache hit" case of the `cache_access` function (which takes in `data` as an argument) with the correct functionality: the data accessed now becomes the most recently used data. By the end of this block of code, all aforementioned invariants should be preserved.

```
if (self._ht.mem?(data)): # Cache hit
    # Get the node in the DLL that has 'data'
    let cache_data = ______________________________

    if self._length > 1 and _cons?(cache_data.prev):
        # Remove 'cache_data' from its original
        # spot in the DLL
        cache_data.prev.next = cache_data.next

        # 'cache_data' now needs to be
        # the most recently used data
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________

    # Return accessed element
    return cache_data.data
```

iii. (2 pts) Complete the missing code inside the `_remove_lru` function to remove the least recently used element from the cache. By the end of this function, all aforementioned invariants should be preserved.

```
def _remove_lru(self):
    # Remove LRU element if cache is at capacity
    if self._length == self._cap:
        # Update hash table
        ------------------------------------

        # If the cache only allowed
        # storage of one element
        if self._cap == 1:
            self._head =
                ---------------------------------------
            self._tail =
                ---------------------------------------
        else:
            ------------------------------------
            ------------------------------------
        self._length = self._length - 1
```

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

Remove this and the following pages to use as a reference and as scratch paper. In doing so, please make sure not to destroy your exam.

A Appendix

A.1 Problem 3: Stack interface

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

A.2 Problem 4: `_cons` struct definition

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

A.3 Problem 4: Dictionary interface

```
interface DICT[K, V]:
    def len(self) -> nat?
    def mem?(self, key: K) -> bool?
    def get(self, key: K) -> V
    # 'put' modifies the dictionary to associate the
    # given key and value. If the
    # key already exists, its value is replaced.
    def put(self, key: K, value: V) -> NoneC
    def del(self, key: K) -> NoneC
```

A.4 Problem 4: Incomplete `LRUCache` class

The incomplete `LRUCache` class from Problem 4 is provided here again for your convenience.

```
class LRUCache:
    let _ht: DICT! # Hash table object that implements DICT
    let _head: OrC(_cons?, NoneC) # Head node of the DLL (initially None)
    let _tail: OrC(_cons?, NoneC) # Tail node of the DLL (initially None)
    let _length: nat? # No. of elements actually in the cache
    let _cap: nat? # No. of allowed cache elements (> 0)

    def _remove_lru(self):
         ... TODO ...

    def cache_access(self, data):
        if (self._ht.mem?(data)): # Cache hit
            ... TODO ...
        else:                     # Cache miss
            self._remove_lru()
            # Not shown: code that inserts new data at head of the
            # list and inserts this new node into the hash table
```

Extra scratch paper

Extra scratch paper