COMP_SCI 214 (Winter 2024)
Exam 2

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 (6 points)	
Problem 2 (5 points)	
Problem 3 (6 points)	
Problem 4 (3 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 second half of) this class. Don't lose track of the big picture.

Problem 1 What Algorithm Should You Use? (6 points)

For each of the following goals, indicate (1) which of these four algorithms you would use: Depth-First Search (DFS), Breadth-First Search (BFS), Kruskal's Algorithm, or Dijkstra's algorithm; and (2) why that algorithm is specifically suited for the goal (in terms of specific properties of the algorithm).

i. (2 pts) Say that everyone at Northwestern University is connected through a social network graph where vertices are users and an edge between two vertices exists if they have corresponded over email.
Goal: You are a student in the Computer Science department and would like to get in touch with a specific faculty member in the History department to discuss a possible collaboration. However, the history department is rather unrelated to the computer science department. Therefore, rather than cold email someone, you want to see if you can get introduced to that faculty member via your connections in the network (e.g., asking someone to introduce you to someone who can introduce you to someone else and so on), while going through as few connections as possible.

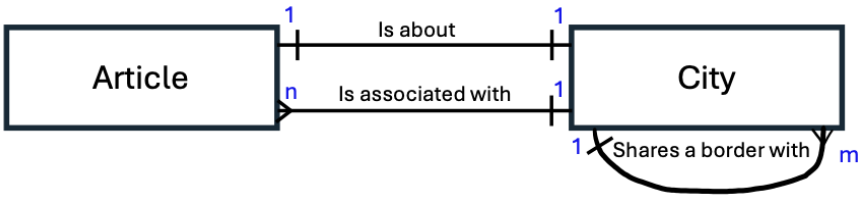
ii. (2 pts) A snowstorm has hit Evanston and the roads need to be plowed and cleared; cars and bikes can then travel around town using these plowed roads. Unfortunately, there are too many roads in Evanston and not enough plowing trucks.
Goal: You are in charge of prioritizing which roads need to be plowed so that everyone can get to everywhere they need to go, while knowing that some people may end up taking a longer route if they stick to only plowed roads.

iii. (2 pts) You are building a website to optimize flight paths, where a flight path from a starting location to a destination may have zero or more stops or layovers. One leg of a flight path corresponds to the direct flight from one location to another. Each leg is associated with a monetary cost. An optimal flight path minimizes the total cost across all the legs.
Goal: For a given start and destination, your site finds the most cost-effective flight path.

Problem 2 Serving Relevant News (5 points)

You are building a news website that serves relevant news articles to users based on the city in which they are located. For a specific city, "relevant" news articles include articles *associated with* that city and articles *associated with* all its neighboring cities (i.e., cities with which it shares a border). "Chicago" and "Evanston" are examples of neighboring cities, as they share a border. The ultimate goal of the website is to display a collection of "relevant" news articles for a given city.

Following the data design recipe we saw in class, the entities are: "article" and "city". Here is an Entity-Relation Diagram. The directions of arrows are: city-to-article (1:n), article-to-city (1:1), and city-to-city (1:m).



i. (1 pts) What ADT would you use to represent the city-to-article and article-to-city relations in the diagram? In other words, how would you store all the articles across all cities such that every article is associated with one city? Assuming each article is of the type `Article`, for each component of the ADT (e.g., nodes, edges, and weights for graphs), tell us what kind of data (e.g., "number", "vector", "linked list") you would use, and what they would represent in the domain (e.g., "a metropolitan area's population").

ii. (1 pts) What ADT would you use to represent the city-to-city relation in the diagram? In other words, how would you store information about what cities neighbor each other? For each component of the ADT (e.g., tasks and priorities for priority queues), tell us what kind of data (e.g., "number", "vector", "linked list") you would use, and what they would represent in the domain (e.g., "a metropolitan area's population").

iii. (3 pts) Given a city name and the two ADTs you described, describe an algorithm (in pseudocode or actual code) below that for a given city name, returns a linked list of all "relevant" articles (according to our definition of "relevant"). Describe your algorithm in the body of the function below. `ADT1` holds a populated instance of your selected ADT from (i) and `ADT2` holds a populated instance of the one from (ii). You should assume that you do not know what their underlying implementations look like. For our reference, indicate what the ADTs are in the first two comments. If you need any additional ADTs or data structures to complete the function, indicate them too.

The `cons` struct for your linked list is provided in Appendix A.1 as reference. You are also free to use any of the interface functions defined in Appendix A.2.

```
def get_relevant_articles(city: str?, ADT1, ADT2) -> Cons.ListC[Article?]:

    # ADT1 is:

    # ADT2 is:
```

Problem 3 Priority Queue with Binary Search Trees (6 points)

We've seen the priority queue ADT implemented with a binary heap. We can also implement priority queues with another binary tree data structure, the Binary Search Tree (BST), which is what you will do in this problem. The ADT interface (`PRIORITY_QUEUE`) which we will implement is given in Appendix A.2.

The only two invariants of the BST are:

- The tree is either: (1) an empty tree (`None`) or (2) a root node with two children, where each child is a valid tree (recursive definition).
- At any node with data d in the BST, the data of all the nodes in the left subtree are less than d and the data of all the nodes in the right subtree are greater than d; "less than" and "greater than" are decided according to when the `lt?` function (defined by the client) returns `True` or `False` respectively.

Here is the definition of one node in the BST and the start of the class implementation.

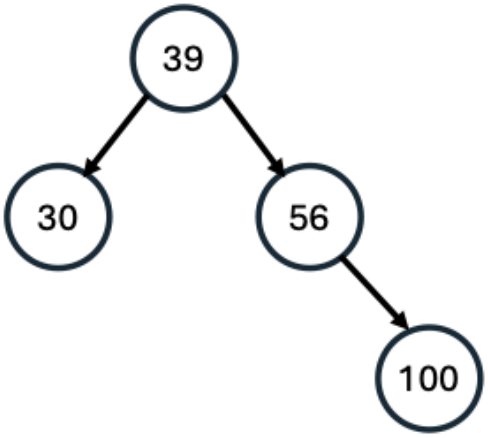
```
struct Node:
    let data # of type X when used in the BST class
    let left: OrC(Node?, NoneC)
    let right: OrC(Node?, NoneC)
```

```
class BST[X] (PRIORITY_QUEUE):
    let root: OrC(NoneC, Node?) # Tree root: initialized to None
    let size: nat?              # No. elements: initialized to 0
    let lt?:  FunC[X, X, bool?] # Comparator function: argument to __init__
```

i. (2 pts) Complete the following implementation of `find_min` such that the two invariants are preserved by the end of the operation and such that the average-case complexity of the operation (i.e., when the tree is balanced) is $O(log\ n)$.

```
def find_min(self) -> X:
    if self.size == 0: error('PQ is empty')
    def find_min_helper(node):
        if _______________________________:
            return _______________________________
        else:
            return _______________________________
    return find_min_helper(self.root)
```

ii. (1 pts) Insert the value 36 into the following BST such that the two invariants are preserved by the end of the operation. The comparator `lt?` here is: `lambda x, y: x < y`. Draw the final BST below.



iii. (1 pts) Draw a possible BST after executing `remove_min` on your BST from Exercise (ii) such that the two invariants are preserved by the end of the operation.

iv. (1 pts) Our priority queue interface requires a function `get_sorted_elements` which returns all the data in the priority queue stored in an array from smallest to largest. What is the time complexity of the `get_sorted_elements` function below as a function of the BST's size n?

```
def get_sorted_elements(self) -> VecC[X]:
    let sorted_elements = [None; self.len()]
    let arr_index = 0

    def traverse(node):
        if not node:
            return

        traverse(node.left)

        sorted_elements[arr_index] = node.data
        arr_index = arr_index + 1

        traverse(node.right)

    traverse(self.root)
    return sorted_elements
```

v. (1 pts) Conversely, what would be the complexity of implementing `get_sorted_elements` if implemented with a binary heap? Explain why that is the complexity.

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

Problem 4 Recording Paths in Graphs (3 points)

In this problem, you will modify the recursive `dfs` function below to report a path between two vertices in a graph if there is one. Specifically, if there is a path from `start` to `target`, the updated code for `dfs` should return a *stack* containing the vertices on this path in **reverse order** — `start` is at the bottom and `target` at the top. If there is no path between `start` and `target`, it returns an empty stack. Appendix A.2 contains the `WUGRAPH` and the `STACK` interface.

One change has been implemented for you on Line 4 in the implementation below.

```
1  def dfs(graph: WUGRAPH!, start: nat?, target: nat?) -> STACK!:
2      let seen = [False; graph.len()]
3      # "path" can be accessed and updated inside dfs_helper()
4      let path = ListStack()
5
6      def dfs_helper(v) -> bool?:
7          seen[v] = True
8          if v == target:
9              return True
10         let neighbors = graph.get_neighbors(v)
11         for neigh in neighbors:
12             if not seen[neigh] and
13                     dfs_helper(neigh):
14                 return True
15         return False
16
17     let connected = dfs_helper(start)
18     return connected
```

i. (3 pts) In the tables below, list the remaining changes that need to be made to satisfy the goal and the `dfs` contract. Use the first table for new code that needs to be inserted between existing lines and specify the line whose indentation the new code should match. Use the second table for lines that need to be deleted and (possibly) replaced with new code. You may not need all the rows provided.

Code to be inserted	between line #	and line #	with the same indentation as line #

Delete line #	and replace with this code (optional)

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

Detach this and the next page for reference and to use as scratch paper. In doing so, please make sure not to destroy your exam.

A Appendix

A.1 `cons` struct for Problem 2

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

A.2 Common ADT interfaces

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

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> NoneC
    def dequeue(self) -> T
    def empty?(self) -> bool?
```

```
interface DICT[K, V]:
    def len(self) -> nat?
    def mem?(self, key: K) -> bool?
    def get(self, key: K) -> V
    def put(self, key: K, value: V) -> NoneC
    def del(self, key: K) -> NoneC
```

```
interface PRIORITY_QUEUE[X]:
    def find_min(self) -> X
    def remove_min(self) -> NoneC
    def insert(self, element: X) -> NoneC
    # Returns all the elements in the PQ
    # sorted from smallest to largest
    def get_sorted_elements(self) -> VecC[X]
```

Continued on next page

```
interface UUGRAPH:
     def add_edge(self, u: nat?, v: nat?) -> NoneC
     def has_edge?(self, u: nat?, v: nat?) -> bool?
     def get_vertices(self) -> VecC[nat?]
     def get_neighbors(self, v: nat?) -> VecC[nat?]
     def get_all_edges(self) -> VecC[TupC[nat?, nat?]]
```

```
let weight? = OrC(num?, inf)
interface WUGRAPH:
     def set_edge(self, src: nat?, w: weight?, dst: nat?) -> NoneC
     def get_edge(self, src: nat?, dst: nat?) -> weight?
     def get_vertices(self) -> VecC[nat?]
     def get_neighbors(self, v: nat?) -> VecC[nat?]
     def get_all_edges(self) -> VecC[TupC[nat?, nat?]]
```

Extra scratch paper. Detach this page carefully.

Extra scratch paper.

Extra scratch paper. Detach this page carefully.

Extra scratch paper.