COMP_SCI 214 (Practice Exam for 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 questions will be taken during the exam; answer the questions based only on the information given (see bullet 2).

Problem 1 (7 points)	
Problem 2 (4 points)	
Problem 3 (5 points)	
Problem 4 (4 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 Designing a mentorship system (7 points)

You are designing a mentorship program to help high school students prepare for college by matching them with mentors who are college students. You are writing software to help you manage this.

Your program has three concrete goals:

1. Receive applications from mentees (high school students) to receive a mentor. The application contains their name, their age, and the area in which they want mentoring. This area can be one of "college-life", "college-apps", or "academics".

2. Receive applications from college students to be mentors in the program. The application contains their name and the area in which they can provide mentoring (from above).

3. Match each mentee, in order from those with the highest need (highest age; ties can be handled in any way) to the lowest, to a mentor who mentors in the area the student wants. A mentee can only be matched once and mentors who match the area criteria should be assigned in the order in which they applied.

Assume that your program stores each mentee application in a `struct` called `"mentee"` that has three fields: `name`, `age`, `area`. Assume that your program stores each mentor's application in a `struct` called `"mentor"` that has two fields: `name`, `area`.

i. (1 pts) What ADT or combination of ADTs would you use to store mentee applicants from goal 1, keeping in mind the requirements of goal 3? For each component (e.g., keys and values for dictionaries) of each ADT, tell us what kind of data (e.g., strings) you would use, and what they would represent in the domain (e.g., a metropolitan area's population). You shouldn't need to modify the `struct` definitions for this, but if you feel that you need to, clearly explain the modifications and why you made them.

Priority queue. Priority is a natural number while the value is a mentee struct object. Priority is the age (higher age is higher priority)

ii. (1 pts) Why did you pick the above ADT(s) over other ADT choices? As part of your answer, you must compare what you chose to at least one other ADT and your answer must describe clearly (and correctly) why your choice suits the given problem.

Another choice of ADT is a regular queue. A regular queue would match users on a first-come first-serve basis, but the program needs the next user to handle based on age priority. Priority queue allows a worklist to output a task in that order.

iii. (1 pts) What ADT or combination of ADTs would you use to store mentor applicants from goal 2, keeping in mind the requirements of goal 3? For each component (e.g., keys and values for dictionaries) of each ADT, tell us what kind of data (e.g., strings) you would use, and what they would represent in the domain (e.g., a metropolitan area's population). You shouldn't need to modify the `struct` definitions for this, but if you feel that you need to, clearly explain the modifications and why you made them.

Dictionary of queues. Key: name = area (String). Value is a queue object of mentor applicants for an area

iv. (1 pts) Why did you pick the above ADT(s) over other ADT choices? As part of your answer, you must compare what you chose to at least one other ADT and your answer must describe clearly (and correctly) why your choice suits the given problem.

Another ADT is a stack. A stack only allows LIFO order. It doesn't match our goal of finding the first mentor for a given area (some degree of FIFO). A dictionary of queues allows us to store FIFO mentor applicants against a specific area.

v. (3 pts) Using the struct definitions and the ADTs you chose (no concrete data structures) from questions i and iii, write a function `match` to aid with goal 3 and that matches one student to a mentor; this function will get called every time the mentee with the next highest need needs to be matched. You can assume that this function only gets called for the first time after goals 1 and 2 have already been executed and applications have been stored.

Your `match` function should return a 2-element vector of strings where the first string is the mentee's name and the second string is the mentor's name (e.g., `["mentee", "mentor"]`). If no mentor is found for a mentee, the mentor returned should be `None`. If there are no mentee applications to match, both elements of the returned vector should be `None`. Your function should take in arguments that hold instances of your selected ADTs, populated in goals 1 and 2; you should assume that you do not know what their underlying implementations look like. For example, you might write `def match(stack, dict)` if your program is using a stack and a dictionary ADT separately or `def match(dict1, dict2, queue)` if your program uses two different dictionaries and a queue (you can name the variables whatever you want, but it should be clear what they represent).

```
def match(pq, dict):
    let mentee_name = None
    let mentor_name = None

    if pq.empty?():
        return [mentee_name, mentor_name]

    let to_match = pq.find_min()
    mentee_name = to_match.name
    pq.remove_min()

    let mentors = dict.get("to_match.area")
    if mentors.empty?():
        mentor_name = None
    else:
        let mentor = mentors.dequeue()

    return [mentee_name, mentor_name]
```

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

Problem 2 Cycles in graphs (4 points)

In graph theory, a cycle is defined as: "a path that starts from a given vertex and ends at the same vertex".

Graphs, both undirected and directed, can contain cycles. However, often to use graphs for specific purposes, a graph needs to be acyclic. For example, a spanning tree is acyclic and a Directed Acyclic Graph (DAG) is a popular type of graph used to represent information flows, a notable example being modeling dependencies in software package managers.

While depth-first search (DFS) is used to traverse a graph from a starting point or check reachability between vertices, it can also be used to detect cycles in a graph (either directed or undirected).

A regular DFS traversal can be modified to detect cycles in the graph. This modified algorithm returns `true` or `false` depending on if a cycle is detected. In addition to a `seen` array, this modified version maintains an array indicating what nodes are being tracked in the current path the DFS is going down. If DFS tries to visit a vertex that was already visited in the current path, then a cycle is detected.

Here is a modified DFS algorithm for cycle detection on a directed graph.

```
Procedure DFS_cycle(g, start):
    seen <- new array of size |V|, filled with false;
    current_path <- new array of size |V| filled with false;
        Procedure Traverse(w):
            seen[w] <- true;
            current_path[w] <- true

            for u in g.get_succs(w) do
                if not seen[u] then
                    if Traverse(u) is true then
                        return true;
                else if current_path[u] is true then
                    return true
                end
            end
            return false;
        end
    Traverse(start);
end
```

i. (1 pts) There is a bug in this algorithm; the bug can either be (1) that it detects a cycle where there is not one, or (2) that it doesn't detect any cycle when the graph has a cycle. (A cycle, if exists, does not necessarily need to include the starting node of the DFS.) Run the above DFS for cycle detection on the graph in Figure 1 **starting from node `1`** and write whether it will return true or false. Based on your answer, which of the two bugs does this DFS exhibit?

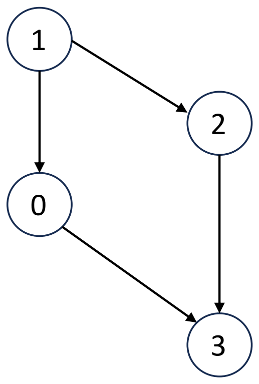


Figure 1: Graph for Problem 2.i.

Returns true.

It is bug 1

ii. **(1 pts)** Describe a small fix for this algorithm either by either changing **one** line of the algorithm, deleting **one** line of the algorithm, or adding **one** line to the algorithm. In your answer, first indicate the line number to fix for either of the first two cases or for the last case, indicate under which line number the new line should go. Then write the full line of the algorithm that constitutes your fix (mention any notes about indentation if applicable). We won't be particular about algorithm or pseudocode syntax but your intent should clearly come through.

Under line 15 add line indented same as line 15:
current_path[w] = False

iii. **(2 pts)** We can also use a Union-Find data structure to detect cycles in an **undirected graph**. Fill in the function definition in DSSL2 for `detect_cycle` which takes in an unweighted undirected graph and an instance of a Union-Find ADT implementation. Your function should return `false` if no cycle is detected. Note that a single edge does not constitute a cycle. The interface and types for the unweighted undirected graph ADT and the interface for the Union-Find ADT are given in Appendix A.1. We won't be finicky about minor DSSL2 syntax, but your intent should clearly come through.

```
def detect_cycle(g: UUGRAPH!, uf: UNION_FIND!) -> bool?:
    for edge in g.get_all_edges():
        let u = edge[0]
        let v = edge[1]
        if uf.find(u) == uf.find(v):
            return true
        uf.union(u, v)
    return false
```

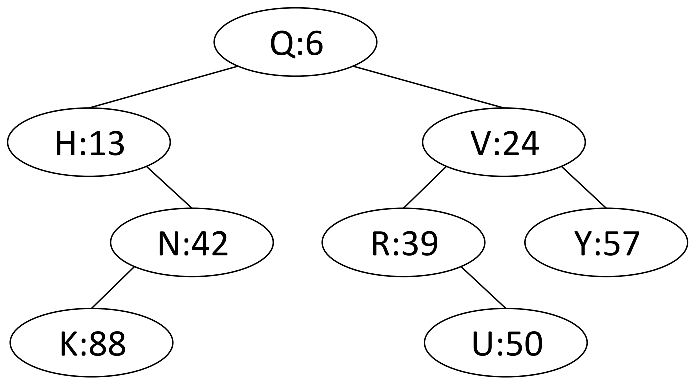
3 Treaps (5 points)

We can use the ideas of binary search trees, heaps and rotations together to create a data structure with an insert operation that on average will produce a tree that is balanced enough to guarantee $O(logn)$ complexity for insert and lookup operations, where n is the number of elements stored in the data structure. This data structure is called a *treap* since it features properties of binary search ***TR****ees* and *h****EAP****s*.

Each node of the treap contains an element that has a key, which is a letter in this example. in addition, each element has a priority, a random integer chosen independently for each element. For this problem, we will assume that all keys are distinct as are all priorities. Elements in a proper treap obey the following three data structure invariants:

- **Invariant 1**: If y is in the left subtree of x, then $key(y) < key(x)$
- **Invariant 2**: If y is in the right subtree of x, then $key(y) > key(x)$
- **Invariant 3**: If y is in either subtree of x, then $priority(y) > priority(x)$

For a given set of elements $e_1, e_2, \ldots, e_n$ (with associated keys and priorities), there is a *unique* treap associated with these elements. Here is an example of a treap that satisfies all of the invariants above. Each node is of the form $(k : p)$ where k is the key and p is the priority.



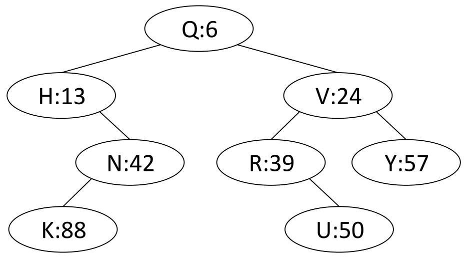
i. (1 pts) Draw the treap obtained out of the following elements:

(K:61), (Q:88), (H:44), (T:27), (F:16), (Z:59), (C:73), (N:40)

See page 17

We insert into a treap by first inserting an element based on its key as we would with a binary search tree. (For letters, this would be based on alphabetical ordering.) But this may violate the set of data structure invariants. Then we perform rotations to restore the invariants.

ii. (1 pts) Insert the element (M:10) into the treap below based on its key value only. (One or more data structure invariants may be violated at this point.) Just draw or indicate where (M:10) is inserted in the treap below; *do not redraw the treap.*



To the bottom and right of K:88

iii. (1 pts) Which of the three data structure invariants are violated after this initial insertion from the previous task?

Invariant 3

iv. (1 pts) Specify the sequence of single rotations needed to restore the invariants. For each rotation, specify left or right and the element that needs to be moved down to make room for the new element (M:10) moving up. You may not need all of the lines below.

Rotate left moving K:88 down and (M:10) up.

Rotate right moving N:42 down and (M:10) up.

Rotate left moving H:13 down and (M:10) up.

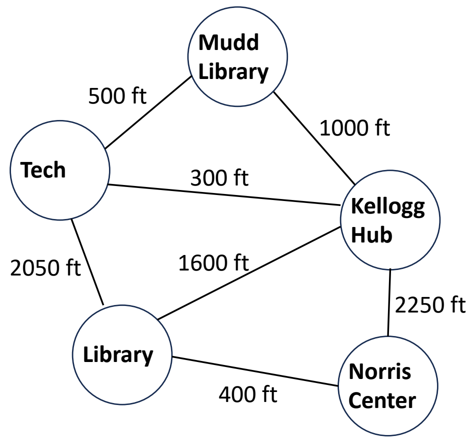
Rotate ____________ moving ____________ down and (M:10) up.

v. (1 pts) Finally draw the final treap below after (M:10) is inserted and the rotations are performed, restoring the invariants.

See page 18

Problem 4 Single-Source Shortest Path (4 points)

Consider the graph below representing a fictionalized Northwestern's campus with buildings and the distances between them.



You are a newcomer to the campus and want to find the shortest distance from Mudd Library to other buildings, assuming that all routes pass through buildings on campus along the way. To do this, you need to run a Single-Source Shortest Path algorithm from Mudd Library to all the other locations in the graph.

vi. (4 pts) Run Dijkstra's algorithm to find the shortest path from Mudd Library to **all other locations** in the graph. As you execute Dijkstra's algorithm, fill in both the tables below with the distances to each node and the predecessors as we did in class, updating values as you go through the algorithm. If you need to update a value in the table, draw an arrow from the old to the updated value (e.g., in "$\infty \to 45 \to 23 \to 2$", 2 is the final value). Each location in the graph is represented by its first letter; you can use the same letters when filling in the predecessor table. For both tables, make sure intermediate and final values are legible and visible.

`v`	`dist[v]`
M	∞ -> 0
T	∞ -> 500
L	∞ -> 2550 -> 2400
N	∞ -> 3050 -> 2800
K	∞ -> 1000 -> 800

`v`	`pred[v]`
M	M/None or empty
T	M
L	T-> K
N	K -> L
K	M -> T

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 Problem 2

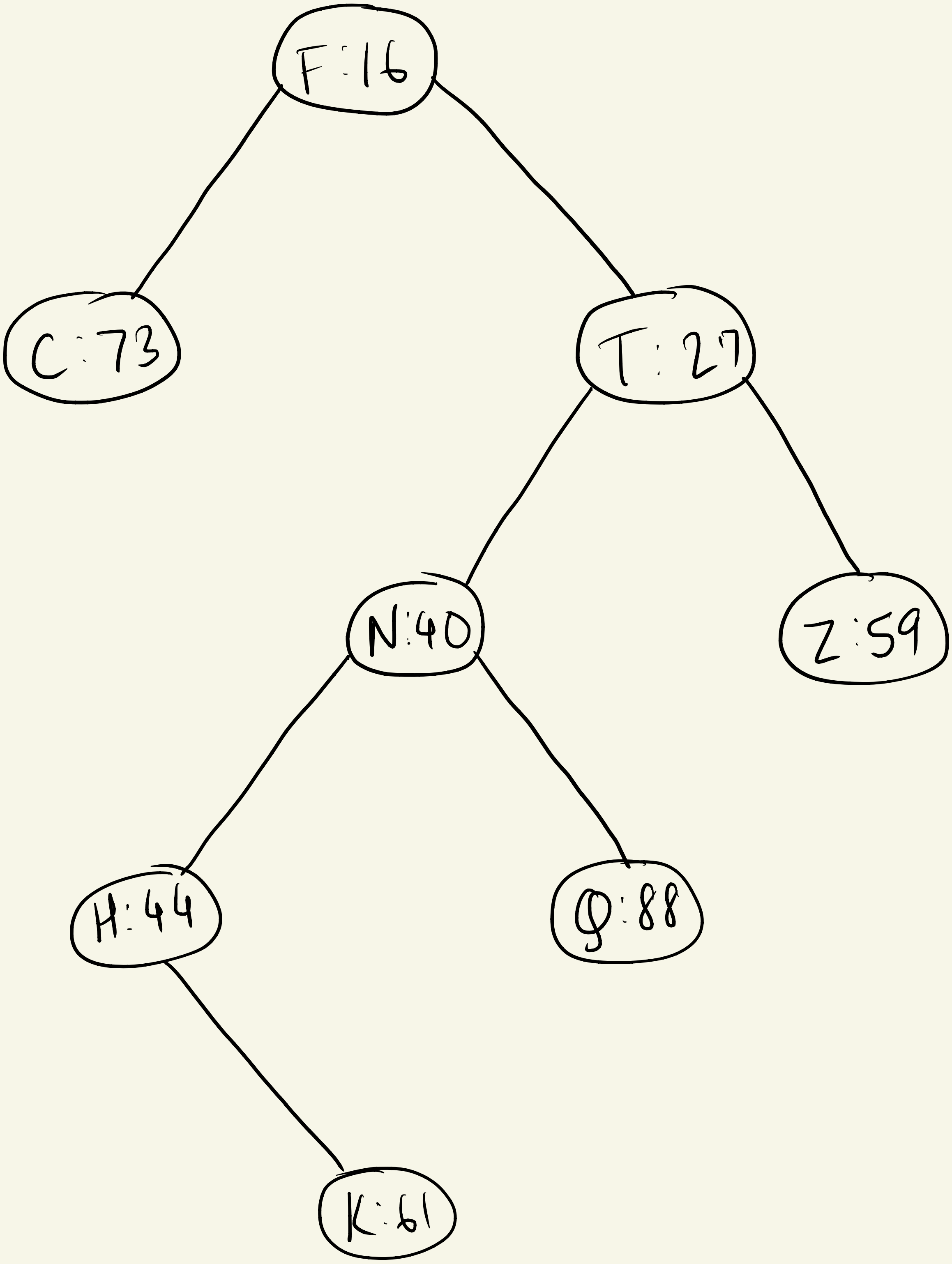
```
interface UUGRAPH:
    def add_edge(self, u: nat?, v: nat?) -> NoneC
    def has_edge(self, u: nat?, v: nat?) -> bool?
    # "get_all_edges" returns a vector of tuples
    # each tuple contains both vertices in an edge
    def get_all_edges(self) -> VecC[TupC[nat?, nat?]]
    def get_adjacent(self, v: nat?) -> VecC[nat?]
```

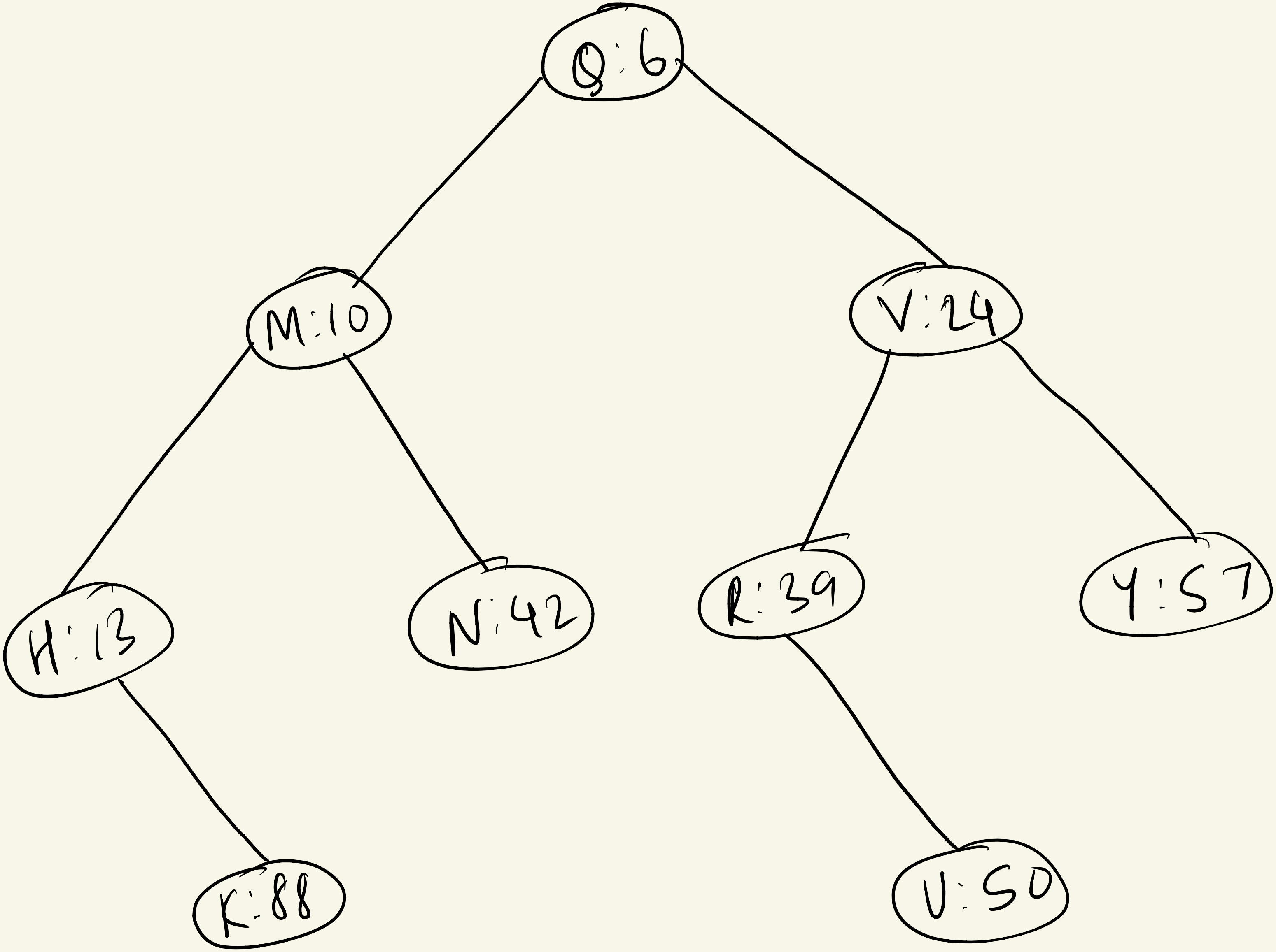
```
interface UNION_FIND:
    def len(self) -> nat?
    def union(self, p: nat?, q: nat?) -> NoneC
    # "find" returns the canonical representative of p's set
    def find(self, p: nat?) -> nat?
```

Extra scratch paper

Extra scratch paper. Detach this page carefully.

Extra scratch paper.


F:16
C:73
T:27
N:40
Z:59
H:44
Q:88
K:61


Q:6
M:10
V:24
H:13
N:42
R:39
Y:57
K:88
U:50