Final Exam

COSC 211: Data Structures, Fall 2021

Instructions. This exam is open book and open note—you may freely use your notes, lecture notes, or textbook while working on it. You may *not* consult any living resources such as other students or web forums. The exam must be submitted via email by **12:00 PM on Friday, December 17th, 2021**. Please submit your scanned or typeset solution **in PDF format** to the Moodle submission site.

In order to receive full credit, be sure to show your work for each problem (where applicable)—an answer without justification is not guaranteed to receive any credit.

Affirmation. I attest that that work presented here is mine and mine alone. I have not consulted any disallowed resources while taking this exam.

Name: Dhyey Dharmendrakumar Mavani

Signature: ______________________

Problem 1. In the spaces below, please answer the following questions.

(a) What is an abstract data type (ADT)?

It is a class which signifies behaviour of certain objects as defined by a set of operations and values. It is like a tech documentation of what operations are to be performed but not how these operations will be implemented. Thus, ADT will not contain code for how memory, algorithms and data interact specifically. The word "abstract" makes it clear that the type/class would give a higher-level view by hiding the details under the rug.

(b) What is a data structure?

A data structure is a particular way of data organization/management in a computer so that we can effectively and efficiently access or edit the stored data according to our specific requirements of time and space. The data structure can be seen as the specific implementation of the functionality mentioned in an ADT.

(c) What does it mean for a data structure to represent an ADT?

A data structure represents an ADT if it provides the functionality mentioned in ADT completely. The data structure has specifics of how the requested functionality (as mentioned in an ADT) is/are implemented. According to my answers to parts (a) and (b) we can simply say that if the data structure reveals and correctly implements all the functionality mentioned in ADT, then it represents an ADT.

Problem 2. Recall that the priority queue ADT (as specified by the `SimplePriorityQueue` interface) supports the following operations:

- `void insert(long k, E x)` add an element `x` with associated priority `k`
- `E removeMin()` removes the element with the smallest associated priority
- `E min()` returns the element with the smallest associated priority

(a) How could you implement a priority queue using a (singly) linked list in such a way that the `removeMin()` and `min()` operations run in time $O(1)$?

→ In SimplePriorityQueue, the lower the associated priority number, the higher the priority. Hence, I would store the element with lowest associated priority next to the head of linked list especially because according to the ADT, I would only provide access to the element with smallest associated priority (highest priority).

→ In the node class, I would store the value, next node reference and priority value. This will allow me to get a reference of where the new node with a unique priority belongs.

→ For $\boxed{\text{E removeMin()}}$, as I know the element near the head is the element with least priority, I will remove it from the linked list. Firstly, I will store head.next into $\boxed{\text{temp}}$. Then, I will ~~do~~ $\boxed{\text{head.next = head.next.next}}$ Then, I will return $\boxed{\text{temp.value}}$

→ For E min(), I will just return $\boxed{\text{head.next.value}}$.

(b) For your suggested implementation of part (a), what is the worst-case running time of the `insert(k, x)` method?

As we are working with a linked list, inserting an element with priority in between would take O(n) worst-case running time because we have to search for the appropriate place to add the node. The actual addition requires just O(1), So, in total $\boxed{\text{insert(k, x)}}$ would take O(n) according to above implementation.

Problem 3. Consider the following partial binary tree implementation:

```
public class BinaryTree<E> {
    Node<E> root;

    public int size() {...}

    protected class Node<E> {
        Node<E> leftChild;
        Node<E> rightChild;
        E value;

        int numDescendants() {...}
    }
}
```

(a) In the space below, write *a recursive method* `numDescendants()` for the `Node<E>` class such that calling `nd.numDescendants()` on a node `nd` will return the number of descendants of `nd`.

```
int numDescendants() {
    int count=0;
    if(leftChild != null){
        count +=1;
        leftChild.numDescendants();
    }
    if(rightChild != null){
        count +=1;
        rightChild.numDescendants();
    }
    return count;
}
```

(b) Write a method `size` for the `BinaryTree<E>` class that returns the total number of nodes in the tree. (Hint: use part (a).)

```
public int size(){
    if (root == null){
        return 0;
    }
    return (1+ root.numDescendants());
}
```

+1 is because we have to include the root itself!!

Problem 4. Recall that a *Bloom filter* of size n with k hash functions $h_1, h_2, \ldots, h_k$ is a Boolean array, `filter`, of size n, with each entry initially set to `false`. Given a list L of m elements that we wish to detect, the array is updated so that for $i = 1, 2, \ldots, k$ and each element x contained in L, we set `filter`$[h_i(x)] =$ `true`. Given an arbitrary element y, we say that `filter` *flags* y if `filter`$[h_i(y)]$ `== true` for all $i = 1, 2, \ldots, k$. If y is contained in L, y will always be flagged; if y is not contained in L, then y is unlikely to be flagged (for suitable choices of n, m, and k).

(a) Suppose we are given two Bloom filters `filter1` and `filter2` that were formed as above for the same choice of n, m, and k, as well as the same hash functions $h_1, \ldots, h_k$. However, `filter1` was initialized to detect elements from a list L_1, while `filter2` was initialized to detect elements for another list, L_2. Given *only* the arrays `filter1` and `filter2`, how could you construct a Boolean array `filterEither` that is guaranteed to flag all elements appearing in either L_1 or L_2? (You should not assume you have access to the lists L_1 and L_2, or the hash functions $h_1, \ldots, h_k$.)

same n, m, k & $h_1, h_2, \ldots, h_k$
$\begin{cases} \text{filter1} \to L_1 \\ \text{filter2} \to L_2 \end{cases}$

filterEither can be constructed by first setting it to a array of n elements of which all are false.

Then, we can say the following logic for any element x...

if $\left(\text{filter1}[h_i(x)] == \text{true} \ \forall\, i = 1 \text{ to } k \ \boxed{\text{OR}} \ \text{filter2}[h_i(x)] == \text{true} \ \forall\, i = 1 \text{ to } k\right)$,
then $(\text{set filterEither}[h_i(x)] = \text{true} \ \forall\, i = 1 \text{ to } k)$

⇒ What we can do instead is we can loop through both filter 1 and filter2 and

$\boxed{\text{set filterEither}[i] = \text{true iff } (\text{filter1}[i] == \text{true} \,||\, \text{filter2}[i] == \text{true})}$

Because (false || false) = false by default!!

(b) Given `filter1` and `filter2` as above, how could you construct a Boolean array `filterBoth` that flags elements appearing on both L_1 and L_2?

same n, m, k & $h_1, h_2, \ldots, h_k$

$$\begin{cases} \text{filter1} \rightarrow L_1 \\ \text{filter2} \rightarrow L_2 \end{cases}$$

filterBoth can be constructed by firstsetting it to a array of n elements of which all are <u>false</u>.

Then, we can say the following logic for any element x...

$$\text{if } \left(\begin{array}{l} \text{filter1}[h_i(x)] == \text{true } \forall\, i = 1 \text{ to } k \ \boxed{\text{AND}} \\ \text{filter2}[h_i(x)] == \text{true } \forall\, i = 1 \text{ to } k \end{array} \right),$$

$$\text{then } (\text{set filterBoth } [h_i(x)] = \text{true } \forall\, i = 1 \text{ to } k)$$

⇒ What we can do instead is we can loop through both filter 1 and ~~fi~~filter2 and

$$\boxed{\text{set filterBoth}[i] = \text{true iff } \left(\begin{array}{r} \text{filter1}[i] == \text{true} \ \&\& \\ \text{filter2}[i] == \text{true} \end{array} \right)}$$

Because

$$\left\{ \begin{array}{l} (\text{false} \ \&\& \ \text{false}) = \text{false} \\ (\text{true} \ \&\& \ \text{false}) = \text{false} \\ (\text{false} \ \&\& \ \text{true}) = \text{false} \end{array} \right\} \text{ by default!!}$$

Problem 5. Recall that in a hash table, two elements are said to *collide* if they hash to the same index in the table. In class, you implemented a hash table that used *chaining* to deal with collisions: each entry in the table stores a (reference to a) linked list of stored elements. Thus, multiple elements can be stored at the same index. Another approach is to store a single element at each index in the table. If when adding an element x that hashes to index i the ith entry of the table is occupied, x is placed in the next unoccupied position in the table. The following program gives an *incorrect* implementation of this strategy.

```
public class BadHash<E> {
    private Object[] table = new Object[8];
    private int size = 0;
    public boolean add(E x) {
        int i = getIndex(x);
        while (table[i] != null) {
            if (x.equals(table[i])) return false;
            i = (i + 1) % table.length;
        }
        table[i] = x;
        size++;
        if (size > table.length / 2) increaseCapacity();
        return true;
    }
    public E find(E x) {
        int i = getIndex(x);
        while (table[i] != null) {
            if (x.equals(table[i])) return (E) table[i];
            i = (i + 1) % table.length;
        }
        return null;
    }
    public E remove(E x) {
        int i = getIndex(x);
        while (table[i] != null) {
            if (x.equals(table[i])) {
                E y = (E) table[i];
                table[i] = null;
                size--;
                return y;
            }
            i = (i + 1) % table.length;
        }
        return null;
    }
    protected int getIndex(E x) {...}
    protected void increaseCapacity() {...}
}
```

(continued on following page)

(a) Suppose a BadHash instance contains n elements. Using big O notation, what is the worst-case running time of the find method? (You may assume that the equals method has running time $O(1)$.)

```
public E find(E x) {
O(1) [ int i = getIndex(x);
       while (table[i] != null) {
O(n)  O(1)[if (x.equals(table[i])) return (E) table[i];
           i = (i + 1) % table.length;
       }
O(1) [ return null;
}
```

worst case runtime for this is O(n) if the initial i=1 and there is no empty spot until end and no match of element until end.

find method seems to have a worst-case runtime of O(n).

(b) Show that add, find, and remove methods for BadHash do not properly implement the unordered set ADT, as specified in the SimpleUSet interface. (Hint: describe a sequence of operations including an add(x) and find(x) such that add(x) succeeds, x is never removed, but nonetheless find(x) fails to find x in the set.)

We are using SimpleUSet ADT, so

- add: checks if element already in set, if no such element found then add and return true, or return false.
- find: check if semantically equal element present, if yes then return it, else return null.
- remove: check if semantically equal element present, if yes then remove and return it, else return null.

Let's say we have 5, 4, 3, 2 hashing to 0 and we added them in the table of size 5.

5	4	3	2	null

Let's say now we remove 3. This works fine!

But, now when we say find(2). It starts from index 0 of the tabular array and then as the index previously containing 3 is now null, it would stop and return null. This is UNDESIRABLE as we never removed 2.

(c) How could you modify the BadHash class so that it still uses the same basic add strategy, but correctly implements add, find, and remove as specified for the unordered set ADT?

We can rewrite the ~~while loop in~~ find method as...

```
public E find(E x){
    int i = getIndex(x);
    while(True){
        if(x.equals(table[i])) return (E) table[i];
        i = (i+1) % table.length;
        if(i == getIndex(x)) return null;
    }
    return null;
}
```

these modification will make sure that we go through each ~~and~~ element to be sure if the semantically equivalent element is present or not.

Similarly, we can update the while loop embedded inside the remove method by change the while-conditional statement to True and adding a condition at very end before we go to another iteration that if we returned at the same position as we were at the start, then we should return null as we were unsuccessful in finding/removing the element because the semantically equivalent element is not in the data table.

Problem 6. In class, we described the Graph ADT specified as the `Graph<E>` interface:

```
public interface Graph<E> {
    int size();
    boolean adjacent(E u, E v);
    List<E> neighbors(E u);
    boolean addVertex(E u);
    List<E> removeVertex(E u);
    boolean hasVertex(E u);
    boolean addEdge(E u, E v);
    boolean removeEdge(E u, E v);
}
```

We also defined the following method that uses breadth-first search (BFS) to determine if a given `Graph<E>` instance contains a given vertex `E x`:

```
    public static <E> E bfsFind(E x, E start, Graph<E> g) {
        if (!g.hasVertex(start)) {
            throw new NoSuchVertexException("Vertex " + start + " not found in graph");
        }

        Queue<E> q = new LinkedList<E>();
        q.add(start);
        Set<E> visited = new HashSet<E>();
        visited.add(start);

        while (q.size() > 0) {
            E cur = q.remove();
            if(x.equals(cur)) {
                return cur;
            }

            for(E nbr : g.neighbors(cur)) {
                if (!visited.contains(nbr)) {
                    visited.add(nbr);
                    q.add(nbr);
                }
            }
        }
        return null;
    }
```

(continued on following page)

The bfsFind method returns a vertex y equal to x if such a vertex is found, and null otherwise. How could you modify the bfsFind method to return a path from start to y in g in the case that such a vertex is found?

⇒ Modifying bfsFind to return a path from start to y in the case we found the vertex in g

```
public static <E> E bfsFind(E x, E start, Graph<E> g) {
    if (!g.hasVertex(start)) {
        throw new NoSuchVertexException("Vertex " + start + " not found in graph");
    }

    Queue<E> q = new LinkedList<E>();
    q.add(start);
    Set<E> visited = new HashSet<E>();
    visited.add(start);
⇒   Map<E,E> prev = new HashMap<E,E>();
    while (q.size() > 0) {
        E cur = q.remove();
        if(x.equals(cur)) {
            return cur;
        }

        for(E nbr : g.neighbors(cur)) {
            if (!visited.contains(nbr)) {
                visited.add(nbr);
                q.add(nbr);      → prev.put(nbr, cur);
            }
        }
    }
    return null;
}
```

Another way is to just make each of the entries: start, nbr and cur ~~of~~ of type E be the nodes storing additional notion of prev, so that it becomes easy to just look for previous at the end to print the path

By modifying the bfs find method as mentioned above, we would be able to keep track of the previous node of each node [storing the process of how we get there]. At last, when we find a element which was expected to be found, we can call recursively the previous of each of the nodes and the path which the bfsfind used to reach the end node will be printed.