UC Berkeley – Computer Science
CS61B: Data Structures

SOLUTIONS

Midterm #2, Fall 2022

Write the statement *"I have neither given nor received any assistance in the taking of this exam."* below.

____

____

Signature: ____________________________

#	Points	#	Points
0	1	6	400
1	250	7	575
2	400	8	750
3	299		
4	575		
5	750		
		TOTAL	4000

Name: ____________________________

SID: ____________________________

GitHub Account #   : fa22-s_____

Person to Left's # : fa22-s_____

Person to Right's #: fa22-s_____

Exam Room: ______________________

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful, and **you may not need all lines**.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not 'does not compile.'**
- **Do not use ternary operators, lambda functions, or streams.**
- ○ indicates that only one circle should be filled in.
- □ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or □, please fill in the shape completely.

0. So it begins (1 point). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your GitHub account # (e.g. fa22-s185) in the corner of every page. If you are taking the exam remotely, make sure that you are screen sharing, are recording your workspace, and your mic is unmuted.

1. **DisjointSets (250 points).** A **correct implementation** of the WeightedQuickUnion idea from lecture is given below. Here we have one array: `arr`. We use the convention that if an element is the root of the set, its array value is the weight of the set negated (as you saw on HW3). Otherwise, its array value is the parent of that element (as you also saw on HW3).

```
1: public void connect(int v1, int v2) {
2:    int rootV1 = find(v1);
3:    int rootV2 = find(v2);
4:    if (rootV1 != rootV2) {
5:       if (arr[rootV1] <= arr[rootV2]) {
6:          arr[rootV1] += arr[rootV2]; // adds rootV2 tree size to rootV1
7:          arr[rootV2] = rootV1;
8:       } else {
9:          arr[rootV2] += arr[rootV1]; // adds rootV1 tree size to rootV2
10:         arr[rootV1] = rootV2;
11:      }
12:   }
13: }
```

a) **(150 points)** Suppose we replace line 7 with: `arr[rootV2] = v1;` What are the potential negative consequences of this change?

□ if `connect` is called when `v1` is not a root, later `isConnected` operations may be incorrect
□ if `connect` is called when `v2` is not a root, later `isConnected` operations may be incorrect
☒ runtime for `connect` operations may be asymptotically worse than with the original line 7
☒ runtime for `isConnected` operations may be asymptotically worse than with the original line 7
□ none of the above

b) **(100 points)** For the same modified code from part a, suppose we have 6 items in our **`WeightedQuickUnion`** object, i.e. the valid arguments for `v1` and `v2` are 0, 1, 2, 3, 4, and 5. What is the worst case height of the **`WeightedQuickUnion`** tree (per convention, height is zero-indexed)?

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ● 5 ○ 10 ○ 11

2. **EvenIterable (400 points).** We want to make an **EvenIterable** class with a constructor that takes in an **Iterable** and creates a new **Iterable** object that only iterates over items in even numbered positions. For example, the output of the code below should be 0 then 10 then 20. See the reference sheet at the end of this exam for definitions of the **Iterable** and **Iterator** interfaces.

```
    public static void main(String[] args) {
        List<Integer> L = List.of(0, 1, 10, 2, 20, 3);
        EvenIterable<Integer> evenIt = new EvenIterable<>(L);
        for (int i : evenIt) {
            System.out.println(i);
        } // prints 0 then 10 then 20
    }
```

Fill in the **EvenIterable** class below. Your constructor may use up to linear space and linear time, i.e. the space and time usage must be O(N). You may not need all lines. Your code should not result in exceptions being thrown for the main function above.

```
public class EvenIterable<T> implements Iterable<T> {
    private List<T> evens_______________;

    public EvenIterable(Iterable<T> iterable) {
        evens = new ArrayList<>()_______;
        Iterator<T> it = iterable.iterator()__________;
        while (it.hasNext()________) {
            evens.add(it.next());_________
            if (it.hasNext()) {___________
                 it.next();_______________
            }_____________________________
            ______________________________
            ______________________________
        }
    }

    public Iterator<T> iterator() {
        return evens.iterator()_______;
    }
}
```

Alternate sol:

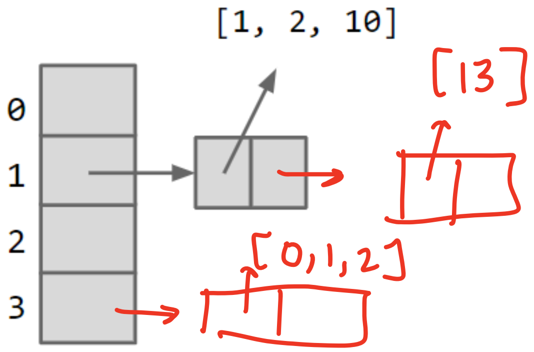
Use a boolean flag/counter, and declare AND instantiate the List in the instance variable line.

3. HashSets (299 Points). Suppose we create a class **HashableArrayList**, given below.

```
public class HashableArrayList<T> extends ArrayList<T> {
    @Override
    public int hashCode() {
        int hc = 0;
        for (int i = 0; i < size(); i += 1) {
            hc += get(i).hashCode();
        }
        return hc;
    }
}
```

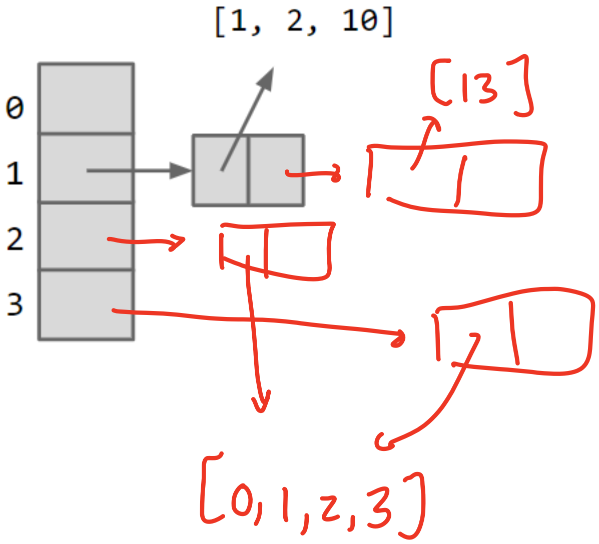
a) For example, if we created a **HashableArrayList** containing [1, 2, 10], the hash code would be 13, since the hashcode of an integer is its own value. Below, draw the results of the following code, assuming the hashcode is reduced (i.e. "reduced" means converted into a bucket number) by modding by the number of buckets. Assume that we are not resizing at any point. Collisions are handled by external chaining (a linked list) where the first instance variable of each node is a link to the stored item, e.g. [1, 2, 10], and the second instance variable is a link to the next node. Additionally, note that the **ArrayList**.equals() method compares the contents. The first add operation has been done for you.

```
1: HashSet<HashableArrayList<Integer>> set = new HashSet<>();
2: set.add(new HashableArrayList<>(1, 2, 10));
3: HashableArrayList<Integer> zeroOneTwo = new HashableArrayList<>(0, 1, 2);
4: set.add(zeroOneTwo);
5: set.add(new HashableArrayList<>(13)); // draw after lines 1-5 all done
```

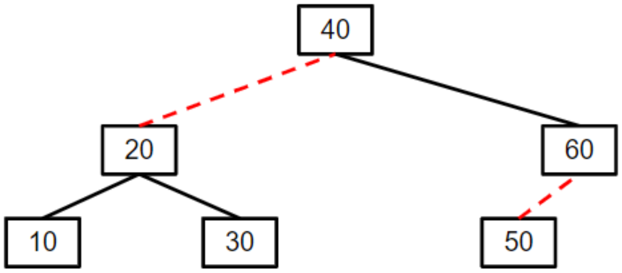


b) **Copy** your diagram from part a to the diagram to the right. Now suppose we also call the following lines of code. Draw the new state of the hash table **after all 8 lines of code have executed.**

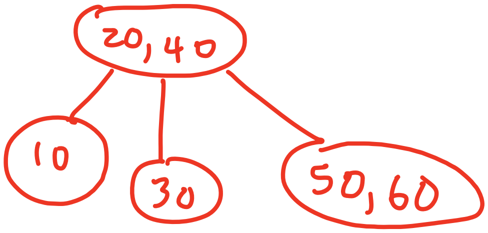
```
6: set.add(zeroOneTwo);
7: zeroOneTwo.add(3);
8: set.add(zeroOneTwo);
```



4. LLRBs (575 Points). Suppose we have the **LLRB <u>of integers</u>** below:



a) **(150 points)** Draw the corresponding 2-3 tree:



b) **(125 points)** Give all integers which, when inserted into the LLRB, result in a single rotateLeft operation and no further cascading operations (i.e. no rotateLefts, rotateRights or color flips). If this is not possible, write "impossible". If it is possible, use inclusive bracket notation, e.g. if your answer is "[62, 64] and [66, 68]", this is equivalent to saying 62, 63, 64, 66, 67, 68.

[11, 19], [31, 39]

c) **(125 points)** Give all integers which, when inserted into the LLRB results in a single rotateRight operation and no further cascading operations (i.e. no rotateLefts, rotateRights or color flips). If this is not possible, write "impossible". If it is possible, use inclusive bracket notation, e.g. if your answer is "[62, 64] and [66, 68]", this is equivalent to saying 62, 63, 64, 66, 67, 68.

Impossible

d) **(125 points)** At most, how many additional inserts can be performed into the LLRB above without triggering any color flip operations? Do not count the operation that results in the first color flip.

2

e) **(50 points)** True or false: If we insert enough values into the LLRB above, eventually we will get a `colorFlip(40)` operation.

○ True ● False

5. Tree Life (750 points). In the heaps lecture, we used an array to represent a complete binary tree (also called "tree representation 3B" in lecture). Suppose we generalize this idea so that our array now represents a trinary tree (where the word **tri**nary means "**three**", as opposed to **bi**nary which means "**two**"). Suppose we then have a trinary heap represented by the array [-, 2, 4, 5, 12, 6, 7, 8, 11, 20]. Recall that in this representation that the leftmost item is unused, i.e. the - indicates the unused position.

a) **(100 points)** What value is 8's parent? 4

b) **(150 points)** If we perform a pre-order traversal of this tree, what are the **last 3 values** we traverse? Give in the order traversed.

11, 20, 12

c) **(150 points)** If we perform a post-order traversal of this tree, what are the **first 3 values** we traverse? Give in the order traversed.

6, 7, 8

d) **(175 points)** If we perform a DFS graph traversal starting from node 12, breaking ties by going to the smallest node first, what will be the DFS pre-order? Give the whole order:

12, 2, 4, 6, 7, 8, 5, 11, 20

e) **(175 points)** Suppose we replace 20 by some arbitrary value X such that the trinary heap is still valid (e.g. 0 would be invalid, but 33 would be valid). If we use the standard heap deletion algorithm, where can X end up after we call `deleteMin` exactly once? Select all that might apply. Recall that heaps can contain duplicates.

□ X might not be in the heap at all after `deleteMin`.
□ In the root position previously occupied by the 2.
☒ In the position previously occupied by the 4.
□ In the position previously occupied by the 5.
□ In the position previously occupied by the 12.
☒ In the position previously occupied by the 6.
□ In the position previously occupied by the 7.
□ In the position previously occupied by the 8.
□ In the position previously occupied by the 11.
□ In the same position, i.e. the X might not move.

6. **Asymptotics and ArrayDeques (400 points).**

a) **(100 points)** Suppose $R(N) = 11 + 22 + 33 + 44 + \cdots + N$, e.g. if $N = 55$, then $R(N) = 165$. Which of the following are true?

- □ $R(N) \in O(1)$
- □ $R(N) \in O(N)$
- ☒ $R(N) \in O(N^2)$
- ☒ $R(N) \in O(N^3)$
- □ $R(N) \in \Theta(1)$
- □ $R(N) \in \Theta(N)$
- ☒ $R(N) \in \Theta(N^2)$
- □ $R(N) \in \Theta(N^3)$

b) **(100 points)** Suppose we use the following resize strategy for an ArrayDeque: When the Deque becomes too full, resize by adding 11 new entries. What is the order of growth of the runtime for the following code? Give your answer in big theta notation in terms of N.

```
ArrayDeque<Integer> ad = new ArrayDeque<>();
for (int i = 0; i < N; i += 1) {
    ad.add(i);
}
```

Answer: Θ(N^2)

c) **(100 points)** Let f(x) be defined as follows:

$$f(x) = \begin{cases} x & \text{if x is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Let $Z(N) = f(1) + f(2) + f(3) + \cdots + f(N)$. For example, if N = 8, then $Z(N) = 1 + 2 + 1 + 4 + 1 + 1 + 1 + 8 = 19$. Which of the following are true about $Z(N)$?

- □ $Z(N) \in O(1)$
- ☒ $Z(N) \in O(N)$
- ☒ $Z(N) \in O(N^2)$
- ☒ $Z(N) \in O(N^3)$
- □ $Z(N) \in \Theta(1)$
- ☒ $Z(N) \in \Theta(N)$
- □ $Z(N) \in \Theta(N^2)$
- □ $Z(N) \in \Theta(N^3)$

d) **(100 points)** Suppose we use the following resize strategy for an ArrayDeque: When the Deque becomes too full, resize by doubling the size of the array. What is the order of growth of the runtime for the following code? Give your answer in big theta notation in terms of N.

```
ArrayDeque<Integer> ad = new ArrayDeque<>();
for (int i = 0; i < N; i += 1) {
    ad.add(i);
}
```

Answer: Θ(N)

7. **Asymptotics and Dynamic Method Selection (575 points).**

Suppose we have the following class definitions:

```
public class Cheese {
    public String cheese;
    public Cheese(String cheese) {
        this.cheese = cheese;
    }

    // String's substring runs in Θ(length of the substring)
    public Cheese slice(int from, int to) {
        return new Cheese(cheese.substring(from, to));
    }

    // String's length runs in Θ(1)
    public int weight() {
        return cheese.length();
    }
}

public class Parmesan extends Cheese {
    public Parmesan(String cheese) {
        super(cheese);
    }
    @Override
    public int weight() {
        int w = super.weight();
        if (w <= 1) {
            return w;
        }
        Cheese mine = slice(0, w / 2);
        Cheese yours = slice(w / 2, w);
        return mine.weight() + yours.weight();
    }
}
```

Exam Sanctuary. Listen to any good music lately? Feel free to praise it here. Or draw a walrus. Or something else.

Clarification (that you might not need, but we're putting it here to avoid having to clarify during the exam): The static type of `slice(int from, int to)` is **`Cheese`** because that is the return type specified in the method signature. The dynamic type of the returned object is the actual runtime type of the instantiated object.

a) **(125 points)** Give the tightest asymptotic runtime of the **weight** method in the **Parmesan** class in terms of N, where N is the length of the underlying **String** cheese? Give your answer in big theta notation.

Runtime: $\Theta(N)$

b) **(125 points)** Now, suppose we override the slice method by including the following method in Parmesan.java:

```
@Override
public Cheese slice(int from, int to) {
    Cheese b = super.slice(from, to);
    return b;
}
```

What is the runtime of the **weight** method previously analyzed in terms of N in big theta notation?

Runtime: $\Theta(N)$

c) **(100 points)** Now, suppose we override the slice method by including the following method in Parmesan.java. This problem is instead of, and thus independent of, part b.

```
@Override
public Cheese slice(int from, int to) {
    String sayCheese = cheese.substring(from, to);
    Parmesan c = new Cheese(sayCheese);
    return c;
}
```

This method will result in an error. What kind of error?

○ Runtime Error ● Compile Time Error ○ Infinite Loop

d) **(100 points)** Now, suppose we override the slice method by including the following method in Parmesan.java. This problem is instead of, and thus independent of, parts b and c.

```
@Override
public Cheese slice(int from, int to) {
    String sayCheese = cheese.substring(from, to);
    Parmesan d = (Parmesan) new Cheese(sayCheese);
    return d;
}
```

This method will result in an error. What kind of error?

● Runtime Error ○ Compile Time Error ○ Infinite Loop'

e) **(125 points)** Now, suppose we override the slice method by including the following method in Parmesan.java. This problem is instead of, and thus independent of, parts b, c, and d.

```
@Override
public Cheese slice(int from, int to) {
    String sayCheese = cheese.substring(from, to);
    Parmesan e = new Parmesan(sayCheese);
    return e;
}
```

This method will not error. What is the asymptotic runtime of the **weight** method previously analyzed in terms? You may use N to represent the length of the underlying **String** cheese. Give your answer in big theta notation.

Runtime: $\Theta(N\log(N))$

f) **(0 points)** *Candidatus* Desulforudis audaxviator (in part discovered by my labmate Dylan Chivian while I was in grad school) is the only species known to be alone in its ecosystem. Where was it discovered?

South Africa

8. **Data Structures Design (750 points).** Suppose we want to add a new operation to the **`LinkedListDeque`** from Proj1A, specifically a `removeEvery(int x)` operation. For simplicity, assume our **`LinkedListDeque`** is hard coded to use integer values. This method should remove all instances of x from the list. Describe an implementation of the modified **`LinkedListDeque`** class below. Your methods must complete in `O(k+log N)` time on average (i.e. it's OK to ignore any occasional resize operation), where k is the number of x's present in the Deque, and N is the number of items in the Deque. In other words, the methods below must be no worse than linear with respect to k, and no worse than logarithmic with respect to N. It's OK if your methods are faster than the requirement. No credit will be given for naïve solutions (e.g. $\theta(N)$ runtime, since this is linear with respect to N).

a) What additional instance variable do you need? List a single instance variable, **<u>including the type and name</u>**. You may use only one! You may use any data structure from the reference sheet or any primitive type.

TreeMap<Integer, ArrayList<Node>> tmap

b) Describe below how the `addFirst(int x)` operation is different (if at all) from the `addFirst` operation in a normal **`LinkedListDeque`**. If you use your instance variable from part a, use the name you gave to that instance variable. If there are no differences, just write "no difference".

Let N represent the Node created in a call to addFirst. If tmap contains x, add n to the existing arraylist. Else, put x in the tmap with a new arraylist with n.

c) Describe below your `removeEvery(int x)` operation. If you use your instance variable from part a, use the name you gave to that instance variable.

Remove x from tmap, and for every Node in the ArrayList returned, remove it from the LinkedListDeque.

As a reminder, the **`LinkedListDeque`** class is partially defined below:

```
public class LinkedListDeque {
    private Node sentinel;
    private int size;
    public void addFirst(int item) { … }
    public int removeFirst() { … }
    public int get(int index) { … }
    …
}
```

Reference Sheet

Map, Set, List (note: all implement Iterable)

```
public interface Map<K, V> { ...
   boolean containsKey(K key)
   V get(Object key)
   V remove(Object key)
   V getOrDefault(Object o, V value)
   void put(K key, V value)
   Set<K> keySet()
   int size()
}

public interface Set<K> { ...
   boolean contains(K key)
   void add(K key)
   boolean remove(Object o)
   Iterator<K> iterator()
   int size()
}

public interface List<T> { ...
   boolean contains(Object o)
   void add(T item)
   void add(int index, T item)
   T get(int i)
   T set(int i, T item)
   int indexOf(Object o)
   boolean remove(Object o)
   Iterator<T> iterator()
   int size()
}
```

Implementations:

```
LinkedList implements List

ArrayList implements List

TreeSet implements Set

HashSet implements Set

TreeMap implements Map

HashMap implements Map
```

Note: **TreeSet** and **TreeMap** are implemented with a red black tree.

Note: For brevity, we've omitted usages of "extends" and "implements" from class and interface definitions.

Other handy data types:

```
public class Stack<T> { ...
   T pop()
   void push(T item)
   Iterator<T> iterator()
   int size()
}
public class Queue<T> { ...
   T dequeue()
   void enqueue(T item)
   Iterator<T> iterator()
   int size()
}
public class MinPQ<T> { ...
   T removeSmallest()
   T smallest()
   void add(T item)
   Iterator<T> iterator()
   int size()
}
```

Iterator, Iterable, Comparator, Comparable:

```
public interface Iterator<T> { ...
    boolean hasNext()
    T next()
}

public interface Iterable<T> { ...
    Iterator<T> iterator()
}

public interface Comparator<T> { ...
      int compare(T o1, T o2)
}
public interface Comparable<T> {
     int compareTo(T obj)
}
```