CS 61B Spring 2023 — Midterm 2

Thursday March 16, 2023

Intro: Welcome to CS61B Midterm 2!

Your name: ____________________

Your SID: ____________________ Login: `sp23-s`__________

Location: ____________________

SID of Person to your Left: ____________________ Right: ____________________

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

____

____

Signature: ____________________

Tips:

- For answers which involve filling in a ◯ or ☐, **please fill in the shape completely.** If you change your response, **erase as completely as possible**. Incomplete marks may affect your score.
- ◯ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- You may not need to use all provided lines, but we **will not give credit for solutions that go over number of provided lines.**
- You may not use ternary operators, lambdas, streams, or multiple assignment.
- 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'.

Q1: Taiko Time!

(1100 Points)

```
public class Taiko {
    private String name;
    private int size;
    public Taiko(String name, int size) {
        this.name = name;
        this.size = size;
    }

    public String play() { return "BOOM"; }
    public String getName() { return name; }
}

public class Chu extends Taiko {
    public Chu(String name, int size) {
        super(name, size);
    }

    public String play() { return "don kon"; }
}

public class Shime extends Taiko {
    public Shime(String name, int size) {
        super(name, size);
    }

    public String play() { return super.play(); }
}
```

(a) Suppose that Crystal plans to take out the equipment items one by one, starting from the item she most recently put in. Which data structure should she use?

◯ Set ◯ LLRB ◯ Heap ◯ Map ◯ Stack

(b) We will use a `Queue` to represent the `equipment` in the `organize` function below. Crystal would like to categorize each item in `equipment` as either a `Chu` or `Shime` and add it to its respective collection (mapping `name` to `Chu` in `chus` or putting a `Shime` into `shimes`). You may assume that all elements in `equipment` are `Chus` and `Shimes`. You may destructively alter the attributes of the `Queue<> Equipment`.

```
public class Cages {
    private Map<String, Chu> chus = new TreeMap<>();
    private Set<Shime> shimes = new HashSet<>();
    public void organize(Queue<____________________________________> equipment) {

        while (_______________________________________________________________) {

            _________________________________________________________________;

            if (______________________________________________________________) {

                _____________________________________________________________;
            } else {

                _____________________________________________________________;
            }
        }
    }
}
```

(c) For performance purposes, Crystal decides to consider `Taiko` to be equal if they produce the same sound using the `play()` function, like:

```
public class Taiko {
    // constructor and other methods not shown
    ...
    @Override
    public boolean equals(Object o) {
        // o typechecking not shown
        ...
        return ((Taiko) o).play().equals(this.play());
    }
}
```

Given the following variables, evaluate each statement in the table below and mark the appropriate box. If the code errors, distinguish whether it is a compiler (CE) or runtime error (RE).

```
Taiko m = new Taiko("miyake", 5);
Shime b = new Shime("beato", 2);
Chu h = new Chu("hachijo", 7);
Taiko y = new Chu("yatai", 9);
Taiko s = new Shime("suwari", 6);
```

`m.equals(b)`	○ true	○ false	○ CE	○ RE
`h.equals(y)`	○ true	○ false	○ CE	○ RE
`s.equals((Chu) s)`	○ true	○ false	○ CE	○ RE
`((Taiko) b).equals((Taiko) h)`	○ true	○ false	○ CE	○ RE
`((Chu) m).equals(y)`	○ true	○ false	○ CE	○ RE
`b.equals((Shime) h)`	○ true	○ false	○ CE	○ RE
`s.equals(h)`	○ true	○ false	○ CE	○ RE

Q2: BSTIterator

(700 Points)

(a) Given a Binary Search Tree, if we want to yield the keys in sorted order, what kind of tree traversal should be implemented?

◯ Level-order ◯ In-order ◯ Pre-order ◯ Post-order

(b) Given the following BST implementation, complete the `BSTIterator` implementation such that it yields the keys in sorted order. You may assume that no elements are added or removed to the BST after the iterator is instantiated. **You may not instantiate any classes other than `Iterator`s**.

Note: `Collections.emptyIterator()` returns an empty `Iterator` (not **`null`**!).

```
public class BST<K extends Comparable<K>> {
    private Node root;
    private class Node {
        K key;
        Node left;
        Node right;
        Node(K key) {
            this.key = key;
        }
    }

    public Iterator<K> iterator() {

        return new BSTIterator(___________________________________________________________);
    }

    private class BSTIterator implements Iterator<K> {
        Iterator<K> iterLeft;
        K current;
        Iterator<K> iterRight;
        BSTIterator(Node n) {
            if (n == null) {
                iterLeft = Collections.emptyIterator();
                current = null;
                iterRight = Collections.emptyIterator();
            } else {

                iterLeft = ______________________________________________________________;

                current = _______________________________________________________________;

                iterRight = _____________________________________________________________;
            }
        }
        @Override
        public boolean hasNext() {

            return _____________________________________________________________________ ||

                   _____________________________________________________________________ ||

                   _____________________________________________________________________;
        }

        @Override
        public K next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            if (iterLeft.hasNext()) {

                return _________________________________________________________________;
            } else if (current != null) {

                K result = _____________________________________________________________;

                current = ______________________________________________________________;

                return _________________________________________________________________;
            } else {

                return _________________________________________________________________;
            }
        }
    }
}
```

Q3: Disjoint Sets

(900 Points)

You are given a Weighted Quick Union data structure with this as the current state of the underlying array.

Index	0	1	2	3	4	5	6	7	8	9
Value	-4	-3	0	-2	0	-1	1	1	3	2

(a) How many disjoint sets are there currently?

(b) We call `connect(4, 9)` **without path compression**. How many disjoint sets are there now?

(c) After running `connect(4, 9)`, suppose we call `connect(4, 1)` **both without path compression**. What is the height of the resulting tree of this specific set? (A tree's height is the greatest number of edges from the root to a leaf of the tree.)

(d) After these 2 operations `connect(4, 9)` and `connect(4, 1)` from parts b - c, what is the state of the underlying array?

Index	0	1	2	3	4	5	6	7	8	9
Value										

Parts e - f are unrelated to parts a - d.

(e) Given the WQU data structure below's before value state, what should the underlying array look like after calling `isConnected(1,9)` **with path compression**?

Index	0	1	2	3	4	5	6	7	8	9
Before Value	-9	-1	0	4	2	4	2	6	2	5
After Value										

(f) What is the output of this call to `isConnected(1,9)`?

Q4: Two Tree, Three Tree, Red Tree, Black Tree

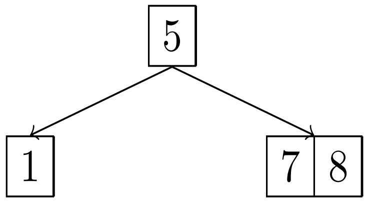
(1100 Points)

Your answer must fit in the box. No points will be awarded for answers outside the box.

(a) What is the maximum number of **leaf nodes** for a 2-3 Tree with height 5? (A tree's height is the greatest number of edges from the root to a leaf of the tree.)

(b) What is the maximum number of **elements** for a 2-3 Tree with height 4?

For parts c - d, consider the following valid 2-3 Tree



(c) What is the **minimum** number of elements to insert to trigger a change in height?

(d) What is the **maximum** number of **distinct** elements that can be successfully inserted **without changing the height of the tree**?

For parts e - g, the operations are **independent of each other**. A **single add operation** is performed on a Left-Leaning Red-Black Tree. Which of the following are true?

(e) If a `rotateLeft` operation happened, it is immediately followed by a `rotateRight` operation (i.e. no operations happened between the `rotateLeft` and `rotateRight` operation):

○ Always True ○ Sometimes True ○ Never True

(f) If a `rotateRight` operation happened, it is immediately followed by a `colorFlip` operation (i.e. no operations happened between the `rotateRight` and `colorFlip` operation):

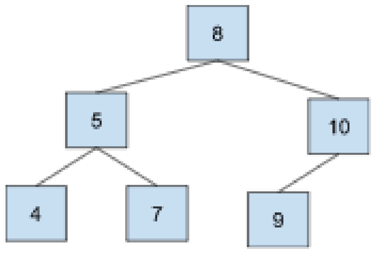
○ Always True ○ Sometimes True ○ Never True

(g) If a `colorFlip` operation happened, it is immediately followed by a `colorFlip` operation. (i.e. no operations happened between the `colorFlip` and second `colorFlip` operation):

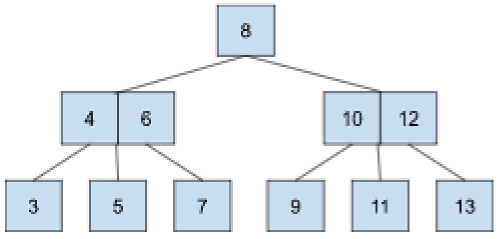
○ Always True ○ Sometimes True ○ Never True

(h) Which of the following are **not** well-formed 2-3 tree(s)? Select all that apply.

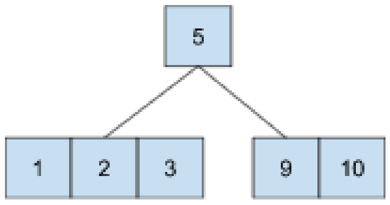
☐ (i) ☐ (ii) ☐ (iii) ☐ (iv) ☐ (v)



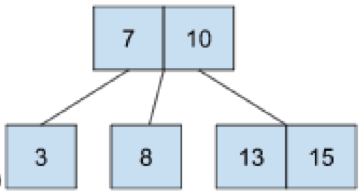
(i)



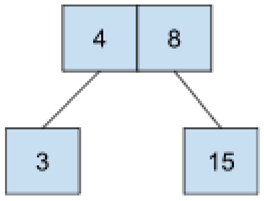
(ii)



(iii)



(iv)



(v)

Q5: Tree Shirts

(900 Points)

Crystal works at 61BestMerch, and she found searching for shirts is incredibly slow. She recalled from lecture that BST helps optimize search, so she decided to store available shirt sizes in a BST.

Fill out the instance method `findShirt(int reqSize)` that finds the available shirt size closest to the requested size **that is larger or equal to** in size. In other words, return the smallest Node that is greater than or equal to requested size in the BST. If no such Node exists, return null.

For example, suppose available shirt sizes are represented with the BST to the right. If Crystal wanted size 13, `findShirt(13)` should return the `Node` with 15.

```
         20
       /    \
    15        27
   /  \         \
  4     17       30
```

```
public class TreeShirt {
    private Node root;
    private class Node {
        int shirtSize;
        Node left;
        Node right;
    }

    public Node findShirt(int reqSize) {
        Node prev = null;
        Node curr = root;

        while (______________________________________________________________) {

            if (_____________________________________________________________) {

                return _______________________________________________________;

            } else if (_______________________________________________________) {

                ______________________________________________________________;

                ______________________________________________________________;

            } else {

                ______________________________________________________________;
            }
        }

        return ________________________________________________________________;
    }
}
```

Q6: Hash-ish

(900 Points)

Part 1: Classify each of the string hash functions that have no compile errors as follows.
Incorrect: The function is not a valid hash function, or is buggy. *If this is the case, write 10 words or less explaining why the hash function is invalid.*
Ineffective: The hash function is valid, but it is easy to find collisions (two words with the same hash, and new words with the same hash as a given word). *If you select this, write 2 distinct words less than 6 characters that yield the same hash.*
Correct: The hash function is valid, and it is not easy to find collisions. *You may select this option on only one of the answers below.*

```
(a)
public int hashCode() {
    Random random = new Random();
    return this.length * random.nextInt();
}
```

◯ Incorrect ◯ Ineffective ◯ Correct

```
(b)
public int hashCode() {
    Random random = new Random(this.hashCode());
    return random.nextInt();
}
```

◯ Incorrect ◯ Ineffective ◯ Correct

```
(c)
public int hashCode() {
    return 0;
}
```

◯ Incorrect ◯ Ineffective ◯ Correct

```
(d)
public int hashCode() {
    Random random = new Random(0);
    int val = 0;
    for (char c : this.toCharArray()) {
        val += c * random.nextInt();
    }
    return val;
}
```

◯ Incorrect ◯ Ineffective ◯ Correct

```
(e)
public int hashcode() {
    int val = 0;
    for (char c : this.toCharArray()) {
        val += c;
    }
    return val;
}
```

◯ Incorrect ◯ Ineffective ◯ Correct

Part 2: You are an evil hacker, and have come across a HashMap of instructor names (which are Strings).

- You are able to insert new elements to the HashMap, and time how long it takes for that insertion to run. You may NOT use any other functions of the HashMap, or modify the hash function used.
- You know how the HashMap is implemented and the hash function used for strings, but don't know the data in this particular HashMap.
- The HashMap uses linked lists for its bins, and resizes when `numberOfElements / numberOfBins` exceeds a certain threshold.
- The HashMap already has a few values in it before you insert anything. You do not know anything about these values, but you do know that not all of them have the same hash.
- You may assume the hash function distributes values evenly.

Classify each of the following actions as follows:

Possible: It is sometimes impossible to do this if strings use a correct hash function, but it is always possible to do this if strings use an ineffective hash function (per the definition in 1).

Impossible: Even if strings use an ineffective hash function, it is sometimes impossible to do this.

(a) Determine how many names were originally in the HashMap
◯ Possible ◯ Impossible

(b) Determine if "Justin" is in the HashMap, without inserting "Justin" yourself
◯ Possible ◯ Impossible

(c) Make it arbitrarily slow for other people to access any name in the HashMap
◯ Possible ◯ Impossible

(d) Assume that "Justin" and "Josh" were already inserted into the HashMap. Make it arbitrarily slow to access "Justin", but still fast to access "Josh" (assuming that "Justin" and "Josh" have the same hash)
◯ Possible ◯ Impossible

This page is intentionally left blank (except for this sentence and the page number).

Q7: SHeap

(900 Points)

We are given the following array representing a min-heap containing 7 values, **where each symbol represents a distinct number**. There is no implicit ordering with the symbols. The last array position is initially unused. (For convenience, we have numbered the array elements starting from 1 rather than 0)

Index	0	1	2	3	4	5	6	7	8
Value	NULL	!	@	#	$	%	?	&	

Answer the following questions about the heap after removing the minimum value.

(a) Which indice(s) might contain the value & in the resulting min-heap?
☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8

(b) Which indice(s) might contain the value % in the resulting min-heap?
☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8

(c) Which symbol(s) could be at index 4 in the resulting heap?
☐ ! ☐ @ ☐ # ☐ $ ☐ % ☐ ? ☐ &

Now return to the original heap (without the min element removed). Suppose we want to insert the element ✓ into the heap.

(d) Where in the array (at what indices) might ✓end up?
☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8

(e) Which symbols(s) could be at index 2 in the resulting min-heap?
☐ ! ☐ @ ☐ # ☐ $ ☐ % ☐ ? ☐ & ☐ ✓

(f) Which indice(s) in the resulting array might contain the 3rd smallest value?
☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8

Q8: Combinatorial Explosion

(1500 Points)

Consider the following functions:

```
public static int multiply(int a, int b) {
    int val = 0;
    for (int i = 0; i < a; i += 1) {
        val = val + b;
    }
    return val;
}

public static int fact(int n) {
    if (n == 0) { return 1; }
    return multiply(n, fact(n - 1));
}

public static int multiplyAll(List<Integer> lst) {
    Stack<Integer> values = new Stack<>(lst);
    while (values.size() > 1) {
        int i = values.remove();
        int j = values.remove();
        values.insert(multiply(i, j));
    }
    return values.remove();
}

public static int grahamHelper(int base, int pow, int iter) {
    if (iter == 0) { return multiply(base, pow); }
    if (pow == 0) { return base; }
    return grahamHelper(base, grahamHelper(base, pow - 1, iter), iter - 1);
}

public static int grahamCracker(int n) {
    g = grahamHelper(3, 3, 4);
    for (int i = 1; i < 64; i += 1) {
        g = grahamHelper(3, 3, g);
    }
    return g;
}
```

Assume that all arithmetic operations (+, $-$, %, and the built-in multiplication operator $*$, but NOT our multiply function) run in constant time, regardless of their inputs.

Summation Reference Guide:
If you have a sum of the form $f(1) + f(2) + ... + f(n)$, then its asymptotic growth is:

- $\Theta(nf(n))$ if f is polynomial or logarithmic. For example, $1 + 2 + 3 + ... + n = \Theta(n * n) = \Theta(n^2)$.
- $\Theta(f(n))$ if f is exponential or larger. For example, $1 + 2 + 4 + ... + 2^n = \Theta(2^n)$.

(a) What is the runtime of `fact` in terms of N?

○ $\Theta(1)$ ○ $\Theta(\log N)$ ○ $\Theta(N)$ ○ $\Theta(N \log N)$ ○ $\Theta(N^2)$
○ $\Theta((N-1)!)$ ○ $\Theta(N!)$ ○ $\Theta(2^N)$ ○ $\Theta(N^{N-1})$ ○ $\Theta(N^N)$

(b) If `lst` has N elements, all of which are less than N, what is the best-case and worst-case asymptotic runtime of `multiplyAll` in terms of N?

Best Case:

○ $\Theta(1)$ ○ $\Theta(\log N)$ ○ $\Theta(N)$ ○ $\Theta(N \log N)$ ○ $\Theta(N^2)$
○ $\Theta((N-1)!)$ ○ $\Theta(N!)$ ○ $\Theta(2^N)$ ○ $\Theta(N^{N-1})$ ○ $\Theta(N^N)$

Worst Case:

○ $\Theta(1)$ ○ $\Theta(\log N)$ ○ $\Theta(N)$ ○ $\Theta(N \log N)$ ○ $\Theta(N^2)$
○ $\Theta((N-1)!)$ ○ $\Theta(N!)$ ○ $\Theta(2^N)$ ○ $\Theta(N^{N-1})$ ○ $\Theta(N^N)$

(c) What is the runtime of `grahamCracker` in terms of N?

○ $\Theta(1)$ ○ $\Theta(\log N)$ ○ $\Theta(N)$ ○ $\Theta(N \log N)$ ○ $\Theta(N^2)$
○ $\Theta((N-1)!)$ ○ $\Theta(N!)$ ○ $\Theta(2^N)$ ○ $\Theta(N^{N-1})$ ○ $\Theta(N^N)$

The previous functions are copied to this page for your convenience:

```
public static int multiply(int a, int b) {
    int val = 0;
    for (int i = 0; i < a; i += 1) {
        val = val + b;
    }
    return val;
}

public static int fact(int n) {
    if (n == 0) { return 1; }
    return multiply(n, fact(n - 1));
}

public static int multiplyAll(List<Integer> lst) {
    Stack<Integer> values = new Stack<>(lst);
    while (values.size() > 1) {
        int i = values.remove();
        int j = values.remove();
        values.insert(multiply(i, j));
    }
    return values.remove();
}

public static int grahamHelper(int base, int pow, int iter) {
    if (iter == 0) { return multiply(base, pow); }
    if (pow == 0) { return base; }
    return grahamHelper(base, grahamHelper(base, pow - 1, iter), iter - 1);
}

public static int grahamCracker(int n) {
    g = grahamHelper(3, 3, 4);
    for (int i = 1; i < 64; i += 1) {
        g = grahamHelper(3, 3, g);
    }
    return g;
}
```

What would the asymptotic runtime of this program be if we make the following changes? Each subpart is independent; changes made in an earlier subpart do not propagate to later subparts.

(d) What is the asymptotic runtime of `fact` in terms of N if the definition of `multiply` was replaced with the following line: **`return`** `a * b;`?

◯ $\Theta(1)$	◯ $\Theta(\log N)$	◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$
◯ $\Theta((N-1)!)$	◯ $\Theta(N!)$	◯ $\Theta(2^N)$	◯ $\Theta(N^{N-1})$	◯ $\Theta(N^N)$

(e) Which of the following correctly describes the runtime of `fact` in terms of N if we swapped the arguments of `multiply`, so that the last line read **`return`** `multiply(fact(n-1), n);`? Select all that apply.

☐ $\Theta(1)$	☐ $\Theta(\log N)$	☐ $\Theta(N)$	☐ $\Theta(N \log N)$	☐ $\Theta(N^2)$
☐ $\Theta((N-1)!)$	☐ $\Theta(N!)$	☐ $\Theta(2^N)$	☐ $\Theta(N^{N-1})$	☐ $\Theta(N^N)$
☐ $O(1)$	☐ $O(\log N)$	☐ $O(N)$	☐ $O(N \log N)$	☐ $O(N^2)$
☐ $O((N-1)!)$	☐ $O(N!)$	☐ $O(2^N)$	☐ $O(N^{N-1})$	☐ $O(N^N)$

(f) Which of the following correctly describe the best-case and worst-case asymptotic runtime of `multiplyAll` in terms of N, **if we replaced the stack with a queue**. As before, `lst` has N elements, all of which are less than N. Select all that apply.

Best Case:

☐ $\Theta(1)$	☐ $\Theta(\log N)$	☐ $\Theta(N)$	☐ $\Theta(N \log N)$	☐ $\Theta(N^2)$
☐ $\Theta((N-1)!)$	☐ $\Theta(N!)$	☐ $\Theta(2^N)$	☐ $\Theta(N^{N-1})$	☐ $\Theta(N^N)$
☐ $O(1)$	☐ $O(\log N)$	☐ $O(N)$	☐ $O(N \log N)$	☐ $O(N^2)$
☐ $O((N-1)!)$	☐ $O(N!)$	☐ $O(2^N)$	☐ $O(N^{N-1})$	☐ $O(N^N)$

Worst Case:

☐ $\Theta(1)$	☐ $\Theta(\log N)$	☐ $\Theta(N)$	☐ $\Theta(N \log N)$	☐ $\Theta(N^2)$
☐ $\Theta((N-1)!)$	☐ $\Theta(N!)$	☐ $\Theta(2^N)$	☐ $\Theta(N^{N-1})$	☐ $\Theta(N^N)$
☐ $O(1)$	☐ $O(\log N)$	☐ $O(N)$	☐ $O(N \log N)$	☐ $O(N^2)$
☐ $O((N-1)!)$	☐ $O(N!)$	☐ $O(2^N)$	☐ $O(N^{N-1})$	☐ $O(N^N)$

Nothing on this page is worth any points.

Q9: 61Brief 61Battle

(0 Points)

The shortest recorded war in history happened in 1896 and lasted for only 38 minutes, after which one side surrendered. What were the 2 countries involved in the war?

Q10: Compensation

(0 Points)

We forgot a fun question on the last exam, so now you get two! Find a polynomial with integer coefficients that is satisfied by $2+\sqrt{3}$ but is NOT satisfied by $2-\sqrt{3}$, or prove that no such polynomial exists.

Feedback

(0 Points)

Leave any feedback, comments, concerns, or drawings below!