UC Berkeley – Computer Science
CS61BL: Data Structures

Midterm 2 Solutions, Summer 2016

This test has 9 questions worth a total of 45 points. The exam is closed book, except that you are allowed to use two double-sided pages of notes as a cheat sheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Write the statement out below in the blank provided and sign. You may do this before the exam begins. **Any plagiarism, no matter how minor, will result in points deducted from your exam.**

"I have neither given nor received any assistance during the taking of this exam."

_____

_____

Signature: _____

Write your name and student ID on the front page. Write the names of your neighbors. Write and sign the given statement. Once the exam has started, write your login in the corner of every page.

Name:	**Sarallahan Kyao**	**Your Login:**	cs61bl-**ab**
SID:	**6170628494**	**Name of person to left:**	**Edwin Liao**
TA:	**머리스 선배님**	**Name of person to right:**	**Joseph Moghadam**

Notes:

- 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.
- Unless otherwise stated, you can use any standard library classes & methods, and can assume imports happen automatically.
- 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 during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'

Optional. Mark along the line to show your feelings on the spectrum between 😟 and 😂.

Before exam: [😟 _|_____ 😂]

After exam: [😟 _____|___ 😂]

1. Pidgey (6 pts)

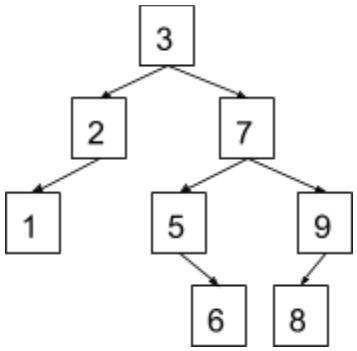
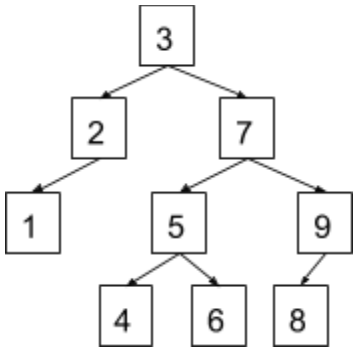
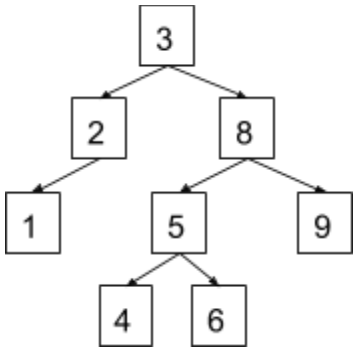
a. For the below problems, assume `N` is the number of items or nodes in the data structure. Do not provide an explanation. Provide an exact answer.

- The number of edges in a rooted tree is ___`N-1`___
- The height of a leaf node in a rooted tree is ___`0`___
- Given a rooted tree, defining a leaf to be the new root leaves a tree of size ___`N`___

Provide answers in Big-Theta notation if possible, otherwise use Big-O. Give the tightest possible bound.

- The height of a binary search tree is in ___`O(N)`___
- The best-case runtime of insertion into a BST is in ___`Θ(1) or O(1)`___
- The best-case runtime of insertion into a red-black tree is in ___`Θ(log N)`___
- The length of a path between two nodes in a red-black tree is in ___`O(log N)`___
- The worst-case runtime of inserting an `Integer` into a `HashSet` is in ___`Θ(N)`___
- The worst-case runtime of `HashSet contains()` on a `String` of length `M` is in ___`Θ(NM)`___
- The runtime of adding `M` `Integer`s into the back of an *empty* `ArrayList` is in ___`Θ(M)`___
- The worst-case runtime of `contains()` on a `HashSet<Integer>` is in ___`Θ(N)`___

b. Draw the result of the standard insert and remove operations, one after another, on this BST.

Original	`insert(4)`	`remove(7)`
		 (Replacing 7 with the inorder predecessor, 6, is also correct)

2. Golduck (3 pts)

A normal generic linked list contains objects of only one type. But we can imagine a generic linked list where entries alternate between two types. `AltList` is an implementation of such a data structure:

```
public class AltList<X, Y> {
    private X item;
    private AltList<Y, X> next;

    AltList(X item, AltList<Y, X> next) {
        this.item = item;
        this.next = next;
    }
}
```

Let's construct an `AltList` instance:

```
AltList<Integer, String> list =
    new AltList<Integer, String>(5,
        new AltList<String, Integer>("cat",
            new AltList<Integer, String>(10,
                new AltList<String, Integer>("dog", null))));
```

This list represents `[5 cat 10 dog]`. In this list, assuming indexing begins at 0, all even-index items are `Integer`s and all odd-index items are `String`s.

Write an instance method called `pairsSwapped()` for the `AltList` class that returns a copy of the original list, but with adjacent pairs swapped. Each item should only be swapped once. This method should be non-destructive: it should not modify the original `AltList` instance.

For example, calling `list.pairsSwapped()` should yield the list `[cat 5 dog 10]`. There were two swaps: "cat" and 5 were swapped, then "dog" and 10 were swapped. You may assume that the list on which `pairsSwapped()` is called has an **even non-zero** length. Your code should maintain this invariant.

```
public class AltList<X, Y> {
    // ... continued from above
    public AltList<Y, X> pairsSwapped() {
        AltList<Y, X> ret = new AltList<Y, X>(next.item, new AltList<X, Y>(item, null));
        if (next.next != null) {
            ret.next.next = next.next.pairsSwapped();
        }
        return ret;
    }
}
```

3. Zubat (4 pts)

Consider the following classes and their hashcodes and equality definitions. There is a problem with each `hashCode()` method below (correctness, distribution, efficiency). **Provide a one-sentence explanation.** Do not list more than one problem. Assume there are no problems with the correctness of `equals`; any code for handling casting is omitted for space.

Code	Problem(s), if any

`class DynamicString {`
`  ArrayList<Character> vals;`
 	
`  public int hashCode() {`
`  int h = 0;`
`  for (int i = 0; i < vals.size(); i++) {`
`  h = 31 * h + vals.get(i);`
`  }`
`  return h;`
`  }`
 	
`  public boolean equals(Object o) {`
`  DynamicString d = (DynamicString) o;`
`  return vals.equals(d.vals);`
`  }`
`}`
	No problems
	
	(This one is done for you as an example using Java's `String::hashCode`.)

`class PokeTime {`
`  int startTime;`
`  int duration;`
 	
`  public int getCurrentTime() {`
`  // Gets the current system clock time`
`  }`
 	
`  public int hashCode() {`
`  return 1021 * (startTime + 1021`
`  * duration + getCurrentTime());`
`  }`
 	
`  public boolean equals(Object o) {`
`  PokeTime p = (PokeTime) o;`
`  return p.startTime == startTime`
`  && p.duration == duration;`
`  }`
`}`
	Incorrect: The `hashCode` is non-deterministic.

```
class Phonebook {
    List<Human> humans;

    public int hashCode() {
        int h = 0;
        for (Human human : humans) {
            // Assume Human hashcode is correct
            h = (h + human.hashCode()) % 509;
        }
        return h;
    }

    public boolean equals(Object o) {
        Phonebook p = (Phonebook) o;
        return p.humans.equals(humans);
    }
}
```

Poor distribution: The `hashCode` only distributes numbers between -508 and 508, which is an inefficient use of the full integer range and will cause more collisions than necessary.

```
class Person {
    Long id;
    String name;
    Integer age;

    public int hashCode() {
        return id.hashCode() + name.hashCode()
               + age.hashCode();
    }

    public boolean equals(Object o) {
        Person p = (Person) o;
        return p.id == id;
    }
}
```

Incorrect: `Person`s that are `equals()` do not necessarily have the same `hashCode()`.

```
class DblCharSeq {
    char[] seq1;
    char[] seq2;

    public int hashCode() {
        int h = 0;
        for (char c1 : seq1) {
            for (char c2 : seq2) {
                h = 31 * (31 * h + c1) + c2;
            }
        }
        return h;
    }

    public boolean equals(Object o) {
        DblCharSeq d = (DblCharSeq) o;
        return Arrays.equals(seq1, d.seq1)
               && Arrays.equals(seq2, d.seq2);
    }
}
```

Inefficient: The `hashCode` takes quadratic time to compute when it could have taken only linear time.

4. Vaporeon (6 pts)

Suppose we have a `HashMap`, but want to be able to undo operations made on it. Implement `HistoryMap` below to have this functionality. The only operations that we care about that modify the structure are `put` and `remove`.

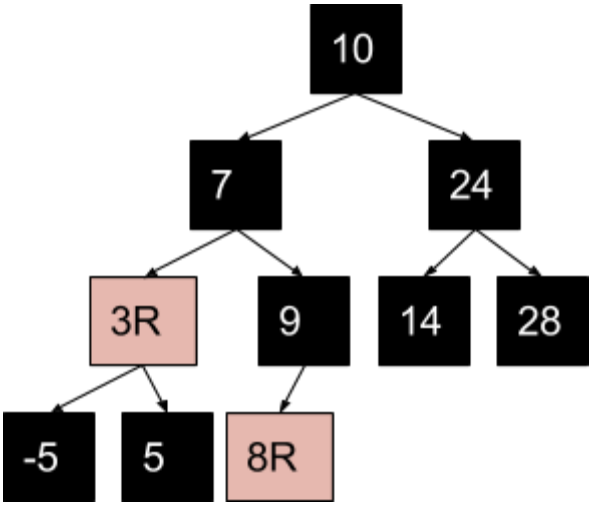
Calling `undo` should revert the state of the `HistoryMap` to before the last `put` or `remove`, whichever was most recent. See the `main` method for example behavior. Assume remove is used correctly; any key removed is assumed to already exist in the `HistoryMap`. You may not need all lines.

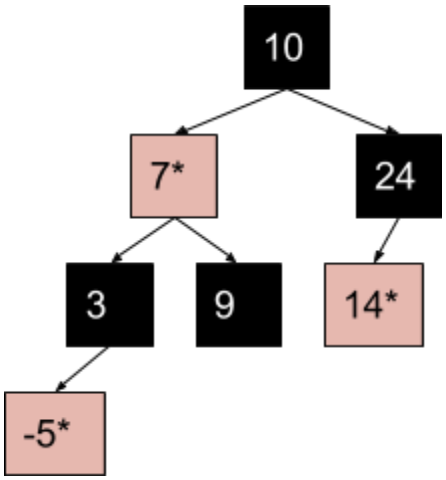
```
public class HistoryMap<K, V> extends HashMap<K, V> {
    Stack<Operation> history = new Stack<>();
    class Operation {  /* Helper class */
        /* Place fields/variables here */
        boolean shouldRemove;
        K key;
        V value;

        /* Place the constructor here */
        Operation (boolean shouldRemove, K key, V value) {
            this.shouldRemove = shouldRemove;
            this.key = key;
            this.value = value;
        }
    }

    @Override
    /** Remember that in a HashMap, a null value is valid */
    public V put(K key, V value) {
        history.push(new Operation(!containsKey(key), key, super.get(key)));
        return super.put(key, value);
    }
    @Override
    public V remove(Object key) {
        history.push(new Operation(false, (K) key, super.get(key)));
        return super.remove(key);
    }
    // Continues on next page
```

```
    @Override
    public boolean containsKey(K key) {
        return super.containsKey(key);
    }

    public void undo() {
        if (history.isEmpty()) {
            return;
        }
        Operation op = history.pop();
        if (op.shouldRemove) {
            super.remove(op.key);
        } else {
            super.put(op.key, op.value);
        }
    }

    public static void main(String[] args) {
        HistoryMap<String, Integer> h = new HistoryMap<>();
        h.put("party", 1);
        h.put("parrot", 2);
        h.put("conga", 4);
        h.put("parrot", 3);
        h.undo();
        h.undo();
        System.out.println(h); // Output: {parrot=2, party=1}
        h.remove("party");
        h.undo();
        System.out.println(h); // Output: {parrot=2, party=1}
    }
}
```

5. Wigglytuff (4 pts)

a. Consider the following Left-Leaning Red-Black tree, where red nodes are marked with an asterisk (*). Draw the resulting tree after inserting 28, 5, and 8 in order. Clearly mark red nodes with the letter R.





b. Now consider these BST operations. Red-Black trees are also valid BSTs, but sometimes require modified operations. Check the boxes where the **BST operations** correctly function on the given data structure without breaking invariants **without any modifications**.

BST Operation	Red-Black Tree	Left-Leaning Red-Black Tree
`void delete(T o);`		
`boolean contains(T o);`	✓	✓
`void insert(T o);`		
`T getElementAt(int i);`	✓	✓
`List getElementsBetween(T s, T t);`	✓	✓
`List getAllElements();`	✓	✓

6. Porygon (8 pts)

Finally! You've garnered a coveted interview with Litman Chen for an internship at Kelp. Determine which data structure(s) are best suited for the scenarios below in terms of performance, taking into account the specific types of inputs listed in each problem. Your descriptions of the data structure(s) chosen should be **brief** but sufficiently detailed so that the runtime is unambiguous. Give the worst-case runtime bound of the solution in Big-Theta notation.

a. Mr. Chen has loaded all the reviews stored on Kelp into a text document of N words. Find the number of occurrences of each unique word.

Data Structures: `HashMap<String, Integer>`
Usage: Iterate through the words, maintaining a mapping from word to number of times encountered.
Runtime: `Θ(N)`

b. Kelp has a collection of N reviews. Each review has a date, author name, number of stars and the text contents of the review. Mr. Chen wants to query the number of reviews within a certain date-time range. Optimize for both query and construction time.

Data Structures: `ArrayList<Review>`
Usage: Maintain an `ArrayList` ordered on the date of the review. At query time, binary search for the indices corresponding to the endpoints; the difference is the number of reviews in the range.
Runtime (construction): `Θ(N log N)` Runtime (query): `Θ(log N)`
Note: `TreeSet<Review>` solutions must mention modification to `TreeSet` that involves storing the sizes of the subtrees at each node in order to get full points.

c. Kelp is releasing a new product Kelp-Komplete, a text based auto-complete engine! It stores a collection of N words. Given a query string of K characters, determine if it is a prefix of any of the words and thus can be kelp-kompleted. Assume that the each of the N words have a maximum length of M. Additionally, give the runtime for both construction and query.

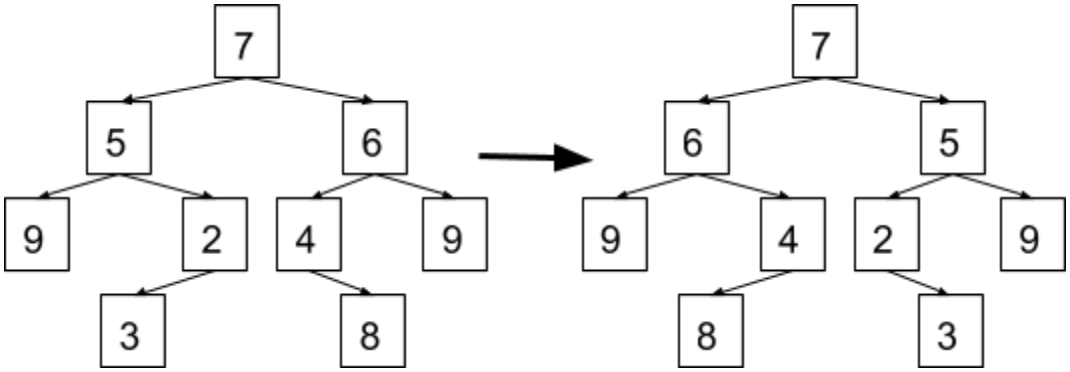
Data Structures: `Trie`
Usage: Construct a trie on the words using the ascii alphabet. Query by traversing until the end of the query string is met.
Runtime (construction): `Θ(NM)` Runtime (query): `Θ(K)`

d. Kelp is now trying to get into the field of visual computing. Mr. Chen gives you N images of size 256x256, each represented as an int[][] array, that you'll need to store in a collection. Each image has associated with it a saturation value, which can be calculated from the pixels. Support the following operations: `add(int[][] img)`, `getAllImgWithSaturation(int saturation)`, and `remove(int[][] img)`.

Data Structures: `HashMap<Integer, HashSet<ImageContainer>>`
Usage: Maintain a mapping from saturation to a set of images with that saturation for easy `add` and `get`. Construct a helper class, `ImageContainer`, that computes the `hashCode` of the image and stores it; `remove` computes the saturation and then attempts to remove the image from the `HashSet`.
Runtime (construction): `Θ(N)`

7. Lapras (6 pts)

Fill in a method, `Tree::flipHorizontally`, which should flip a **symmetric** binary tree's values destructively about the root in linear time. Some helper methods (`swapNumbers` and `safePush`) are given. You may not define your own helper methods. See the example:




```
public class Tree {
    private TreeNode root;

    private static class TreeNode {
        private int num;
        private TreeNode left, right;

        private TreeNode(int num, TreeNode left, TreeNode right) {
            this.num = num;
            this.left = left;
            this.right = right;
        }
    }

    private static void swapNumbers(TreeNode t1, TreeNode t2) {
        int temp = t1.num;
        t1.num = t2.num;
        t2.num = temp;
    }

    private static void safePush(TreeNode t, Stack<TreeNode> s) {
        if (t != null) {
            s.push(t);
        }
    }

    // Continues on next page
```

```
    /* This is the two stack solution. */
    public void flipHorizontally() {
        Stack<TreeNode> forwardsTraversal = new Stack<>();
        Stack<TreeNode> backwardsTraversal = new Stack<>();
        safePush(root.left, forwardsTraversal);
        safePush(root.right, backwardsTraversal);
        while (!forwardsTraversal.isEmpty()) {
            TreeNode first = forwardsTraversal.pop();
            TreeNode second = backwardsTraversal.pop();
            swapNumbers(first, second);

            safePush(first.right, forwardsTraversal);
            safePush(first.left, forwardsTraversal);
            safePush(second.left, backwardsTraversal);
            safePush(second.right, backwardsTraversal);
        }
    }

    /* This is the one stack solution. */
    public void flipHorizontally() {
        Stack<TreeNode> fringe = new Stack<>();
        safePush(root.right, fringe);
        safePush(root.left, fringe);
        TreeNode leftNode, rightNode;
        while (!fringe.isEmpty()) {
            leftNode = fringe.pop();
            rightNode = fringe.pop();
            swapNumbers(leftNode, rightNode);

            safePush(rightNode.right, fringe);
            safePush(leftNode.left, fringe);
            safePush(rightNode.left, fringe);
            safePush(leftNode.right, fringe);
        }
    }
}
```

Alakazam (0 pts)

This is a designated ExamFunZone™©. Draw or write whatever you want.

8. Magikarp (3 pts)

Fill out `DuplicateIterator` so that it works as used in the example `main` method. (The `main` method is at the bottom of the next page.) When given two **sorted** input `Iterator`s, the `DuplicateIterator` returns the elements that are within both iterators. Note the helper method `findNextElement`. You may not need all lines.

```
public class DuplicateIterator<T extends Comparable<? super T>> implements Iterator<T> {
    private Iterator<T> iter1, iter2;
    private T nextElement = null;

    public DuplicateIterator(Iterator<T> iter1, Iterator<T> iter2) {
        this.iter1 = iter1;
        this.iter2 = iter2;
        findNextElement();
    }

    public boolean hasNext() {
        return nextElement != null;
    }

    public T next() {
        T toReturn = nextElement;
        findNextElement();
        return toReturn;
    }

    /** Sets the nextElement instance variable to the next duplicate element
      * (or null if there is no remaining duplicate element). */
    private void findNextElement() { ... }

    public static void main(String[] args) {
        Iterator<Integer> iter1 = Arrays.asList(1, 2, 4, 5, 6, 9).iterator();
        Iterator<Integer> iter2 = Arrays.asList(1, 2, 3, 5, 7, 10).iterator();
        DuplicateIterator<Integer> di = new DuplicateIterator<>(iter1, iter2);
        di.forEachRemaining(o -> System.out.print(o.toString() + " ")); // Prints 1 2 5
    }
}
```

9. Dragonite (5 pts)

a. Write a method, `makeArrayReducer`, that takes in a non-empty array `E[] arr` and returns a `Function`. The returned `Function f` should take in a `BinaryOperator<E>` and have a return type of `E`. `f` should compute the result of the reduce operation on `arr`, given the `BinaryOperator`. You may not need all lines. **You do <u>not</u> have to use any stream related methods**, but if you do, there is no partial credit for incorrect syntax**.**
Recall that a `BinaryOperator<E>` takes in two arguments of type `E` and outputs an argument of type `E`. A `Function<T, R>` takes in an argument of type `T` and returns an argument of type `R`. Both are functional interfaces that have a single `apply` method, and instance references can be replaced with lambda statements and method references.

```
public static <E> Function<BinaryOperator<E>, E> makeArrayReducer(E[] arr) {
    assert arr.length != 0;
    return f -> {
        E reduced = arr[0];
        for (int i = 1; i < arr.length; i++) {
            reduced = f.apply(reduced , arr[i]);
        }
        return reduced;
    };
}
```

Alternatively (you were not expected to remember the syntax for this):

```
public static <E> Function<BinaryOperator<E>, E> makeArrayReducer(E[] arr) {
    assert arr.length != 0;
    return f -> Arrays.stream(arr).reduce(f).get();
}
```

b. Now given an example array below, fill in the blanks to compute the array max and array sum.

```
final Double[] arr = {1.0, 0.99, 0.98, 0.97, 0.96, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1};
Function<BinaryOperator<Double>, Double> reducer = makeArrayReducer(arr);

double arr_max = reducer.apply(Math::max);

double arr_sum = reducer.apply((x, y) -> x + y);
```

c. Using `makeArrayReducer`, write a one line method, `join`, that concatenates all the `String`s in a string array together using the delimiter to separate the entries in the array.

```
public static String join(String[] arr, String delimiter) {

    return makeArrayReducer(arr).apply((s1, s2) -> s1 + delimiter + s2);
}
```

Example usage:

```
final String[] strs = {"Na", "na", "nah", "Nah", "BATMAN!"};
String batmobile = join(strs, "NA");
System.out.println(batmobile);
// Output: NaNAnaNAnahNANahNABATMAN!
```