CS 61BL Data Structures & Programming Methodology

Summer 2018

Midterm 3 Solution

This exam has 9 questions worth a total of 25 points and is to be completed in 80 minutes. The exam is closed book except for three double-sided, handwritten cheat sheets. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement below in the blank provided and sign. You may do this before the exam begins.**

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

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

Signature: Blear Hug

Question	Points
1	½
2	½
3	2
4	6
5	5
6	3
7	4
8	4
9	0
Total	25

Name	Blear Hug
Student ID	1234567890
Lab Section	0 0 1
Name of person to left	Christine Zhou
Name of person to right	Kevin Lin

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are much more complicated than necessary.
- **Work through the problems 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**. For code-writing questions, **write only one statement per line** and **do not write outside the lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed, but in the event that we do 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.'**

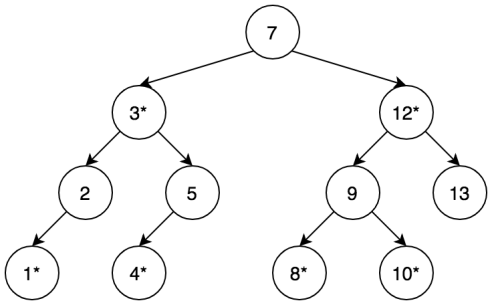
1. (½ pt) **Your Thoughts** If you were a food, what food would you be and why?

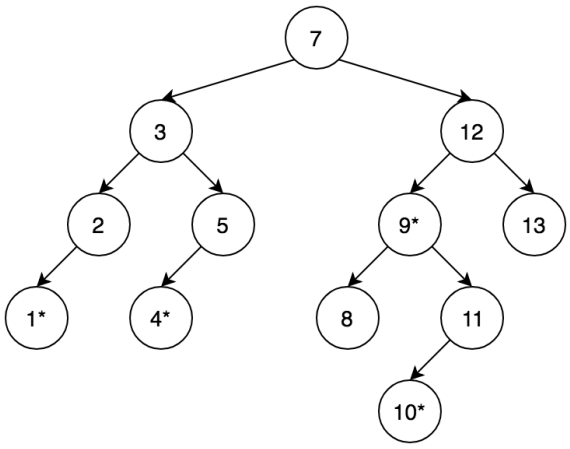
   A golden pell bepper.

2. (½ pt) **So It Begins** Write the statement on the front page and sign. Write your name, ID, and your lab section. Write the names of your neighbors. Write your name in the corner of every page.

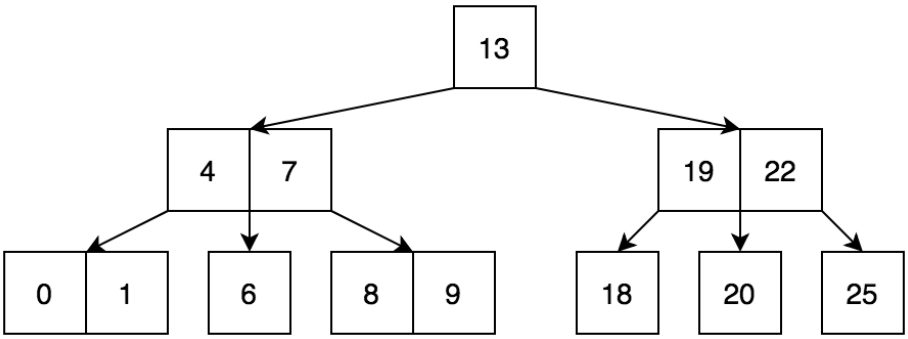
3. **Balanced Search Trees** If you need to split a node in a *2-3 tree*, insert the new element in the corresponding node and send up the resulting middle element. If you need to split a node in a *2-3-4 tree*, send up the middle element, and then insert the new element in the corresponding node.

   (a) (1 pt) Draw the resulting red-black tree in the box after inserting 11 into the tree below. Indicate red nodes by putting an asterisk (*) next to the number of the node. **If a node has one red child, draw it as left-leaning.**





   (b) (1 pt) Answer the following questions about *2-3 trees* **with exact values**. 2-3 trees may contain any integer value as an element.



   (a) What is the **maximum** number of items that we can insert and **not increase** the height?

   (a) 13

   (b) What is the **minimum** number of items that we can insert and **increase** the height?

   (b) 3

Name: Blear Hug

4. **Design** Choose the data structure that would best solve each scenario in terms of execution time and ease of use and give the **worst-case runtime** in $\Theta(\cdot)$ notation. Then, explain how you would use the data structure in *no more than three lines*. Assume linked lists have a sentinel and references to the first and last node. Assume each item can be hashed in constant time with a uniform distribution and can be compared with each other. There may be multiple correct answers, but choose only one.

   (a) (2 pts) Consider a smartphone notification system. Notifications can be *added* and *processed* one at a time. *Processing* a notification removes the most recently-added notification from the system. At any given time, there will be at most N notifications.

   ● Linked list ○ Fixed-size array ○ Red-black tree ○ Binary heap ○ Hash table

   Add: $\Theta($1$)$ *Process*: $\Theta($1$)$

   Linked list. We have pointers to the front and the back of the list, so adding or removing elements from the front or back will be fast. We can always add to and remove from the front of the list whenever we add a notification and process a notification respectively because we only need to know about the most recent notification.

   (b) (2 pts) You are designing a game where users can play for free, but they can also pay real money to unlock different characters. The game has a total of N different software bugs which need to be fixed. Users can submit bug reports for any number of these N bugs. Only one bug can be fixed at a time, so we want to choose the fix that resolves the most bug reports.

   However, you consider paying users twice as important as non-paying users. Any bug report submitted by a paying user is worth two bug reports from a non-paying user. Design a system that will let users *report* bugs, *find* the *most important bug* with the most bug reports — with special consideration to reports from paying users, and *remove* the most important bug.

   ○ Linked list ○ Fixed-size array ○ Red-black tree ● Binary heap ○ Hash table

   Report: $\Theta($N$)$ *Find*: $\Theta($1$)$ *Remove*: $\Theta(\log N)$

   Heap. Each bug is an item in the heap with the number of bug reports submitted for that bug as the priority value. We will be using a max heap in order to keep track of which bug has been reported by the most people. When a new bug is reported, insert it into the heap with priority 1 or 2 (depending on if the user is paying or non-paying). When a bug already in the system is reported, find the item in the heap, increment the priority of the bug by 1 (non-paying) or 2 (paying), and reheapify that item. Returning the next bug to fix will be a constant time peek operation, but removing will be a logarithmic time poll operation.

   (c) (2 pts) Design a `DataType` that can *add* items and return all added items in *sorted* order.

   ```
   DataType dt = new DataType(); dt.add(10); dt.add(4); dt.add(3); dt.add(8);
   dt.sorted(); // 3 4 8 10
   dt.add(5); dt.add(7); dt.sorted(); // 3 4 5 7 8 10
   ```

   Assume the `DataType` already contains N items.

   ○ Linked list ○ Fixed-size array ● Red-black tree ○ Binary heap ○ Hash table

`add`: $\Theta($log N $)$ `sorted`: $\Theta($ N $)$

Red-black tree. If there are N items in `DataType`, then the height of the tree will be $\log N$. Inserting an item into the tree will take $\log N$ time, and the sorted order of all the items will be an inorder traversal of the red-black tree, which will take N time.

Name: Blear Hug

5. (5 pts) **Hashing** Recall that the `String::equals` method is defined as follows.

```
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof String)) return false;
    String other = (String) o;
    if (length() != other.length()) return false;
    for (int i = 0; i < length(); i += 1)
        if (charAt(i) != other.charAt(i)) return false;
    return true;
}
```

Suppose we have three hash table `Set` implementations that use linked lists for external chaining: (i) resizing when load factor $L = 1$, (ii) resizing when $L = \sqrt{N}$, and (iii) *without resizing*, $L = \varnothing$. Give the **worst-case runtime for inserting one** `String` in terms of M, length of the string, and N, number of items. **For this entire question, assume this insertion does not cause a resize.**

(a) For part (a), assume that the `String::hashCode` given below **distributes values uniformly**.

```
public int hashCode() {
    int h = 0;
    for (int i = 0; i < length(); i += 1)
        h = 31 * h + charAt(i);
    return h;
}
```

i. $L = 1$: $\Theta($*M*$)$ ii. $L = \sqrt{N}$: $\Theta($*M*$\sqrt{N}$$)$ iii. $L = \varnothing$: $\Theta($*MN*$)$

(b) Suppose that we changed the `String::hashCode` to the following.

```
public int hashCode() {
    return charAt(0) + charAt(length() - 1);
}
```

i. $L = 1$: $\Theta($*MN*$)$ ii. $L = \sqrt{N}$: $\Theta($*MN*$)$ iii. $L = \varnothing$: $\Theta($*MN*$)$

A `MutableString` can change its *current value* by calling `mutate(String value)`. Fill in a `newValue` so that the *current value* of `test` changes and `main` prints `true` **for any initial capacity**, `C`. Assume `MutableString::equals` calls `String::equals` on its *current value*, and `MutableString::hashCode` calls the `hashCode` defined in part (b) on its *current value* as well.

```
public static void main(String[] args) {
    HashSet<MutableString> set = new HashSet<>(C);
    MutableString test = new MutableString("pell bepper");
    set.add(test);
    String newValue = "pear          ";
    test.mutate(newValue);
    System.out.println(set.contains(new MutableString(newValue))); // true
}
```

6. (3 pts) **Restaurants** It's lunch time and you're hungry. But you don't want to eat just anywhere, you want to eat at the highest-rated places! Implement `kBestRestaurants`, which takes a list of all the `restaurants` in the area and an integer, `k`, and returns an array of the `k` highest-rated restaurants **sorted from highest-rated to lowest-rated** in $O(N \log k)$ time. The `Restaurant` class is defined below, where restaurants with larger `rating` values are rated higher than restaurants with smaller `rating` values. The `PriorityQueue` class is also defined below and is implemented with a binary min heap.

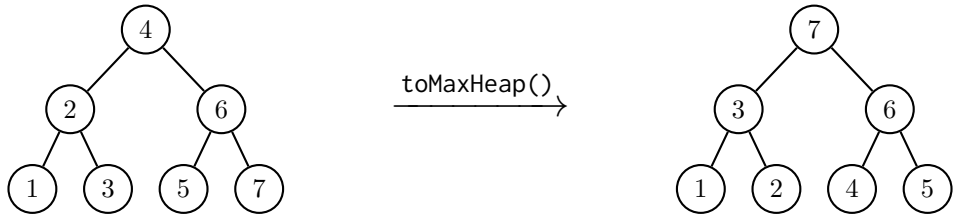
```
public class Restaurant {
    String name;
    Double rating;
}

public static Restaurant[] kBestRestaurants(List<Restaurant> restaurants, int k) {
    PriorityQueue<Restaurant> pq = new PriorityQueue<>(k,

            (o1, o2) -> o1.rating.compareTo(o2.rating)                    );

    for (Restaurant r : restaurants                                       ) {

        pq.add(r)                                                         ;

        if (pq.size() > k                                                 ) {

            pq.poll()                                                     ;
        }
    }

    Restaurant[] result = new Restaurant[k]                               ;

    for (int i = 0; i < k; i += 1                                         ) {

        result[k - 1 - i] = pq.poll()                                     ;
    }

    return result                                                         ;
}

public class PriorityQueue<E> {
    PriorityQueue(int initialCapacity, Comparator<? super E> comparator);
    boolean add(E e);
    E peek();
    E poll();
    int size();
}
```

Name: Blear Hug

7. (4 pts) **Tree Traversals** In lab, we explored a case of the B-tree called a 2-3-4 tree. Implement `inorderTraversal` for a **B-tree of any number of items**, which returns a list of all the items in sorted order. Recall that each internal node in a B-tree has one more child than number of items. The children of a leaf node are all `null`. You may find the `List` instance method, `boolean addAll(Collection<E> c)`, useful, though this method is not necessary to solve the problem.

```
public class BTree<T extends Comparable<T>> {
    private Node root;

    private class Node {
        private List<T> items;
        private List<Node> children;

        private List<T> helper(List<T> lst) {
            int i;

            for (i = 0; i < items.size(); i += 1                    ) {

                if (children.get(i) != null                         ) {

                    children.get(i).helper(lst)                     ;
                }

                lst.add(items.get(i))                               ;
            }

            if (children.get(i) != null                             ) {

                children.get(i).helper(lst)                         ;
            }
            return lst;
        }
    }

    public List<T> inorderTraversal() {

        if (root != null                                            ) {

            return root.helper(new ArrayList<>())                   ;
        }
        return null;
    }
}
```

8. (4 pts) **Heapify** Implement `toMaxHeap`, which destructively converts a **complete binary search tree** into a valid max heap in time linear to the number of nodes, N, with one additional invariant: for each node, all elements in the left subtree must be smaller than the elements in the right subtree. The `inorderIterator` method is an instance method of `BinaryTree` that will return an `Iterator` object that outputs next element in the inorder traversal of the tree whenever `next` is called.




```
public class BinaryTree<T> {
    private TreeNode root;
    private class TreeNode {
        T item;
        TreeNode left;
        TreeNode right;
    }

    public void toMaxHeap() {
        if (root != null) {
            Iterator<T> iter = inorderIterator();

            helper(root, iter)                                                    ;
        }
    }

    private void helper(TreeNode n, Iterator<T> iter                              ) {

        if (n != null                                  ) {

            helper(n.left, iter)                                                  ;

            helper(n.right, iter)                                                 ;

            n.item = iter.next()                                                  ;
        }
    }
}
```

9. (0 pts) **HBD** Which staff member's birthday is closest to today's date, and how old will they be?

Jonathan Murata, pretty old.

This page intentionally left blank. Please enjoy this space.

(0 pts) Who would win: course staff or a T-rex? Explain your answer pictorally, and give the worst-case runtime bound of the encounter in $\Theta(\cdot)$ notation.

Exam scratch paper.