UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #2, Spring 2019

This test has 11 questions across 12 pages worth a total of 480 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use two double sided written cheat sheet (front and back). 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 your name. You may do this before the exam begins.**

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

Signature: ___________________________

#	Points	#	Points
0	1	6	24
1	33	7	40
2	49	8	56
3	54	9	35
4	46	10	84
5	0	11	60
		TOTAL	480

```
Name: ______________________________
SID: _______________________________
GitHub Account #   : sp19-s_____
Person to Left's # : sp19-s_____
Person to Right's #: sp19-s_____
Exam Room: _________________________
```

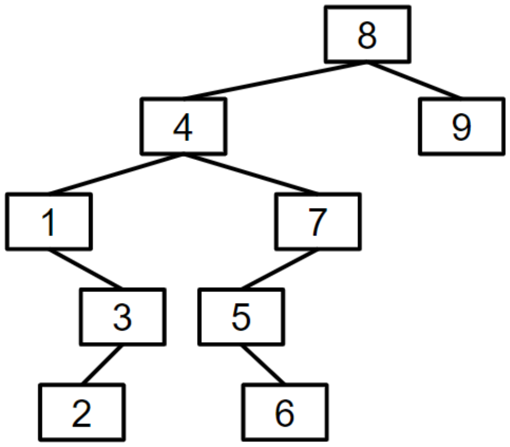
Tips:
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones you are comfortable with first. 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.'**
- ○ 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.

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_______________☺].
After exam: [☹_______________☺].

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 login in the corner of every page. Enjoy your free point ☺.

1. Tree Removal.
a) **(13 points).** Suppose we have the BST shown below. Give the tree that results from deleting "4" using the procedure from class (a.k.a. Hibbard deletion). **Use the successor, not the predecessor**. Draw your answer in the box located to the right of the tree below.



b) **(20 points).** Suppose we want to write a method that deletes the **smallest** item in a BST rooted at **`TreeNode`** `x`. Fill in the code below. The method should return the root of the BST after deletion. Assume the **`TreeNode`** class has three fields: `item`, `left`, and `right`. You may assume `x` is never `null`. If deletion results in an empty tree, you should return null.

```
private TreeNode deleteMin(TreeNode x) {
    if (x.left == null) { return ____________; }
    ___________________ = deleteMin(__________);
    return _____________;
}
```

For reference, the **`TreeNode`** class is defined as shown below:

```
public class TreeNode {
   public TreeNode left;
   public TreeNode right;
   public int item;
   ...
}
```

2. Hash Tables. Suppose we have a correct hash table implementation of a set as seen in lecture, where:

- We store the buckets in an array, where each bucket is a linked list (i.e. separate chaining).
- We resize to 1.5x the number of buckets at the end of an add operation if the load factor is ≥ 1.
- We reduce our hashcode with `floorMod(hashCode(), M)`. Recall that `floorMod` is just %, but handles negative numbers correctly.
- For parts a-c: We currently have N = 19 items and M = 41 buckets. Assume calls to `add` add unique items to the hash table

a) **(5 points).** What is the current load factor? It is OK to leave your answer as a fraction: _________

b) **(10 points).** Suppose a user calls `contains`. Give the minimum and maximum number of times the `contains` method would need to call `equals`. **Do not assume anything about the distribution of items in the hash table.**

Min: _______________ Max: ________________

c) **(10 points).** Give the minimum and maximum number of `add` calls that can cause the next `resize`.

Min: _______________ Max: ________________

d) **(12 points).** Suppose we define an `ExamOomage` class that has 4 instance variables `int r`, `int g`, `int b`, `int a`, each in the range [0, 255]. Assume that it provides a correct `equals` method as well as a `hashCode` method that returns some integer. Do not assume anything about the quality of the `hashCode` method. Suppose we run a timing experiment using a `Problem2HashSet` (as defined in the bulleted list above) as follows:

```
public static void timingTest(int N) {
    Stopwatch sw = new Stopwatch();
    Problem2HashSet<ExamOomage> hsp = new Problem2HashSet<>();
    for (int i = 0; i < N; i += 1) {
        hsp.add(ExamOomage.randomPoint()); //returns ExamOomage with
    }                                      //random r/g/b/a in [0, 255]
    System.out.println(sw.elapsedTime());
}
```

Using the code above we get the runtimes below:

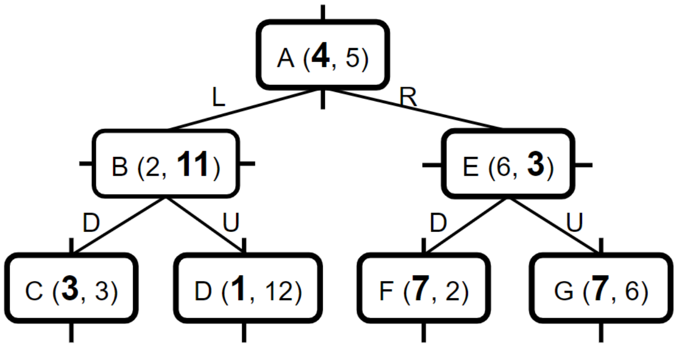
```
N = 1000  : 0.026 seconds
N = 10000 : 1.00 seconds
N = 49123 : 26.96 seconds
N = 100000: 118.43 seconds
```

From the given data, which of the following best matches the **average** time **<u>per add</u>** operation?
<u>Each add</u> on average costs: ○ Θ(1) ○ Θ(log N) ○ Θ(N) ○ $\Theta(N^2)$ ○ $\Theta(2^N)$

e) **(12 points).** Give an example of a `hashCode()` function that would yield timing results similar to those shown in the table above.

```
return _________________________________
```

3. **KdTree**. Suppose we have the KdTree shown below.



a) **(8 points).** Give the result of the call to `nearest(new Point(2, 2))`.

○ A (4, 5) ○ B (2, 11) ○ C (3, 3) ○ D (1, 12) ○ E (6, 3) ○ F (7, 2) ○ G (7, 6)

b) **(10 points).** Assuming we're searching for `nearest(new Point(2, 2))`. Which child of A is the "good child", i.e. should definitely be explored? Briefly explain your answer.

○ B ○ E Explanation: ______________________________________

c) **(10 points).** Assuming we're searching for `nearest(new Point(2, 2))`. Without actually exploring A's children, what is the closest possible point to (2, 2) that could exist in A.right (or its children)? That is, provide an answer based only on how A partitions the space. Give an x and y value.

Best possible point: (_____, _____)

d) **(16 points). Give the order in which the vertices are visited** to complete the call to `nearest` from part a. If some nodes are not explored due to pruning, do not include them. For example, if the KdTree nearest method visited A, then visited E, then pruned both of E's children, then pruned A's left child, you'd write only A then E. The first node has been provided for you. You may not need all provided entries. You can use either the perpendicular or diagonal pruning rule (both give same answer).

A				

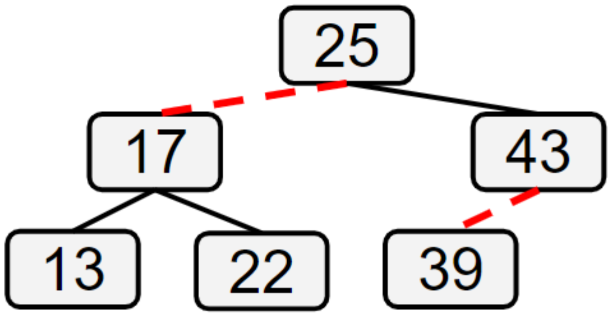
e) **(10 points).** If we added the point (5, 1) to the tree above, where would it go?

○ C.left ○ C.right ○ D.left ○ D.right ○ F.left ○ F.right ○ G.left ○ G.right

4. **LLRBs.**

a) **(10 points).** Suppose we have the LLRB below, where red links are given as thicker dashed lines. If we add 15, what **single** LLRB operation will we need to perform? Give your answer as `rotateLeft(X)`, `rotateRight(X)`, or `colorFlip(X)` where X is the node which we rotate or color flip. Reminder: `colorFlip(17)` would mean flipping all edges touching 17. Don't forget to fill in the value in parenthesis!

○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`



b) **(24 points).** Starting from the figure of the LLRB above (i.e. without adding 15), what LLRB operations will we need to complete if we add the value 40? By LLRB operation, we mean `rotateLeft`, `rotateRight`, and `colorFlip`. Give your answer by filling in one bubble per line and filling in the value in parenthesis, where line 1 corresponds to operation #1, etc. You may not need all five operations.

Op #1: ○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`
Op #2: ○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`
Op #3: ○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`
Op #4: ○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`
Op #5: ○ `rotateLeft(    )` ○ `rotateRight(    )` ○ `colorFlip(    )`

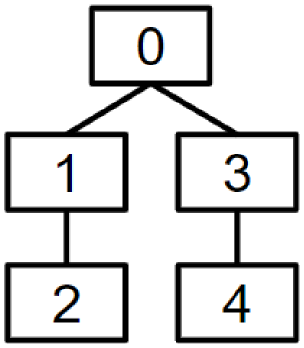
c) **(12 points).** Draw the 2-3 tree corresponding to the LLRB from part a (i.e. without adding 15 or 40).

5. **PNH (0 Points).** What is a "desire path"?

6**. WQUzaaaaaaaaaaaaaaaaaaaaapp (24 points).**

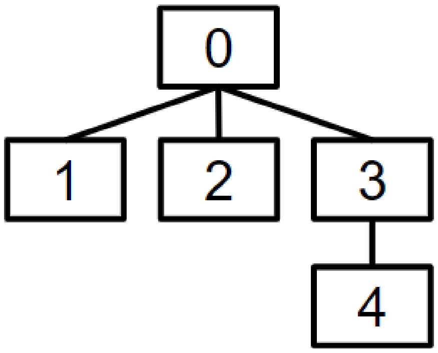
For each of the three figures below, give a sequence of 4 Weighted Quick Union `union` calls (without path compression) that could result in the figure shown. If two trees are unioned with the same weight, assume the **first argument's root is put below the second argument** (i.e. first side's new parent is the second side). If the figure is impossible to create using 4 or fewer `union` calls, fill in the circle marked "Not possible". You may not need all blanks.

Assume that before your `union` calls, all items are originally disconnected.



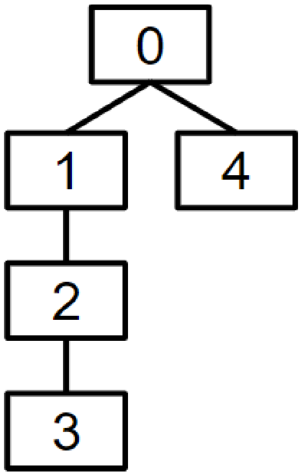
Union(______,______)
Union(______,______)
Union(______,______)
Union(______,______)

○ Not possible



Union(______,______)
Union(______,______)
Union(______,______)
Union(______,______)

○ Not possible

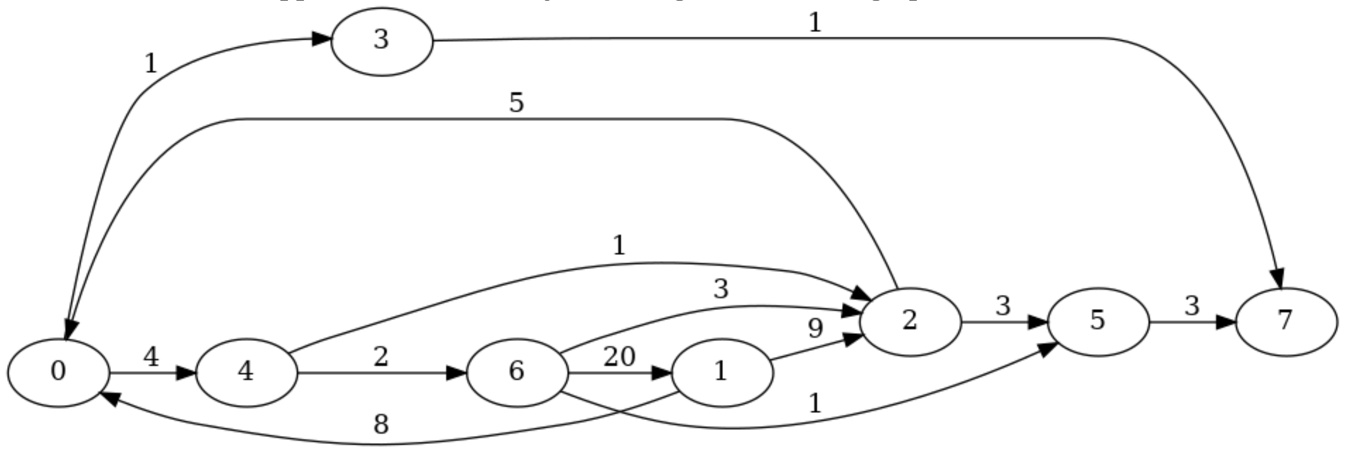


Union(______,______)
Union(______,______)
Union(______,______)
Union(______,______)

○ Not possible

Reminder: For each figure, either fill in the blanks for the `union` calls, or mark "Not possible".

7. Shortest Paths. Suppose that we run Dijkstra's algorithm on the graph below from the vertex 0.



The table below shows the results right after vertex 2's edges have been relaxed.

vertex #	0	1	2	3	4	5	6	7
distTo	0	∞	5	1	4	8	6	2
edgeTo	-	-	4	0	0	2	4	3

a) **(16 points).** In what order were the vertices visited? The last value has been filled in for you.

				2

b) **(4 points).** What vertex will be visited next? _____________

c) **(16 points).** Give the distTo array after the next vertex is visited (i.e. after all its edges are relaxed).

vertex #	0	1	2	3	4	5	6	7
distTo	0							

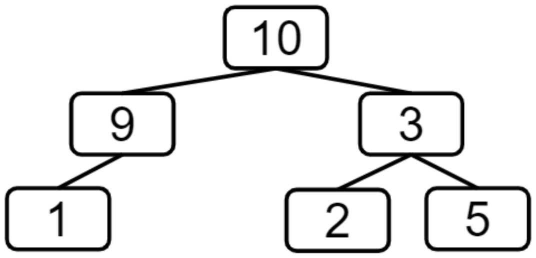
d) **(4 points).** Suppose all of the edges in the graph above were undirected.

How many edges would be in the minimum spanning tree? ________________

Tranquility Area

Mark The Paper In The Way That You Like Best

8. Les Visiteurs. Suppose we have the tree below. Suppose it is implemented using TreeNodes.



```
public class TreeNode {
   public TreeNode left;
   public TreeNode right;
   public int item;
   ...
}
```

In lecture, our implementation for the preorder traversal printed a node when it was visited, but this can be generalized so that "visiting" can be any arbitrary action. Code for this idea is given below.

```
static void preorder(TreeNode n, Visitor v) {
    if (n == null) { return; }
    v.visit(n);
    preorder(n.left, v);
    preorder(n.right, v);
}
```

```
interface Visitor {
  void visit(TreeNode x);
}
```

a. **(8 points).** Suppose we perform a **preorder** traversal on the tree above using the `Printer` visitor below. Write the output of the program in the blank provided.

```
class Printer implements Visitor {
    public void visit(TreeNode x) {
        System.out.print(x.item + " ");
    }
}
```

____

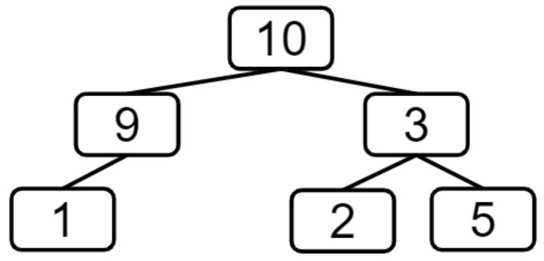
b. **(16 points).** Suppose we perform an **inorder** traversal using the `Adder` visitor defined below. Draw the tree that results in the provided box.

```
static void inorder(TreeNode n, Visitor v) {
    if (n == null) { return; }
    inorder(n.left, v);
    v.visit(n);
    inorder(n.right, v);
}
class Adder implements Visitor {
    public void visit(TreeNode x) {
        if (x.left != null)  { x.item += x.left.item; }
        if (x.right != null) { x.item += x.right.item; }
    }
}
```

c. **(20 points).** Suppose we perform a preorder traversal using a `GlorpGlorp` on the tree to the right, i.e. we call `TreeNode.preorder(n, new GlorpGlorp())`, where `n` is the root node of the tree.

```
class GlorpGlorp implements Visitor {
  Printer printer = new Printer();

  public void visit(TreeNode x) {
      TreeNode.inorder(x, printer);
  }
}
```



Write the output of the program in the blank provided below.

____

d. **(12 points).** Give the DFS preorder and postorder starting from 5 if we were to treat the tree at the top of this page as a graph. If a node has multiple neighbors, traverse the smallest neighbor first.

DFS Preorder: ____________________

DFS Postorder: ____________________

9. Heaping. a) **(20 points).** In lecture and project 2A, we implemented a Heap as an array, but we could also implement as a recursive data structure, e.g. a tree of `TreeNode` objects. Write a `sink` method that sinks the value in the given `TreeNode`. You'll need to use the `min` helper method given in the text box.

```
public static void sink(TreeNode n) {
    if (n == null) { return; }
    ______________________________
    if (______________________) {
        return;
    }
    if (______________________) {
        ___________________________
        ___________________________
        ___________________________
        ___________________________
    }
}
```

`TreeNode min(TreeNode a, TreeNode b)`:
Returns the treenode with the smaller .item.
If both equal, breaks ties arbitrarily.
If both are null, returns null.
If only one is null, returns the non-null node.

`min` is a static method that is part of the `TreeNode` class.

b) **(15 points).** If we create a `SinkVisitor` whose `visit` method simply calls `sink(x)`. Give a **complete tree with 4 nodes** for which `TreeNode.preorder(n, new SinkVisitor())` does not result in a heap if `n` is the root of your tree. **No credit will be given to trees that are not complete.**

10. Asymptotics

a) **(44 points).** Give the runtime of the following functions in Θ notation. Your answer should be a function of N that is as simple as possible with no unnecessary leading constants or lower order terms. **<u>Don't spend too much time on these</u>**!

```
_Θ_____   public static void g1(int N) {
                for (int i = 0; i < N; i += 3) {
                    for (int j = 0; j < i; j += 1) {
                        System.out.print("ca3");
                    }
                }
            }

_Θ_____   public static void g2(int N) {    // ASSUME N IS A POWER OF 2.
                for (int i = 1; i < N; i = i * 2) {
                    g2(i);
                }
                System.out.print("power egg");
            }

_Θ_____   public static void g3(int N) {
                int j = N / 2;
                for (int i = 0; i < j; i += 4) {
                    for (j = 0; j < N; j += 1) {
                        System.out.print("moo");
                    }
                }
            }

_ Θ_____   public static void g4(ArrayList<Integer> x) {
                int N = x.size(); // size takes constant time
                if (N == 1) { System.out.println(x.get(0)); return; }

                // subListCopy(P, Q) returns a copy of the list from [P, Q),
                // i.e. exclusive of Q, and takes Θ(Q – P) time.
                ArrayList<Integer> leftHalf = x.subListCopy(0, N / 2);
                ArrayList<Integer> rightHalf = x.subListCopy(N / 2, N);

                g4(leftHalf);
                g4(rightHalf);

                for (int i = 0; i < N; i += 1) {
                    System.out.println(x.get(i)); // get takes constant time
                }
            }
```

b) In the first few weeks of class, when discussing `AList`, I claimed that resizing by multiplying by a constant factor works well, but resizing our array by adding a constant factor results in an unusably slow data structure. Let's now work on understanding why by examining the cost of the `resize` method. To do so, we'll use number of values copied as our cost model.

i) **(10 points).** Suppose we have an AList that starts with 1 item and doubles in size whenever an item is added that won't fit in the old array. How many values need to be copied to handle each of the first 16 add operations? The first 4 values are provided for you. For example, on the 3rd add we resize from size 2 to 4, so must copy the 2 old values.

add #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# copied	0	1	2	0												

ii) **(10 points).** Let $C_d(N)$ be the **total number of copy operations** needed to handle a sequence of N add operations using this technique where we resize by doubling. For example, $C_d(4) = 3$. Give a theta bound for $C_d(N)$.

$C_d(N) = \Theta($_______$)$

iii) **(10 points).** Suppose we instead have an AList that starts with 1 item and instead increases in size by 3 whenever an item is added that won't fit in the old array. How many values need to be copied to handle each of the first 16 add operations? The first 5 values are provided for you. For example, on the 5th add we resize from size 4 to 7, so must copy the 4 old values.

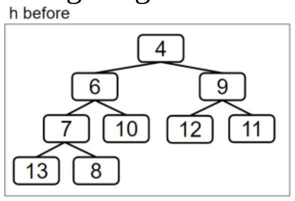
add #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# copied	0	1	0	0	4											

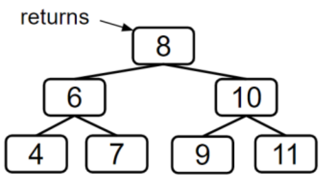
iv) **(10 points).** Let $C_{a3}(N)$ be the **total number of copy operations** needed to handle a sequence of N add operations using this technique where we resize by adding 3 more array entries. For example, $C_{a3}(5) = 5$. Give a theta bound for $C_{a3}(N)$.

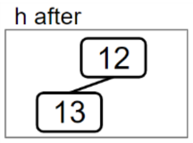
$C_{a3}(N) = \Theta($_______$)$

Sometime after exam, use the answers to the problems above to convince yourself that multiplicative resizing is good, and constant factor resizing is bad.

11. Yggdrasil (60 points). **<u>This is a very challenging problem</u>**. Write a function that takes an integer k and a min-heap `h` (in tree representation) and removes **the k smallest values** and returns them organized into **valid perfectly balanced BST.** For example, if we call `heapToBBST(7, h)` on the MinHeap in the left figure, it returns the Tree in the middle figure, and as a side-effect, `h` becomes the MinHeap in the right figure.







This should be done in-place, i.e., reusing the TreeNodes from the min-heap. Your function should complete in O(N log N) time, where N is the number of items in the min-heap, and use no more than O(log N) additional memory while it is running. For full credit, it must work for arbitrary values for k, but you can earn **almost full credit if your solution works for k = 2^H – 1** (i.e. powers of 2 minus 1).

To receive 10% credit and skip this problem, fill in this box and leave the code below blank: ☐

```
public class TreeNode {
    public int item;
    public TreeNode left;
    public TreeNode right;
}

public class MinHeap {
    /* Even though a MinHeap is made up of TreeNodes, the instance
     * variables are private. You can't directly access them. */

    /* removes the minimum node and returns it */
    public TreeNode removeMin() { /* ... */ }
    ...
}

public static TreeNode heapToBBST(int k, MinHeap h) {
     if (k == 0) {
         return null;
     }
     _________________________________
     _________________________________
     _________________________________
     _________________________________
     _________________________________
     _________________________________
     _________________________________
     _________________________________
     _________________________________
}
```