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.”
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.

   ![Binary Search Tree Diagram]

   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.

   ```java
   private TreeNode deleteMin(TreeNode x) {
     if (x.left == null) {
       return x.right;
     }
     x.left = deleteMin(x.left);
     return x;
   }
   ```

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

   ```java
   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.

a) (5 points). What is the current load factor? It is OK to leave your answer as a fraction: 19/41

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: 0  Max: 19

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

Min: 22  Max: 22

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:

```java
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 per add operation?

Each add on average costs: ○ Θ(1)  ○ Θ(log N)  ● Θ(N)  ○ Θ(N^2)  ○ Θ(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 1; // the goal here was to make a bad hash code
```
3. **KdTree.** Suppose we have the KdTree shown below.

![KdTree Diagram]

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: Node A splits on x values, and since 2 < 4 left is the “good child”

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: (4, 2)

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 | B | C |

- A  
- B  
- C  
- D  
- E  
- F  
- G

```
  A
  B
  C
```

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  

- C.right
- D.right
- F.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(13)  ○ 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(39)  ○ rotateRight( )  ○ colorFlip( )
Op #2: ○ rotateLeft( )  ● rotateRight(43)  ○ colorFlip( )
Op #3: ○ rotateLeft( )  ○ rotateRight( )  ● colorFlip(40)
Op #4: ○ rotateLeft( )  ○ rotateRight( )  ● colorFlip(25)
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).

```
  17 25
 /    |
13    22    39 43
```

5. PNH (0 Points). What is a “desire path”?
A desire path is a path created as a consequence of erosion from human/animal traffic. It usually represents the shortest or most easily navigated route.
6. WQUzaaaaaaaaaaaaaaaaaaaaaaaaapp (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(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)  Union(____,____)

● Not possible  ○ Not possible  ● Not possible

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

GitHub Account #: sp19-s______
7. Shortest Paths. Suppose that we run Dijkstra’s algorithm on the graph below from the vertex 0.

![Graph Diagram]

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

<table>
<thead>
<tr>
<th>vertex #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>distTo</td>
<td>0</td>
<td>∞</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>edgeTo</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

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

0 3 7 4 2

b) (4 points). What vertex will be visited next? _____6_____

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

<table>
<thead>
<tr>
<th>vertex #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>distTo</td>
<td>0</td>
<td>26</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

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

How many edges would be in the minimum spanning tree? _____7_____

Tranquility Area

Mark The Paper In The Way That You Like Best
Les Visiteurs. Suppose we have the tree below. Suppose it is implemented using TreeNodes.

```java
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.

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

```java
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.

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

_____10 9 1 3 2 5_____________________

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

```java
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.

```java
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.

`1 9 10 2 3 5 1 9 1 2 3 5 2 5`

---

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: `5 3 2 10 9 1`

DFS Postorder: `2 1 9 10 3 5`

---

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.

```java
public static void sink(TreeNode n) {
    if (n == null) { return; }
    TreeNode minN = min(n.left, n.right)
    if (minN == null) {
        return;
    }
    if (n.item > minN.item) {
        int temp = n.item;
        n.item = minN.item;
        minN.item = temp;
        sink(minN);
    }
}
```

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. Don’t spend too much time on these!

\[ \Theta N^2 \] 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");
        }
    }
}

\[ \Theta N \] public static void g2(int N) {
    // ASSUME N IS A POWER OF 2.
    for (int i = 0; i < N; i = i * 2) {
        g2(i);
    }
    System.out.print("power egg");
}

\[ \Theta N^2 \] 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");
        }
    }
}

\[ \Theta N \log N \] 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 \(\Theta(Q - P)\) 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
    }
}
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.

<table>
<thead>
<tr>
<th>add #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td># copied</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 \( C_d(N) \).

\[ C_d(N) = \Theta(\_\_N\_\_) \]

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.

<table>
<thead>
<tr>
<th>add #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td># copied</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 \( C_{a3}(N) \).

\[ C_{a3}(N) = \Theta(\_\_N^2\_) \]

Note problem 10.a was a hint for this problem, which is why that problem prints out “ca3”. They have the exact same pattern / runtime.

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). **This is a very challenging problem.** 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 \( \text{heapToBST}(7, h) \) 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: □

```java
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;
    }
    TreeNode left = heapToBBSt(k/2, h);
    TreeNode middle = h.removeMin();
    TreeNode right = heapToBBSt(k - (k/2) - 1, h);
    middle.left = left;
    middle.right = right;
    return middle;
}
```