

# Final

● Graded

Student

AKSHAJ KAMMARI

Total Points

78.6 / 150 pts

## Question 1

### Problem 1 - Misc

18 / 40 pts

1.1 1.1

0 / 10 pts

+ 5 pts Algorithm:

- For each item in m array, do a binary search in n array

+ 5 pts  $O(m \log n)$

✓ + 0 pts Incorrect

1.2 1.2

6 / 6 pts

✓ + 3 pts  $O(n)$

✓ + 3 pts In the worst case each node is to the left (or each node to the right) of its parent forming a list of n nodes.

+ 0 pts Incorrect

1.3 1.3

0 / 6 pts

+ 3 pts  $O(\log_d(n))$

+ 3 pts Each node has d children therefore the tree height is  $\log_d(n)$ . During insert a new key is compared against the parent (1 compare) and can swim from the leaf level to the root.

✓ + 0 pts Incorrect

+ 2 pts | if(root == null) Has base case

```
private static void print(TreeNode root) {  
    if (root == null) return; // 2 point for base case  
  
    // 4 points for printing before left and right calls  
    System.out.println(root.getData().getProbOcc() + " ");  
  
    // 4 points for going left before going right  
    print (root.getLeft()); // 4 points  
    print (root.getRight()); // 4 points  
}
```

+ 4 pts | System.out.println(root.getData().getProcOcc() + " "); Prints before left and right calls.

```
private static void print(TreeNode root) {  
    if (root == null) return; // 2 point for base case  
  
    // 4 points for printing before left and right calls  
    System.out.println(root.getData().getProcOcc() + " ");  
  
    // 4 points for going left before going right  
    print (root.getLeft()); // 4 points  
    print (root.getRight()); // 4 points  
}
```

✓ + 4 pts Calls left before right

```
private static void print(TreeNode root) {  
    if (root == null) return; // 2 point for base case  
  
    // 4 points for printing before left and right calls  
    System.out.println(root.getData().getProbOcc() + " ");  
  
    // 4 points for going left before going right  
    print (root.getLeft()); // 4 points  
    print (root.getRight()); // 4 points  
}
```

✓ + 4 pts | print(root.getLeft()); Goes left

```
private static void print(TreeNode root) {  
    if (root == null) return; // 2 point for base case  
  
    // 4 points for printing before left and right calls  
    System.out.println(root.getData().getProbOcc() + " ");  
  
    // 4 points for going left before going right  
    print (root.getLeft()); // 4 points  
    print (root.getRight()); // 4 points  
}
```

✓ + 4 pts | print(root.getRight()); Goes right

```
private static void print(TreeNode root) {  
    if (root == null) return; // 2 point for base case  
  
    // 4 points for printing before left and right calls  
    System.out.println(root.getData().getProbOcc() + " ");  
  
    // 4 points for going left before going right  
    print (root.getLeft()); // 4 points  
    print (root.getRight()); // 4 points  
}
```

+ 0 pts Incorrect

## Question 2

### Problem 2 - Priority Queue

0 / 16 pts

2.1 (a)

0 / 4 pts

+ 2 pts  $O(n)$

+ 2 pts Heaps have no symmetric order (inorder traversal yields words in ascending order) so linear search is performed.

✓ + 0 pts Incorrect

2.2 (b)

0 / 6 pts

+ 3 pts  $O(1)$

+ 3 pts A word is inserted with frequency 1 at the first empty array slot from left to right. The MAX-heap order invariant is not violated because the parent word will not have a smaller frequency than 1.

✓ + 0 pts Incorrect

2.3 (c)

0 / 6 pts

+ 3 pts  $O(\log(n))$

+ 3 pts Suppose that there are  $n$  words in the heap. All words have the same frequency. When a word at the leaf level has its frequency increased it can swim all the way to the root.

✓ + 0 pts Incorrect

**Question 3**

Problem 3 - Hash Tables

**20 / 20 pts**

3.1

**3.1**

16 / 16 pts

Before Rehash

✓ + 1 pt 90 at index 0

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 15 at index 0

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 27 at index 0

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 10 at index 1

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 8 at index 2

Before rehashing

```
[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key
```

After rehashing

```
[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key
```

✓ + 1 pt 14 at index 2

Before rehashing

```
[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key
```

After rehashing

```
[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key
```

After Rehash

✓ + 1 pt 12 at index 0

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 90 at index 0

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 7 at index 1

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 14 at index 2

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 8 at index 2

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 27 at index 3

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 15 at index 3

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 10 at index 4

Before rehashing

[0] ---> 90 ---> 15 ---> 27  
[1] ---> 10  
[2] ---> 8 ---> 14  
// 1 point for each key

After rehashing

[0] ---> 12 ---> 90  
[1] ---> 7  
[2] ---> 14 ---> 8  
[3] ---> 27 ---> 15  
[4] ---> 10  
[5] ---> 17 ---> 11  
// 1 points for each key

✓ + 1 pt 17 at index 5

Before rehashing

```
[0] --> 90 --> 15 --> 27  
[1] --> 10  
[2] --> 8 --> 14  
// 1 point for each key
```

After rehashing

```
[0] --> 12 --> 90  
[1] --> 7  
[2] --> 14 --> 8  
[3] --> 27 --> 15  
[4] --> 10  
[5] --> 17 --> 11  
// 1 points for each key
```

✓ + 1 pt 11 at index 5

Before rehashing

```
[0] --> 90 --> 15 --> 27  
[1] --> 10  
[2] --> 8 --> 14  
// 1 point for each key
```

After rehashing

```
[0] --> 12 --> 90  
[1] --> 7  
[2] --> 14 --> 8  
[3] --> 27 --> 15  
[4] --> 10  
[5] --> 17 --> 11  
// 1 points for each key
```

+ 0 pts Incorrect

3.2

4 / 4 pts

✓ + 1 pt O(n)

✓ + 3 pts Each of the n keys will need to be remapped to the new table

+ 0 pts Incorrect

**Question 4**

Problem 4 - Graphs

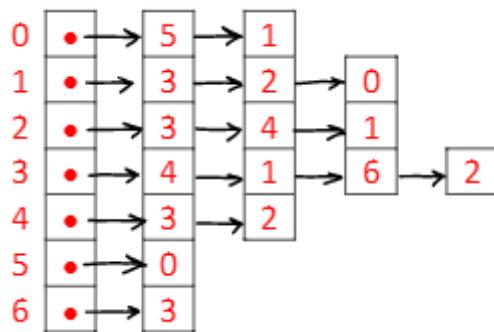
**23.6 / 40 pts**

4.1

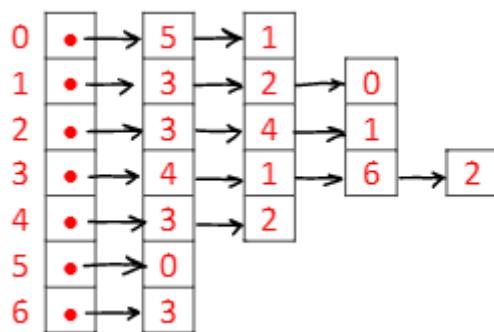
**4.1**

6 / 6 pts

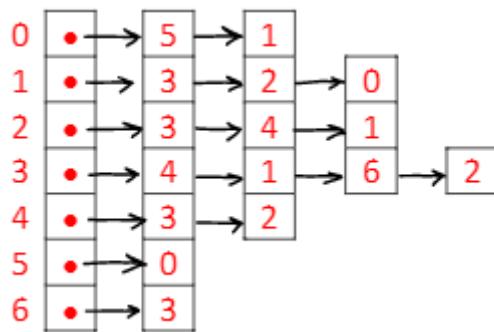
✓ + 1 pt Index 0 has 5, 1



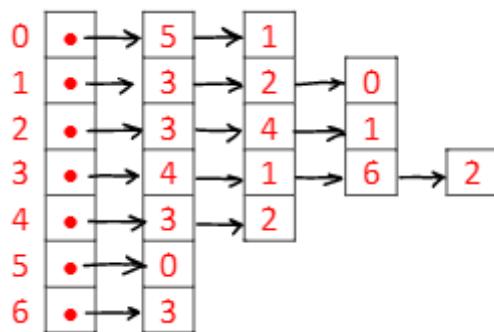
✓ + 1 pt Index 1 has 3, 2, 0



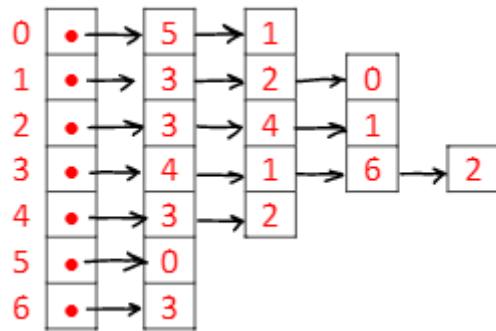
✓ + 1 pt Index 2 has 3, 4, 1



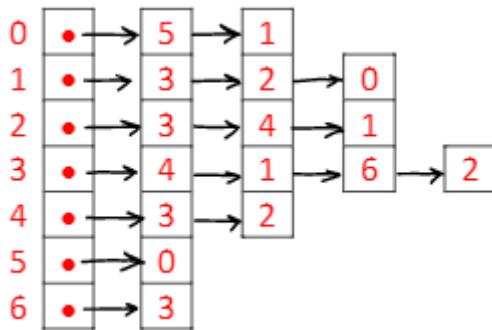
✓ + 1 pt Index 3 has 4, 1, 6, 2



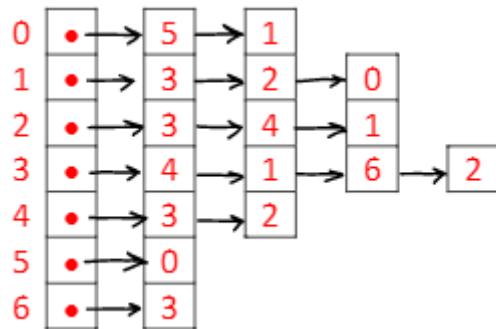
✓ + 0.5 pts Index 4 has 3, 2



✓ + 1 pt Index 5 has 0



✓ + 0.5 pts Index 6 has 3



+ 0 pts Incorrect

4.2

4.2a

0 / 2 pts

+ 2 pts  $\text{degree}(v)$

OR the number of adjacent

vertices to  $v$

OR the number of neighbors to the vertex  $v$

✓ + 0 pts Incorrect

4.3

4.2b

3 / 3 pts

✓ + 3 pts  $2 \times 8$  times

OR  $2E // G.E() = 8;$

+ 0 pts Incorrect

4.4 4.3a 0 / 5 pts

+ 5 pts 0 -> 5 -> 1 -> 3 -> 4 -> 2 -> 6

✓ + 0 pts Incorrect

4.5 4.3b 1.6 / 5 pts

✓ + 0.8 pts Value 0 at index 1

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0.8 pts Value 4 at index 2

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0.8 pts Value 1 at index 3

0	1	2	3	4	5	6
-	0	4	1	3	0	3

✓ + 0.8 pts Value 3 at index 4

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0.8 pts Value 0 at index 5

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0.8 pts Value 3 at index 6

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0.2 pts All correct

0	1	2	3	4	5	6
-	0	4	1	3	0	3

+ 0 pts Incorrect

4.6 4.4a 0 / 5 pts

+ 5 pts 0 -> 5 -> 1 -> 3 -> 2 -> 4 -> 6

✓ + 0 pts Incorrect

4.7 4.4b

4 / 5 pts

✓ + 0.8 pts Value 0 at index 1

0	1	2	3	4	5	6
-	0	1	1	2	0	3

✓ + 0.8 pts Value 1 at index 2

0	1	2	3	4	5	6
-	0	1	1	2	0	3

✓ + 0.8 pts Value 1 at index 3

0	1	2	3	4	5	6
-	0	1	1	2	0	3

✓ + 0.8 pts Value 2 at index 4

0	1	2	3	4	5	6
-	0	1	1	2	0	3

+ 0.8 pts Value 0 at index 5

0	1	2	3	4	5	6
-	0	1	1	2	0	3

✓ + 0.8 pts Value 3 at index 6

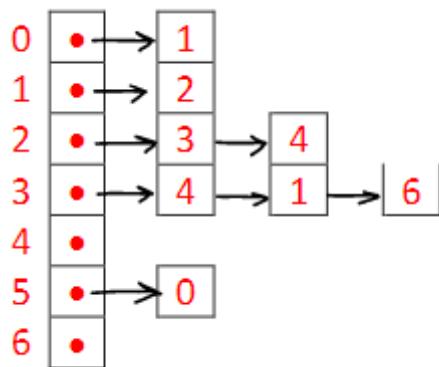
0	1	2	3	4	5	6
-	0	1	1	2	0	3

+ 0.2 pts All correct

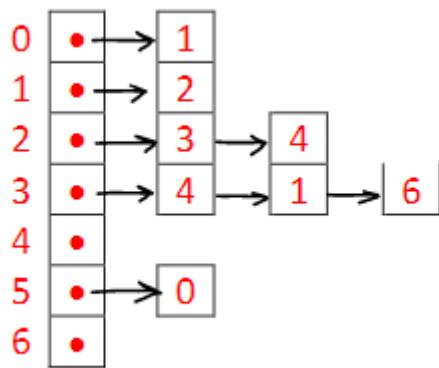
0	1	2	3	4	5	6
-	0	1	1	2	0	3

+ 0 pts Incorrect

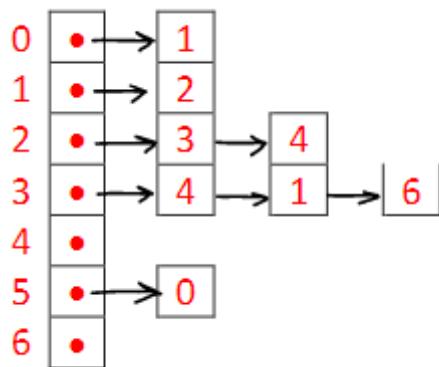
✓ + 1 pt Index 0 has 1



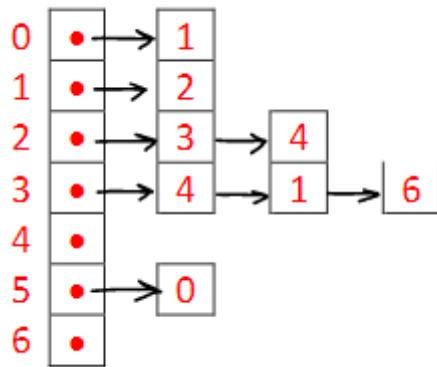
✓ + 1 pt Index 1 has 2



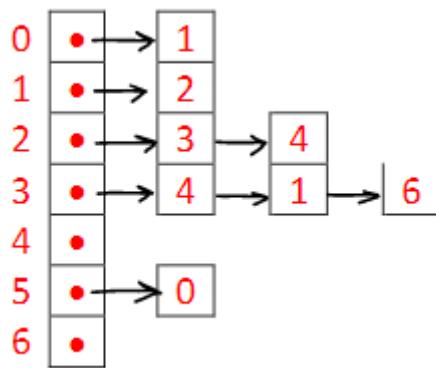
✓ + 1 pt Index 2 has 3 and 4



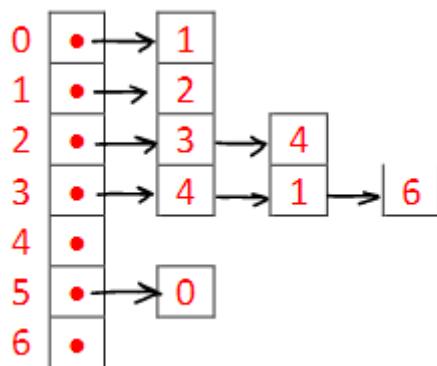
✓ + 1 pt Index 3 has 4, 1, and 6



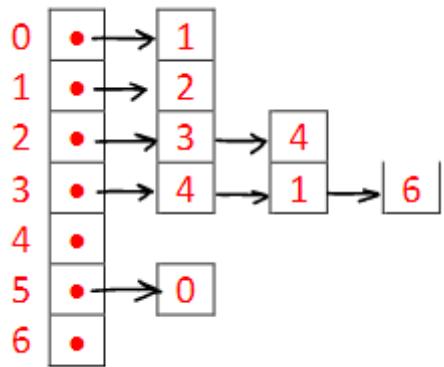
✓ + 0.5 pts Index 4 has none



✓ + 1 pt Index 5 has 0



✓ + 0.5 pts Index 6 has none



+ 0 pts Incorrect

4.9 4.5b

3 / 3 pts

✓ + 3 pts  $7 * (7 - 1) = 42$

+ 0 pts Incorrect

## Question 5

### Problem 5 - Sorts

17 / 34 pts

5.1 5.1

0 / 5 pts

+ 2 pts Insertion sort

+ 3 pts Because the best-case scenario for insertion sort is O(n) when the array is already sorted.

✓ + 0 pts Incorrect

5.2 5.2

2 / 2 pts

✓ + 2 pts Merge sort

+ 0 pts Incorrect

5.3 5.3

3 / 3 pts

✓ + 3 pts Merge sort uses O(n) extra storage to sort an array while heap sort does not use extra storage.

OR

Merge sort uses  $2n$  storage to sort an array while heap sort uses  $n$ .

OR

Merge sort uses  $n+n/2$  storage to sort an array while heap sort uses  $n$

+ 0 pts Incorrect

5.4 5.4a

12 / 12 pts

✓ + 4 pts Swap 89, 2

	25	8	89	28	15	9	2
Swap 89, 2	->	25	8	2	28	15	9
Swap 28, 9	->	25	8	2	9	15	28
Swap 25, 15	->	15	8	2	9	25	28

✓ + 4 pts Swap 28, 9

	25	8	89	28	15	9	2
Swap 89, 2	->	25	8	2	28	15	9
Swap 28, 9	->	25	8	2	9	15	28
Swap 25, 15	->	15	8	2	9	25	28

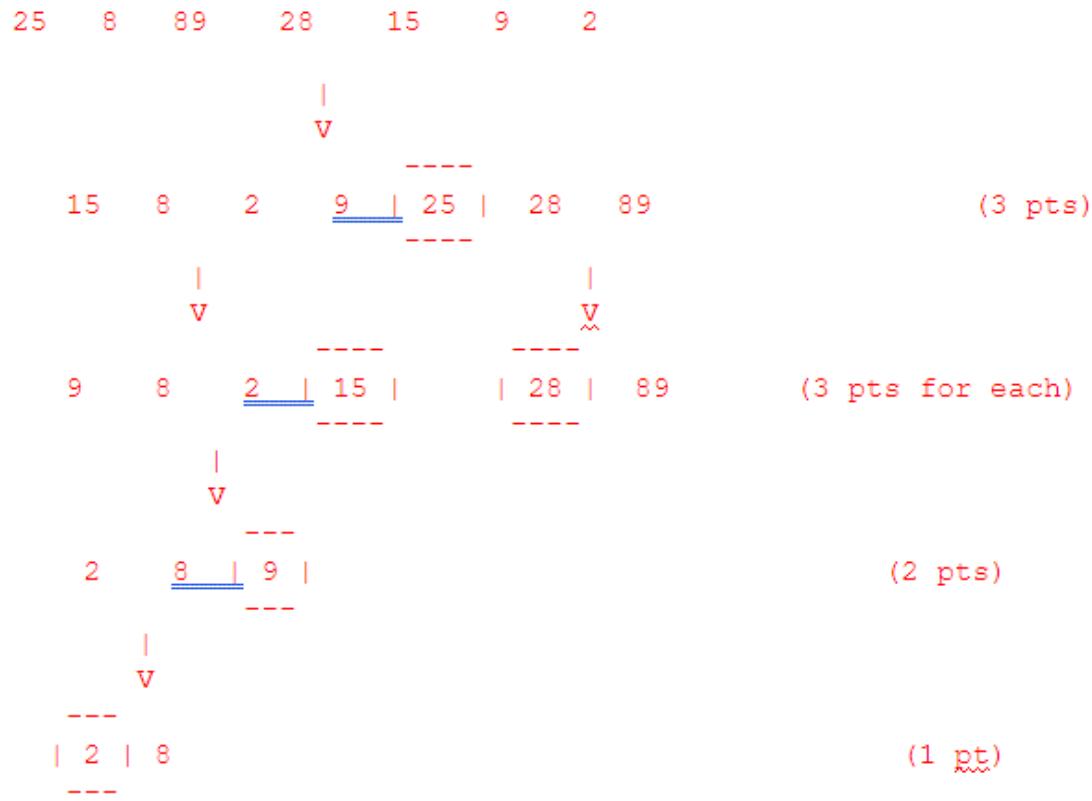
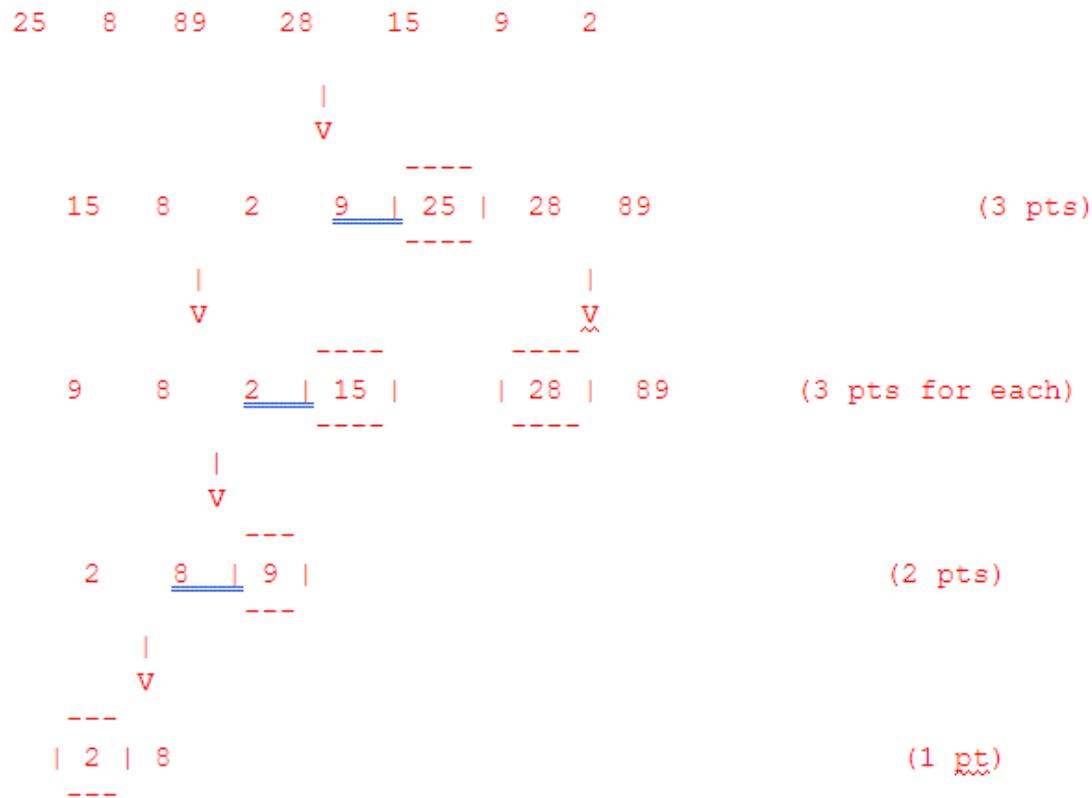
✓ + 4 pts Swap 25, 15

	25	8	89	28	15	9	2
Swap 89, 2	->	25	8	2	28	15	9
Swap 28, 9	->	25	8	2	9	15	28
Swap 25, 15	->	15	8	2	9	25	28

+ 0 pts Incorrect

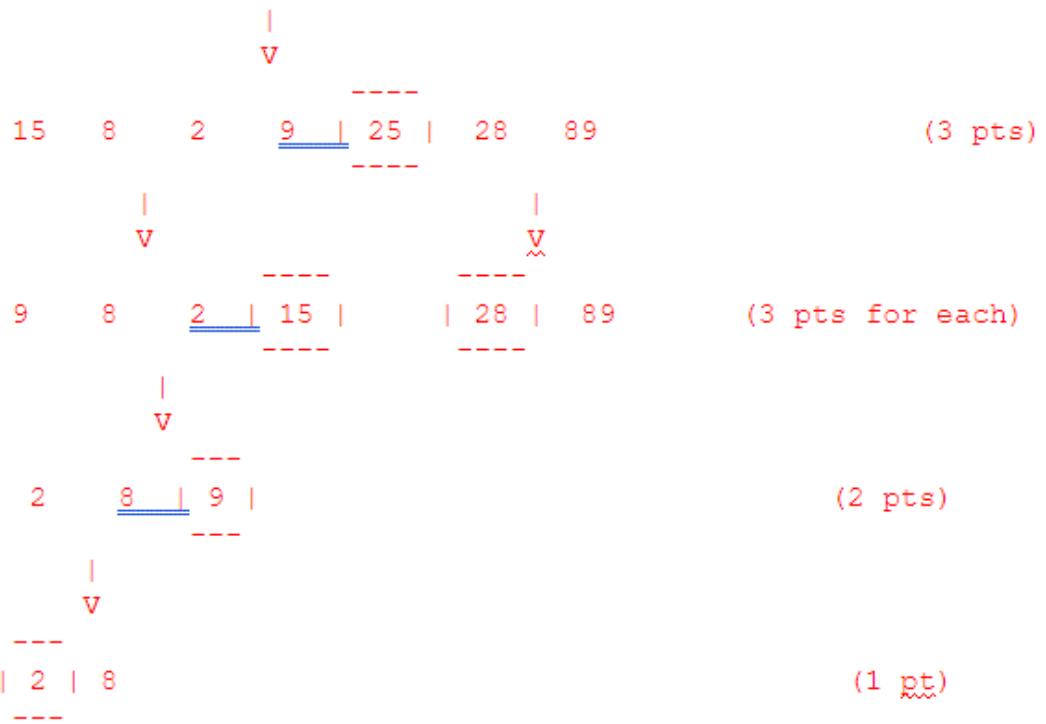
5.5 L 5.4b

0 / 12 pts

**+ 3 pts** Split/pivot at 25**+ 3 pts** Split/pivot at 15

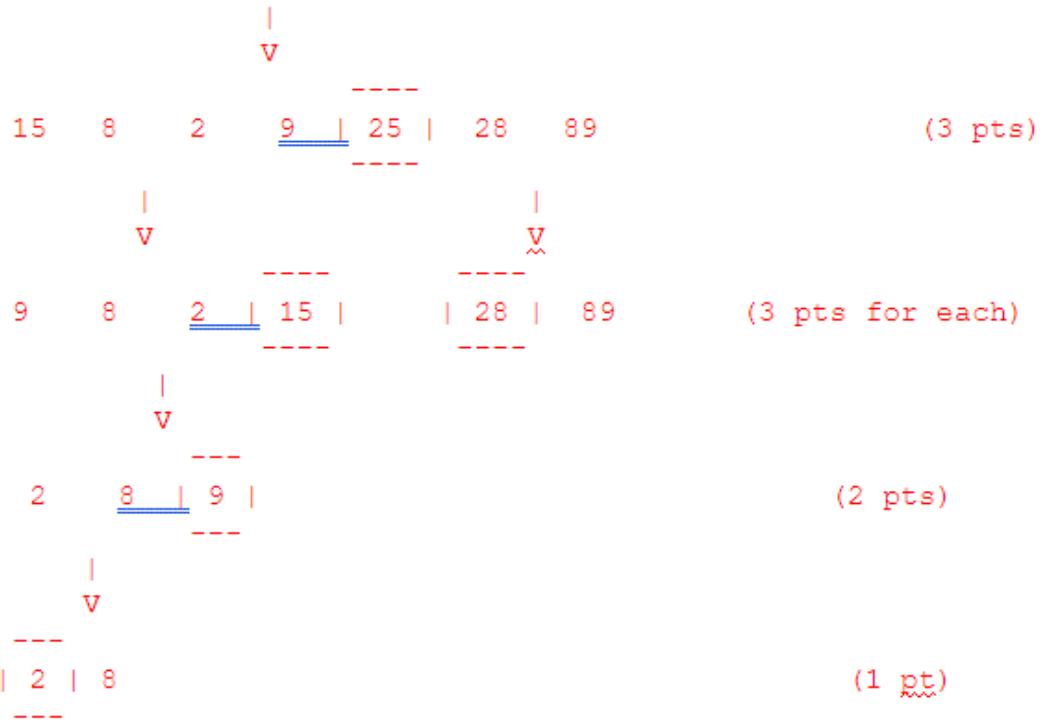
+ 3 pts Split at 28

25    8    89    28    15    9    2



+ 2 pts Split/pivot at 9

25    8    89    28    15    9    2



+ 1 pt Split/pivot at 2

25      8      89      28      15      9      2

$\begin{array}{c}   \\ \text{v} \\ \hline 15 & 8 & 2 & \underline{\underline{9}} & 25 &   & 28 & 89 \\ \hline \end{array}$	(3 pts)
$\begin{array}{c}   \\ \text{v} \\ \hline 9 & 8 & \underline{\underline{2}} & 15 &   &   & 28 & 89 \\ \hline \end{array}$	(3 pts for each)
$\begin{array}{c}   \\ \text{v} \\ \hline 2 & \underline{\underline{8}} &   & 9 &   \\ \hline \end{array}$	(2 pts)
$\begin{array}{c}   \\ \text{v} \\ \hline \hline   & 2 &   & 8 \\ \hline \end{array}$	(1 pt)

✓ + 0 pts Incorrect

Name: Akshaj Icommarri NetID: AK1990

- WRITE your name and NetID on EVERY page.
- DO NOT REMOVE the staple/crimp in your exam.
- DO NOT BEGIN until instructed to do so.
- WRITE NEATLY AND CLEARLY. If we cannot read your handwriting, you will not receive credit. Please plan your space usage. No additional paper will be given.
- This exam is worth 150 points.

**Problem 1 – Miscellaneous (40 points)**

1. (10 points) For this problem write the fastest algorithm (measured by worst-case big O). Describe an algorithm (give concise steps, do not write code) that given an unsorted array of length  $m$  and a sorted array of length  $n$ , finds the common items. Assume there are no duplicates in either array and,  $m$  is much larger than  $n$ . What is the worst-case big O running time?

heap sort,  $O(m \log m)$  to sort the unsorted array and search through the sorted array would be  $O(n \log n)$  so the worst case big O would be  $O(m \log m)(\log n)$

2. (6 points) In the worst case, how many compares, in big O notation, to insert in a BST? Explain.

$O(n)$ , because the whole tree might have to be traversed.

3. (6 points) In the worst case, how many compares, in big O notation, to insert in a d-way heap? Explain.

$O(\log_3 n)$ , because a d-way heap has an extra child node.

4. (18 points) On the Huffman Coding assignment the TreeNode class houses a CharFreq object "data" representing a certain character and its frequency, and TreeNodes "left" and "right" representing the left and right subtrees of the binary tree.

Name: Akshay Kammar NetID: AK1990

```

public class TreeNode {
    private CharFreq data;
    private TreeNode left;
    private TreeNode right;

    // We can create with data and both children
    public TreeNode(CharFreq d, TreeNode l, TreeNode r) {
        data = d;
        left = l;
        right = r;
    }

    // We can create with only data, children are null
    public TreeNode(CharFreq d) { this(d, null, null); }

    // No arguments sets everything to null
    public TreeNode() { this(null, null, null); }

    // Getters and setters
    public CharFreq getData() { return data; }
    public TreeNode getLeft() { return left; }
    public TreeNode getRight() { return right; }

    public void setData(CharFreq d) { data = d; }
    public void setLeft(TreeNode l) { left = l; }
    public void setRight(TreeNode r) { right = r; }
}

public class CharFreq implements Comparable<CharFreq> {
    private Character character;
    private double probOcc;

    // We can set both the Character and double at once
    public CharFreq(Character c, double p) {
        character = c;
        probOcc = p;
    }

    // No arguments makes a null character and prob 0
    public CharFreq() { this(null, 0); }

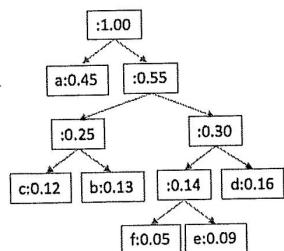
    // Allows us to use Collections.sort() to sort by probOcc
    public int compareTo(CharFreq cf) {
        Double d1 = probOcc, d2 = cf.probOcc;
        if (d1.compareTo(d2) != 0) return d1.compareTo(d2);
        return character.compareTo(cf.character);
    }

    // Getters and setters
    public Character getCharacter() { return character; }
    public double getProbOcc() { return probOcc; }

    public void setCharacter(Character c) { character = c; }
    public void setProbOcc(double p) { probOcc = p; }
}

```

Write the following RECURSIVE print method that prints in preorder the probabilities in the Huffman Tree.



The print method prints the probabilities in this tree as follows:

1.0 0.45 0.55 0.25 0.12 0.13 0.30 0.14 0.05 0.09 0.16

```

// Prints the probabilities according to a preorder
// traversal of the tree
private static void print (TreeNode root) {
    // COMPLETE THIS RECURSIVE METHOD

    int i = root;
    if (i.left != null) {
        i = i.left;
        System.out.print(i.data);
        print(i); // recursive call
    }

    if (i.right != null) {
        i = i.right;
        System.out.print(i.data);
        print(i); // recursive call
    }
    return -1; // 
}

```

Name: Akshaj Kammari NetID: AK1990**Problem 2 - Priority Queue (16 points)**

Suppose that a MAX heap is implemented using an array. The heap is used to keep track of the frequency of the words in a book, the word that appears most often in the book has the highest frequency and is located at the root (array index 1).

During input every word is inserted into the heap as follows: a word is read from the book and then the word is searched in the heap:

- (i) if the word is not found it is then inserted into the heap with frequency 1.
- (ii) if the word is found its frequency is increased by 1.

Answer the following questions.

- a. (4 points) What is the worst-case running time, in big O notation, of searching for a word in the heap? Explain.

$\Theta(\log n)$ , because the word may be a leaf node.

- b. (6 points) Assume that a word has been searched and determined that it is not in the heap. What is the worst-case running time, in big O notation, of inserting a new word into the heap (after heap invariants are restored)? Explain.

$\Theta(\log n)$  because it may have to be inserted high in the heap.

- c. (6 points) Assume that a word has been searched and determined that it is not in the heap. What is the worst-case running time, in big O notation, to update a word's frequency (heap invariants are expected to be restored after insert is complete)? Explain.

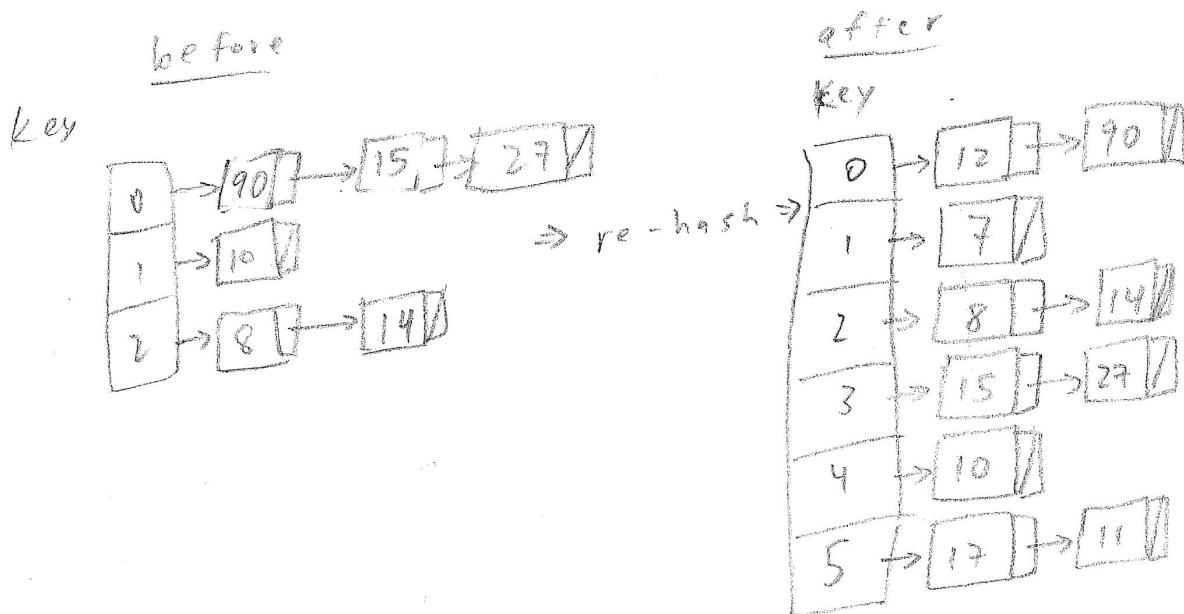
$O(1)$ , because it doesn't need to be searched again.

Name: Akshay Kammani NetID: AK1990**Problem 3 - Hash table (20 points)**

The following keys will be inserted in sequence to a hash table. For simplicity, we omit the "values" associated with the keys.

14 8 27 10 15 90 11 7 12 17

1. (16 points) Assume the Separate-Chaining Symbol Table API discussed in class is used. The table size is denoted by  $m$  and the hash function is  $\text{hash}(\text{key}) = \text{key} \% m$ . The initial table size is 3. Note that keys are inserted at the front of the list. The threshold of the load factor is 2. So, when the load factor is larger than 2, rehashing should be performed. Suppose we would double the table size when we do rehashing. Show the contents of the two hash tables before rehashing and after rehashing.

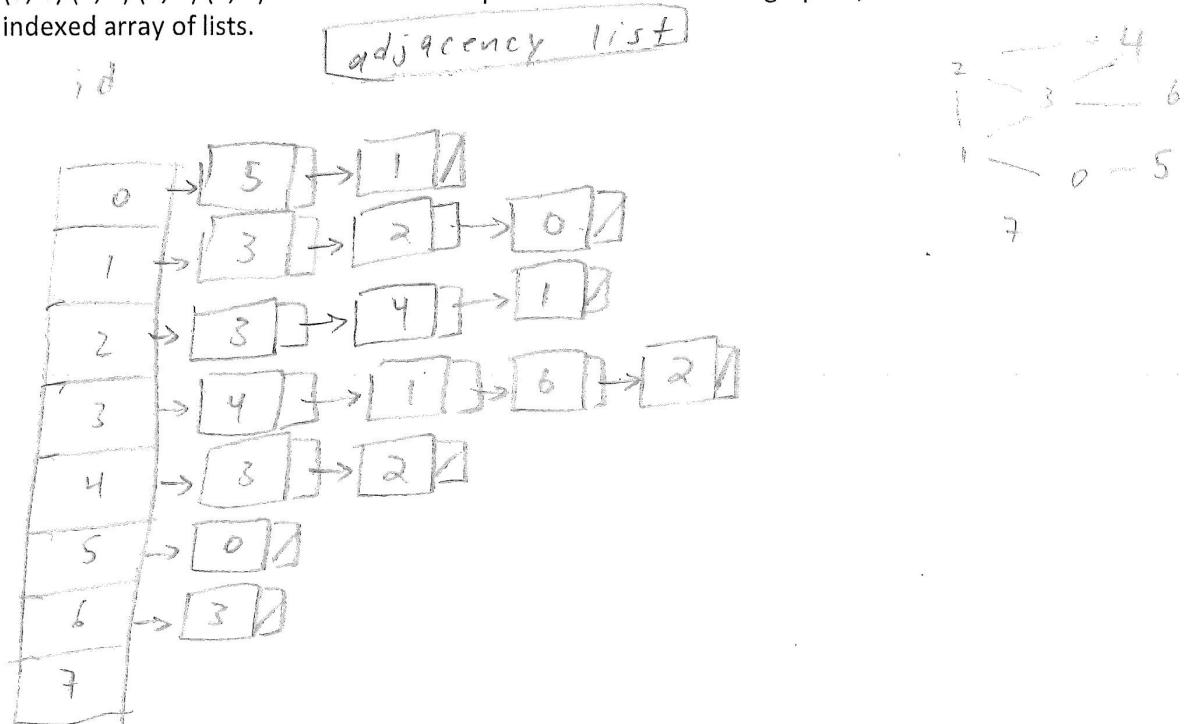


2. (4 points) What is the running time (big O) for rehashing given the input size  $n$ ? Give the reasoning.

$O(n)$ , all of the elements must be run through the hash function again.

Name: Akshaj Kammari NetID: Ak1990**Problem 4 – Graph (40 points)**

1. (6 points) Assume an undirected graph G with 7 vertices and 8 edges  $(v,w)$  (an edge between vertices  $v$  and  $w$ ). Based on the adjacency list Java implementation discussed in class where a new edge is added to front of list. If the list of 8 edges  $(0, 1)$   $(1, 2)$   $(2, 4)$   $(2, 3)$   $(3, 6)$   $(5, 0)$   $(3, 1)$   $(3, 4)$  were added in sequence to construct the graph G, show the vertex-indexed array of lists.



2. Answer the following questions based on the adjacency list in problem 4.1.

- a. (2 points) If a single for loop is used to iterate over the list of vertices adjacent to a given vertex  $v$ , what is the number of iterations?

$$V - 1 = 6 \text{ iterations}$$

- b. (3 points) Given the code below, what is the number of times `StdOut.println` statement is executed, assuming G is the undirected graph constructed in problem 4.1?

```

for ( int v = 0; v < G.V(); v++ )
    for ( int w : G.adj(v) )
        StdOut.println( v + " - " + w );
  
```

16 times

Name: Akshaj kannan NetID: Ak1990

3. Answer the following questions based on the Java code below and the adjacency list representation of graph G constructed in problem 4.1.

```
private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

- a. (5 points) Write the sequence of vertices visited of a method call to  $\text{dfs}(G, 0)$ .

0    1    2    3    4    5

- b. (5 points) write the contents of the array  $\text{edgeTo}[]$  as a consequence of the method call in 3.a.

(0, 1)

(1, 2)

(2, 3)

(3, 4)

(3, 6)

(4, 5)

0, 1, 2, 3, 3, 1

Name: Akshaj Kammari NetID: AK1990

4. Answer the following questions based on the Java code below and the adjacency list representation of graph G constructed in problem 4.1.

```

private void bfs(Graph G, int s) {
    Queue<Integer> q = new Queue<Integer>();
    distTo[s] = 0;
    marked[s] = true;
    q.enqueue(s);

    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                q.enqueue(w);
            }
        }
    }
}

```

- a. (5 points) Write the sequence of vertices visited of a method call to `bfs(G, 0)`.

0 1 5 2 3 4 6

- b. (5 points) Write the contents of the array `edgeTo[]` as a consequence of the method call in 4.1.

(0, 1)

(0, 5)

(1, 2)

(1, 3)

(3, 4)

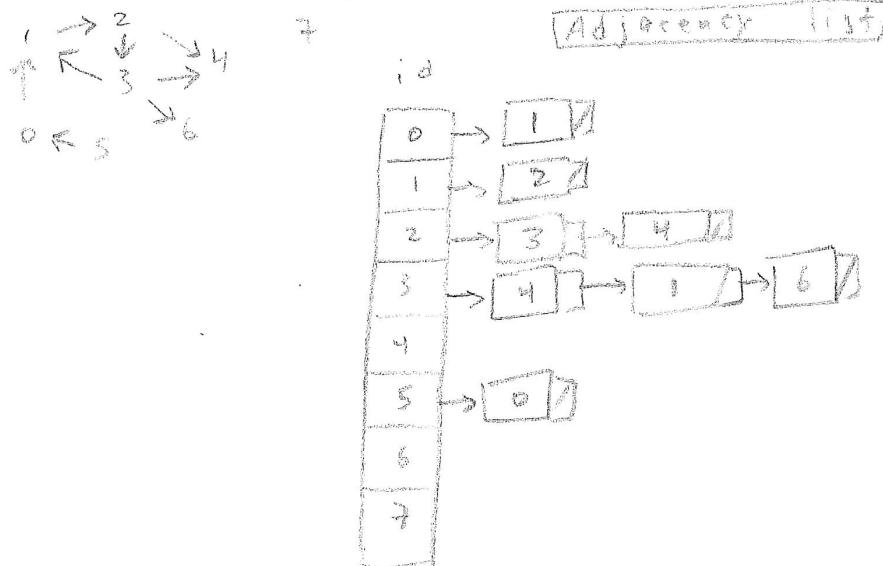
(3, 6)

0, 0, 1, 1, 3, 3

Name: Akshaj Kammari NetID: AK1990

5. Assume a **directed** graph G with 7 vertices and 8 edges  $(v,w)$  (an edge between vertices v and w). Based on the adjacency list Java implementation discussed in class where a *new edge is added to front of list*.

- a. (6 points) If a list of 8 edges  $(0, 1) (1, 2) (2, 4) (2, 3) (3, 6) (5, 0) (3, 1) (3, 4)$  with 7 vertices were added in sequence to construct the graph G, show the vertex-indexed array of list.



- b. (3 points) What is the maximum number of edges one can add to the directed graph in 5.1?

$$v(v-1)$$

$$7 \times 6 = 42 \text{ edges}$$

Name: Akshaj Kammani NetID: Ak1990**Problem 5 - Sorts (34 points)**

1. (5 points) Which sorting algorithm would you use, insertion sort, selection sort, merge sort or quick sort, to sort a large array that is known to be almost sorted? Justify your answer.

selection sort because the array is already almost sorted so as the unsorted part is being sorted it will be added to the sorted until the whole array is sorted.

2. (2 points) Which sorting algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves?

merge sort

3. (3 points) Explain how does merge sort compare to heap sort, in big O notation, with respect of storage consumption?

merge sort  $\rightarrow \Theta(n)$

heap sort  $\rightarrow \Theta(n^2)$

heap sort = merge sort

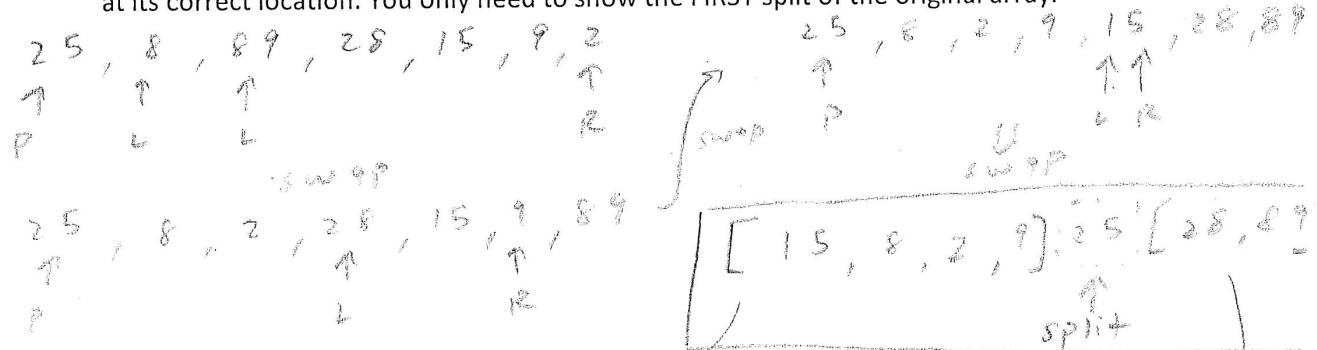
Name: Akshaj KammeriNetID: AK1990

4. (24 points) Trace the quicksort algorithm on the following array. Use the first item as the pivot when doing a split.

25, 8, 89, 28, 15, 9, 2

$$\begin{array}{l} L > P \\ P < R \end{array}$$

- a. (12 points) Show the series of item swaps that are performed in the FIRST split process, and the array after each of these swaps, up to and including the step of placing the pivot at its correct location. You only need to show the FIRST split of the original array.



- b. (12 points) Show the full recursion tree, i.e. all splits, on original array and all subarrays.

