Question 1:

Input size **n = 11**

Data Range 0:6, therefore, **k = 7**

We will define 3 arrays for the counting sort:

Array A: input array, size = n = 11

Array B: Output array, size = n = 11

Array C: Count array, since = k = 7

A – Input Array:

6	0	2	0	1	3	4	6	1	3	2

C – Count Array

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

Modify the count array by adding the previous counts

2	4	6	8	9	9	11

Insert values:

A[0] = 6

B – Output Array

1	2	3	4	5	6	7	8	9	10	11
										6

C – Count Array

0	1	2	3	4	5	6
2	4	6	8	9	9	**10**

A[1] = 0

B – Output Array

1	2	3	4	5	6	7	8	9	10	11
	0									6

C – Count Array

0	1	2	3	4	5	6
1	4	6	8	9	9	10

A[2] = 2

B – Output Array

1	2	3	4	5	6	7	8	9	10	11
	0				**2**					6

C – Count Array

0	1	2	3	4	5	6
1	4	**5**	8	9	9	10

A[3] = 0

B – Output Array

1	2	3	4	5	6	7	8	9	10	11
0	0				2					6

C – Count Array

0	1	2	3	4	5	6
0	4	5	8	9	9	10

A[6] = 4

B – Output Array

1	2	3	4	5	6	7	8	9	10	11
0	0				2			**4**		6

C – Count Array

0	1	2	3	4	5	6
0	4	5	8	**8**	9	10

…

Final Result:

B – Output Array

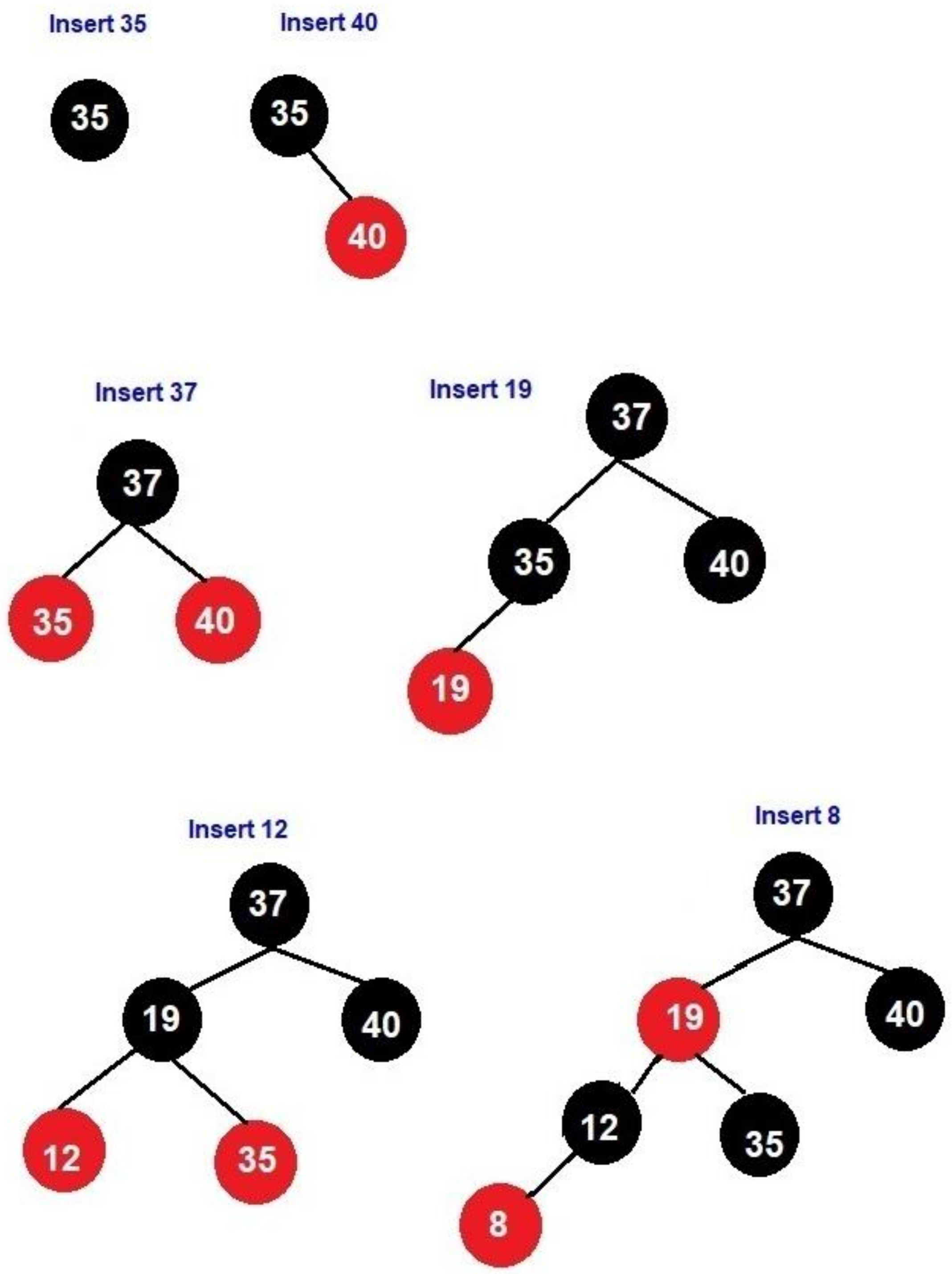
1	2	3	4	5	6	7	8	9	10	11
0	**0**	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**6**	**6**

C – Count Array

0	1	2	3	4	5	6
0	**2**	**4**	**6**	**8**	**9**	**9**

Question 2:

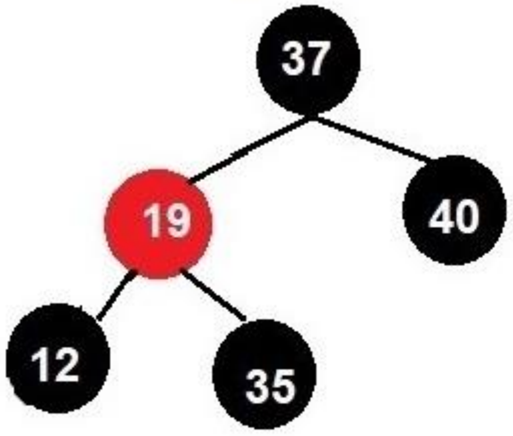
Red-black trees after each insertion: 35, 40, 37, 19, 12, 8



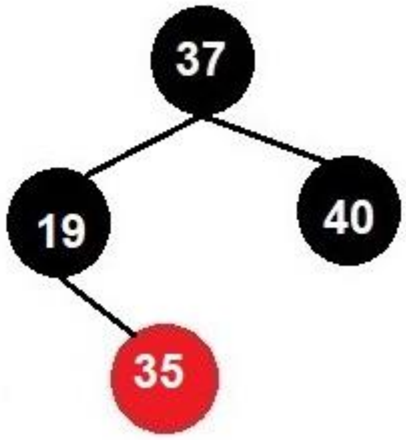
Question 3:

Red-black trees after each deletion: 8, 12, 19, 37, 40, 35

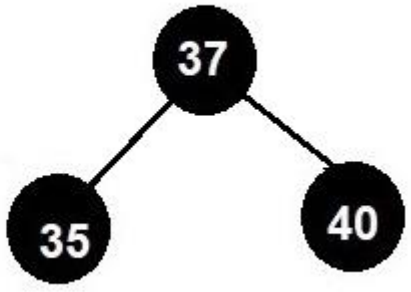
Delete 8



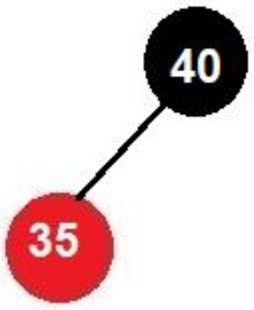
Delete 12



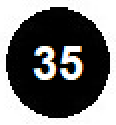
Delete 19



Delete 37



Delete 40



Delete 35

NIL

Question 4:

Algorithm Description

Input: A[n], k - an unsorted array with n elements, integer k
Output: a sorted sequence of smallest k elements with time complexity O(n)

Algorithm:

Build a Max-Heap of the first k elements of array A - A[0: k-1]

Iterate other elements – A [k: n-1]; compare each one with the root of the Max-Heap:

- If the element is smaller than the root, replace it with the root and heapify the heap.
- Otherwise, ignore it.

This will return the smallest k elements in the end

Max-Heap:

A Max-Heap is a binary tree where each node is greater than or equal to its children. That means the root will be the largest element of all tree nodes.

Build a Max-Heap:

Here is how to build the Max-Heap:

Start with the last non-leaf node in the array A[floor(n/2):1]; Call Heapify on this node to fix the node and its children to satisfy the Max-Heap property

Recursively call Heapify on the left/right child of the current node until all nodes in the heap satisfy the Max-Heap property

Heapify:

Input: Array A, index i

First build a binary tree structure from the input array A. The left child index will be 2i and the right child index will be 2i+1 where i is the current node index

Then we fix the binary tree to satisfy the heap property:

- Start from the last parent node and compare it with its child, If the parent is not greater than both of its children, swap it with the larger child
- Recursively call heapify to check all parent nodes in all subtrees until the entire tree satisfied the heap property

Algorithm Correctness

To show the correctness of this algorithm, we need to show:

1. This algorithm returns a sequence of smallest k elements
2. This algorithm runs in O(n) time complexity (this will be explained at the next section)

We start to build a Max-Heap of the fist k elements of the input array A. So we partition the array into two parts: A[0:k-1] and A[k:n-1]. Therefore, in the first part, the root node is the largest element of k elements. Then we compare every element in the second part with the root node in the Max-Heap, if it is smaller than the root node, we will swap them, so the smaller one will stay in the first part. We Heapify the first part to keep it as a Map-Heap. By recursing on this, we keep the smaller item in the first part and eject the larger element as part of the k smallest elements.

Therefore, the algorithm always returns a list of the k smallest elements.

Algorithm Complexity

Build a Max-Heap of the first k elements: O(k)

For the remaining n-k elements:

- Compare the element with the root of the Max-Heap: O(1)
- Heapify: O(logk)

$$\begin{aligned} k + (n-k)(1 + \log k) &= k + (n-k) + (n-k)\log k \\ &= k + n - k + (n-k)\log k \\ &= n + (n-k)\log k \\ &\le n + n\log k \\ &= n + n \\ &= 2n \\ &= O(n) \end{aligned}$$

Therefore, the time complexity of this algorithm is **O(n)**

Question 5:

To design an efficient data structure using modified red-black trees, we can modify the standard red-back tree data structure and add a size property for tree node object (node.size).

node.size represents the number of nodes in its subtree including itself. We can design each function as below:

Insert(x)

We can do a standard binary search tree insertion into a red-black tree and update the node.size for all nodes that are inside the subtree on the path from the root to the insert node.

The time complexity of insertion is **O(logn)**, n is the number of items in the tree.

Delete(x)

Same as insertion, we do a standard binary search tree deletion from a red-black tree and update the node.size of all nodes in the subtree on the path from the root to the delete node.

The time complexity of deletion is **O(logn)**

Find_Smallest(x)

Start at the root and compare x.size with the size of nodes(n.size) that in the left subtree rooted at the current node. If x.size ≤ n.size, we can keep searching the left subtree recursively for the xth smallest item, otherwise we subtract the size of the left subtree plus one from x.size and search the right subtree for the (x.size – leftSubTree.size -1)th smallest item. We keep searching until we reach a leaf node or the xth smallest element is found.

The time complexity of find smallest is also **O(logn)**

Question 6.

Theorem: Every node has rank at most ⌊lgn⌋.

Define x.rank and x.size to be the rank and size of nodes in the tree rooted at node x.

To show the above theorem, we will prove below two theorems first:

1. X.size ≥ $2^{X.rank}$
2. There are at most $n/2^r$ nodes of rank r. r ∈ integer and r > 0

Proof (of(1)): Induction on the sequence of Make_set and Union operations.

Base case: the first operation must be Make_set, we have one node with rank 0:

x.size = 1
$2^{X.rank} = 2^0 = 1$
Therefore, the property holds true for the base case

Induction step: Assume that the theorem holds before performing the Union(X_a, X_b) operation.

Let rank, size be the ranks and sizes before the Union operation, and the rank' and size' be the ranks and sizes after Union operation.

Case1: *X_a.rank = X_b.rank*:

Assume X_b be the root of the new tree after Union operation, we get
X_b.size' = X_a.size + X_b.size ≥ $2^{Xa.rank} + 2^{Xb.rank} = 2^{Xb.rank+1} = 2^{Xb.rank'}$

Case2: *X_a.rank ≠ X_b.rank*:

Assume X_a.rank < X_b.rank; so X_b will be the root of the new tree after Union operation, we get
X_b.size' = X_a.size + X_b.size ≥ $2^{Xa.rank} + 2^{Xb.rank} \geq 2^{Xb.rank} = 2^{Xb.rank'}$

□

Proof (of(2)): Induction on the sequence of Make_set and Union operations.

Base case: the first operation must be Make_set, we have one node with rank 0:
The property holds true since the number of nodes of rank 0 = n/2r = 1/10 = 1

Induction step: Assume that the theorem holds before performing the Union(X a, X b) operation

Case1: *X_a.rank ≠ X_b.rank*: The theorem is true since there is no rank and size change

Case2: *X_a.rank = X_b.rank*:

Follows by the Theorem1 and there are a total of n nodes, plus the root's descendants of two nodes on the same level are disjoint. There are at most $n/2^r$ disjoint sets with at least 2^r nodes.

□

Proof Every node has rank at most ⌊lgn⌋.
Let r be the rank and according to Theorem2 that we proved before, there are at most $n/2^r$ nodes of rank r. Since r ∈ integer and r > 0, we get

$n/2^r \geq 1 \Rightarrow 2^r \leq n \Rightarrow r \leq \lg n$

Thus, every node has rank at most ⌊lgn⌋.

□

Question 7.

Since every node has rank at most $\lfloor \lg n \rfloor$, the number of bits required to store the rank for each node x will be

$$\lfloor \lg \lfloor \lg n \rfloor \rfloor + 1 = \lfloor \lg \lg n \rfloor + 1$$

For Question 9a, we use 1 byte which is 8 bits to store rank which is

$$= \lfloor \lg \lg n \rfloor + 1 = 8$$

$$\text{Lg}(\lg n) = 7$$

$$\text{Lgn} = 2^7$$

$$n = 2^{128}$$

This is big enough to store the rank for question 9a

Question 8.

15.3-3

An optimal Huffman code for the first 8 Fibonacci numbers is as below:

Frequency	Character	Code
1	a	0000000
1	b	0000001
2	c	000001
3	d	00001
4	e	0001
5	f	001
13	g	01
21	h	1

15.3-4

Theorem: The total cost B(T) of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

Proof Induction on the number of merge operations.

Let T be the binary tree, I(T) be the set of internal nodes of T and C(T) be the set of leaves of T.

We need to show that B(T) = $\sum_{x\in I(T)} x.freq,$ where B(T) is defined as B(T) = $\sum_{x\in I(T)} x.freq.\mathrm{d}_T(c)$

Base case: T has one internal node x, others are all leaves (l and r), we have

$$\sum_{x\in I(T)} x.freq = z.freq = l.freq + r.freq = \big(l.freq.\mathrm{d}_T(l)\big) + \big(r.freq.\mathrm{d}_T(r)\big)$$

$$= \sum_{x\in I(T)} c.freq.\mathrm{d}_T(c) = \mathrm{B(T)}$$

Let L and R be the left and right subtree of T. The inductive hypothesis holds for

B(L) = $\sum_{x\in I(L)} x.freq$ and B(R) = $\sum_{x\in I(R)} x.freq.\mathrm{d}_T(c)$

Let T has root z, so that $z.freq = \sum_{c\in C(T)} c.freq$, we have

$$BT = \sum_{c\in C(L)} c.freq.\mathrm{d}_T(c) + \sum_{c\in C(R)} c.freq.\mathrm{d}_T(c)$$

$$= \sum_{c\in C(L)} c.freq.(\mathrm{d}_L(c)+1) + \sum_{c\in C(R)} c.freq.(\mathrm{d}_R(c)+1)$$

$$= \sum_{c\in C(L)} c.freq.\mathrm{d}_L(c) + \sum_{c\in C(R)} c.freq.\mathrm{d}_R(c) + \sum_{c\in C(L)} c.freq + \sum_{c\in C(R)} c.freq$$

$$= B(L) + B(R) + \sum_{c\in C(L)} c.freq$$

$$= \sum_{x\in I(L)} x.freq + \sum_{x\in I(R)} x.freq + z.freq \qquad \text{(by the inductive hypothesis)}$$

$$= \sum_{x\in I(T)} x.freq$$

□

15.3-5

We can use a full binary tree as the data structure, by a preorder walk on the tree and mark 1 for the internal node and 0 for the leaves. A full binary tree has n-1 internal nodes and n leaves which total is 2n-1 nodes. Since the C code is on n characters and , $\lceil lgn \rceil$ bits are needed for each character, so if each leaf represents a character, n$\lceil lgn \rceil$ bits represents all the characters. That is total number of 2n-1+ n$\lceil lgn \rceil$.

Question 9.

The algorithms and code are implemented in ***uandf.java*** and ***question9.java***

Function ***final_set()*** is designed to finalized the data structure, it resets the representatives of the sets so that integers from 1 to size will be used as representatives.
It will first go through the parents list to find the representatives and reset the value if needed; it also finds the set count and updates the rank value accordingly. Finally, it returns the total number of current sets.

final_set() is implemented in the class uandf.java which implements a Disjoint-Set data structure. Integer array is used for parents and ranks, Disjoint-set data structure functions like make_set(i), find_set(i), union_sets(I, j) and final_sets(i) are implemented in class uandf.java

The time complexity of this function is O(nlgn). Going through the parents list takes O(n), invoke *find_set(i)* takes log(n) since there is recursive function call, other updates takes O(1).
The overall time complexity of ***final_set()*** function is **O(nlogn)**