Section B

Problem Description

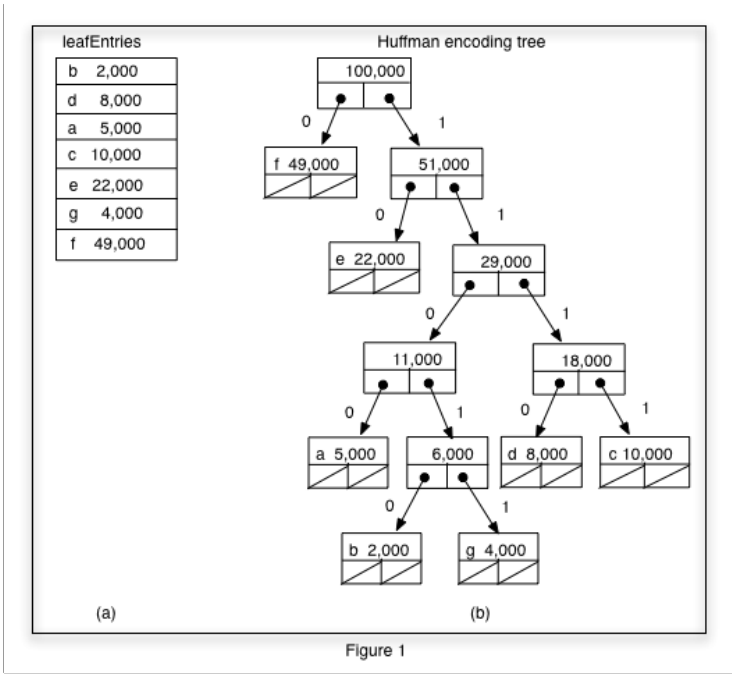


Figure 1

The Huffman encoding is a data compression strategy. Unlike the UCS-2 (old Unicode) encoding that uses the same number of bits (i.e. 0 and 1) to encode a character, the Huffman encoding uses different number of bits to encode a character, depending on the frequency that the character has in a given message. Binary trees can be used to represent Huffman codes. For instance, for the sample frequencies of characters ‘a’ to ‘g’ given in Figure 1(a), the Huffman encoding tree representation would the binary tree in Figure 1(b).

All the characters and their respective frequencies are in the leaf nodes. Each non-leaf node includes only a frequency value given by the sum of the frequency values of its two children. The frequency value of the left child is smaller than the frequency value of the right child. The Huffman encoding for a character is the binary string formed by traversing the tree from the root node to the leaf containing that character. Each time you go left, you append a 0, and each time you go right, you append a 1.

For instance, the Huffman encoding for the letter ‘b’ in Figure 1 is 11010.

The algorithm for building a Huffman tree makes use of a priority queue of binary trees where the element with highest priority is the tree whose root node has the smallest frequency. Assume this priority queue to be implemented as an appropriate Heap.

The algorithm consists of the following steps:

1. Store in an array each character and its frequency (see `setFrequencies()`).

2. For each element in the array construct a binary tree with root node containing the individual character and its frequency and place the binary tree in a priority queue (see `setPriorityQueue()`).

3. Create the Huffman tree as follows:
   While the priority queue has more than one item

   Remove the two binary trees with the smallest frequencies.

   Combine them into a new binary tree on which the frequency of the tree root is the sum of the frequencies of its children.

   Insert the newly created tree back into the priority queue.

Step 2 and the first iteration of Step 3 are illustrated respectively in Figure 2(a) and (b).

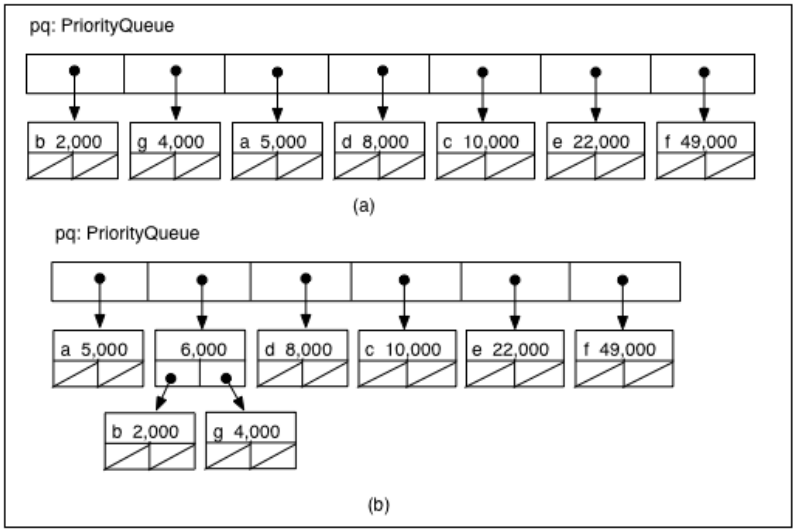


Figure 2

Your task is to complete the provided implementation. In addition to this specification, you should carefully read the commented source files. *Do not rename or rearrange the source code given; in particular do not introduce any packages (that is, the keyword "package" should not appear anywhere in your code).*

You are given:

- For your convenience, a UML class-diagram describing the architecture of the system. Your implementation will build upon the given design.
- The class `HuffmanData` that includes a character and its frequency. This implements Comparable<HuffmanData>.
- The interface `PriorityQueueInterface<K extends Comparable<K>>`.
- A partial implementation of the class `PriorityQueue<K>` that realises the interface `PriorityQueueInterface<K extends Comparable<K>>`.
- The interface `BinaryTreeInterface<K extends Comparable<K>>` that extends Comparable<BinaryTreeInterface<K>>.

- The class `BinaryTree<K>` that realises the interface `BinaryTreeInterface <K extends Comparable<K>>`. This class uses an inner class TreeNode.
- A partial implementation of the class `Huffman` that is responsible for building an HuffmanTree and printing the Huffman codes of the characters.

You are asked to write:

1) The methods `add(E newEntry)` and `heapRebuild(int root)` within the class `PriorityQueue<E>`.

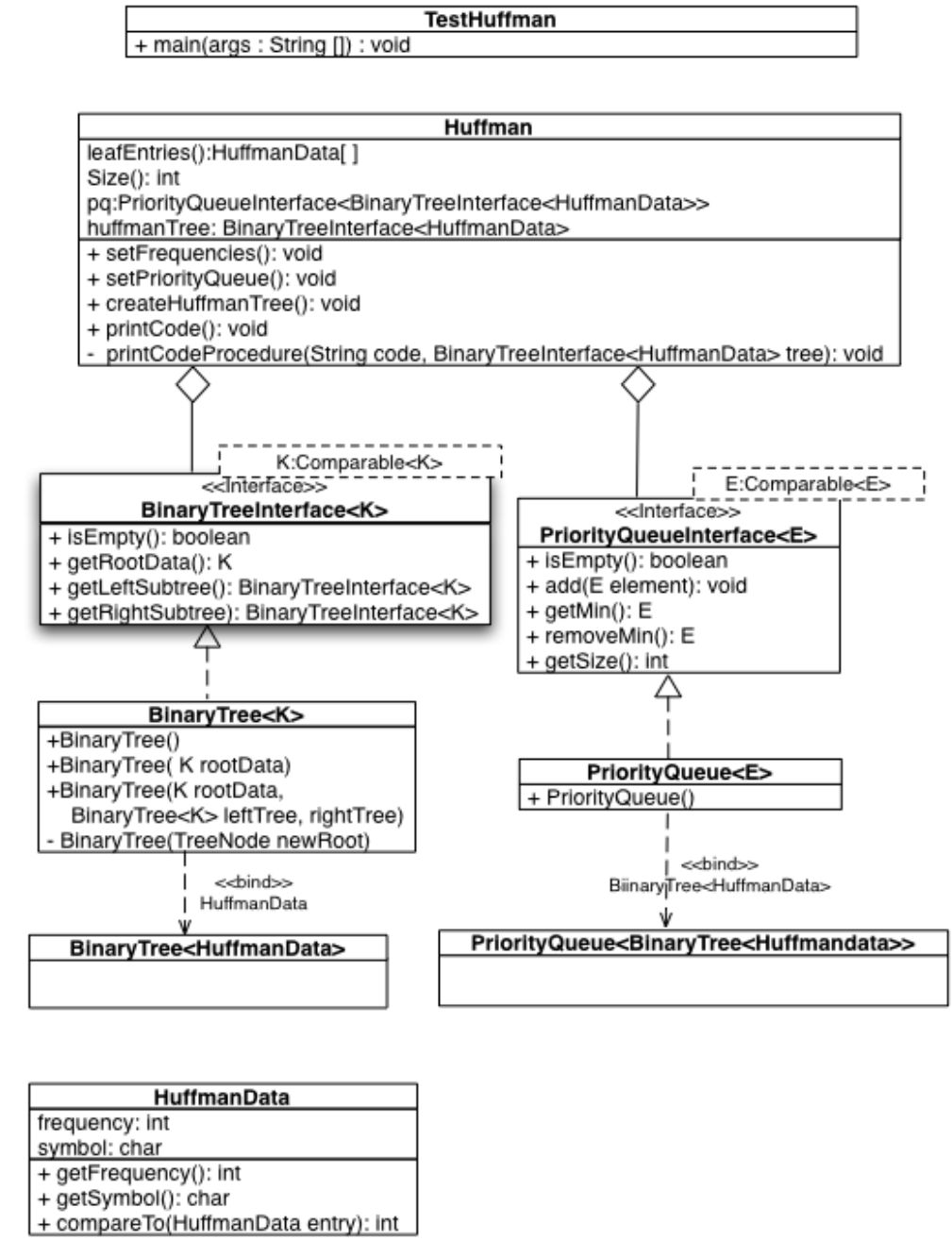
2) The method `createHuffmanTree()` that implements Step (3) in the algorithm given above.

3) The method `printCode()` that prints the Huffman codes of all the characters using the Huffman tree created by the method in point (2) above.

For the symbols and frequencies given in Figure1(a), your implementation should produce the following output:

```
f: 0
e: 10
a: 1100
b: 11010
g: 11011
d: 1110
c: 1111
```

System Architecture


TestHuffman
+ main(args : String []) : void
Huffman
leafEntries():HuffmanData[ ]
Size(): int
pq:PriorityQueueInterface<BinaryTreeInterface<HuffmanData>>
huffmanTree: BinaryTreeInterface<HuffmanData>
+ setFrequencies(): void
+ setPriorityQueue(): void
+ createHuffmanTree(): void
+ printCode(): void
- printCodeProcedure(String code, BinaryTreeInterface<HuffmanData> tree): void
K:Comparable<K>
<<Interface>>
BinaryTreeInterface<K>
+ isEmpty(): boolean
+ getRootData(): K
+ getLeftSubtree(): BinaryTreeInterface<K>
+ getRightSubtree): BinaryTreeInterface<K>
E:Comparable<E>
<<Interface>>
PriorityQueueInterface<E>
+ isEmpty(): boolean
+ add(E element): void
+ getMin(): E
+ removeMin(): E
+ getSize(): int
BinaryTree<K>
+BinaryTree()
+BinaryTree(K rootData)
+BinaryTree(K rootData,
BinaryTree<K> leftTree, rightTree)
- BinaryTree(TreeNode newRoot)
PriorityQueue<E>
+ PriorityQueue()
<<bind>>
HuffmanData
<<bind>>
BiinaryTree<HuffmanData>
BinaryTree<HuffmanData>
PriorityQueue<BinaryTree<Huffmandata>>
HuffmanData
frequency: int
symbol: char
+ getFrequency(): int
+ getSymbol(): char
+ compareTo(HuffmanData entry): int