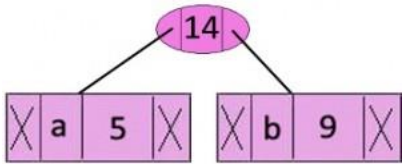
Huffman Coding | Greedy Algo

Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.
See this for applications of Huffman Coding.
There are mainly two major parts in Huffman Coding
1) Build a Huffman Tree from input characters.
2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree
Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
Let us understand the algorithm with an example:

```
character   Frequency
    a            5
    b           9
    c           12
    d           13
    e           16
    f           45
```

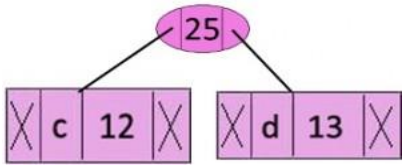
Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

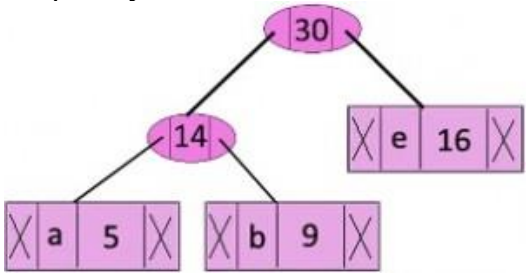
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

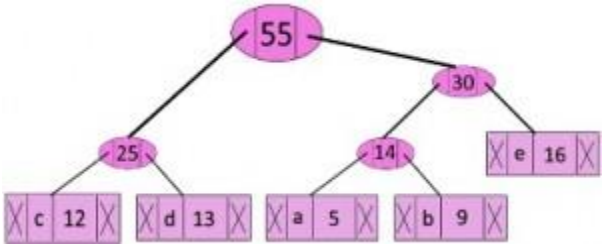


Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30

```
      f                45
```

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



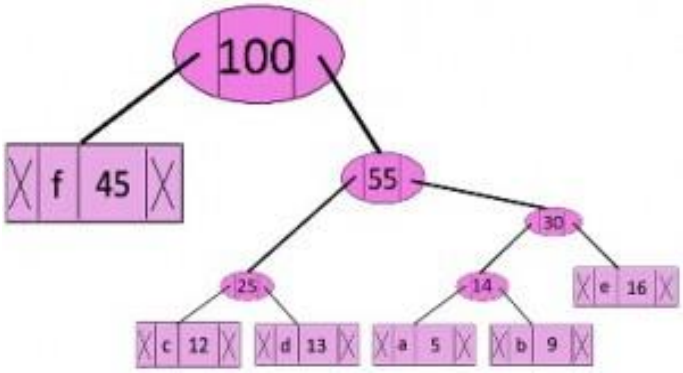
Now min heap contains 2 nodes.

```
character      Frequency

       f          45

Internal Node     55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



Now min heap contains only one node.

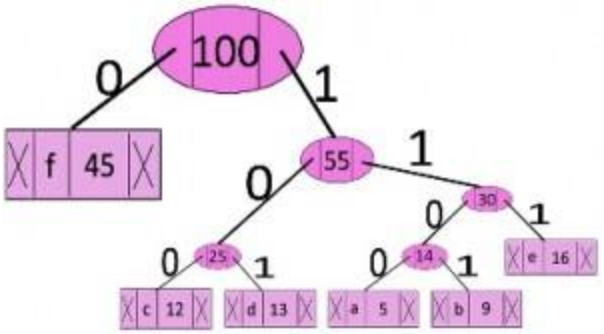
```
character      Frequency

Internal Node     100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character   code-word

    f          0
```

```
    c          100
    d          101
    a          1100
    b          1101
    e          111
```

Below is the implementation of above approach:

```
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;

// node class is the basic structure
// of each node present in the Huffman - tree.
class HuffmanNode {

    int data;
    char c;

    HuffmanNode left;
    HuffmanNode right;
}

// comparator class helps to compare the node
// on the basis of one of its attribute.
// Here we will be compared
// on the basis of data values of the nodes.
class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y)
    {

        return x.data - y.data;
    }
}

public class Huffman {

    // recursive function to print the
    // huffman-code through the tree traversal.
    // Here s is the huffman - code generated.
    public static void printCode(HuffmanNode root, String s)
    {

        // base case; if the left and right are null
        // then its a leaf node and we print
        // the code s generated by traversing the tree.
        if (root.left
                == null
            && root.right
                   == null
            && Character.isLetter(root.c)) {
```

```
            // c is the character in the node
            System.out.println(root.c + ":" + s);

            return;
        }

        // if we go to left then add "0" to the code.
        // if we go to the right add"1" to the code.

        // recursive calls for left and
        // right sub-tree of the generated tree.
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }

    // main function
    public static void main(String[] args)
    {

        Scanner s = new Scanner(System.in);

        // number of characters.
        int n = 6;
        char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
        int[] charfreq = { 5, 9, 12, 13, 16, 45 };

        // creating a priority queue q.
        // makes a min-priority queue(min-heap).
        PriorityQueue<HuffmanNode> q
            = new PriorityQueue<HuffmanNode>(n, new MyComparator());

        for (int i = 0; i < n; i++) {

            // creating a Huffman node object
            // and add it to the priority queue.
            HuffmanNode hn = new HuffmanNode();

            hn.c = charArray[i];
            hn.data = charfreq[i];

            hn.left = null;
            hn.right = null;

            // add functions adds
            // the huffman node to the queue.
            q.add(hn);
        }

        // create a root node
        HuffmanNode root = null;

        // Here we will extract the two minimum value
        // from the heap each time until
```

```
        // its size reduces to 1, extract until
        // all the nodes are extracted.
        while (q.size() > 1) {

            // first min extract.
            HuffmanNode x = q.peek();
            q.poll();

            // second min extarct.
            HuffmanNode y = q.peek();
            q.poll();

            // new node f which is equal
            HuffmanNode f = new HuffmanNode();

            // to the sum of the frequency of the two nodes
            // assigning values to the f node.
            f.data = x.data + y.data;
            f.c = '-';

            // first extracted node as left child.
            f.left = x;

            // second extracted node as the right child.
            f.right = y;

            // marking the f node as the root node.
            root = f;

            // add this node to the priority-queue.
            q.add(f);
        }

        // print the codes by traversing the tree
        printCode(root, "");
    }
}
```

```
f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111
```

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).