```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;

/**
 *
 * @author Rasmus Bartholin og Mads Mikael Keinicke
 * Rasmus: rbart17
 * Mads: makei17
 */
public class HuffTree {

    // ArrayList containing the bitcodes for our HuffTree
    ArrayList<String> codes = new ArrayList<String>();

    // Empty constructor
    public HuffTree() {

    }

    /**
     * Creates the actual tree, returning the rootnode of the tree
     *
     *
     * @param heap a PQHeap used to create the tree from, taking keys as frequencies.
     * @return HuffNode that represents the root of the tree
     */
    public Element HuffUnify(PQHeap heap)
    {
        // Retrieve the size of the given Heap
        int n = heap.getSize();

        // Copy the Heap
        PQHeap q = heap;

        // Loop creating the Huffman tree
        for(int i = 1; i < n; i++)
        {
            // Create new parentnode
            HuffNode parent = new HuffNode();

            // Create the new left child, by extracting the minimum from the heap
            HuffNode lChild = new HuffNode(q.extractMin());

            parent.setLchild(lChild);
            lChild.setParent(parent);

            // Create the new right child, by extracting the minimum from the heap, which is higher than the left child
            HuffNode y = new HuffNode(q.extractMin());
            parent.setRchild(y);
            y.setParent(parent);

            // set frequency of the parentnode
            parent.setFreq(lChild.getFreq() + y.getFreq());

            // Insert into the heap
            q.insert(new Element(parent.getFreq(), parent));
        }
        // return the final node as the rootnode of the tree
        return q.extractMin();
    }

    /**
     * Returns a String[] containing the bitcode of the Huffman Tree hte given node is a root of
     *
     * Uses a helper method of same name, but is overloaded with different parameters
     *
     * @param rootNodeA HuffNode that is the rootnode of a Huffman tree
     * @return String[] that contains bitcode for the ASCII character of the given index
     */
    public String[] findCode(HuffNode rootNode)
```

```
    {
        // set a HuffNode as the root, which is the node given as parameter
        HuffNode root = rootNode;

        // Initialize the List to conatin the bitCode
        String[] list = new String[256];

        // starts the traversal, updating the bitCode list with each call, if the root is not null
        if(root != null)
        {
            // Call the recursive helepr method, starting the bitcode with "0", using the left child of the given root
            list = findCode(root.getLchild(), list, "0");

            // Call the recursive helepr method, starting the bitcode with "1", using the right child of the given root
            list = findCode(root.getRchild(), list, "1");
        }
        return list;

    }
    /**
     * Helper method for the public findCode. Calls itself recursively, returning the new String[] of bitcodes each call
     *
     * Each recursion calls itself using both children in the genreal case, update the bitcode with the proper number.
     * Base case updates the list given as parameter, and returns it.
     *
     *
     * @param node the current node the iteration has gotten to
     * @param oldList the old String[] of bitcode
     * @param bitCode the String representing the current bitCode that has yet to be added
     * @return the String[] of the bitcode
     */
    private String[] findCode(HuffNode node, String[] oldList, String bitCode)
    {
        // Retrieve the bitcode of this call
        String newBit = bitCode;

        // Retrieve the list of the current call
        String[] newList = oldList;

        // Base case, testing if the root is a leaf, by checking if it has two children that are null
        if(true && node.getLchild() == null && node.getRchild() == null)
        {
            // get the index of the ASCII character
            int index = (Integer) node.getData();

            // add the current bitcode to list of bitCode
            newList[index] = newBit;

            // return the new list
            return newList;
        }
        // general case both childs exist, there will never be a case of only having one child that exists.
        else {
            // if node exists
            if(node != null)
            {
                // go left with new bitcode and update list
                newList = findCode(node.getLchild(), newList, newBit + "0");

                // go right with new bitcode and update list
                newList = findCode(node.getRchild(), newList, newBit + "1");
            }

            return newList;
        }
    }


    /**
     * Method that gives the frequency of the different chars in a file, using bytes.
     *
     * @param Str
```

```
     * @return int[] containing the frequency of the index's byte
     */
    public static int[] getFrq(String filePath)
    {
        // try statement in case path does not exist
        try {
            // create input stream
            FileInputStream fin = new FileInputStream(filePath);

            // Loop that increases frequency by one, each time a number has been encountered
            int x = 0;
            int[] freqs = new int[256];
            while((x = fin.read()) != -1)
            {
                freqs[x]++;
            }

            // return the list of frequencies
            return freqs;
        } catch (IOException e) {
            System.out.println(e);
        }
        // In case a file was not found, return null
        return null;
    }

    /**
     * Creates a heap with 256 nodes, using the index's value as key, and it's index as data.
     *
     * One element for each possible unicode character.
     *
     * @param list
     * @return
     */
    public PQHeap createHeap(int[] list)

    {
        // Initialize the heap
        PQHeap heap = new PQHeap(256);


        // Inserts unicode frequency into heap, with the ASCII unicode as data, and frequency as key
        int i = 0;
        for(int x : list)
        {
            heap.insert(new Element(x, i));
            i++;
        }
        return heap;
    }

    /**
     * Prints out the frquency of each number that has been encountered
     *
     * Used for testing purposes, ignores frquencies of 0
     *
     * @param list
     */
    public static void printFreq(int[] list)
    {
        ArrayList<String> arList = new ArrayList<String>();
        int i = 0;
        String tmp;
        for(int str : list)
        {
            tmp = String.valueOf(i);
            if(str == 0)
            {
            }
            else
            {
```

```
                tmp += ": ";
                tmp += String.valueOf(str);
                arList.add(tmp);
            }
            i++;
        }
        System.out.println(Arrays.toString(arList.toArray()));
    }


}
```