```java
/*Alan Stoloff
 * Data Structures
 * Dr. Benjamin
 *
 *    class HEncode − A program to encode a file using Huffman Code Algorithm.
 */

import java.io.*;
import java.lang.reflect.Array;
public class HEncode {

    private Node root = null;               // Root of the Huffman Code Tree.

    private Node[] leafPtr = new Node[256]; // Array of pointers to leaf nodes.
                                            // used to traverse up the code
                                            // tree during the encoding process.


    private int[] freq = new int[256]; // Frequency of the bytes being encoded.
                                       //    used to build the initial trees
                                       //    when building the code tree.

    private String inputFilename;   // The name of the file to encode.

                      // pq is a priority queue of root nodes to trees.  It
                      // is used during the building of the code tree to
                      // select the roots with minimum frequency count.

    private PriorityQueue<Node> pq = new PriorityQueue<Node>();

                      // stk is a stack used to store the 0’s and 1’s
                      // of the code for a byte being encoded.  As a byte
                      // is being encoded, we travel from a leaf node, up
                      // the parent pointers to the root, pushing 0’s and 1’s
                      // as appropriate for the encoding.  When the root
                      // is reached, we pop the stack of "bits" into the
                      // output file.

    private Stack<Integer> stk = new Stack<Integer>();

    private BitWriter bitw;   // Writes bits to the outputfile.

    public final boolean DEBUG = true; // When true debugging info
                                       // is displayed.





    public static void main(String[] args)
            throws FileNotFoundException, IOException
    {
        if (args.length != 1) {
            System.out.println("Incorrect program argument");
            System.exit(0);
        }

        HEncode coder = new HEncode(args[0]);  // Construct a Huffman Encoder

        coder.getFrequencies();  // Get the frequencies of bytes in inputfile.
        if (coder.DEBUG)
            coder.showFreq();    // For debug - Let’s see if we got the freqs.
        coder.getLeafPtrs();     // Get initial trees used to build code tree.
        if (coder.DEBUG)         // For debug - print priority queue of roots.
            coder.showPQ();
        coder.buildTree();       // Build the code tree.
        if (coder.DEBUG)         // For debug - print the code tree.
            coder.printTree();
        coder.encodeFile();      // Read the inputfile a second time, encoding
                                 // the inputfile.
    }

    /*
     *    Constructor - The argument is the name of the file to encode.
     */

    public HEncode(String inputFilename)
    {
        this.inputFilename = inputFilename;
    }


```

```java

    /*
     *   getFrequencies() - Open the given file and determine the frequency
     *                      with which each byte (character) occurs.
     *
     *   The frequencies are store in the array freq at the index
     *   location corresponding to the byte value 0 to 255.
     */

    public void getFrequencies()
    {
        FileInputStream inF;   // File object to read from.
        int nextByte;          // Next byte from the file.

        // Initialize the frequencies

        for (int i = 0; i < 256; i++)
            freq[i] = 0;

        try {
            inF = new FileInputStream(inputFilename);  // Open the input file.

            do {
                nextByte = inF.read();    // Read the next byte (-1 on EOF)
                if (nextByte != -1)       //
                    freq[nextByte]++;     //    Increment frequency counter
            } while (nextByte != -1);     //    for the byte.

            inF.close();                  //  Close the file.
        }
        catch (FileNotFoundException e) {
            System.out.printf("Error opening file %s\n", inputFilename);
            System.exit(0);
        }
        catch (IOException e) {
            System.out.printf("IOException reading from: %s\n", inputFilename);
            System.exit(0);
        }
    }



    /*
     *   showFreq() - display the byte frequency array.
     *
     *   For debugging purposes we want to show the frequency with
     *   which each byte (or character) occurs.
     */

    public void showFreq()
    {
        for (int i = 0; i < 256; i++) {   // Only show the bytes
            if (freq[i] != 0)             // having non-zero frequency.
                System.out.printf("byte: %3d  char: %c  freq: %d\n",
                                               i, (char) i, freq[i]);
        }
    }


    /*
     *    getLeafPtrs() - Create root nodes containing the bytes.  These nodes
     *                    are the roots of the initial trees used to build the
     *                    code tree.  They will become the leaf nodes of the
     *                    Huffman Code tree.
     *
     *                    Enter the nodes in the priority queue of roots.
     *                    as each node is created.
     *                    Nodes with smaller frequency have higher priority.
     */

    public void getLeafPtrs()
    {
        //need a ponter to priority Queue
        for(int i=0;i<256;i++){
            if(freq[i]!=0){
                Node temp=new Node();
                leafPtr[i]=temp;
                temp.data=(byte)i;
                temp.frequency=freq[i];
                pq.enqueue(temp);
            }
        }
    }

    public void showPQ()
    {
        System.out.println(pq.toString());
    }

    /*
     *    buildTree() - A function to build the Huffman Code Tree.
     *                  We start with a priority queue of the leaf nodes.
     */

    public void buildTree()
    {
        Boolean keepGoing=true;
        Node temp=pq.dequeue();
        while(keepGoing){
            if(pq.isEmpty()){
                pq.enqueue(temp);
                this.root=temp;
                keepGoing=false;
                break;
            }
            Node rootTemp=new Node();
            Node rtemp=pq.dequeue();
            rootTemp.rchild=rtemp;
            rootTemp.lchild=temp;
            temp.parent=rootTemp;
            rtemp.parent=rootTemp;
            rootTemp.frequency=rootTemp.rchild.frequency+rootTemp.lchild.frequency;
            temp=rootTemp;
        }
    }

    /*
     *  encodeFile() - a function to encode a file - that is to
     *                 create a compressed file using the Huffman
     *                 algorithm. The function:
     *  1. Creates a BitWriter to write the compressed file.
     *  2. Uses the BitWriter to write the size of the original file.
     *  3. Calls writeTree() to write the code tree.
     *  4. Opens the original file.
     *  5. Repeatedly reads a byte from the file and
     *         and calls writeCode to write the coded bits
     *         for the byte to the compressed file.
     *  6. Closes the BitWriter
     */

    public void encodeFile()
    {
        bitw=new BitWriter("book.txt.huf");
        bitw.writeInt(root.frequency);
        writeTree(root);
        FileInputStream inF;
        int nextByte;
        try {
            inF = new FileInputStream(inputFilename);  // Open the input file.

            do {
                nextByte = inF.read();    // Read the next byte (−1 on EOF)
                if (nextByte != −1)       //
                    writeCode((byte)nextByte);     //    Increment frequency counter
            } while (nextByte != −1);     //    for the byte.

            inF.close();
            bitw.close();                  //  Close the file.
        }
        catch (FileNotFoundException e) {
            System.out.printf("Error opening file %s\n", inputFilename);
            System.exit(0);
        }
        catch (IOException e) {
            System.out.printf("IOException reading from: %s\n", inputFilename);
            System.exit(0);
        }
    }



    /*
     *   writeCode() − A function to encode byte b.  The function
     *                 uses b as an index into the array of pointers
     *      to leaf nodes of the Huffman Code Tree.  Once at the leaf,
     *      parent pointers are used to climb to the root, pushing
     *      0’s and 1’s on a stack according to the encoding.  Once
     *      at the root, the o’s and 1’s on the stack are popped off
     *      and written to the outputfile using the bitWriter.
     */

    public void writeCode(byte b)
    {
        Node ptr=leafPtr[b];
        while(ptr!=root){
            Node ptrParent=ptr.parent;
            if(ptrParent.rchild==ptr){
                stk.push(1);
            }
            if(ptrParent.lchild==ptr){
                stk.push(0);
            }
            ptr=ptr.parent;
        }
        while(!stk.isEmpty()){
            bitw.writeBit(stk.pop());
        }
    }
    /*
     *    writeTree() − A recursive function to write the Huffman
     *                  Code Tree to the output file.  The tree
     *       must be stored with the encoded file so that it can
     *       be used to decode the file.
     */


    public void writeTree(Node root)
    {
        if(root==null){
            return;
        }
        else if(root.lchild==null && root.rchild==null){
            bitw.writeBit(0);
            bitw.writeByte(root.data);
        }
        else{
            bitw.writeBit(1);
            writeTree(root.lchild);
            writeTree(root.rchild);
        }
    }


```

```java

    /*
     *   printTree() − Print the Huffman Code Tree to
     *                 standard output.
     */


    public void printTree()
    {
        rPrintTree(root,0);
    }

    /*
     *    rPrintTree() − the usual quick recursive method to print a tree.
     */

    public void rPrintTree(Node r, int level)
    {

        if (r == null)          // Empty tree.
            return;

        rPrintTree(r.rchild, level + 1);   // Print the right subtree.

        for (int i = 0; i < level; i++)
            System.out.print("         ");

        if (r.data > (byte) 31)
            System.out.printf("%c−%d\n", (char) r.data, r.frequency);
        else
            System.out.printf("%c−%d\n", ’*’, r.frequency);

        rPrintTree(r.lchild, level + 1);
    }


    /*
     *    Node − an inner class to represent a node of
     *           a Huffman Code Tree.
     */

    private class Node implements Comparable<Node>
    {
        byte data;           // A byte of data − stored in an Integer.
        Node lchild;         // Left child pointer.
        Node rchild;         // Right child pointer.
        Node parent;         // Pointer to parent node.
        Integer frequency;   // Frequency the data within
                             // a file being encoded.
        /*
         *   Basic node constructor.
         */

        public Node()
        {
            data = 0;          // Each Huffman Code Tree node
            lchild = null;     // contains data, pointers to
            rchild = null;     // children and parent nodes
            parent = null;     // plus a frequency count
            frequency = 0;     // associated with the data.
        }

        /*
         *   Constructor specifying all values
         *   of the node instance variables.
         */

        public Node(byte data, Node lchild, Node rchild,
                               Node parent, int frequency)
        {
            this.data = data;
            this.lchild = lchild;
            this.rchild = rchild;
            this.parent = parent;
            this.frequency = frequency;
        }

        /*
         *    compareTo() − Compare two frequency values.  We want Nodes
         *                  with lower frequencies to have higher priority
         *                  in the priority queue.
         *
         */

        public int compareTo(Node other)
        {
            if(this.frequency>other.frequency){
                return −1;
            }
            else if(this.frequency<other.frequency){
                return 1;
            }
            else{
                return 0;
            }
        }

        public String toString()
        {
            char ch = (char) this.data;

            String str = "byte: " + data + "  char: ";

            if (data > (byte) 31)
```

```
                str = str + (char) data + "  freq: " + frequency;
            else
                str = str + " " + "  freq: " + frequency;

            return str;
        }
    }
}
```