**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileOutputStream;  
**import** java.io.IOException;

*/\*\*  
 \* Encode implementation  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/*  
**public class** Encode {  
  
 **public static void** main(String[] args) {  
 **int**[] freqTable = **new int**[256]; *//Indexes are byte values (ints) and values are frequency* String[] variableLengthCodeWords = **null**;  
  
 **try** (FileInputStream inFile = **new** FileInputStream(args[0])) {  
  
 **while** (inFile.available() > 0) {  
 **int** index = inFile.read();  
 freqTable[index]++;  
 }  
 Element<Node> rootElement = Huffman.*huffman*(freqTable);  
 variableLengthCodeWords = Huffman.*convertToVariableLengthCodeWords*(rootElement);  
 } **catch** (FileNotFoundException f) {  
 f.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 **try** (FileOutputStream outFile = **new** FileOutputStream(args[1]);) {  
 BitOutputStream outBitStream = **new** BitOutputStream(outFile);  
 **for** (**int** i = 0; i < freqTable.**length**; i++) {  
 outBitStream.writeInt(freqTable[i]);  
 }  
 **try** (FileInputStream inFile = **new** FileInputStream(args[0])) {  
 **while** (inFile.available() > 0) {  
 **int** index = inFile.read();  
 String codeWord = variableLengthCodeWords[index];  
 **for** (**int** charIndex = 0; charIndex < codeWord.length(); charIndex++) {  
 **char** charAt = codeWord.charAt(charIndex);  
 **int** bit = Integer.*parseInt*( **""**+charAt);  
 outBitStream.writeBit(bit);  
 }  
 }  
  
 }  
 outBitStream.close();  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileOutputStream;  
**import** java.io.IOException;  
*/\*\*  
 \* Decode implementation  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/*  
**public class** Decode {  
  
 **public static void** main(String[] args) {  
 String compressedFilePath = args[0];  
 String outputFilePath = args[1];  
 **int**[] freqTable = **new int**[256];  
 **try** (FileInputStream inFile = **new** FileInputStream(compressedFilePath)) {  
 BitInputStream bitInputStream = **new** BitInputStream(inFile);  
  
 *//Read the first 256 Bytes, which represent the frequency table.* **int** fileByteLength = 0;  
 **for** (**int** i = 0; i < 256; i++) {  
 freqTable[i] = bitInputStream.readInt();  
 fileByteLength += freqTable[i];  
 }  
 Element<Node> rootElement = Huffman.*huffman*(freqTable);  
 **try**(FileOutputStream outputStream = **new** FileOutputStream(outputFilePath)) {  
 **int** bytesWritten = 0;  
 **while** (bytesWritten <= fileByteLength) {  
 *decodeNextSequence*(rootElement.**data**, bitInputStream, outputStream);  
 bytesWritten++;  
 }  
 }  
 bitInputStream.close();  
  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **private static void** decodeNextSequence(Node node, BitInputStream bitInputStream, FileOutputStream outputStream) **throws** IOException {  
 **if**(node.getRight() == **null** && node.getLeft() == **null**){  
 *//Leaf node reached. Write to output. End of sequence, return out.* outputStream.write(node.getKey());  
 **return**;  
 }  
 **int** bit = bitInputStream.readBit();  
 **if**(bit == 0){  
 *decodeNextSequence*(node.getLeft(),bitInputStream,outputStream);  
 } **else** {  
 *decodeNextSequence*(node.getRight(),bitInputStream,outputStream);  
 }  
  
 }  
}

*/\*\*  
 \* Element implementation  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/*

**public class** Element<T> {  
 **public int key**;  
 **public** T **data**;  
  
 **public** Element(**int** freq, T data) {  
 **this**.**key** = freq;  
 **this**.**data** = data;  
 }  
}

*/\*\*  
 \* Huffman implementation  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/*

**public class** Huffman {  
  
 **private static** StringBuilder sb = **new** StringBuilder();  
  
 **public static** Element<Node> huffman(**int**[] freqTable) {  
 **int** n = freqTable.**length**;  
 PQ queue = **new** PQHeap(n);  
  
 **for** (**int** byteValue = 0; byteValue < freqTable.**length**; byteValue++) {  
 **int** freq = freqTable[byteValue];  
 Element<Node> element = **new** Element<>(freq, **new** Node(**null**, **null**, byteValue)); *//Node is the data of elements. No children present at the beginning.* queue.insert(element);  
 }  
  
 **for** (**int** i = 0; i < n - 1; i++) {  
 Element<Node> left = queue.extractMin();  
 Element<Node> right = queue.extractMin();  
 **int** freqSum = left.**key** + right.**key**;  
 Node subTree = **new** Node(left.**data**, right.**data**, -1);  
 Element<Node> newElement = **new** Element<>(freqSum, subTree);  
 queue.insert(newElement);  
 }  
  
 **return** queue.extractMin(); *//Return the root of the tree* }  
  
 **public static** String[] convertToVariableLengthCodeWords(Element<Node> rootElement) {  
 String[] variableLengthCodeWords = **new** String[256];  
 *//Recursive inorder tree walk, that create variable length code words for each byte (Leaf in the huffman tree).  
 inorderTreeWalk*(variableLengthCodeWords, rootElement.**data**);  
 **return** variableLengthCodeWords;  
 }  
  
 */\*\*  
 \* Recursive inorder tree walk  
 \** ***@param variableLengthCodeWords*** *The array that is filled with variable length code words, where the index is the byte value  
 \** ***@param x*** *a node  
 \*/* **private static void** inorderTreeWalk(String[] variableLengthCodeWords, Node x) {  
 **if**(x != **null**){  
 **if**(x.getLeft() == **null** && x.getRight() == **null**){  
 *//Leaf node. Insert code word* variableLengthCodeWords[x.getKey()] = sb.toString();  
 }  
 *//Append 0 if we go left in the tree* sb.append(**"0"**);  
 *inorderTreeWalk*(variableLengthCodeWords,x.getLeft());  
 *//Clean when we go back up the tree* sb.deleteCharAt(sb.length() - 1);  
  
 *//Append 1 if we go right in the tree* sb.append(**"1"**);  
 *inorderTreeWalk*(variableLengthCodeWords,x.getRight());  
 *//Clean when we go back up the tree. The StringBuilder is empty at the root node.* sb.deleteCharAt(sb.length() - 1);  
 }  
 }  
}

*/\*\*  
 \* Node implementation  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/***public class** Node {  
 **private** Node **left**;  
 **private** Node **right**;  
 **private int** key;  
  
 */\*\*  
 \* Creates a new node with left child, right child and the given key value  
 \** ***@param left*** *\** ***@param right*** *\** ***@param key*** *\*/* **public** Node(Node left, Node right, **int** key) {  
 **this**.**left** = left;  
 **this**.**right** = right;  
 **this**.key = key;  
 }  
  
 */\*\*  
 \* Creates a new node with the given key value, and null values for children  
 \** ***@param key*** *\*/* **public** Node(**int** key) {  
 **this**(**null**,**null**,key);  
 }  
  
 */\*\*  
 \* Set the left child  
 \** ***@param left*** *\*/* **public void** setLeft(Node left) {  
 **this**.**left** = left;  
 }  
  
 */\*\*  
 \* Set the right child  
 \** ***@param right*** *\*/* **public void** setRight(Node right) {  
 **this**.**right** = right;  
 }  
  
 */\*\*  
 \* get the left child  
 \** ***@return*** *\*/* **public** Node getLeft() {  
 **return left**;  
 }  
  
 */\*\*  
 \* get the right child  
 \** ***@return*** *\*/* **public** Node getRight() {  
 **return right**;  
 }  
  
 */\*\*  
 \* Gets the key of the node  
 \** ***@return*** *\*/* **public int** getKey() {  
 **return** key;  
 }  
}

*/\*\*  
 \* PQ Interface  
 \*  
 \* Lavet af Christian Skafte Beck Clausen (chcla15) og Daniel Johansen (dajoh16)  
 \*/*

**public interface** PQ {  
 **public** Element extractMin();  
 **public void** insert(Element e);  
}

*/\*\*  
 \* Priority Queue Heap implementation.  
 \* Lavet af Christian Skafte Beck Clausen Chcla15 og Daniel Johansen Dajoh16  
 \*/***public class** PQHeap **implements** PQ {  
  
 **private** Element[] **elements**;  
 **private int heapSize**;  
  
 **public** PQHeap(**int** maxElements) {  
 **elements** = **new** Element[maxElements];  
 **heapSize** = -1;  
  
 }  
  
 */\*\*  
 \* Extracts the root element which is the min element  
 \*  
 \** ***@return*** *\*/* @Override  
 **public** Element extractMin() {  
 Element min = **elements**[0];  
 **elements**[0] = **elements**[**heapSize**];  
 **heapSize**--;  
 minHeapify(0);  
 **return** min;  
 }  
  
 */\*\*  
 \* Inserts an element into the priority queue heap  
 \*  
 \** ***@param element*** *\*/* @Override  
 **public void** insert(Element element) {  
 **heapSize**++;  
 **int** i = **heapSize**;  
  
 **elements**[i] = element;  
  
 **if**(parent(i) < 0){  
 **return**;  
 }  
  
 **if** (**elements**[parent(i)] != **null** && **elements**[i] != **null**) {  
 **while** (i > 0 && **elements**[parent(i)].**key** > **elements**[i].**key**) {  
 Element temp = **elements**[parent(i)];  
 **elements**[parent(i)] = **elements**[i];  
 **elements**[i] = temp;  
 i = parent(i);  
 }  
 }  
 }  
  
 */\*\*  
 \* Rebuilds the heap based on a min heap structure. The call is recursive.  
 \*  
 \** ***@param i*** *\*/* **private void** minHeapify(**int** i) {  
 **int** left = left(i);  
 **int** right = right(i);  
 **int** smallest;  
 **if** (left <= **heapSize** && **elements**[left].**key** < **elements**[i].**key**) {  
 smallest = left;  
 } **else** {  
 smallest = i;  
 }  
 **if** (right <= **heapSize** && **elements**[right].**key** < **elements**[smallest].**key**) {  
 smallest = right;  
 }  
 **if** (smallest != i) {  
 Element temp = **elements**[smallest];  
 **elements**[smallest] = **elements**[i];  
 **elements**[i] = temp;  
 minHeapify(smallest);  
 }  
 }  
  
 */\*\*  
 \* returns the left node index  
 \** ***@param i*** *\** ***@return*** *\*/* **private int** left(**int** i) {  
 **return** 2 \* i + 1;  
 }  
  
 */\*\*  
 \* Returns the right node index  
 \** ***@param i*** *\** ***@return*** *\*/* **private int** right(**int** i) {  
 **return** 2 \* i + 2;  
 }  
  
 */\*\*  
 \*  
 \* Returns -1 if at root.  
 \*  
 \** ***@param i*** *\** ***@return*** *\*/* **private int** parent(**int** i) {  
 **int** parent = (**int**) (Math.*ceil*(i / 2D) - 1);  
 **return** parent;  
 }  
}