public static void main(String[] args) {

Huffman.compress("src/com/company/Main.java",filePath+"fileOut.txt");

Huffman.decompress(filePath+"fileOut.txt",filePath+"fileDec.txt"); }

# Huffman CLASS

import huffman\_toolkit.\*;

import java.util.PriorityQueue;

public class Huffman {

private static final int CHARS = InputTextFile.CHARS;

public static int[] charFreq(String src){

int[] freq = new int[CHARS];

for (int c=0;c<CHARS;c++) freq[c] = 0;

InputTextFile inputFile = new InputTextFile(src);

while(inputFile.textAvailable()){

char c = inputFile.readChar();

freq[c]++; }

inputFile.close();

return freq; }

\* @param charFreq ARRAY DI FREQUENZE PER OGNI CARATTERE

\* @return Node huffmanTree

public static Node huffmanTree(int[] charFreq){

// CODA VUOTA CHE CONTERRà DEI NODI

PriorityQueue<Node> coda = new PriorityQueue<Node>();

for (int c =0;c<CHARS;c++){

if (charFreq[c]>0){

Node n = new Node((char)c,charFreq[c]);

coda.add(n); }}

while (coda.size()>1){

Node l = coda.poll(); // PESCA IL NODO A PRIORITà MAGGIORE E LO RIMUOVE DALLA CODA

Node r = coda.poll();

Node tree = new Node(l,r); // COSTRUZIONE ALBERO

coda.add(tree); } // AGGIUNGE ALLA CODA UN NODO

return coda.peek(); // ALBERO DI HUFFMAN

\* @param root albero di huffman

\* @return tabella con i codici di huffamn generati in base alla priorità

public static String[] huffmanCodes(Node root){

String[] table = new String[CHARS];

// CODICE DI HUFFMAN INIZIALMENTE VUOTO

buildTable(root,"",table);

return table; }

\* @param root NODO

\* @param code CODICE DI HUFFMAN DEGLI SPOSTAMENTI

\* @param table charCodes

private static void buildTable(Node root,String code, String[] table) {

if (root.isLeaf()){

char c = root.symbol();

table[c] = code;

} else {

// 0 = SPOSTAMENTO SX 1 = SPOSTAMENTO DX

buildTable(root.left(),code+"0",table);

buildTable(root.right(),code+"1",table); } }

\* @param src FILE SORGENTE

\* @param dst FILE DESTINAZIONE

public static void compress(String src, String dst){

int[] charFreq = charFreq(src);

Node tree = huffmanTree(charFreq);

String[] codes = huffmanCodes(tree);

InputTextFile inFile = new InputTextFile(src);

OutputTextFile outFile = new OutputTextFile(dst);

outFile.writeTextLine(""+tree.weight());

outFile.writeTextLine(flatTree(tree));

while(inFile.textAvailable()){

char c = inFile.readChar();

outFile.writeCode(codes[c]); }

inFile.close();

outFile.close(); }

\* @param src FILE SORGENTE COMPRESSO

\* @param dst FILE DESTINAZIONE DECOMPRESSO =? FILE SORGENTE DI COMPRESS()

public static void decompress(String src, String dst){

InputTextFile inFile = new InputTextFile(src);

OutputTextFile outFile = new OutputTextFile(dst);

// NUMERO TOTALE DI CARATTERI CHE DEVONO COMPARIRE NEL FILE OUT

int numChar = Integer.parseInt(inFile.readTextLine());

Node tree = restoreTree(inFile);

String d = inFile.readTextLine();

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

Node node = tree;

do {

int b = inFile.readBit();

node = (b == 0)? node.left() : node.right();

} while(!node.isLeaf());

outFile.writeChar( node.symbol()); }

inFile.close();

outFile.close(); }

\* @param n ALBERO

\* @return CONFIGURAZIONE STRINGA DELL'ALBERO

public static String flatTree(Node n){

if (n.isLeaf()){

char c = n.symbol();

if (c=='@' || c == '\\') return "\\"+c;

else return ""+c;

} else return "@"+flatTree(n.left())+flatTree(n.right());

}

private static Node restoreTree(InputTextFile in){

char c = in.readChar();

if (c == '@'){

Node l = restoreTree(in);

Node r = restoreTree(in);

return new Node(l,r);

} else {

if (c == '\\') c = in.readChar();

return new Node(c,0); }}}

# NODE CLASS

public class Node implements Comparable<Node> {

private final char sym; // carattere testuale

private final int wgt; // occorrenze

private final Node lft; // figlio sx

private final Node rgt; // figlio dx

public Node(char c, int w){

sym = c;

wgt = w;

lft = null;

rgt = null; }

public Node(Node l, Node r){

sym = (char) 0;

wgt = l.weight() + r.weight();

lft = l;

rgt = r; }

@Override // sostituisce il metodo compareTo della classe Comparable

public int compareTo(Node n) {

if (weight() < n.weight()) return -1;

else if (weight() == n.weight()) return 0;

else return +1; }

public boolean isLeaf(){return (lft == null); }

public char symbol(){ return sym; }

public int weight(){return wgt; }

public Node left(){return lft; }

public Node right(){return rgt;

}