Huffman Εργασία 2020

ΟΜΑΔΑ 1: Αντωνόπουλος Διογένης

Στεφανίδου Άρτεμις

Χύσκαϊ Βασίλης

Μέρος 1ο

Η παράγραφος αυτή αναφέρεται στον τρόπο σκέψης και υλοποίησης του πρώτου μέρους της εργασίας.

1) Αρχικά,δημιουργήσαμε έναν πίνακα 3ων θέσεων για να μπορέσουμε να αποθηκεύσουμε τα τρία διαφορετικά url που δόθηκαν από την εκφώνηση.
```
URL[] url = new URL[3];
```
2) Στη συνέχεια δημιουργήσαμε τα αντικείμενα για τα url καθώς και τρεις BufferedReader για να μπορέσουμε να τα διαβάσουμε και να καταγράψουμε το πόσες φορές εμφανίζεται το κάθε γράμμα(από τα πρώτα 128 του ASCII) και στα τρία url μαζί.
```
url[0] = new URL ("https://www.gutenberg.org/files/1342/1342-0.txt");
url[1] = new URL ("https://www.gutenberg.org/files/11/11-0.txt");
url[2] = new URL ("https://www.gutenberg.org/files/2701/2701-0.txt");

BufferedReader[] reader = new BufferedReader[3];
```
3) Για την διαδικασία αυτή ορίσαμε:

   - Έναν πίνακα 128 θέσεων ,που η κάθε θέση αντιστοιχεί στον counter εμφάνισης του κάθε γράμματος.
```
int[] chars = new int[128];
```
   - Μία μεταβλητή που κρατάει την int τιμή του γράμματος που επέστρεψε το read().
```
int charValue = 0;
```
   - Μία for που θα εκτελεστεί 3 φορές με το σκεπτικό ότι η ίδια διαδικασία θα γινεί και για τα τρία url.
   - Ένα while και μία read() για να διαβάσουμε χαρακτήρα-χαρακτήρα και τα τρία url.

```
for (int i = 0; i < 3; i++) {

    reader[i] = new BufferedReader(new InputStreamReader(url[i].openStream()));

    while ((charValue = reader[i].read()) != -1) {
        if (charValue < 128) {
            chars[charValue]++;
        }
    }
```

- Ένα if για να πάρουμε μόνο όσα είναι μεταξύ 0 και 127 και να αυξήσουμε τον αντίστοιχο μετρητή του γράμματος.

4) Στη συνέχεια κλείσαμε τα αρχεία-url.

```
reader[i].close();
```

5) Δημιουργήσαμε έναν BufferedWriter για να τυπώσουμε τις θέσεις του πίνακα με τις αντίστοιχες συχνότητες για κάθε γράμμα-ειδικό χαρακτήρα ,μέσω μιας for ,στο αρχείο “frequencies.dat” όπως μας ζητήθηκε από την εκφώνηση .(μικρή αλλαγή στον κώδικα του 1ου μέρους για να μας διευκολύνει στο 2ο μέρος)

```
//Create a file and print the results there
try (BufferedWriter outputStream = new BufferedWriter(new FileWriter("frequencies.dat"))) {
    for (int i = 0; i < 128; i++) {
        outputStream.write(i + " -> " + chars[i] + "\n");
        outputStream.flush();
    }
}
```

```
//Create a file and print the results there
try (BufferedWriter outputStream = new BufferedWriter(new FileWriter("frequencies.dat"))) {
    for (int i = 0; i < 128; i++) {
        outputStream.write( chars[i] + "\n");
        outputStream.flush();
    }
}
```

Παραδείγματα Εκτέλεσης του Κώδικα μας:

Ακολουθεί παράδειγμα από το file frequencies.dat ,που δημιουργείτε στο φάκελο του project (εκεί που είναι και το pom.xml):

2621

2098

1450

1876

1106

1082

Μέρος 2ο

Η παράγραφος αυτή αναφέρεται στον τρόπο σκέψης και υλοποίησης του δεύτερου μέρους της εργασίας.

1) Ορίσαμε μία κλάση για τους κόμβους του δέντρου ,όπως μας ζητήθηκε στην εκφώνηση,με το όνομα Node η οποία περιέχει:

```
public class Node implements Comparable<Node>, Serializable {
```

- Ένα χαρακτήρα (σε περίπτωση που ο κόμβος είναι φύλλο ο, χαρακτήρας είναι ένα γράμμα από τα πρώτα 128 του ASCII, ενώ αν δεν είναι τότε ο χαρακτήρας ισούτε με το '\0').
- Μια συχνότητα (που θα είναι ή η συχνότητα του γράμματος ,αν είναι φύλλο,ή το άθροισμα των συχνοτήτων των δύο παιδιών του node)
- Δύο παιδιά,ένα δεξί και ένα αριστερό (που θα είναι null αν είναι leaf)

Αυτά τα χαρακτηριστικά του Node τα βάλαμε ***private*** γιατί δε θέλουμε να είναι προσβάσιμα εκτός κλάσης και ***final*** γιατί δεν αλλάζουν ,παρά μόνο κατά την αρχικοποίηση τους που γίνεται μέσω του ***Constructor***.

```
//Node Characteristics
private final char character;
private final int frequency;
private final Node left;
private final Node right;

//Constructor
public Node(char character, int frequency, Node left, Node right) {
    this.character = character;
    this.frequency = frequency;
    this.left = left;
    this.right = right;
}
```

- Την μέθοδο compareTo επειδή κάναμε ***implement*** το ***Interface Comparable<T>*** γιατί θέλαμε να έχουμε τη δυνατότητα σύγκρισης μεταξύ των αντικειμένων που θα δημιουργήσουμε.Στην ουσία έπρεπε να υλοποιήσουμε και να κάνουμε override την μέθοδο αυτή η οποία μας γυρνάει τη διαφορά δύο συχνοτήτων δύο διαφορετικών κόμβων.Δηλαδή συγκρίνει τις δύο συχνότητες και γυρνάει <0 αν η συχνότητα του n είναι μεγαλύτερη ,>0 αν είναι μικρότερη και 0 αν είναι ίσες.

```
//Compare the frequencies from two nodes
@Override
public int compareTo(Node n) {
    return this.frequency - n.frequency;
}
```

- Μία μέθοδο που ελέγχει αν ο κόμβος αυτός που της δώσαμε ως παράμετρο είναι φύλλο ή όχι.Για να είναι φύλλο ένας κόμβος πρέπει και τα δύο του παιδιά να είναι null.Τα φύλλα είναι αυτά που περιέχουν ένα χαρακτήρα από τους 128 του ASCII.Προς το παρόν όμως τη μέθοδο αυτή δε τη χρειαστήκαμε κάπου αλλά θεωρήσαμε πως θα μας είναι χρήσιμη στη συνέχεια της εργασίας και ότι είναι καλό να την έχουμε για κάθε καινούριο αντικείμενο Node που θα δημιουργούμε.

```
//Check if node is leaf:Has character
public boolean isLeaf() {
    return right == null && left == null;
}
```

- Τους getters για τα χαρακτηριστικά του Node που χρειαστήκαμε να χρησιμοποίησουμε τις τιμές τους και σε άλλη κλάση.

```
public char getCharacter() {
    return character;
}

public int getFrequency() {
    return frequency;
}

public Node getLeftChild() {
    return left;
}

public Node getRightChild() {
    return right;
}
```

2) Ορίσαμε επίσης μία κλάση για το δέντρο Huffman όπως μας ζητήθηκε στην οποία υλοποιούμε όλη τη διαδικασία δημιουργίας ενός δέντρου Huffman.Η κλάση περιέχει:

```
public class Huffman
```

- Μία μέθοδο makeTree για να μπορέσουμε να υλοποιήσουμε τον αλγόριθμο του Huffman που να παράγει ένα δέντρο χρησιμοποιώντας ως είσοδο τον πίνακα συχνοτήτων.

```
public Node makeTree(int[] array) {
```

- ***Υλοποίηση μεθόδου :***

  - ✔ Δημιουργήσαμε έναν πίνακα από Node με 128 θέσεις(μία για τον κάθε χαρακτήρα)έτσι ώστε να τα αρχικοποιήσουμε εύκολα με μία for loop και να μπορέσουμε να φτιάξουμε ένα δέντρο (ενός κόμβου) για τον κάθε ASCII χαρακτήρα βάζοντας κάθε φορά , πέρα από τον χαρακτήρα,την αντίστοιχη συχνότητα από τον πίνακα που δεχτήκαμε ως παράμετρο ,και τα παιδιά του να είναι null.

```
//Create an array of nodes.Each node represents a character
Node[] treeNodes = new Node[128];
for(char i = 0; i<array.length; i++) {
    treeNodes[i] = new Node(i,array[i],null,null);
}
```

- ✔ Χρησιμοποιήσαμε την MinHeap του εργαστηρίου για να ταξινομήσουμε τα 128 Nodes και να έχουμε στην κορυφή του δέντρου το Node με τη μικρότερη συχνότητα.

```
//Create the MinHeap for our Nodes using the right constructor to use heapify
MinHeap<Node> h =new ArrayMinHeap<>(treeNodes);
```

- ✔ Όσο το size της MinHeap είναι μεγαλύτερο από 1(αντίστοιχο στις διαφάνειες του μαθήματος:Μέχρι να μείνει ένα δέντρο)καλούμε την deleteMin η οποία διαγράφει και γυρνάει το πρώτο Node από τη MinHeap άρα και αυτό με τη μικρότερη συχνότητα και το αποθηκεύουμε σε ένα προσωρινό Node .Αφού η deleteMin περιέχει στον κώδικα της την fixDown πάει να πει ότι έγιναν οι απαραίτητοι έλεγχοι και ενέργιες έτσι ώστε πρώτο στοιχείο στην h να ξαναείναι το μικρότερο από όσα έμειναν.Άρα ξανακαλούμε την deleteMin και γίνεται η ίδια διαδικασία που αναφέρθηκε και παραπάνω. Στη συνέχεια τα συγχωνεύσαμε σε ένα νέο δυαδικό δέντρο που έχει αριστερό παιδί ρίζας το 1ο και δεξί παιδί ρίζας το 2ο και δώσαμε συχνότητα ίση με το άθροισμα των συχνοτήτων των παιδιών του και χαρακτήρα τον κενό '\0'. Τέλος,προσθέσαμε το νέο node στην MinHeap για να μπορέσουμε και πάλι να βρούμε τα δύο μικρότερα και να συνεχίσουμε την διαδικασία μέχρι το size να γίνει ένα.

```
//Create the Huffman Tree with the help of MinHeap
while(h.size() > 1) {
    Node leftChild = h.deleteMin();
    Node rightChild = h.deleteMin();
    Node parent = new Node('\0',leftChild.getFrequency() + rightChild.getFrequency(),leftChild,rightChild);
    h.insert(parent);
}
```

- ✔ Στο τέλος,καλούμε την deleteMin για να πάρουμε το root του δέντρου Huffman που δημιουργήσαμε.

Παρατήρηση:Στην ουσία το size μειώνεται κατά ένα κάθε φορά αφού διαγράφουμε 2 και προσθέτουμε ένα node στην MinHeap.Ο πατέρας ,το node δηλαδή που έχει δύο παιδιά και δώσαμε συχνότητα ίση με το άθροισμα των συχνοτήτων των παιδιών του ,ναι μεν μπαίνει μόνο αυτός με τη βοήθεια του insert στην MinHeap αλλά δεν παύει να δείχνει και στα δύο παιδιά που είχε πριν μπει.

3) Τέλος, χρησιμοποιήσαμε την κλάση App από το πρώτο μέρος της εργασίας:
   - Δημιουργήσαμε σε αυτή ένα αντικείμενο Scanner στο οποίο δώσαμε σαν όρισμα το αρχείο frequencies.dat για να μπορέσουμε στη συνέχεια να το διάβασουμε και να πάρουμε την απαραίτητη πληροφορία που χρειαζόμαστε για το δέντρο Huffman.

```
Scanner scanner = new Scanner(new File("frequencies.dat"));
```

   - Για να το πετύχουμε αυτό ορίσαμε έναν πίνακα με 128 θέσεις για να αποθηκεύσουμε τις συχνότητες των 128 χαρακτήρων ASCII που θα πάρουμε από το αρχείο.
   - Χρησιμοποιήσαμε μία for και το nextInt() για να διαβάσουμε κάθε φορά έναν ακέραιο από το αρχείο ,δηλαδή μία συχνότητα και να την αποθηκεύουμε στην αντίστοιχη θέση του πίνακα array.

```
int[] array = new int[128];
for(int i= 0; i < 128; i++) {
    array[i] = scanner.nextInt();
}
```

- Δημιουργήσαμε ένα αντικείμενο Huffman για να μπορέσουμε στη συνέχεια να χρησιμοποίησουμε την μέθοδο που έχει αυτή η κλάση.

```
//Call class HuffmanTree to create an object for that by the dafault constructor
Huffman tree = new Huffman();
```

- Αποθηκεύσαμε το αντικείμενο του δέντρου στο αρχείο tree.dat χρησιμοποιώντας Java Object Serialization(γι αυτό το λόγο έχουμε κάνει και 'implements Serializable' στην κλάση Node).Η makeTree φτιάχνει ένα δέντρο Huffman και μας γυρνάει το root του δέντρου αυτού και στην ουσία όλο το δέντρο αφού έχει pointers που δείχνουν σε αυτό.Έτσι, καλώντας την με όρισμα τον πίνακα συνχοτήτων των χαρακτήρων,μας δημιουργεί το Huffman με αυτές τις συχνότητες και μας γυρνάει το root του.Το node δηλαδή που χρειαζόμαστε για να αποθηκεύσουμε ολόκληρο το δέντρο στο tree.dat.

```
//Create the huffman tree by calling makeTree and write it in tree.dat
try (ObjectOutputStream objectOut = new ObjectOutputStream(new FileOutputStream("tree.dat"))) {
    objectOut.writeObject(tree.makeTree(array));
}
```

<u>Δύο λόγια για την MinHeap αφού την έχουμε υλοποιήσει και στο εργαστήριο:</u>

- ✔ Η insert που την χρησιμοποίησαμε και στην εργασία αυξάνει κατά ένα το size του πίνακα της MinHeap και αποθηκεύει εκεί το Node(στην προκειμένη περίπτωση)που προσθέσαμε. Στη συνέχεια,καλεί την fixup η οποία λέει όσο k > 1 δηλαδή όσο το αντικείμενο που έδωσα δεν είναι η ρίζα του δέντρου και όσο το αντικείμενο είναι πιο μεγάλο από τον πατέρα του τότε κάνε swap τον πατέρα με το παιδί του και πήγαινε στην θέση που ήταν πριν ο πατέρας για να ξαναγίνει το while.Με αυτόν τον τρόπο εξασφαλίζουμε ότι διαρκώς το root θα είναι και το μικρότερο αλλά και ότι η MinHeap είναι διατεταγμένη σωστά κάθε φορά που προσθέτουμε ένα αντικείμενο σε αυτήν.
- ✔ Η deleteMin αποθηκεύει σε μια προσωρινή μεταβλητή το πρώτο στοιχείο του πίνακα της MinHeap(άρα και το ελάχιστο)και στη θέση του τοποθετεί το τελευταίο στοιχείο.Στη συνέχεια,διαγράφει την τελευταία θέση του πίνακα και καλεί τη fixdown.Η fixdown με τη σειρά της κάνει τους απαραίτητους ελέγχους για να βρει το μικρότερο από τα δύο παιδιά του Node που το βάλαμε στο root και μετά συγκρίνει το min των παιδιών αυτών με τον πατέρα.Άμα ο πατέρας είναι μεγαλύτερος τότε κάνει swap με το παιδί αλλιώς παραμένει στη θέση που ήταν.Τέλος,η deleteMin γυρνάει την προσωρινή μεταβλητή.
- ✔ ***<u>public ArrayMinHeap(E[] array):</u>***Τη χρησιμοποίησαμε όταν θελήσαμε να κάνουμε μία MinHeap με τα 128 Nodes.Δημιουργήσαμε έναν πίνακα από τα Nodes αυτά για να μπορέσουμε να τον βάλουμε σαν όρισμα στον Constructor της ArrayMinHeap και έτσι να δημιουργηθεί αυτόματα η MinHeap χωρίς την χρήση της μεθόδου insert.Αυτό το κάναμε γιατί ακολουθεί τον κανόνα του big-Oh notation ,που μάθαμε στη θεωρία,δηλαδή προσπαθούμε να έχουμε όσο μικρότερο order γίνεται γιατί τόσο πιο γρήγορος θα είναι ο κώδικας μας.

Μέρος 3ο

Η παράγραφος αυτή αναφέρεται στον τρόπο σκέψης και υλοποίησης του τρίτου μέρους της εργασίας.

1. Αρχικά όπως λέει και η εκφώνηση θεωρήσαμε πως θα χρειαστούμε μία στοίβα.Έτσι, χρησιμοποιήσαμε την ***Deque*** και της ορίσαμε να δέχετε ***Characters*** για να μας διευκολύνει στην υλοποίηση του τρίτου μέρους.
2. Καταλήξαμε στο ότι είναι πιο σωστό να χρησιμοποιήσουμε τις λέξεις ***RIGHT*** (συμβολίζει ότι πάμε δεξιά στο δέντρο και με βάση τη δική μας υλοποίηση το δεξιά σημαίνει ότι η ακμή αυτή θα έχει το 1)και ***LEFT*** (συμβολίζει ότι πάμε αριστερά στο δέντρο και με βάση τη δική μας υλοποίηση αυτό σημαίνει ότι η ακμή αυτή θα έχει το 0)από ότι 1 και 0 αντίστοιχα.Το σκεφτήκαμε αυτό γιατί άμα χρειαζόταν να αλλάξουμε την υπόθεση που κάναμε (Δεξιά→1,Αριστερά→0)δε θα χρειαζόταν να αλλάζαμε ένα ένα τα 0 και 1 που είχζαμε βάλει σο πρόγραμμα αλλά θα αρκούσε μόνο να αλλάζαμε τα final .

```
public static final Character RIGHT = '1';
public static final Character LEFT = '0';
```

3. Δημιουργήσαμε την μέθοδο createCode ,η οποία παίρνει ως παράμετρο ένα node, για να μπορέσουμε να διασχίσουμε το δέντρο και να κωδικοποιήσουμε κάθε χαρακτήρα με μια συμβολοσειρά από 0 και 1.Τις κωδικοποιήσεις τις αποθηκεύουμε σε έναν πίνακα για να μπορέσουμε μετά να τις επεξεργαστούμε στο App.Έτσι:

- Για αρχή ελέγξαμε άμα το node που είμαστε είναι φύλλο δηλαδή άμα περιέχει κάποιον χαρακτήρα από τους 128 του ASCII και άμα τα παιδιά του είναι null.Άμα δεν είναι τότε βάζουμε στην στοίβα ***dequeCode*** το 0 άρα πάμε μία αριστερά (γιατί θέλουμε να κάνουμε inorder διάσχιση)και ξανακαλούμε την printCode αλλά με το αριστερό παιδί του προηγούμενου node(μέθοδος της αναδρομής).

```
//check if the node is leaf->has one of 128 ASCII characters
if (!curRoot.isLeaf()) {

    dequeCode.push(LEFT);
    createCode(curRoot.getLeftChild());
```

- Επαναλαμβάνοντας αυτή τη διαδικασία κάποια στιγμή θα φτάσουμε στο πιο αριστερό node του δέντρου το οποίο θα είναι και φύλλο.Άρα θα μπει στο ***else*** και με τη βοήθεια του ***for*** για

τα αντικείμενα που έχουν ***iterator*** θα μεταφέρουμε έναν έναν τους χαρακτήρες σε έναν ***StringBuffer(stBuffer)***μέσω της μεθόδου ***append*** **,**η οποία προσθέτει κάθε φορά στο τέλος του buffer τον επόμενο χαρακτήρα από την ***deque.***

```
else {

    //the for of the objects that have iterator
    for (Character x : dequeCode) {
        //add in buffer the next Character  of representation
        stBuffer.append(x);
    }
```

- Στη συνέχεια,αποθηκεύσαμε σε έναν πίνακα (***private String[] array = new String[128])*** τις κωδικοποιήσεις με βάση όμως τον χαρακτήρα τους έτσι ώστε στην main να μπορούμε να τα επεξεργαστούμε με τη σειρά από το 0 μέχρι και το 127.Για να επιτύχουμε μία σωστή αναπαράσταση 0 και 1 του κάθε χαρακτήρα χρησιμοποιήσαμε την μέθοδο ***reverse()*** για να μας τα αποθηκεύσει με την ανάποδη σειρά από ότι μας τα εμφάνιζε το for.Λόγω του ότι το for θα εκτυπώσει πρώτο αυτό που μπήκε τελευταίο στην ***dequeCode*** ενώ εμείς θέλουμε το πρώτο να εκτυπωθεί πρώτο και το τελευταίο τελευταίο.Επίσης ,χρησιμοποιήσαμε την μέθοδο toString() για να αναπαραστήσουμε τα περιεχόμενα του Buffer σε String μορφή. Επιπλέον,κάναμε ***delete*** το περιεχόμενο του ***Buffer*** έτσι ώστε να μπορέσουμε να ξαναγράψουμε στη συνέχεια την επόμενη κωδικοποίηση.

```
//save the character representation in the corresponding position of the array with the correct way
array[curRoot.getCharacter()] = stBuffer.reverse().toString();

//delete the contents of the buffer so reused it in the next character representation
stBuffer.delete(0, stBuffer.length());
```

- Για να συνεχίσουμε σωστά την αναπαράσταση έπρεπε να επισκεφτούμε και το δεξί παιδί του τελευταίου node που μπήκε στο if πριν συναντήσουμε ένα node φύλλο.Πριν από αυτό όμως θα πρέπει να αφαιρέσουμε το τελευταίο 0 που μπήκε στην ***dequeCode*** δηλαδή να αναιρέσουμε μία αριστερή ακμή και να προσθέσουμε μία δεξιά.Γι αυτό το λόγο κάναμε ένα pop και έτσι όταν η αναδρομή γυρίσει στο τελευταίο node που μπήκε στο if θα κάνουμε ένα ***push(RIGHT)*** για να πάμε σε αυτό το μονοπάτη τώρα και συνεχιστεί η διαδικασία.

```
//check if deque has characters to pop
if (!dequeCode.isEmpty()) {

    dequeCode.pop();

}
```

```
dequeCode.push(RIGHT);
createCode(curRoot.getRightChild());
```

- **ΓΕΝΙΚΗ ΣΚΕΨΗ:**

Ο τρόπος που θέλαμε να διασχίσουμε το δέντρο και τον υλοποιήσαμε και σε κώδικα ήταν:

1. Να πηγαίνουμε με τη διάσχιση του δέντρου τέρμα αριστερά ,άρα να καταλήγουμε σε φύλλο και να το εκτυπώνουμε.
2. Μετά ένα πάνω ,άρα στην ουσία μία pop και μετά ένα δεξιά.
3. Άμα είναι φύλλο το εκτυπώνουμε άμα δεν είναι κάνουμε πάλι όλο αριστερά.
4. Επαναλαμβάνουμε αυτή τη διαδικασία μέχρι και το τελευταίο φύλλο.Μέχρι δηλαδή να τελειώσει η αναδρομή ,γιατί και η αναδρομή θα μπορούσαμε να πούμε πως είναι μία στοίβα που κάθε φορα θυμάται τα node που χρωστάει.

Για την κλάση App:

- Δημιουργήσαμε ένα ***ObjectInputStream*** για να μπορέσουμε να διαβάσουμε από το αρχείο ***tree.dat*** και αποθηκεύσαμε στο root (κάνοντας cast σε node)ό,τι διαβάσαμε από αυτό,κοινώς την ρίζα του δέντρου μας.

```
//create a stream to read file tree.dat
ObjectInputStream input = new ObjectInputStream(new BufferedInputStream(new FileInputStream("tree.dat")));
Node root = (Node) input.readObject();
```

- Δημιουργήσαμε ένα ***BufferedWriter*** για να μπορέσουμε να γράψουμε στο αρχείο ***codes.dat*** και στη συνέχεια καλέσαμε την μέθοδο ***createCode*** με παράμετρο την ρίζα του δέντρου που διαβάσαμε από το αρχείο ***tree.dat*** και αποθηκεύοντας τον πίνακα που γυρίζει σε ένα αντίστοιχο ***String[] arrayCode.***

```
//Create a BufferWriter to write in file codes.dat
try ( BufferedWriter output = new BufferedWriter(new FileWriter("codes.dat"))) {

    //save the array that return from class Huffman method printCode
    String[] arrayCode = tree.createCode(root);
```

- Τέλος,χρησιμοποιώντας τις μεθόδους ***write*** και ***flush*** καθώς και μία for για τις 128 αναπαραστάσεις των χαρακτήρων ,εκτυπώσαμε-γράψαμε στο αρχείο ***codes.dat*** τις κωδικοποιήσεις όλων των χαρακτήρων που τις δημιούργησε η μέθοδος ***createCode.***

Παραδείγματα Εκτέλεσης του Κώδικα μας:

Ακολουθεί παράδειγμα από το file codes.dat :

```
0->1101100011001100000111101111110
1->1101100011001100000111101111111
2->110110001100110000011111111100001010
3->11011000110011000000111111010
4->1101100011001100000011111111000111
5->110110001100110000001111111100001011
6->1101100011001100000011111111100001
7->11011000110011000000111100
8->1101100011001100000011111110111
9->1101100011001100000011111111100010
10->110001
```

Μέρος 4ο

Η παράγραφος αυτή αναφέρεται στον τρόπο σκέψης και υλοποίησης του τέταρτου μέρους της εργασίας.

1. Πρώτα από όλα ελέγχουμε άμα ο χρήστης μας έδωσε το σωστό πλήθος ορισμάτων-αρχείων.Αν όχι του εκτυπώνουμε το αντίστοιχο μήνυμα λάθους και κάνουμε exit από το πρόγραμμα.

```
//Check for correct input from the terminal
if (args.length != 2) {
    System.out.println("Usage: program filename filename\nAnd the first file must already exist");
    System.exit(-1);
    }
```

2. Στη συνέχεια,αφού κάναμε μία μικρή αλλαγή στον τρόπο εκτύπωσης του τρίτου μέρους για να εκτυπώνει στο αρχείο μόνο τις αναπαραστάσεις των χαρακτήρων με 0 και 1 ,διαβάσαμε γραμμή-γραμμή(σε κάθε γραμμή υπάρχει μόνο μία κωδικοποίηση) το αρχείο και τις αποθηκεύσαμε στην αντίστοιχη θέση του πίνακα **code**.

```
//Read the Huffman coding from the file codes.dat
String[] code = new String[128];
try (Scanner myScanner = new Scanner(new File("codes.dat"))) {
    int i = 0;
    while (myScanner.hasNextLine()) {
        code[i++] = myScanner.nextLine();
    }
}
```

3. Χρησιμοποιήσαμε έναν ***BufferedReader myReader*** για να μπορέσουμε να διαβάσουμε χαρακτήρα-χαρακτήρα το αρχείο που θα μας δώσει ο χρήστης και να το αποθηκεύσουμε σε έναν ***StringBuffer encode*** για να μας βοηθήσει στη συνέχεια στην επεξεργασία bit προς bit.

```
//Store in encode,char by char,the coresponding huffman encoding
try{
    File file = new File(args[0]);

    StringBuffer encode = new StringBuffer();
    try (BufferedReader myReader = new BufferedReader(new FileReader(file))) {
        int ch;
        while ((ch = myReader.read()) != -1) {
            encode.append(code[ch]);
        }

    }
```

ΓΕΝΙΚΗ ΣΚΕΨΗ:

Για να μπορέσουμε να χειριστούμε σωστά τα bytes σκεφτήκαμε ότι θα πρέπει να έχουμε κάθε φορά 8 bits δηλαδή ένα byte και τότε να το εκτυπώνουμε στο αρχείο που θα μας το έχει δώσει κι αυτό ο χρήστης.Έτσι όμως,θα πρέπει να έχουμε αποθηκεύσει πολλαπλάσια του 8 στον ***StringBuffer*** αυτό όμως δε μπορούμε να το ξέρουμε και επίσης δε γίνεται να είναι συνέχεια η κωδικοποίηση του αρχείου πολλαπλάσια του 8.Έτσι ο υπολογιστής για να ολοκληρώσει τα 8 bits προσθέτει στο τέλος τόσα 0 όσα και λείπουν.Για να αποφύγουμε να διαβάσουμε ,στο επόμενο part,παραπάνω bits από ότι πρέπει,σκεφτήκαμε ότι θα ήταν καλό στην αρχή του αρχείου να βάλουμε το ποσό των χρήσιμων bits,των bits δηλαδή που πρέπει να διαβάσουμε από το τελευταίο byte.

4. Για να βρούμε αυτά τα χρήσιμα bits κάναμε modulo το μέγεθος του αρχείου, δηλαδή το μέγεθος του ***StringBuffer ,*** με το 8 και το αποθηκεύσαμε σε έναν ***Integer*** για να μπορέσουμε στη συνέχεια να το εκτυπώσουμε στην αρχή του αρχείου όπως γράφεται και στην εκφώνηση.

```
//Find the useful bits of the last byte
Integer tmp;
tmp = encode.length() % 8;
```

5. Δημιουργούμε ένα ***DataOutputStream dataOut*** για να μπορέσουμε να γράψουμε στο αρχείο που μας έδωσε ο χρήστης ξεκινώντας με το να γράψουμε τα ωφέλιμα bits ,χρησιμοποιώντας το ***dataOut.writeBytes ,***το ***toString()*** και ένα newline.

```
//Create and output the new bytes in the file
String newLine=System.getProperty("line.separator");
try (DataOutputStream dataOut = new DataOutputStream(new FileOutputStream(args[1]))) {
    //Print the usesful bits and a newline
    dataOut.writeBytes(tmp.toString());
    dataOut.writeChars(newLine);
```

6. Αρχικοποιούμε δύο μετρητές ***int i = 0 και j = 0 ,***τον πρώτο για να μπορέσουμε να διατρέξουμε τον buffer και να ξέρουμε κάθε φορά που βρισκόμαστε και τον δεύτερο για να έχουμε δυναμικά τον αριθμό των shifts που θα πρέπει να κάνουμε για να "ενεργοποιήσουμε"(να κάνουμε δηλαδή 1)το σωστό bit.Αρχικοποιήσαμε επίσης ,μία μεταβλητή την ***byte currentByte*** που θα μας βοηθήσει στο να εκτυπώνουμε κάθε φορά τα σωστά 8 bits στο αρχείο.

```
byte currentByte = 0;
int i = 0, j = 0;
```

7. Χρησιμοποιήσαμε ένα ***while*** για να διατρέχουμε τον ***buffer*** και ελέγχαμε κάθε φορά το στοιχείο που πήραμε από αυτόν.Άμα ήταν το 0 δεν αλλάζαμε το ***currentByte*** γιατί είχε ήδη το bit που χρειαζόμασταν(το 0),άμα όμως ήταν το 1(00000001 σε μορφή byte) τότε κάναμε shift τις κατάλληλες θέσεις για να μπορέσουμε το 1 να το βάλουμε στη σωστή θέση στο byte.Γι αυτή την υλοποίηση χρειαστήκαμε και την ιδιότητα της or που ενεργοποιεί συγκεκριμένα bit σε ένα σχήμα από bits(δηλαδή επιβολή του 1 σε αυτά):

Bitwise Or Operations

Operation	Result
0 | 0	0
1 | 0	1
0 | 1	1
1 | 1	1

```
while (i < encode.length()) {
    //Do the required bitwise operations
    if (encode.charAt(i) == '1') {
        currentByte |= 1 << (7 - j);

    }
```

8. Στη συνέχεια,με τη χρήση ενός if ελέγξαμε άμα έχουμε ολοκληρώσει τα 8 bits για να μπορέσουμε να τα βάλουμε στο αρχείο ή άμα είμαστε στα τελευταία bits που βρίσκονται στο buffer για να τα γράψουμε κι αυτά (αφού το σύστημα προσθέσει τα bits που υπολύπονται για να συμπληρωθεί ένα byte)με τη βοήθεια του ***dataOut.writeByte.***
9. Τέλος,αρχικοποιήσαμε και πάλι τις μεταβλητές ***j*** και ***currentByte*** για να επαναλάβουμε και πάλι τη διαδικασία και για την υπόλοιπη πληροφορία του buffer μέχρι το length του να γίνει 0.

Παραδείγματα Εκτέλεσης του Κώδικα μας:

```
Hello World!
Bye
```

```
      @laptop:~/Desktop/ergasia$ java -cp target/ergasia-1.0-SNAPSHOT.jar org.hua.ergasia.App
Usage: program filename filename
And the first file must already exist
      @laptop:~/Desktop/ergasia$ java -cp target/ergasia-1.0-SNAPSHOT.jar org.hua.ergasia.App denuparxei.txt
Usage: program filename filename
And the first file must already exist
      @laptop:~/Desktop/ergasia$ touch testfile.txt
      @laptop:~/Desktop/ergasia$ java -cp target/ergasia-1.0-SNAPSHOT.jar org.hua.ergasia.App testfile.txt
Usage: program filename filename
And the first file must already exist
```