**”Algorithm Huffman report”**

**Names:** Mahmoud Mohamed **IDs:** 5560

Youssef Ahmed 5491

Marwan Medhat 5365

Tony Fayez 5327

1. **Code Organization:**
   1. Node Class:

* Code:

class Node

{

Character ch;

Integer freq;

Node left = null, right = null;

Node(Character ch, Integer freq) {

this.ch = ch;

this.freq = freq;

}

public Node(Character ch, Integer freq, Node left, Node right) {

this.ch = ch;

this.freq = freq;

this.left = left;

this.right = right

* Code description:

1. Attributes: ch-> holds the character for each node of Huffman tree. Freq-> holds frequency of each character. Left/right-> holds the left and right children for each node.
2. Constructor 1: builds the Node after getting the values of ch and frequency and sets these values.
3. Constructor 2: builds Node after getting the values of ch, freq, left, and right.

* Use: when building the Huffman tree to encode the text file a priority queue is created taking a new Node with its ch value and freq value (these nodes are the leaf nodes so there left and right values are null), then 2 nodes are poped out of the queue and there sum is used to create a new node whose ch is null and freq is the sum of the frequencies of the poped nodes, and left and right are the two poped nodes. This would lead to creating Huffman tree as presented in diagram 1.1.1

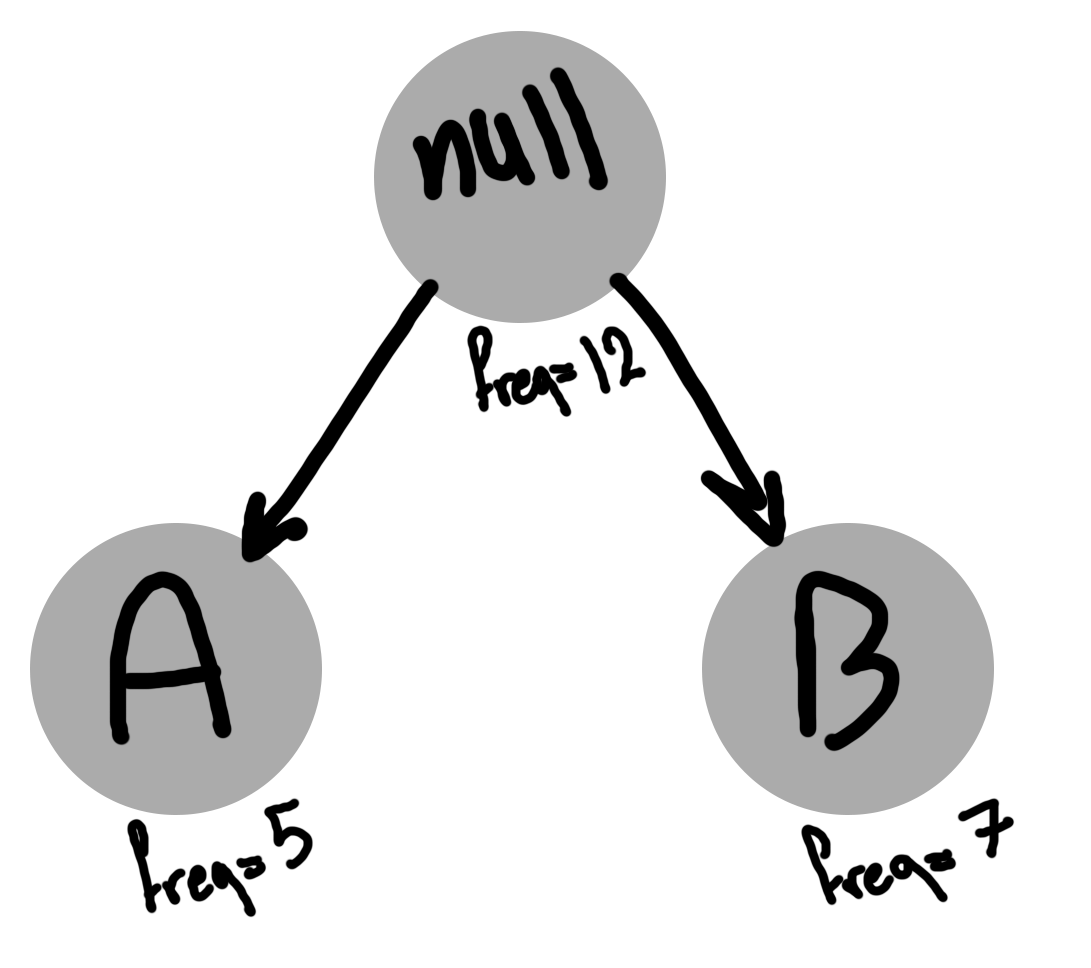


Diagram 1.1.1 (represent how the Huffman tree is created).

* 1. CreateFile Class:
* Code:

public class CreateFile {

public void MakeFile() {

try {

File myObj = new File("Encoded.txt");

if (myObj.createNewFile()) {

} else {

System.***out***.println("File already exists.");

}

} catch (IOException e) {

System.***out***.println("An error occurred.");

e.printStackTrace();

}

}

public void writeToFile( Map<Character, String> map , int addZeros) {

try {

PrintWriter myWriter = new PrintWriter(new FileWriter("encoded.txt", true));

myWriter.append(addZeros+"->"+map);

myWriter.close();

System.***out***.println("Successfully wrote to the file.");

} catch (IOException e) {

System.***out***.println("An error occurred.");

e.printStackTrace();

}

}

public void MakeDecodedFile() {

try {

File myObj = new File("Decoded.txt");

if (myObj.createNewFile()) {

} else {

System.***out***.println("File already exists.");

}

} catch (IOException e) {

System.***out***.println("An error occurred.");

e.printStackTrace();

}

}

public void writeToDecodedFile(String s) {

try {

FileWriter myWriter = new FileWriter("Decoded.txt");

myWriter.write(s);

myWriter.close();

System.***out***.println("Successfully wrote to the file.");

} catch (IOException e) {

System.***out***.println("An error occurred.");

e.printStackTrace();

* Code Description/use: (MakeFile,MakeDecodedFile) methods creates both Endoded.txt(for saving file after compression) and Decoded.txt (for saving file after decompressing). (WriteToFile, WriteToDecodedFile ) are used to write to encoded.txt only the values of the map and a value called additional zero (carries the value of added zeros for the last byte in case it is not 8 bits) , and write to decoded.txt the final text after decompressing normally.
* Note: in writeTofile a PrintWriter.append method is used in order to not overwrite the file.
  1. ByteFiles Class:
* Code:

public class ByteFiles {

static String *FILEPATH* = "Encoded.txt";

static File *file* = new File(*FILEPATH*);

static void writeByte(byte[] bytes,Map<Character, String> map , int addZeros)

{

try {

OutputStream os = new FileOutputStream(*file*);

os.write(bytes);

CreateFile c = new CreateFile();

c.writeToFile( map, addZeros);

System.***out***.println("Successfully" + " byte inserted");

os.close();

}

catch (Exception e) {

System.***out***.println("Exception: " + e);

* Code Description/use: writeByte method is used in order to write the original text as a byte array into encoded.txt file then calling writeToFile method to write Huffman map and additional zeros as described before.
  1. ReadTextFile Class
* Code:

public class ReadTextFile {

public static void readFromFile(String fileName) throws IOException {

int nextLineCounter=0;

try {

File myObj = new File(fileName);

Scanner myReader = new Scanner(myObj);

while (myReader.hasNextLine()) {

if(nextLineCounter==0)

Main.*text* = Main.*text* + myReader.nextLine();

else

Main.*text* = Main.*text* +"\n" + myReader.nextLine();

nextLineCounter++;

}

myReader.close();

} catch (FileNotFoundException e) {

System.***out***.println("FILE NOT FOUND");

e.printStackTrace();

* Code Description/use: readFromFile method is used to read the original file required to be compressed.
  1. ByteFileRead Class:
* Code:

public class ByteFileRead{

public static void readFromFile(String fileName) throws IOException {

int nextLineCounter=0;

int counter=0;

try {

Main.*additionalZero* = new byte[1];

int k=0;

Main.*array* = Files.*readAllBytes*(Paths.*get*(fileName));

for(int i = 0 ;i<Main.*array*.length;i++) {

k++;

if(Main.*array*[i+2]==45&&Main.*array*[i+3]==62&&Main.*array*[i+4]==123) {

break;

}

}

Main.*binaryByteArray*=new byte[k];

for(int i = 0 ;i<Main.*array*.length;i++) {

if(Main.*array*[i+1]==45&&Main.*array*[i+2]==62&&Main.*array*[i+3]==123) {

counter=i+4;

Main.*additionalZero*[0]=Main.*array*[i];

break;

}

Main.*binaryByteArray*[i]=Main.*array*[i];

}

Main.*mapArray* = new byte[Main.*array*.length];

for( int j = counter ;j<Main.*array*.length;j++) {

if(Main.*array*[j]==125) {

break;

}

Main.*mapArray*[j]=Main.*array*[j];

}

File myObj = new File(fileName);

Scanner myReader = new Scanner(myObj);

while (myReader.hasNextLine()) {

if(nextLineCounter==0)

Main.*text* = Main.*text* + myReader.nextLine();

else

Main.*text* = Main.*text* +"\n" + myReader.nextLine();

nextLineCounter++;

}

myReader.close();

} catch (FileNotFoundException e) {

System.***out***.println("FILE NOT FOUND");

e.printStackTrace();

* General Use: used to read from the encoded.txt file into Byte 3 Byte arrays, first carries the encoded text, second carries the additional zeros, third carries the Huffman map.
* Code Description: ReadFromFile method is used to read the entire file as Bytes and added to a Byte array Main.array, then three essential for loops are used, first one adds the bytes from Main.array to Main.binaryBiteArray terminating when (Main.*array*[i+2]==45 && Main.*array*[i+3]==62 && Main.*array*[i+4]==123 ) which is when the following string is reached (->{) which I added as a separator for the encoded text and the Huffman map in the encoded.txt file. Second loop is used to add the byte used for additional zero from Main.array to Main.additionalZero and terminates at( Main.*array*[i+1]==45 && Main.*array*[i+2]==62 && Main.*array*[i+3]==123) on termination we save the value i+4 in a variable named counter which indicates the start of the Huffman map bytes in the Main.array Byte array. Third for loop is used to add the Huffman map from Main.array to Main.mapArray terminating at (Main.*array*[j]==125) indicating char ’}’ which represents the end of map bytes in encoded.txt
  1. Main Class:
* Attributes:

public static String *text*=""; //used to carry text from readFiles.

public static byte[] *array*;//used for ByteFileRead as described.

public static byte[] *mapArray*; //used for ByteFileRead as described.

public static byte[] *additionalZero*; //used for ByteFileRead as described.

public static byte[] *binaryByteArray*; //used for ByteFileRead as described.

static float *compressionRatio*;//used to carry compression ratio value.

* Encode method:

1. Code:

public static void encode(Node root, String str, Map<Character, String> huffmanCode)

{

if (root == null) {

return;

}

if (*isLeaf*(root)) {

huffmanCode.put(root.ch, str.length() > 0 ? str : "1");

}

*encode*(root.left, str + '0', huffmanCode);

*encode*(root.right, str + '1', huffmanCode);

1. Code Description:

* First call: encode(root,””,huffmanCode) root is the root of the Huffman tree, HuffmanCode is an empty hashmap whose keys are the ch values of leaf nodes of Huffman tree and whose values are the coded string using Huffman algorithm.
* The recursive call is done terminating if the node is leaf node and for each iteration the method is called twice, once for left of root giving it a value str+’0’ and for right of root giving value str+’1’.
* When terminating the ch of root is added to huffmanCode as key whose value is str.

}

* IsLeaf method: simply checks if node is leaf by checking left and right children if they are null then it’s a leaf node.
* BuildHuffmanTreeToEncode method:

1. Code:

public static void buildHuffmanTreeToEncode(String text)

{

if (text == null || text.length() == 0) {

return;

}

Map<Character, Integer> freq = new HashMap<>();

for (char c: text.toCharArray()) {

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

PriorityQueue<Node> pq;

pq = new PriorityQueue<>(Comparator.*comparingInt*(l -> l.freq));

for (Map.Entry<Character, Integer> entry : freq.entrySet()) {

pq.add(new Node(entry.getKey(), entry.getValue()));

}

while (pq.size() != 1)

{

Node left = pq.poll();

Node right = pq.poll();

int sum = left.freq + right.freq;

pq.add(new Node(null, sum, left, right));

}

Node root = pq.peek();

*huffmanCode* = new HashMap<>();

*encode*(root, "", *huffmanCode*);

System.***out***.println("Huffman Codes are: " + *huffmanCode*);

//System.out.println("Original string is: " + text);

StringBuilder sb = new StringBuilder();

for (char c: text.toCharArray()) {

sb.append(*huffmanCode*.get(c));

for ( Character key : *huffmanCode*.keySet() ) {

System.***out***.println("BYTE= "+(int)key + " NORMAL CODE= 0"+Integer.*toBinaryString*((int)key) + " HUFFMAN CODE= "+*huffmanCode*.get(key));

}

String temp = "";

String encodedText ="";

int decimal;

int additionalZeros=0;

byte[] bytes ;

if(sb.length()%8==0) {

bytes = new byte[(sb.length()/8)];

}

else {

bytes = new byte[(sb.length()/8)+1];

}

int k=0;

ByteFiles bfile = new ByteFiles();

CreateFile createFile = new CreateFile();

for(int i =0 ; i< sb.length();i++)

{

temp=temp+sb.charAt(i);

if(temp.length()==8) {

decimal = Integer.*parseInt*(temp, 2);

bytes[k]=(byte)decimal;

k++;

encodedText=encodedText+String.*valueOf*(Character.*toChars*(decimal));

temp="";

}

if(i==sb.length()-1) {

additionalZeros = 8-temp.length();

if(temp!="") {

decimal = Integer.*parseInt*(temp, 2);

bytes[k]=(byte)decimal;

k++;

encodedText=encodedText+Character.*toString*((char)decimal);

temp="";

}

}

}

*compressionRatio*=text.length()\*8/bytes.length;

createFile.MakeFile();

ByteFiles byteFile = new ByteFiles();

byteFile.*writeByte*(bytes, *huffmanCode* , additionalZeros);

}

1. Code Description:

* A freq map is created mapping each ch to it frequency.
* A priority queue of nodes is created adding nodes from created map in a sorted fashion, then two nodes are removed adding there sum to the priority queue inorder to create the Huffman tree as described in diagram 1.1.1.
* This process is done till one node is left in the queue which would be the root node.
* Encode method is then called as previously described.
* A stringBuilder sb is created encoding the text that is read from the original file with the Huffman codes.
* A Byte array bytes is created whose size equals sb/8 incase sb is divisible by 8, and equals sb/8 +1 incase its not divisible.
* An essential for loop is implemented that reads characters from sb and put them into a temp string, when this string size = 8, this string is added to the bytes array, then temp value is reseted to null.
* For the last iteration the additionalZeroes value is added by calculating 8-temp.length.
* At last compression ratio is calculated then encoded.txt file is created then the byte array is saved into is as described earlier.
* BuildHuffmanTreeToDecode method:
* Code:

public static void buildHuffmanTreeToDecode(String text) {

String binaryString="";

String mapString= new String(*mapArray*);

for (int d = 0 ; d<mapString.length() ; d++) {

if(mapString.charAt(d)=='=') {

mapString=mapString.substring(d-1, mapString.length()-1);

break;

}

}

String textToSave="";

String temp1="";

String carry = new String(*additionalZero*);

int additionalZeros=Integer.*parseInt*(carry);

*huffmanCode* = new HashMap<Character, String>();

System.***out***.println(*binaryByteArray*.length);

for(int d = 0 ; d<*binaryByteArray*.length-1;d++) {

temp1=temp1+String.*format*("%8s", Integer.*toBinaryString*(*binaryByteArray*[d] & 0xFF)).replace(' ', '0');

}

String x="";

x=String.*format*("%8s", Integer.*toBinaryString*(*binaryByteArray*[*binaryByteArray*.length-1] & 0xFF)).replace(' ', '0');

temp1=temp1+x.substring(additionalZeros);

System.***out***.println("toBinaryString ended");

binaryString=temp1;

System.***out***.println("HUFFMAN MAP="+mapString);

*huffmanCode*=*createMap*(mapString);

/\*for ( Character key : huffmanCode.keySet() ) {

System.out.println("BYTE= "+(int)key + " NORMAL CODE= 0"+Integer.toBinaryString((int)key) + " HUFFMAN CODE= "+huffmanCode.get(key));

}\*/

String temp="";

for(int count = 0 ; count<binaryString.length();count++)

{

temp=temp + Character.*toString*(binaryString.charAt(count));

for ( Character key : *huffmanCode*.keySet() ) {

if(temp.equals(*huffmanCode*.get(key))) {

if(key=='\n')

{

textToSave=textToSave+'\r';

}

textToSave=textToSave+Character.*toString*(key);

temp="";

break;

}

}

}

System.***out***.println("textTosave ended");

/\*String tempString1="";

String tempString2="";

for(int i = 0; i< textToSave.length();i++) {

if(textToSave.charAt(i)=='\n') {

tempString1 = textToSave.substring(0, i);

tempString2 = textToSave.substring(i);

textToSave = tempString1+"\r"+tempString2;

i++;

tempString1="";

tempString2="";

}

}\*/

CreateFile createFile = new CreateFile();

createFile.MakeDecodedFile();

createFile.writeToDecodedFile(textToSave);

}

* An essential for loop is used looping for the size of ByteArray-1 storing the encoded text in a temp string, after loop terminates the last byte is added after removing the additional zeros then the resulted temp string is stored in binaryString.
* Then a huffmancode map is created using the mapArray Byte array and createMap method.
* Then another essential for loop is used to loop around the length of binaryString temporally storing each character into a string and checking this string with the Huffman map values, if the string is found in the map the values key is added to the string used to be stored in the decoded.txt file.
* Then the decoded.txt file is created and the text is added.

1. **Data Structures used:**

2.1- priority queue (pq): used to create the Huffman tree as described in section 1.6 (buildHuffmanTreeToEncode)

2.2- hashmap huffmanCode: used to store the ch of all nodes as keys whose value is there Huffman code.

2.3- map freq: used to store the ch of nodes as key whose value is the frequency of the node.

2.4- multiple byte arrays: in order to store/read data to/from file.

1. **Algorithm used / Complexity:**

Huffman algorithm is used to encode the text file whose complexity equals O(nlogn).

->note: most of the execution time is spent in the file management.

1. **Compressed file format:**

Encoded text + additional zeros + separator “->” + huffmanCode map

Ex: ûûßJÍ©6åaòwwj­#Ãu€$Ežq¬–6->{ =01, a=1100, b=10101, d=11110, e=0010, '=000110, h=11010, i=101110, k=10110, m=00001, n=10100, o=100, O=111111, r=00010, t=1110, u=0011, U=101111, v=00000, w=11011, y=000111, Y=111110}

1. **Problems faced:**

5.1- At first we encoded the binary string by turning it into an integer then getting the ascii value for it (no byte array used) which produced a wrong output reason being that java characters end at 127 and anything past it was stored as ‘?’

Solution: instead of encoding it ourselves we stored it in a byte array.

5.2- an execution time of approximately 1 hour 30 minutes was resulted when running the program with the given test case for decompressing, which we believed to be unacceptable.

Reason: we noticed that a huge time is spent specifically in 2 for loops, first one used to store the encoded text from the byteArray(at first an if condition was used to check whether it’s the last byte from the array or not if so the additional zero condition is used),second was a for loop checking whether \n character is reached in the resulted text string if so it would ad \r before it.

Solution: for the first for loop the if condition was removed and loop terminated before last element as shown in current code, for second for loop we completely removed it and added its function to the for loop that changes the binaryString to the textToSave string. -> this lead to decreasing the execution time from 1 hour 30 minutes to 20 minutes (we understand it is still unacceptable but a massive improvement nonetheless) .

5.3- execution still took a long time of 20 minutes which we understand to be the result of the following for loop: for(int d = 0 ; d < *binaryByteArray*.length-1 ;d++) { temp1=temp1+String.*format*("%8s", Integer.*toBinaryString*(*binaryByteArray*[d] & 0xFF)).replace(' ', '0');}

From our understanding String.format method takes a long time to execute (since it does a lot of string manipulation) but since it’s essential to the program we didn’t know how to improve upon it.