REPORT

Huffman Encoding and Decoding

Advanced Data Structures
COP 5536 Spring 2017

Prepared By:

Anuja Salunkhe

UF Id – 3213-0171

UF Email Id – ansalunk@ufl.edu

INDEX

1. Structure of Program

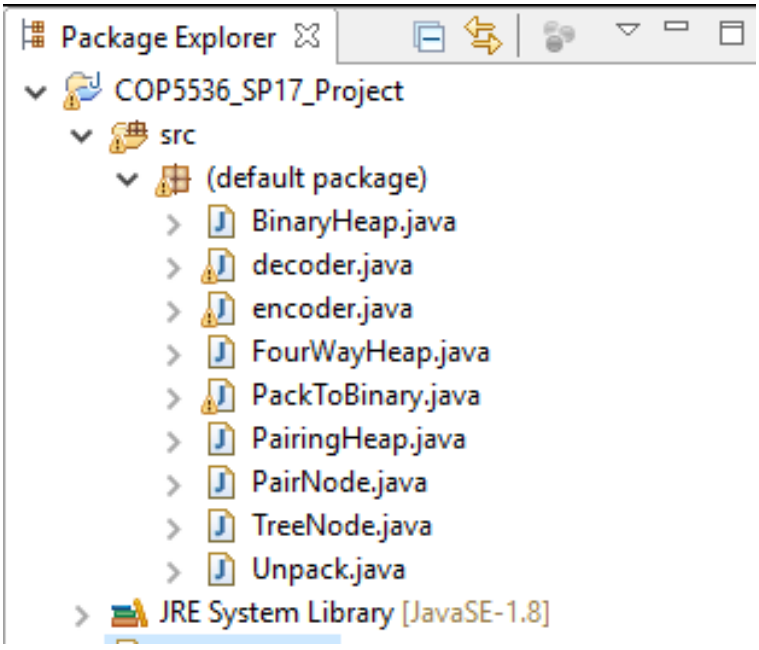
The program consists of different classes as shown in the below image :

1. **datastructure** – This consists of three different types of data structure used namely **BinaryHeap.java**, **FourWayHeap.java** – Optimised, **Pairing Heap.java**. **TreeNode.java** and **PairNode.java** is the class to store the actual value of the node and the frequency of the each of node.

2. **huffmann** – This consist of two main classes:
    a. **encoder.java** – for Encoding
    b. **decoder.java** - for Decoding

3. **encode** – The class **PackToBinary.java** consists of functions that reads from command line a file containing a sequence of the ASCII characters '0' and '1', representing a sequence of bits. It converts each character to a bit, and packs the bits into bytes. The resulting packed bytes are written to encoded.bin file.

4. **decode** – The class **Unpack.java** has functions that reads a binary file from encoded.bin and converts each byte of the input into a string of the ASCII characters '0' and '1'. The resulting sequence of characters is then decoded using a decode tree to output a decoded.txt file.



2. Data-Structures

The three data structures used for Performance analysis are :

1. Binary Heap -> BinaryHeap.java and TreeNode.java
2. Four Way Heap -> FourWayHeap.java and TreeNode.java
3. Pairing Heap -> PairingHeap.java and PairNode.java

BinaryHeap.java

This program uses an Arraylist of TreeNode while creating a binary heap for the same.

```
private static final int d = 2;
private ArrayList<TreeNode> bHeap;
```

The functions inside BinaryHeap.java are :

a. A constructor : To initialise the Arraylist of TreeNode.

```
public BinaryHeap() {
    bHeap = new ArrayList<TreeNode>();
}
```

b. isEmpty() – to check if heap is empty.

```
public boolean isEmpty() {
    return bHeap.size() == 0;
}
```

c. parent(int i) – to get parent index of i.

```
private int parent(int i) {
    return (i - 1) / d;
}
```

d. insert(TreeNode node) – to insert the element inside the heap.

e. deleteMin() – Function to delete minimum element i.e. root.

f. heapifyDown(int posParent) – This method finds the element which is the next minimum and swaps the position

of the minimum element i.e. root with that of the next minimum element and removes then removes it.

g. getHeapSize() – this function returns the current heapsize of the structure.

FourWayHeap.java

This program uses an Arraylist of TreeNode while creating a binary heap for the same. The project however have implemented a cache optimised heap which has the first three indexed as empty to align the children's in the same cache line.

```
ArrayList<TreeNode> fHeap = new ArrayList<TreeNode>();
static FourWayHeap fourWayHeap;
```

The functions inside BinaryHeap.java are :

a. A constructor : To initialise the Arraylist of TreeNode.

```
public FourWayHeap() {
    fHeap.add(null);
    fHeap.add(null);
    fHeap.add(null);
}
```

b. isEmpty() – to check if heap is empty.

```
/** Function to check if heap is empty **/
public boolean isEmpty() {
    return fHeap.size() == 3;
}
```

c. parent(int i) – to get parent index of i.

```
private int parent(int i) {
    return (int) Math.floor(((i - 4) / 4) + 3);
}
```

d. insert(TreeNode node) – to insert the element inside the heap.

e. deleteMin() – Function to delete minimum element i.e. root.

f. Heapify(int posParent) – This method finds the element which is the next minimum and swaps the position of the

minimum element i.e. root with that of the next minimum element and removes then removes it.

```
public void heapify(int posParent) {
    int childPos = 4 * (posParent - 3) + 3;
    int i = 1;
    int minValue = posParent;
    while (i < 5) {
        childPos++;
        if (childPos < fHeap.size()) {
            if (childPos < fHeap.size()
                    && fHeap.get(childPos).getFreq() < fHeap.get(minValue)
                            .getFreq()) {
                minValue = childPos;
            }
        }
        i++;
    }
    if (minValue != posParent) {
        TreeNode temp = fHeap.get(posParent);
        fHeap.set(posParent, fHeap.get(minValue));
        fHeap.set(minValue, temp);
        heapify(minValue);
    }

}
```

g. getHeapSize() – this function returns the current heapsize of the structure.

h. findMin() – To find the minimum of the heap.

```
public TreeNode findMin() {
    if (isEmpty())
        throw new NoSuchElementException("Underflow Exception");
    return fHeap.get(3);
}
```

3. Performance Analysis :

This project have implemented three data structures to build a Huffman Tree. We are calculating the time required to generate the Huffman tree using these three data structures.

```
/*
 * Time to execute the method using 4Way Heap
 */

long startTime = System.nanoTime();
for (int i = 0; i < 10; i++) { // run 10 times on given data set

    generateBinaryTree();
}

long elapsedTime = ((System.nanoTime() - startTime) / 10);

System.out.println("Time using 4way heap " + elapsedTime);
double seconds = (double) elapsedTime / 1000.0;

System.out.println("which is " + (int) seconds + " microseconds");
```

```
/*
 * Time to execute the method using Binary Heap
 */
startTime = System.nanoTime();

for (int i = 0; i < 10; i++) { // run 10 times on given data set

    generateBinaryTree();
}

elapsedTime = ((System.nanoTime() - startTime) / 10);

System.out.println("Time using binary heap " + elapsedTime);
seconds = (double) elapsedTime / 1000.0;

System.out.println("which is " + (int) seconds + " microseconds");
```

```
// Time to execute the method using Pairing Heap

startTime = System.nanoTime();
for (int i = 0; i < 10; i++) { // run 10 times on * given data set

    generatePairingTree();
}

elapsedTime = ((System.nanoTime() - startTime) / 10);

System.out.println("Time using pairing heap " + elapsedTime);
seconds = (double) elapsedTime / 1000.0;

System.out.println("which is " + (int) seconds + " microseconds");
```

According to the results as per the below snapshots which are run on storm for different input files having different sizes.

1. **<u>Small input</u>**

```
stormx:11% java -jar Ads_HuffTree.jar sample_input_small.txt
16 map size 6
Time using 4way heap 239861
which is 239 microseconds
Time using binary heap 108466
which is 108 microseconds
Time using pairing heap 148298
which is 148 microseconds
```

For the small input with number of elements -16 and the no o unique elements as 6 we can see that :

Binary Heap – Fastest → 108 micro sc

Fourway Heap – Slowest → 239 micro sc

Pairing Heap – Intermediate → 148 micro sc

2. Large Input

```
stormx:16% java -jar Ads_HuffTree.jar sample_input_large.txt
10000000 map size 990619
Time using 4way heap 740413035
which is 740413 microseconds
Time using binary heap 727904754
which is 727904 microseconds
Time using pairing heap 29389508801
which is 29389508 microseconds
```

Binary Heap – Fastest → 727904 micro sc

Fourway Heap – Intermediate → 740413 micro sc

Pairing Heap – Slowest → 29389508 micro sc

```
stormx:17% java -jar Ads_HuffTree.jar sample_input_large.txt
10000000 map size 990619
Time using 4way heap 827444282
which is 827444 microseconds
Time using binary heap 843368548
which is 843368 microseconds
Time using pairing heap 33288857006
which is 33288857 microseconds
```

Binary Heap – Intermediate → 843368 micro sc

Fourway Heap – Slowest → 827444 micro sc

Pairing Heap – Fastest → 33288857 micro sc

For the large input with number of elements -10000000 and the no o unique elements 990619 as we can see that, however the timings for binary and fourway almost remain similar where sometimes it takes less time for Binary and sometimes less time for fourway.

Hence keeping in the mind the similar timings for Binary and Fourway heap, in project I chose to implement this using **the Binary heap** since it performs better for both the large and the small inputs.

4. Huffman Encoding :

Implemented using encoder.java , PackToBinary.java and BinaryHeap.java.

1. To implement Huffman Encoding in this project I use Binary Heap which first takes input in form of the input file from command line.

2. The input file which is taken is then traversed to calculate the corresponding frequency if the elements from the file and it is saved into a map as shown in below snapshot :

   Main method inside **encoder.java** –

```
public static void main(String[] args) throws Exception {

    freqMap = new HashMap<Integer, Integer>();
    input = new ArrayList<Integer>();
    packMessage = new PackToBinary();

    @SuppressWarnings("resource")
    Scanner scanner = new Scanner(new File(args[0]));

    int totalAmountOfNumbers = 0;

    while (scanner.hasNext())

    {

        int currentNumber = scanner.nextInt();
        input.add(currentNumber);
        Integer count = freqMap.get(currentNumber);
        if (count == null)
            freqMap.put(currentNumber, 1);
        else
            freqMap.put(currentNumber, count + 1);

        totalAmountOfNumbers++;
    }

    System.out
            .println(totalAmountOfNumbers + " map size " + freqMap.size());
```

3. **Build Huffman Tree** : After we save the frequencies of all the elements in a freqMap we generate the huffman tree by calling series of methods in **encoder.java** i.e. for each entry in the frequency map, we create a TreeNode containing the character and the frequency. Insert each of these TreeNodes in a BinaryHeap<TreeNode>.

```
private static void buildBinaryHeap(Map<Integer, Integer> freqMap) {

    bheap = new BinaryHeap();

    for (Entry<Integer, Integer> entry : freqMap.entrySet()) {
        Integer key = entry.getKey();
        Integer value = entry.getValue();
        TreeNode node = new TreeNode(key, value);
        bheap.insert(node);
    }

}
```

4. **Huffman Tree Algorithm** : While implementing the Huffman tree when we insert the nodes into the Binaryheap we compare the frequencies and then the one with the smallest frequency is heapified.

   Huffman Algorithm to generate Huffman tree.

   a. Remove two nodes from the priority queue BinaryHeap
   b. Create a new TreeNode with these two nodes as its children. The frequency of the node is then set to sum of the frequencies of the two children.
   c. Add the new node to the priority queue BinaryHeap.

```java
/**
 * Huffman Algorithm
 *
 * @param heap
 *            minimum heap
 * @return huffman tree
 */
private static BinaryHeap buildBinaryTree(BinaryHeap heap) {
    int n = heap.getHeapSize();
    for (int i = 0; i < (n - 1); ++i) {
        TreeNode z = new TreeNode();
        z.setLeft(heap.deleteMin());
        z.setRight(heap.deleteMin());
        z.setFreq((int) (z.getLeft().getFreq() + z.getRight().getFreq()));
        heap.insert(z);
        // heap.printHeap();
    }
    return heap;
}
```

5. **Code Table** : Once the Huffman tree is generated we then generate the code table in form of code_table.txt which is needed to generate the encoded.bin is the binary file.

   Once the code table is saved in the codeMap we then write it inside a file called **code_table.txt**.

```java
/**
 * Generates huffman codes for all elements in the file.
 *
 * @param node
 *            root of Huffman tree
 *
 */
private static void generateCode(TreeNode node, String code) {
    if (node.left == null && node.right == null) {
        codeMap.put(node.getNode(), code);
        return;
    }
    generateCode(node.left, code + '0');
    generateCode(node.right, code + '1');
}
```

```
public void generateCodeTable(Map<Integer, String> codeMap)
        throws IOException {

    FileWriter f = new FileWriter(
            "code_table.txt");

    BufferedWriter out = new BufferedWriter(f);

    for (Entry<Integer, String> codeTable : codeMap.entrySet()) {
        String code = codeTable.getValue();

        int key = codeTable.getKey();

        String line = key + " " + code
                + System.getProperty("line.separator");

        out.write(line);
    }
    out.flush();
    out.close();

}
```

6. Encode Input **encoded.bin** : We use the code_table generated to generate the binary file i.e. encoded.bin.

```
public static String encode(ArrayList<Integer> input) {
    // TODO: Write me!

    final StringBuilder stringBuilder = new StringBuilder();

    for (int i = 0; i < input.size(); i++) {

        if (codeMap.containsKey(input.get(i))) {
            stringBuilder.append(codeMap.get(input.get(i)));
        }
    }
    return stringBuilder.toString();

}
```

```
public void encodeToBinary(String encodedMessage) throws IOException {

    ArrayList<Byte> array = new ArrayList<Byte>();

    System.out.println(encodedMessage);
    StringBuilder encodedStrBuildMsg = new StringBuilder(encodedMessage);

    DataOutputStream out = new DataOutputStream(new FileOutputStream(
            "encoded.bin"));

    packToBinary(encodedStrBuildMsg, array);

    for (byte c : array) {
        out.write(c);

    }

    out.flush();
    out.close();
}
```

5. Huffman Decoding :

Implemented using decoder.java, UnPack.java and BinaryHeap.java.

1. To implement Huffman Decoding in this project we use class **decoder.java** which first takes input in form of the encoded.bin and code_table.txt file from command line.

2. **De-serialize encoded.bin file** : We unpack the encoded.bin file to get the actual string on form of 0's and 1's.

```
public static void main(String[] args) throws IOException {

    unpack = new Unpack();

    /* Encoded file */
    ArrayList<Byte> packed = new ArrayList<Byte>();
    StringBuilder codeBuilder = new StringBuilder();
    InputStream fis1 = new FileInputStream(new File(args[0]));

    BufferedReader reader1 = new BufferedReader(new InputStreamReader(fis1));

    int b = reader1.read();
    while (b >= 0) {
        packed.add((byte) b);
        b = reader1.read();
    }
    unpack.unpack(codeBuilder, packed);

    //System.out.print(codeBuilder);

    reader1.close();
```

3. **Generate decoded.txt using Huffman decode tree** : Decode the String using the code_table.txt : To decode a string, I will create a Huffman decode tree. The '0's and '1's in the string dictate a path to follow, starting from the root, through the nodes of the tree. Again, '0' means go left and '1' means go right. When the path reaches a leaf, the node in the leaf is the node corresponding to the input code, so we can then add it to the decoded string and then write it to the **decoded.txt** file.

a. Generate Huffman decode tree

```
public TreeNode buildDecodeTree(Map<Integer, String> codeMap) {

    root = new TreeNode();

    for (Entry<Integer, String> node : codeMap.entrySet()) {

        int key = node.getKey();
        String code = node.getValue();
        char[] codeChar = code.toCharArray();
        temp = root;

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

            temp = insertNode(codeChar[count],
                    count == (codeChar.length - 1), temp, key);

        }

    }

    return root;
}
```

b. Using the Huffman decode tree, traverse the string to get the corresponding element/node.

```
public StringBuilder generateDecodedFile(StringBuilder codeBuilder, TreeNode root) {
    // TODO Auto-generated method stub

    String binaryString = codeBuilder.toString();

    //System.out.println(binaryString);
    StringBuilder output = new StringBuilder();
    TreeNode base = root;
 while (!binaryString.isEmpty()){
    if (binaryString.charAt(0) == '1'){
        base = base.right;
        binaryString = binaryString.substring(1);
    }
    else {
        base = base.left;
        binaryString = binaryString.substring(1);
    }
    if (base.left == null && base.right == null){
        output.append(base.getNode()+ "\n");
        base = root;
    }

  }

 return output;
}
```

c. Write the node to the **decoded.txt** file.

```
TreeNode root = unpack.buildDecodeTree(codeMap);
//unpack.printTree(root);

StringBuilder decodeString = unpack.generateDecodediFile(codeBuilder, root);
//System.out.println(decodeString);

String dString = decodeString.toString().trim();
FileWriter f = new FileWriter(
        "decode.txt");

BufferedWriter out1 = new BufferedWriter(f);

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

    if(c=='\n')
        out1.write(System.getProperty("line.separator"));
    else{
    out1.write(c);
    }

            }

System.out.println("Finish");
out1.flush();
out1.close();


reader2.close();
```

6. **Decoding Algorithm to build Decode Tree :**
    1. In this project I use a variant of Binary Search Tree to build the Decode Tree.
    2. First we take the input from code_table.txt and store the corresponding node value and the code in the hashmap named codeMap.
    3. Let the number of elements/nodes in the codeMap be 'n'. Start with the empty TreeNode i.e. root.
    4. Repeat('**N**' number of elements/nodes)
        a. For each "**code**" for the corresponding element/node we convert the code to an array of characters.
        b. **M**= length of the code
        c. Repeat (**M** != 0)
            i. The for each character which can have a value 1 or 0, we create a new node go to
            ii. Left if value is 0
            iii. Else right if value id 1
        d. If the character is the last in the code, then we create a node and store the corresponding element inside the node.
    5. **Time Complexity :**
        a. **N *M**
        b. **N – Number of nodes**
        c. **M – Length of the code for each node**
        d. **In wrost case M = logN i.e. height of the tree**
        e. **Hence Worst Case – N(logN)**

Hence Worst case the time complexity is N(logN)