# Advanced Data Structures (COP 5536)

# Fall 2020

# Programming Project Report

Name Mingjun Yu

UFID 61707843

## 1.Problem Definition

The goal of this project is to implement Huffman encoder and decoder.

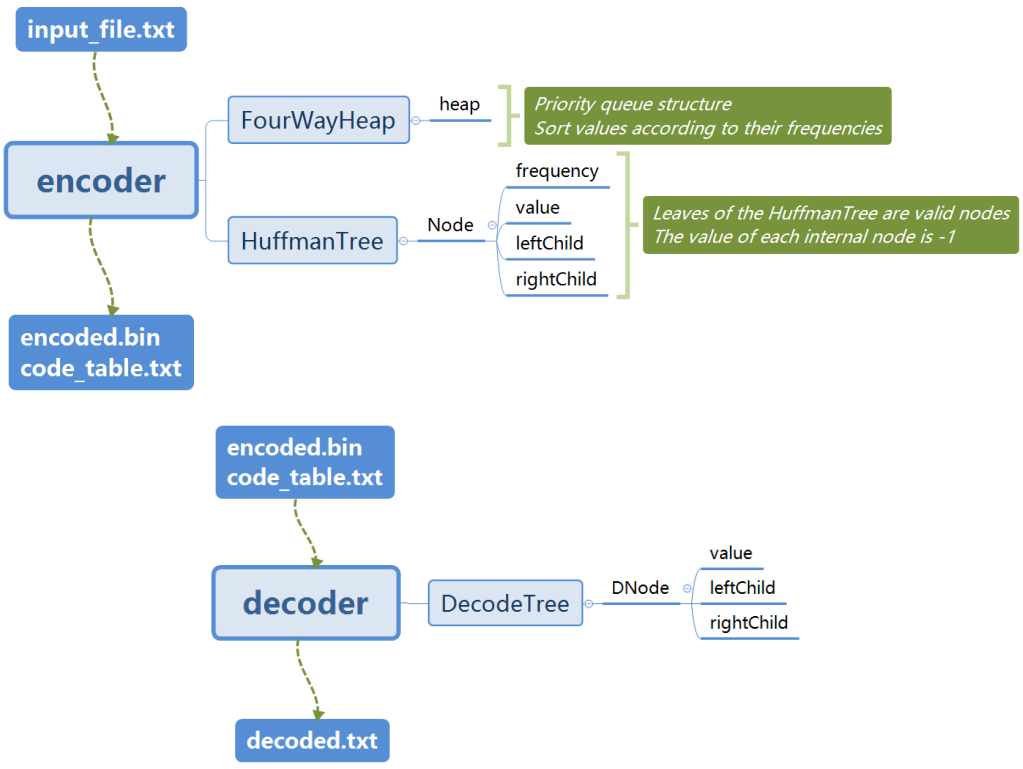
The first step of this project is to develop a program that generates Huffman codes. Three kinds of priority queue structures are implemented: Binary Heap, 4-way cache optimized heap and Pairing Heap. Their performance needs to be evaluated. Then select the data structure whose performance is the best, and use this data structure to write the encoder.

The second step of this project is to write the encoder. The encoder reads an input file that is to be compressed. Then, the corresponding frequency table and code table will be generated. The input file will be compressed using the code table. Finally, the encoder generates two output files: the compressed version of the input file and the code table.

The third step is to write the decoder. The decoder reads two input files – encoded message and code table. To begin with, the decoder will construct corresponding decode tree using the code table. Then the decoded message is able to be generated while traversing the decode tree. The decoder will generate a output file: the decoded version of the input file.

## 2.Function Prototypes and Program Structure

Five classes are implemented in this project: encoder, decoder, HuffmanTree, FourWayHeap and DecodeTree. The encoder is realized with two auxiliary classes: HuffmanTree and FourWayHeap. The decoder is also realized with one auxiliary class: DecodeTree. The structure of these classes is shown as follows.



The function prototypes are shown in the following charts.

**FourWayHeap.java**

| Prototype | Function |
|---|---|
| public FourWayHeap() | Constructor |
| public boolean isEmpty() | If the four-way-min heap is empty, return true; otherwise return false. |
| public void insert(Node n) | Insert node n into the four-way-min heap. |
| public HuffmanNode removeMin() | Return the node with minimum frequency, i.e. the root node, and delete it from the four-way-min heap. |
| private int parent(int n) | Return the index of the parent of node n. |
| private int minChild(int n) | Return the index of node n's child with minimum frequency among its children. |
| private void swap(int i, int j) | Swap the position of node i and node j. |

**encoder.java**

| Prototype | Function |
|---|---|
| private static HashMap<Integer, Integer>buildFreqTable(String inputfile) | Count the number of occurrences of each integer in the input file and store them in the hash table. Return this frequency table. |
| private static HashMap<Integer, String> buildHuffmanTreeAndCodeTable(HashMap<Integer, Integer> freqTable) | Generate Huffman tree with frequency table, and then traverse the Huffman tree to generate the code table. Generate code_table.txt. Return the code table. |
| private static Node buildTreeUsing4wayHeap(HashMap<Integer, Integer> freqTable) | Generate Huffman tree with frequency table. Return the root of the Huffman tree. |
| private static HashMap<Integer, String> genereteHuffmanCode(Node root) | Traverse the Huffman tree to generate code table. Return the code table. |
| private static void encodeData(HashMap<Integer, String> codeTable, String inputfile) | Compress input file using code table. Generate encoded.bin. |

**decoder.java**

| Prototype | Function |
|---|---|
| private static DNode constructDecodeTree(String codeTable) | Generate decode tree using code table. Return the root of decode tree. |
| private void decodeData(DNode tree, String encodedfile) | Decode the encoded data with decode tree. Generate decoded.txt. |

## 3.Performance

The first step in this project is to evaluate which of the following priority queue structures gives the best performance: Binary Heap, 4-way cache optimized heap, and Pairing Heap. I measured run time of each data structure as suggested. The run time of each data structure is shown in the following picture.

```
Time using binary heap (microsecond): 53
Time using pairing heap (microsecond): 20
Time using 4-way heap (microsecond): 6
```

We can see that 4-way heap gives the best performance. Compared with Binary Heap, removeMin operations now do 4 compares per level rather than 2 in the 4-way heap. But number of levels is half. Thus 4-way heap gives a better performance. So, I use 4-way heap as the priority queue structure.

## 4.Decoding Algorithm

The decoding algorithm includes two steps: construct the decode tree with code table and traverse the decode tree to decode.

Step one: construct a decode tree

1. Read <value, code> pairs from code table, where value represents an integer and code represents its corresponding code.

2. Create decode tree according to each character of the code. If the character is '0' and the left child of current node doesn't exist, create a new node and append this node to current node as its left child; if left child exists, advance to it. If the character is '1', the procedure is the same, and the only different is we need to create or advance to right child. When reaching the end of the code, assign value of the pair to current node.

Assuming that n is the number of leaf nodes, i.e. the number of <value, code> pairs in the code table, we can know in the worst case, creating a leaf node takes O(logn) time, since it is possible that we need to traverse each level of the tree. And there are n leaf nodes. So in the worst case, the time complexity of step one is O(nlogn).

Step two: traverse the decode tree to decode data

1. Read encoded.bin into a byte array. Convert each byte into an 8-bit string.
2. Traverse decode tree from the root node according to each bit in the 8-bit string. If current bit is 0, advance to left child; if current bit is 1, advance to right child.

3. If a leaf node is encountered, write the value of this node into decoded.txt, and start

traversal from the root node again.

Since step two need to traverse each bit in the encoded.bin, the time complexity of step two is O(number of bits of encoded.bin).

## 5. Summary

After encode and decode the input file, we can find out the size of encoded message is unique; output message is same as input message.

## 6. Reference


[1] http://www.sanfoundry.com/java-program-implement-d-ary-heap/
[2] http://www.sanfoundry.com/java-program-implement-pairing-heap/
[3] http://www.cs.ubc.ca/~hoos/PbO/Examples/Java/DaryHeap/DHeap.pbo-j
[4] http://stackoverflow.com/questions/12310017/how-to-convert-a-byte-to-itsbinary-string-representation
[5] http://stackoverflow.com/questions/27677519/java-how-to-write-bits-andnot-characters-in-a-file