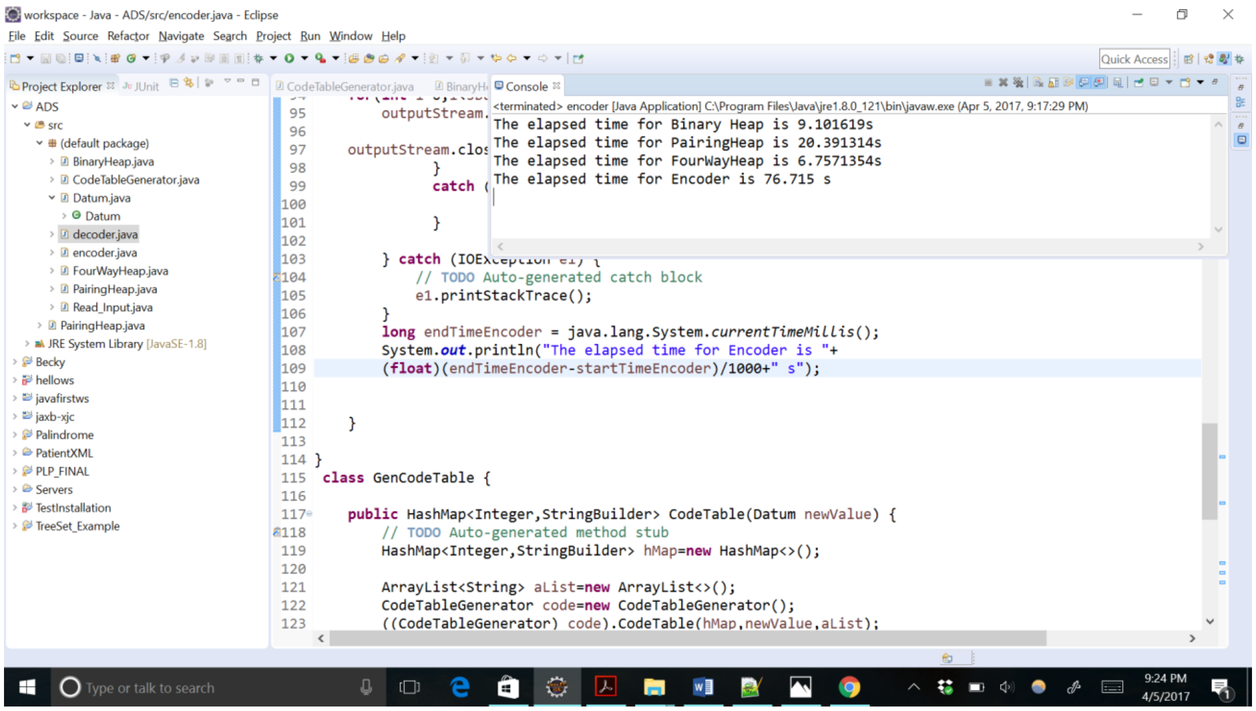
COP 5536 ADVANCED DATA STRUCTURES PROJECT REPORT

HAMSIKA GANDAMALLA
UFID: 5110-1448

Introduction:

In this project, I have implemented three data structures: Binary Heap, 4-way cache optimized heap, and Pairing Heap. Generation of Huffman trees using 4-way cache optimized heap was finalized as 4-way cache optimized heap gave better performance than Binary Heap and Pairing Heap. 4-way cache optimized heap was hence used for encoding and decoding. The encoding process starts at encoder.java that takes sample_input_large.txt as input and produces encoded.bin and code_table.txt as output. The decoding process starts encoded.bin and code_table.txt as input and produces decoded.txt as output.



```
<terminated> encoder [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Apr 5, 2017, 9:17:29 PM)
The elapsed time for Binary Heap is 9.101619s
The elapsed time for PairingHeap is 20.391314s
The elapsed time for FourWayHeap is 6.7571354s
The elapsed time for Encoder is 76.715 s
```

```java
        outputStream.
    
    outputStream.clos
                }
                catch (

                }

        } catch (IOException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        long endTimeEncoder = java.lang.System.currentTimeMillis();
        System.out.println("The elapsed time for Encoder is "+
        (float)(endTimeEncoder-startTimeEncoder)/1000+" s");


    }

}
 class GenCodeTable {

    public HashMap<Integer,StringBuilder> CodeTable(Datum newValue) {
        // TODO Auto-generated method stub
        HashMap<Integer,StringBuilder> hMap=new HashMap<>();

        ArrayList<String> aList=new ArrayList<>();
        CodeTableGenerator code=new CodeTableGenerator();
        ((CodeTableGenerator) code).CodeTable(hMap,newValue,aList);
```

Function Prototypes and Structure of Program:

Datum:

The class Datum defines the structure of the nodes in the Huffman tree.

encoder:

The class encoder takes the inputfile and passes it to Read_InputMethod in the class Read_Input which returns the Frequency Table in the form of a hash map containing the input elements as keys and the frequency of their occurrences as their corresponding values.

The method buildTree in class FourWayHeap.java takes the frequency table and returns the root of the Four way heap constructed using the input.

The class GenCodeTable takes the Four way heap and passes a new hash map, the root of 4 way heap and a newly created array list to CodeTable method in the class CodeTableGenerator which calculates the Huffman codes for the data in the 4 way heaps. The data is stored as keys and their corresponding Huffman codes are stored as values in the hash map. This hash map is used to write into code_table.txt.

Using the input file and the code table, the encoded.bin is generated. This is done by first searching the hash map for the input record and then getting its corresponding Huffman code from the hash map's value field. Then the value is appended to a string builder. This process is repeated for every record in the input file. After reading the input file the string builder is converted into bytes and written into encoded.bin.

decoder:

The class decoderDatum in decoder.java defines the structure of nodes in the decoder tree.
The class decoder takes encoded.bin and code_table as inputs to produces decoded.txt as output.
First, we place the data and their Huffman codes from the code_table into an array list of type decoderDatum then pass this array list to the decodeTree method in

huffManTree class.This method constructs the decoder tree. The encoded.bin is sent as an argument to decoder method which generates decoded.txt.

Performance analysis results and explanation:

As mentioned above the 4-way cache optimized heap gave the fastest running time in constructing the Huffman Tree among 4-way cache optimized heap, binary heap and pairing heap. The time elapsed on my system for the three data structures is given below:

The elapsed time for Binary Heap is 9.101619s
The elapsed time for Pairing Heap is 20.391314s
The elapsed time for Four Way Heap is 6.7571354s

The results above were obtained using the sample_input_large.txt provided and by taking the average after running each heap algorithm for 10 times. Hence, the 4-way cache optimized heap was chosen as the suitable data structure for generating Huffman Code. The encoder was observed to have a running time of around 75 seconds on using sample_input_large.txt. Next, on construction of the decoder, a similar analysis was conducted which gave an average running time of around 23.9 seconds.

My analysis behind the reason why the 4-way cache optimized heap gave the best performance is because it is cache optimized which reduces the number of cache misses when we want to access the children of particular node. And if the cache line is of size 4 then all the children of a node will be brought at once from the disk. Also the height of the tree is less for 4-way heap when compared with the binary heap with equal number of nodes.

Decoding algorithm and its complexity:

The decoding algorithm is of two parts:

1) The Decoder Tree –

The code table generated in the encoding step is used to create the decoder tree. The leaves of the tree has data elements from the code table and the path to these leaves consists of the Huffman code.

The code of each element is taken and is traversed sequentially with a new node being constructed if the bit in the code is the last bit of the code corresponding to the data element. Before constructing a new node it is checked whether any node is already present, if it is then a simple traversal is made and the next bit of the code is considered. If bit is '0' then a left node is constructed, else if it is a '1' a right node. By the time the tree is constructed, all the leaves correspond to each of the data element in the code table and the path from the root being the corresponding code.

2) Decoder tree and encoded file used for Decoding -

After the decoder tree is constructed, the code from the encoded.bin file is taken as a String builder object. Then a traversal is made based on the each bit of the encoded string. The traversal starts initially from the root and also each time a leaf is reached the traversal is done from the root. Each bit is checked in a sequential manner and a left traversal is made if the bit is '0', else a right traversal is made if the bit is '1'. After each traversal, it checked if a leaf node is reached, in which case the data of the leaf node is written to the decoded.txt file.

The decoded.txt and the sample_input_large.txt turn out to be the same.

Complexity of Decoding algorithm:

The complexity to decode= Number of bits in the encoded.bin.
The time to construct decode tree= Sum of Number of bits in the huffman codes in code_table.txt