Anurag Bihani **UFID:** 9831-9647

Advanced Data Structures

Project Report

Project Description

The problem statement was to help transfer enormous amount of data by using Huffman coding to reduce data size. Huffman coding is a lossless data compression algorithm with variable length codes. The codes are selected in a manner such that most frequently occurring data items get the smallest code.

The following data structures were used to implement this project.

- Hash Map: Used for storing the frequency table
- Huffman Tree: Used for encoding and decoding the input file.
- Binary Heap: Used as a priority queue to generate the Huffman Tree.

The following data structures were coded and tested out but were not used to perform encoding or decoding.

- 4-way Cache Optimized Heap.
- Pairing Heap.

The Project consisted of 3 parts.

Huffman Coding

This section involved finding the best heap structure to implement as the priority queue used for generation of the Huffman Tree.

Encoding

After selection of the best data structure, which empirically turned out to be binary heap. We generated an encoded binary file to represent the data in a compressed manner using the Huffman codes.

Decoding

A decoding algorithm was designed and realized which would decode the binary file generated by the encoder and obtain the original file.

Programming Environment

The project has been implemented in Java Language. The code has been tested in Java environment on thunder.cise.ufl.edu. The code produced expected results in tolerable time. The make file compiles all the java files and produces executables for ”encoder” and “decoder” as asked in the project requirement.

$ make

This command produces binary executable files “encoder” and “decoder”

These can be executed as follows:

$ java encoder <input_file_name>

Running the encoder produces output files with names “encoded.bin” and “code_table.txt”

$ java decoder <encoded_file_name> <code_table_file_name>

Running the decoder this way generates a file called “decoded.txt”

Program Structure

Classes present in the code.

Node.java
BinaryHeap.java
FourwayHeap.java
FourwayHeapNode.java
PairingHeap.java
PairingHeapNode.java
TestHeaps.java
decoder.java
encoder.java

Packages used:

```
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.nio.file.Files;
```

Node.java

This class contains the structure of a node of the Huffman Tree.

The Huffman tree is made up of objects of this class. The binary heap is also made up of these nodes.

It contains the following properties:

private Node LeftChild;
private Node RightChild;
private Integer frequency;
private Integer data;

It contains the following Methods:

- *public Node()*
  - Constructor that initializes the node .
- *public Integer getFrequency()*
  - Method to get the frequency of the node (returns the frequency)
- *public Integer getData()*
  - Method to get the data of the node (returns the data)
- *public void setData(Integer d)*
  - Method to set the data of the node.
- *public void setFrequency(Integer f)*
  - Method to set the frequency of the node.
- *public Node getLeftChild()*
  - Method to get the left child (returns the reference to left child)
- *public Node getRightChild()*
  - Method to get the left child (returns the reference to right child)
- *public void setLeftChild(Node l)*
  - Method to set the left child (sets the parameter as the left child)
- *public void setRightChild(Node r)*
  - Method to set the right child (sets the parameter as the right child)

BinaryHeap.java

This class contains the methods to implement the Binary Heap data structure. It implements a min-heap and maintains the property that the frequency of the child is always greater than the frequency of the parent.

It has the following properties:

private ArrayList<Node> heapArray; //array in which heap is stored.
private HashMap<Integer, Integer> frequencyTable; //used by buildHeap for testing.
public long startTime; //used by buildHeap method for testing.

It contains the following Methods:

- *public BinaryHeap()*
  - Constructor that initializes the heap
- *public void buildHeap(String inputFilePath)*
  - Method used by TestHeaps.java to compute the Frequency Table and insert all the data.
- *private Integer left(Integer n)*
  - Method to compute the index of the left child in the array.
- *private Integer right(Integer n)*
  - Method to compute the index of the right child in the array.
- *private Integer parent(Integer n)*
  - Method to compute the index of the parent in the array.
- *public void insert(Node number)*
  - Method to insert an element in the heap.
- *public Node extractMin()*
  - Method to remove the minimum element from heap and adjust heap accordingly.
- *private Integer minChildIndex(Integer n)*
  - Method to get the index of the minimum child.
- *private void bubbleUp(Integer n)*
  - Method to maintain heap property after insert() operation
- *public void bubbleDown (Integer numberIndex)*
  - Method to maintain heap property after extractMin() operation.
- *private void swapNodes (Integer idx1, Integer idx2)*
  - Method to swap 2 nodes in the heap.
- *public Node generateHuffmanTree()*
  - Method used by TestHeaps.java to generate the Huffman Tree.
- *public boolean isEmpty()*
  - Method to check whether heap is empty
- *public Integer getSize()*
  - Method to get the current Heap size.
- *public void generateHuffmanCode(Node root, String c, FileWriter w)*
  - Method used by TestHeaps.java to generate huffman codes from the huffman tree and print it to a file.
- *public Node createNode(Node n1, Node n2)*

- Method used by generateHuffmanTree to build merge two nodes into one and create a node for the Huffman Tree.

FourwayHeap.java

This class is used to implement the 4-way cache optimized heap. The 4-way heap has been implemented by shifting all the heap elements by 3 indices. It implements a min-heap and maintains the property that the frequency of the child is always greater than the frequency of the parent.

The class contains the following properties:

public long startTime; //used by buildHeap() to measure the time
private List<FourwayHeapNode> heap;//Arraylist that implements the Heap
private HashMap<Integer, Integer> frequencyTable;//frequency table used by buildheap() for testing.

The class contains the following Methods:

- *public FourwayHeap()*
  - Constructor that initializes the heap.
  - It adds 3 null elements to shift the heap by 3.
- *public Node generatehuffmanTree()*
  - Method used by TestHeaps.java to generate the Huffman Tree.
- *public Integer getHeapsize()*
  - Method to get the current Heap size.
- *public void generateHuffmanCode(Node root, String c, FileWriter w)*
  - Method used by TestHeaps.java to generate huffman codes from the huffman tree and print it to a file.
- *public FourwayHeapNode createNode(FourwayHeapNode n1, FourwayHeapNode n2)*
  - Method used by generatehuffmanTree to build merge two nodes into one and create a node for the Huffman Tree.
- *public FourwayHeapNode extractMin()*
  - Method to remove the minimum element from heap and adjust heap accordingly.
- *public void buildHeap(String inputFilePath)*
  - Method used by TestHeaps.java to compute the Frequency Table and insert all the data.
- *public void insert(FourwayHeapNode number)*
  - Method to insert an element in the heap.
- *public void minHeapify(Integer numberIndex)*
  - Method to maintain heap property after extractMin() operation.
- *private void swapNodes(Integer node1Idx, Integer node2Idx)*
  - Method to swap 2 nodes in the heap.
- *public Integer getChild1Idx(Integer idx)*
  - Method to compute the index of the child1 in the array.
- *public Integer getChild2Idx(Integer idx)*
  - Method to compute the index of the child2 in the array.
- *public Integer getChild3Idx(Integer idx)*
  - Method to compute the index of the child3 in the array.
- *public Integer getChild4Idx(Integer idx)*
  - Method to compute the index of the child4 in the array.
- *public Integer getParentIdx(Integer idx)*
  - Method to compute the index of the parent in the array.
- *private Integer getminChildIdx(Integer numberIndex)*

- Method to get the index of the minimum child.

FourwayHeapNode.java

The 4-way heap is implemented using an arraylist of objects of this class. It encapsulates the node to implement the Huffman tree.

The class contains the following properties

private Node node;

It contains the following methods:

- *public FourwayHeapNode ()*
  - Constructor that initializes the node.
- *public Integer getFrequency()*
  - Method to get the frequency of the node (returns the frequency)
- *public Integer getData()*
  - Method to get the data of the node (returns the data)
- *public void setData(Integer d)*
  - Method to set the data of the node.
- *public void setFrequency(Integer f)*
  - Method to set the frequency of the node.
- *public Node getLeftChild()*
  - Method to get the left child (returns the reference to left child)
- *public Node getRightChild()*
  - Method to get the left child (returns the reference to right child)
- *public void setLeftChild(Node l)*
  - Method to set the left child (sets the parameter as the left child)
- *public void setRightChild(Node r)*
  - Method to set the right child (sets the parameter as the right child
- *public Node getNode()*
  - Method to get the object of Node class stored inside the FourwayHeapnode.

PairingHeap.java

This class implements the pairing heap. It implements a min-heap and maintains the property that the frequency of the child is always greater than the frequency of the parent.

The class contains the following properties:

public long startTime;//used by buildHeap() for testing.
private Integer size;//current size of the Heap is maintained in this variable
private HashMap<Integer, Integer> frequencyTable; //used by buildHeap() for testing
private ArrayList<PairingHeapNode> fifoQueue;//used by meld() to meld children
private PairingHeapNode root;// the root of the heap is maintained in this reference.

It also contains the following methods:

- *public PairingHeap()*
  - Constructor that initializes the Heap.
- *public PairingHeapNode meld(PairingHeapNode n)*
  - Method that splits all children and melds them together using a FIFO queue and returns the melded tree.
- *public void buildHeap(String inputFilePath)*
  - Method used by TestHeaps.java to compute the Frequency Table and insert all the data.
- *public PairingHeapNode extractMin()*
  - Method to remove the minimum element from heap and adjust heap accordingly.
- *public void insert(PairingHeapNode number)*
  - Method to insert an element in the heap.
- *public PairingHeapNode getRoot()*
  - Method to get the current root of the heap.
- *public Node huffmanTreeGenerate()*
  - Method used by TestHeaps.java to generate the Huffman Tree.
- *public void generateHuffmanCode(Node root, String c, FileWriter w)*
  - Method used by TestHeaps.java to generate huffman codes from the huffman tree and print it to a file.
- *public PairingHeapNode createNode(PairingHeapNode n1, PairingHeapNode n2)*
  - Method used by generatehuffmanTree to build merge two nodes into one and create a node for the Huffman Tree.
- *public Integer getHeapsize()*
  - Method to get the current Heap size.

PairingHeapNode.java

This class contains the node structure of a pairing heap. It encapsulates the node to implement the Huffman Tree.

- *public FourwayHeapNode ()*
  - Constructor that initializes the node.
- *public Integer getFrequency()*
  - Method to get the frequency of the node (returns the frequency)
- *public Integer getData()*
  - Method to get the data of the node (returns the data)
- *public void setData(Integer d)*
  - Method to set the data of the node.
- *public void setFrequency(Integer f)*
  - Method to set the frequency of the node.
- *public Node getLeftChild()*
  - Method to get the left child (returns the reference to left child in the Huffman Tree)
- *public Node getRightChild()*
  - Method to get the left child (returns the reference to right child in the Huffman Tree)
- *public void setLeftChild(Node l)*
  - Method to set the left child (sets the parameter as the left child in the Huffman Tree)
- *public void setRightChild(Node r)*
  - Method to set the right child (sets the parameter as the right child in the Huffman Tree)
- *public Node getNode()*
  - Method to get the object of Node class stored inside the PairingHeapNode.
- *public PairingHeapNode getRight()*
  - Method to get the right of the pairing heap node in the pairing heap.
- *public PairingHeapNode getLeft()*
  - Method to get the left of the pairing heap node in the pairing heap.
- *public void setLeft(PairingHeapNode l)*
  - Method to set the left of the pairing heap node in the pairing heap.
- *public void setRight(PairingHeapNode r)*
  - Method to set the right of the pairing heap node in the pairing heap.
- *public PairingHeapNode getChild()*
  - Method to get the child of the pairing heap node in the pairing heap.
- *public void setChild(PairingHeapNode c)*
  - Method to set the child of the pairing heap node in the pairing heap.

TestHeaps.java

This class contains a *main()* method to test all the heaps and find out the fastest heap. The *Main()* method instantiates all the Heaps and computes the code tables using the buildHeap() methods, *generateHuffmanTree()*, *generateHuffmancodes()* method defined for each heap.

encoder.java

This class contains the methods to implement the encoder.

The class contains the following properties:

private static String inputFile

The class also implements the following Methods:

- *private static HashMap <Integer, Integer> buildFrequncyTable(String inputFilePath)*
    - This method opens the file and populates a Frequency Table and returns it.
- *private static Node generateHuffmanTree(BinaryHeap bh)*
    - This method generates a Huffman tree by using binary heap as a priority queue. It repeatedly extracts minimum elements from the heap using *extractMin()*, calls the *createHuffmanNode()* method which returns a Huffman tree node. This node is inserted back into the heap and successively 2 new nodes are extracted from the heap. This is done until only one element is left in the heap and that node is returned to the calling Method.
- *private static Node createHuffmanNode(Node n1, Node n2)*
    - This method creates a new Node and sets its frequency as the sum of the frequencies of n1 and n2. It then sets the new Node's left and right child pointers to n1 and n2 respectively.
- *private static void generateHuffmanCode(Node root, String c, HashMap<Integer, String> codeTable, FileWriter codeTableFile)*
    - This method traverses through the Huffman Tree inorder and along the way it computes a code for each leaf node. For each node, the path from root '0' is assigned to each left navigation and a '1' is assigned for each right navigation.
- *public static void encode()*
    - This method first instantiates a heap. Then it calls the build frequency table method to obtain the frequency table. Then it reads all the frequency table entries. For each entry it encapsulates the data and frequency into an object of Node class and inserts this object into the heap. Then it calls the *generateHuffmanTree()* method and obtains the root of the generated Huffman tree. It then calls the *generateHuffman code()* and obtains a code table as a HashMap. It then opens the inputFile and then for each line of data it obtains the code from code table and appends it to a single string. Finally, it reads this string character by character and groups 8 characters together into a byte array. And, then finally it traverses through this byte array and writes each byte into a binary file.

decoder.java

The class contains the following properties:

private static String _codeTable; // The code table that is input to this program.
private static String _encodedFile; // The encoded file that is input to the program.

The class contains the following Methods:

- *private static Node generateHuffmanTree(String codeTablePath)*
  - This is the method that reads the codetable file and generates the decode tree.
- *private static void printHuffmanTree(Node root)*
  - This method takes the root of the Huffman tree and traverses it inorder and prints each elements. This method is used for testing purposes that prints the tree using inorder traversal.
- *public static String getEncodedString(String encodedPath)*
  - This method translates the encoded binary file and returns the complete file contents as a single string.
- *private static void decode()*
  - This method first calls the generateHuffmanTree() method that generates the decode tree. It calls the getEncodedString() method to get the encoded file contents as a string. Finally, it traverses through the encoded file and simultaneously traverses through the decode tree to obtain the decoded data and writes it to a file.

Performance Analysis

Upon execution the following was the time obtained for tree generation using different heaps.

Heap Name:	Time for tree generation (miliseconds)
Binary Heap	1582
4-way cache optimized heap	2031
Pairing Heap	189573

Pairing Heap takes a lot of time because compared to the other two heaps as the melding operation was implemented using a FIFO queue. The first extractmin() was observed to consume a lot of time. It reworked the complete Heap and subsequent extractmin() operations were observed to be executed faster. All pairing heap operations (other than extractMin() and remove()) can be done in O(1) time.The complexity of extractmin() in pairing heap is O(n) because the number of subtrees that have to be combined following the removal of a node is O(n). Whereas the complexity of extractMin() is $O(4 \log_4 n)$ in 4-way heap and $O(\log_2 n)$ in Binary Heap.

Theoretically, 4-way cache optimized heap should have been faster than binary heap. However, across multiple executions the performance was very similar to Binary Heap. But it was slower than Binary Heap most of the times. And hence, Binary Heap was chosen. There were times when 4-way cache optimized heap gave better performance. But depending on how busy the processor was during the time of execution, the results varied. My hypothesis for this contradiction is that the 4-way heap was not optimized for the processor on which I was testing the program on.

Algorithm used for Decoding:

Algorithm for generating decode tree.

We initially create a root node of the tree. For each entry in the code table we extract the data and the code. We then traverse through the code bit by bit. Correspondingly, we traverse through the Decode tree starting from the root. If the code bit is '1' we traverse to the right child of the current node and the code bit is '0' we traverse left child of the current node. For each bit we before traversing we check whether a child node exists and if the validation fails, we create the child node and traverse to that node and successively keep on doing this until we reach the end of the code. At that node, we store the data extracted from the code table.After traversing through the entire code table. We would generate the Decode tree.

If we think of the algorithm as a function of height of the tree and the number of entries in the code table. The complexity of the algorithm is O(nh) where n is the height of the tree. And n is the number of entries in the code table. The maximum height of the tree will be n-1 in a completely skewed decode tree. Therefore, the complexity will be $O(n(n-1)) = O(n^2)$.
The complexity can also be thought of O(C) where C is the sum of all lengths of the codes in the code table.

The following is the code of the algorithm in java.

private static Node generateHuffmanTree(String codeTablePath) **throws** NumberFormatException, IOException{

```
            BufferedReader br = new BufferedReader(new
FileReader(codeTablePath));
            String line;
            Node root = new Node();
            Node traverser;
          while ((line = br.readLine())!= null) {
            if(!line.isEmpty() && line !=null){
                  Integer number = Integer.parseInt(line.split(" ")[0]);
                  String code = line.split(" ")[1];
                  traverser = root;
                  for(int i=0; i< code.length()-1;i++){
                        if(code.charAt(i) == '0'){
                              if(traverser.getLeftChild() == null){
                                    traverser.setLeftChild(new Node());

                              }
                              traverser = traverser.getLeftChild();

                        }
                        else if(code.charAt(i) == '1'){
                              if(traverser.getRightChild() == null){
                                    traverser.setRightChild(new Node());

                              }
                              traverser = traverser.getRightChild();
                        }
                  }
                  if(code.charAt(code.length()-1) == '0'){
                        traverser.setLeftChild(new Node());
```

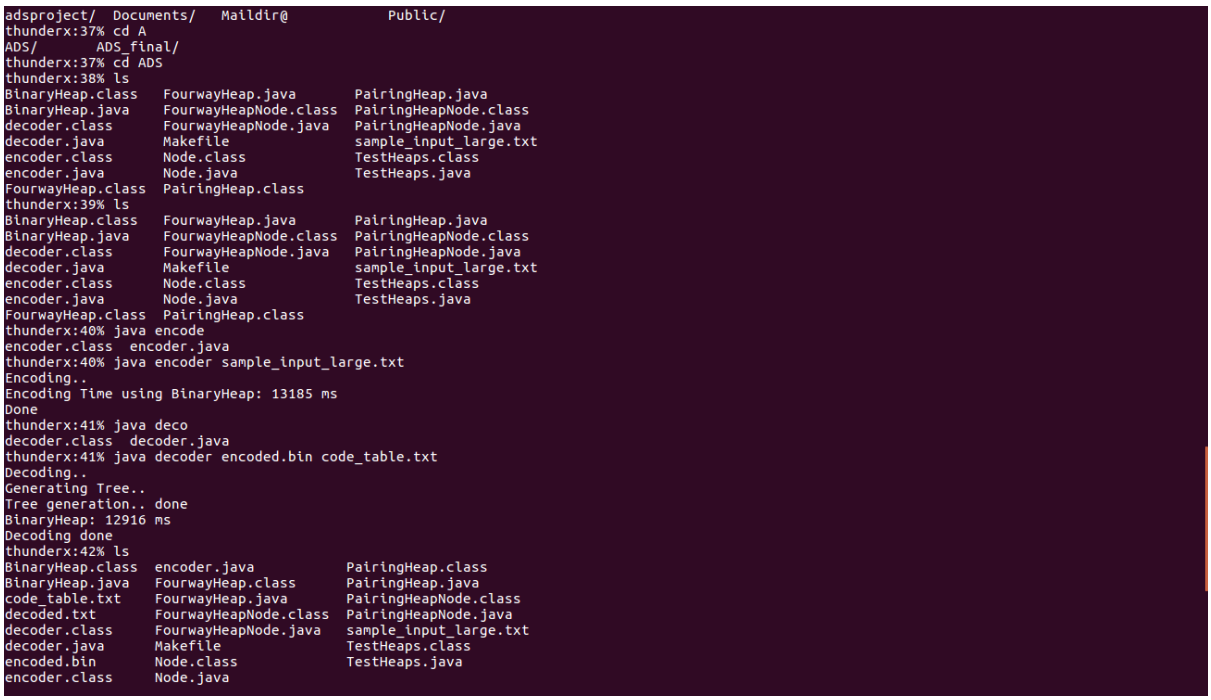
```
                        traverser = traverser.getLeftChild();
                        traverser.setData(number);
                }
                else{
                        traverser.setRightChild(new Node());
                        traverser = traverser.getRightChild();
                        traverser.setData(number);
                }
                }
        }
        System.out.println("Tree.. done");
        return root;

}
```

Algorithm Used for Decoding.

We read all the contents of the binary file bit by bit and traverse through the decode the tree until we fall of the tree. The moment we fall of the tree we write the data at the corresponding node in the decode tree and start the traversal from the root's leftchild or rightchild.
This will always work because of the property of Huffman codes that no code is a perfect prefix of any other code generated from the same tree.

The complexity of this algorithm is O(n) +O(C) where n is the number of bits in the input (encoded file) and C is the sum of all lengths of all Huffman codes in the code table.

Screenshot



```
adsproject/  Documents/   Maildir@            Public/
thunderx:37% cd A
ADS/       ADS_final/
thunderx:37% cd ADS
thunderx:38% ls
BinaryHeap.class    FourwayHeap.java       PairingHeap.java
BinaryHeap.java     FourwayHeapNode.class  PairingHeapNode.class
decoder.class       FourwayHeapNode.java   PairingHeapNode.java
decoder.java        Makefile               sample_input_large.txt
encoder.class       Node.class             TestHeaps.class
encoder.java        Node.java              TestHeaps.java
FourwayHeap.class   PairingHeap.class
thunderx:39% ls
BinaryHeap.class    FourwayHeap.java       PairingHeap.java
BinaryHeap.java     FourwayHeapNode.class  PairingHeapNode.class
decoder.class       FourwayHeapNode.java   PairingHeapNode.java
decoder.java        Makefile               sample_input_large.txt
encoder.class       Node.class             TestHeaps.class
encoder.java        Node.java              TestHeaps.java
FourwayHeap.class   PairingHeap.class
thunderx:40% java encode
encoder.class  encoder.java
thunderx:40% java encoder sample_input_large.txt
Encoding..
Encoding Time using BinaryHeap: 13185 ms
Done
thunderx:41% java deco
decoder.class  decoder.java
thunderx:41% java decoder encoded.bin code_table.txt
Decoding..
Generating Tree..
Tree generation.. done
BinaryHeap: 12916 ms
Decoding done
thunderx:42% ls
BinaryHeap.class  encoder.java           PairingHeap.class
BinaryHeap.java   FourwayHeap.class      PairingHeap.java
code_table.txt    FourwayHeap.java       PairingHeapNode.class
decoded.txt       FourwayHeapNode.class  PairingHeapNode.java
decoder.class     FourwayHeapNode.java   sample_input_large.txt
decoder.java      Makefile               TestHeaps.class
encoded.bin       Node.class             TestHeaps.java
encoder.class     Node.java
```

Conclusion

Huffman codes are an effective way to reduce the size of the data to be transferred. The successful execution of the program resulted in a given sample file of 6,96,38,842 bytes to be compressed to a file size of 2,46,27,695 bytes.

The tests conducted unveiled Binary Heap to be the fastest data structure to implement the priority queue for generating the Huffman tree and was chosen to be implement the encoder.