

# **Advanced Data Structures (COP 5536)**

## **Programming Project Report**

### **Huffman Coding**

Archana Nagarajan

UFID: 14569491

University of Florida

Department of Computer & Information Science & Engineering

Gainesville, FL 32611, USA

archana.nagaraj@ufl.edu

April 6, 2017

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose . . . . .	1
<b>2</b>	<b>Performance Analysis</b>	<b>1</b>
<b>3</b>	<b>Function Prototypes and Structure of the Program</b>	<b>1</b>
3.1	HeapNode.java . . . . .	1
3.2	Heaps.java . . . . .	2
3.3	encoder.java . . . . .	2
3.4	BinaryHeap.java . . . . .	2
3.5	FourWayCacheOptimizedHeap.java . . . . .	3
3.6	PairingHeap.java . . . . .	4
3.7	decoder.java . . . . .	4
<b>4</b>	<b>Decoding Algorithm</b>	<b>5</b>
<b>5</b>	<b>Results</b>	<b>5</b>
<b>6</b>	<b>Analysis and Conclusion</b>	<b>6</b>

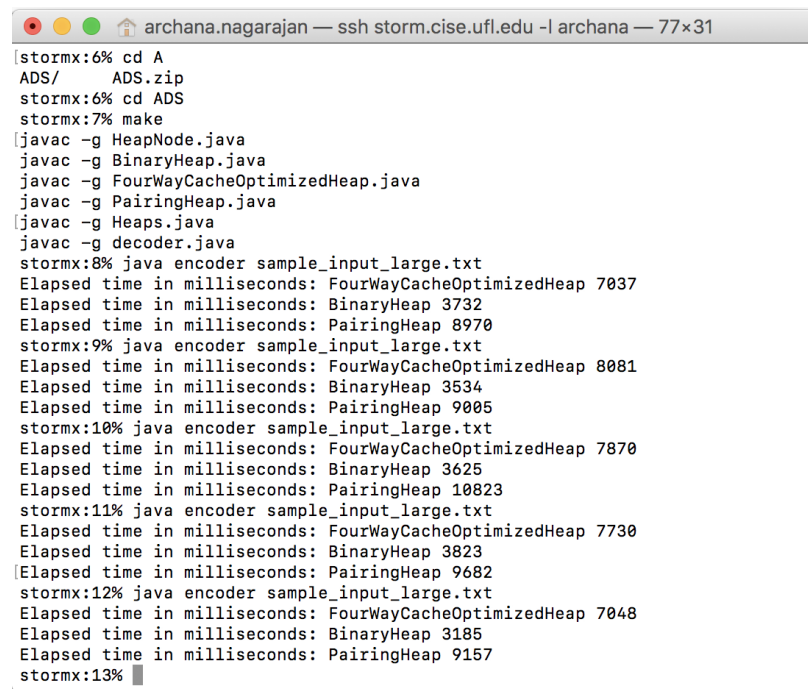
# 1 Introduction

## 1.1 Purpose

The purpose of this project is to implement Huffman Tree coding using the best performing priority queue data structure. Priority queues were implemented using Binary heap, Four way cache optimized heap and Pairing heap. A text file containing the message to be sent was given. This data needs to be encoded at the sender's end and decoded at the receiver's end.

## 2 Performance Analysis

Out of the 3 heaps, the best performing heap while constructing the heap and its corresponding Huffman tree was Binary heap. Please find below the analysis of performance of the 3 heaps.



```
archana.nagarajan — ssh storm.cise.ufl.edu -l archana — 77x31
stormx:6% cd A
ADS/ ADS.zip
stormx:6% cd ADS
stormx:7% make
[javac -g HeapNode.java
javac -g BinaryHeap.java
javac -g FourWayCacheOptimizedHeap.java
javac -g PairingHeap.java
javac -g Heaps.java
javac -g decoder.java
stormx:8% java encoder sample_input_large.txt
Elapsed time in milliseconds: FourWayCacheOptimizedHeap 7037
Elapsed time in milliseconds: BinaryHeap 3732
Elapsed time in milliseconds: PairingHeap 8970
stormx:9% java encoder sample_input_large.txt
Elapsed time in milliseconds: FourWayCacheOptimizedHeap 8081
Elapsed time in milliseconds: BinaryHeap 3534
Elapsed time in milliseconds: PairingHeap 9005
stormx:10% java encoder sample_input_large.txt
Elapsed time in milliseconds: FourWayCacheOptimizedHeap 7870
Elapsed time in milliseconds: BinaryHeap 3625
Elapsed time in milliseconds: PairingHeap 10823
stormx:11% java encoder sample_input_large.txt
Elapsed time in milliseconds: FourWayCacheOptimizedHeap 7730
Elapsed time in milliseconds: BinaryHeap 3823
Elapsed time in milliseconds: PairingHeap 9682
stormx:12% java encoder sample_input_large.txt
Elapsed time in milliseconds: FourWayCacheOptimizedHeap 7048
Elapsed time in milliseconds: BinaryHeap 3185
Elapsed time in milliseconds: PairingHeap 9157
stormx:13%
```

In the above figure the time taken by to construct the heaps and then the Huffman trees are given. The time shown in the above figure is the time taken for 10 iterations for each heap. It can be observed that, of the multiple times the construction was done, it was always the fastest in the case of binary heap. Binary heap takes about 3-4 seconds on an average for 10 iterations whereas pairing heap takes 8-10 seconds and Four way cache optimized heap takes 7-8 seconds.

## 3 Function Prototypes and Structure of the Program

There are a total of 7 classes. Below are the methods and the structure of each class:

### 3.1 HeapNode.java

This is a bean class. This has 4 class variables namely data, frequency, left and right and their corresponding getter, setter methods.

- **data : integer** : Holds the data of the node.
- **frequency : integer** : Holds the frequency(number of occurrence of the message in the file) of the node
- **left : HeapNode** : Holds pointer to its left child.
- **right : HeapNode** : Holds pointer to its right child.

### 3.2 Heaps.java

This class contains methods corresponding to all the 3 heaps. It is responsible for calling the corresponding methods for each heap. In a way, this is a pseudo driver class. The class methods are listed as below:

- **constructBinaryHeap(Map<Integer,Integer> frequencyMap) : void** - This methods calls a set of functions sequentially. Calls are made to methods to build the heap, construct the Huffman tree, traverse the tree, encode the data, write the encoded data and codetable to file.
- **constructFourWayCacheOptimizedHeap(Map<Integer,Integer> frequencyMap) : void** - This methods calls a set of functions sequentially. Calls are made to methods to build the heap, construct the Huffman tree, traverse the tree, encode the data, write the encoded data and codetable to file.
- **constructPairingHeap(Map<Integer,Integer> frequencyMap) : void** - This methods calls a set of functions sequentially. Calls are made to methods to build the heap, construct the Huffman tree, traverse the tree, encode the data, write the encoded data and codetable to file.

### 3.3 encoder.java

This is the encoder driver class. It is executed when to obtain the code table and the encoded.bin files. The functions defined in this class are as follows:

- **main(String[] args) : void** - The main method reads the data from the file and stores in a HashMap.
- **writeCodeTableToFile(Map<String,String> codeTableMap) : void** - This methods writes the code table generated from the encoding step to a file named code\_table.txt.
- **writeEncodedDataToFile(String encodedString) : void** - This methods writes the encoded data to a file called encoded.bin.
- **encodeData(Map<String,String> codeTableMap) : String** : This method reads the codeTableMap which contains the code corresponding to each element and encodes the data and sends back a string of encoded data which is then written to the file using writeEncodedDataToFile(String encodedString) method.

### 3.4 BinaryHeap.java

This class contains all the methods relating to Binary heap. This also contains methods to construct and traverse the Huffman tree using the heap.

#### Class Variables:

- **children : int** - This is a final variable. This stores the number of children that a node in a binary heap can have which is 2.
- **heap : List< HeapNode >** - This list represents the heap.

#### Functions:

- **children : int** - This is a final variable. This stores the number of children that a node in a binary heap can have which is 2.
- **printHeap() : void** - This method is used to print the heap.

- **buildHeap(Map<Integer,Integer> frequencyMap) : void** - This method is used to build the heap by iterating over the values in the frequencyMap.
- **getParent(int child) : integer** - This returns the index of the parent of a given node. **Complexity : O(1)**
- **getNthChild(int parent, int n) : integer** - This returns the index of the nth child of the given parent node. **Complexity : O(1)**
- **insert(HeapNode element) : void** - This methods inserts element into the heap. **Complexity : O(log n)**
- **heapifyInsert(int value) : void** - This method is used to move the newly inserted element into its correct position in the heap.
- **isEmpty() : boolean** - This methods returns true if the heap is empty. **Complexity : O(1)**
- **removeMin() : HeapNode** - This method returns the root of the heap. **Complexity : O(log n)**
- **heapifyRemoveMin(int minIndex) : void** - This method is used to readjust the heap when the root is removed by the removeMin operation.
- **getMinChild(int minIndex) : integer** - This method returns the index of the child with minimum frequency given a parent. **Complexity : O(1)**
- **constructHuffmanTree() : HeapNode** - This method is used to construct the Huffman tree. It returns the root of the Huffman Tree. **Complexity : O(n)**
- **traverseHuffmanTree(HeapNode root, String item,Map< String,String > codeTableMap) : void** - This method is used to traverse the Huffman Tree to assign a code to every leaf node which contains the elements in the text file. **Complexity : O(n)**

### 3.5 FourWayCacheOptimizedHeap.java

This class contains all the methods relating to Four way cache optimized heap. This also contains methods to construct and traverse the Huffman tree using the heap.

#### Class Variables:

- **children : int** - This is a final variable. This stores the number of children that a node in a binary heap can have which is 2.
- **heap : List< HeapNode >** - This list represents the heap.

#### Functions:

- **printHeap() : void** - This method is used to print the heap.
- **buildHeap(Map<Integer,Integer> frequencyMap) : void** - This method is used to build the heap by iterating over the values in the frequencyMap.
- **getParent(int child) : integer** - This returns the index of the parent of a given node. **Complexity : O(1)**
- **getNthChild(int parent, int n) : integer** - This returns the index of the nth child of the given parent node. **Complexity : O(1)**
- **insert(HeapNode element) : void** - This methods inserts element into the heap. **Complexity : O(log n)**
- **heapifyInsert(int value) : void** - This method is used to move the newly inserted element into its correct position in the heap.
- **isEmpty() : boolean** - This methods returns true if the heap is empty. **Complexity : O(1)**
- **removeMin() : HeapNode** - This method returns the root of the heap. **Complexity : O(log n)**

- **heapifyRemoveMin(int minIndex) : void** - This method is used to readjust the heap when the root is removed by the removeMin operation.
- **getMinChild(int minIndex) : integer** - This method returns the index of the child with minimum frequency given a parent.
- **constructHuffmanTree() : HeapNode** - This method is used to construct the Huffman tree. It returns the root of the Huffman Tree. **Complexity : O(n)**

### 3.6 PairingHeap.java

This class contains all the methods relating to Pairing heap. This also contains methods to construct and traverse the Huffman tree using the heap.

**Class Variables:**

- **size : int** - This is used to store the size of the array.
- **pairArray : Array of HeapNodes** - This list represents the heap.

**Functions:**

- **printHeap() : void** - This method is used to print the heap.
- **buildHeap(Map<Integer,Integer> frequencyMap) : void** - This method is used to build the heap by iterating over the values in the frequencyMap.
- **add(HeapNode element) : void** - This methods inserts element into the heap. **Complexity : O(log n)**
- **isEmpty() : boolean** - This methods returns true if the heap is empty. **Complexity : O(1)**
- **removeMin() : HeapNode** - This method returns the root of the heap. **Complexity : O(log n)**
- **constructHuffmanTree() : HeapNode** - This method is used to construct the Huffman tree. It returns the root of the Huffman Tree. **Complexity : O(n)**
- **getMinChild(int minIndex) : integer** - This method returns the index of the child with minimum frequency given a parent.
- **compareAndLink( HeapNode first, HeapNode second ) : HeapNode** - This method is used to insert the element as the leftmost element of the root and adjust its corresponding sibling pointers. **Complexity : O(1)**
- **combineSiblings( PairingHeapNode firstSibling ) : HeapNode** - This method is used to rearrange the elements of the heap when a removeMin() operation is performed. **Complexity : O(m)** where m is the number of child nodes

### 3.7 decoder.java

This is the decoder driver class. This contains methods that reads the code table and the encoded text to reconstruct the message. The functions defined in the class are as follows:

- **main(String args[]) : void** - This main method is takes as input the encoded.bin and code\_table.txt files and calls the methods that reconstruct the tree, decodes the message and writes it to a file name decoded.txt
- **reconstructHuffmanTree(Map< String,String >codeMap) : integer** - This method is used to reconstruct the Huffman tree from the code table given as input.
- **readEncodedMessage(String encodedBin) : String** - This method is used to read the encoded.bin file and generate the encoded text which it returns as a String.
- **decodeMessage(String encodedBin) : List< String >** - This method is used to traverse the tree using the encoded data obtained and list down the elements which will be same as the elements in the sample input file given.
- **writeDecodedMessageToFile(List< String > decodedMessage) : void** - This method is used to write the decoded list to a file called decoded.txt.

## 4 Decoding Algorithm

- The bits are combined 8 each to form a byte which is written to the file.
- The decoder reconstructs the Huffman tree from the code table given.
- The encoded.bin file is converted to a String and is parsed using the Huffman tree to obtain the leaf elements in order.
- Left of the node is traversed if the bit is a '0' and right of the node is traversed if the bit is a '1'.
- It is then written to the decoded.txt file which will match the input file given to the encoder.

**The complexity associated with this algorithm is:**

- Reconstructing tree:  $O(m)$  where  $m$  is the number of bits in the encoded string.
- Decode message :  $O(n)$  where  $n$  is the number of nodes in the Huffman tree. As this requires that we traverse all the root to leaf paths in the tree.
- Hence the complexity of the decoding algorithm is  $O(n+m)$

## 5 Results

Below are the results obtained from the encoder and decoder.

- The size of the encoded.bin matches the size of the encoded.bin given.

```
archana.nagarajan — ssh storm.cise.ufl.edu -l archana — 114x32
device. All activities performed on this device
are logged. Any violations of access policy can
result in disciplinary action.
-----
archana@storm.cise.ufl.edu's password:
Last login: Wed Apr  5 18:38:04 2017 from ip70-171-42-151.ga.at.cox.net
stormx:1% cd A
ADS/ ADS.zip
stormx:1% cd ADS
stormx:2% make
javac -g encoder.java
Error occurred during initialization of VM
java.lang.OutOfMemoryError: unable to create new native thread
make: *** [encoder.class] Error 1
stormx:3% make
javac -g encoder.java
Error occurred during initialization of VM
java.lang.OutOfMemoryError: unable to create new native thread
make: *** [encoder.class] Error 1
stormx:4% make
javac -g encoder.java
stormx:5% java encoder sample_input_large.txt
stormx:6% ls
BinaryHeap.class  encoded.bin          HeapNode.class  PairingHeap.class
BinaryHeap.java  encoder.class        HeapNode.java   PairingHeap.java
code_table.txt   encoder.java         Heaps.class     PairingHeap$PairingHeapNode.class
decoder.class    FourWayCacheOptimizedHeap.class  Heaps.java     sample_input_large.txt
decoder.java     FourWayCacheOptimizedHeap.java   Makefile       sample_input_small.txt
stormx:7% ls -l encoded.bin
-rw-r--r-- 1 archana grad 24627695 Apr  5 18:46 encoded.bin
stormx:8%
```

- The input and the output file matches.

```

[stormx:6% make clean
rm -f *.class
[stormx:7% make
javac -g HeapNode.java
javac -g BinaryHeap.java
javac -g FourWayCacheOptimizedHeap.java
javac -g PairingHeap.java
javac -g encoder.java
javac -g decoder.java
[stormx:8% java encoder sample_input_large.txt
Elapsed time in milliseconds: BinaryHeap 29259
[stormx:9% java decoder encoded.bin code_table.txt
Elapsed time in milliseconds: Decoding and writing to file 14095
[stormx:10% ls
BinaryHeap.class    encoder.class        Heaps.java
BinaryHeap.java    encoder.java         Makefile
code_table.txt     FourWayCacheOptimizedHeap.class  PairingHeap.class
decoded.txt        FourWayCacheOptimizedHeap.java   PairingHeap.java
decoder.class      HeapNode.class       PairingHeap$PairingHeapNode.class
decoder.java       HeapNode.java        sample_input_large.txt
encoded.bin        Heaps.class          sample_input_small.txt
[stormx:11% ls -l encoded.bin
-rw-----+ 1 archana grad 24627695 Apr  5 22:46 encoded.bin
[stormx:12% diff decoded.txt sample_input_large.txt
[stormx:13%

```

## 6 Analysis and Conclusion

As seen from the results, the time taken to encode is approximately 29 seconds and the time taken to decode is approximately 14 seconds. This is in the case with a file with 10 million records. The time achieved is because of binary heap being the underlying data structure. Binary heaps are a form of balanced binary search trees with the root element always being minimum in case of a min heap. This is the most important reason why binary heaps are faster. Time to find a node will be  $O(\log n)$  in the worst case. In case of pairing heap, there is no restricting to the number of children a node can have and hence one needs to traverse all the siblings to find a node which can be  $O(n)$  in worst case, if all the nodes are in the same level. Cache optimized four way heap is better than four way heap when it comes to cache misses. But the performance is still not good as a binary heap although it is closer.

Thus it can be concluded that binary heap is the most efficient heap when handling such large files.