COP 5536 Advance Data Structures

Spring 2017

Programming Project Report

Huffman Encoder and Decoder

Jay Shah

UFID: 14613930

Jayshah953010@ufl.edu

Table of Contents

1. Goals of the project

- Implement Huffman tree using three priority queue data structures binary heap, 4-way cache optimized heap and pairing heaps.

- Evaluate performance of each algorithm used to implement three data structures in order to find out best fit to implement Huffman tree.

2. Programming Environment

Language : Java

Operating System: Linux Ubuntu, Windows

Data Structure Used: 4-way heap

3. Function Prototypes

3.1 Class Encoder:

Encoder Data Variables		
Name	**Type**	**Description**
Length	Int	Denotes the heap size
hm1	HashMap	Used to store code table for input data
S	StringBuilder	Temporarily used to data bits in order to store them in encoded.bin file
Line	String	Read all lines from input file
hm	HashMap	Used to store frequency of occurrence for input data
r	Input_Freq Read*	It is an object which retrieves frequency of occurrence for each data.
h	Encoder*	Initializes heap size
br	Buffered Reader	Used to read input file data
bw	Buffered Writer	Used to write code table
bw1	Buffered Writer	Used to write encoded.bin file

Public functions:

1	**Void insert (String Value , int frequency)**
	Insert a new node into the heap according to the value and frequency of an item
2	**Void goup(int length)**
	It adjust the node according to their values in heap array after inserting the data.
3	**Void buildtree()**
	Create a huffman tree with this.
4	**Void removeMin()**
	Remove the minimum element from the 4 way heap array.

5	**Int Parent(int child)**
	Used to find parent index in heap array using their child index
6	**int l1(int p), int l2(int p), int l3(int p), int l4(int p)**
	It is used to find out child index using their parent's index
7	**Void shiftDown(int nodeIndex)**
	Processes the tree and make a node with minimum frequency as a root
8	**Void print_codetable(Node root, StringBuffer s,BufferedWriter bw)**
	Generates the code table for an input data which can be used at the time of decoding
9	**Void insert(String value,int frequency,Node a,Node b)**
	Used to add two frequency of two nodes and add them in a heap array again

3.2 Class Node

Data Variables		
Value	String	Value of a node
Frequency	Integer	Frequency
Left	Node*	Left Child of a Node
Right	Node*	Rigth Child of a Node

Public functions:

1	**Boolean isLeaf()**
	Check whether node is a leaf or not
2	**Node* (String Value,Int Frequency)**
	Create a new node with specified value and frequency
3	**Node* (String Value, Int Frequency, Node Left, Node right)**
	Create a new node with specified value, frequency, left child and right child

3.3 Class Node1

Data Variables		
Name	**Type**	**Description**
Value	Frequency	Value of a node
Frequency	Integer	Frequency
Left	Node1*	Left Child of a Node
Right	Node1*	Right Child of a Node

Public functions:

1	**Boolean isLeaf()**
	Checks whether node is leaf or not
2	**Void tree_for_decoder(Map<String,String> hm, Node1 root_node)**
	Generate Huffman tree for decoding purpose
3	**Void insert1(Node1 x,char c)**
	Inset a node in a huffman tree based on a character of a string that is retrieved from an encoded.bin file

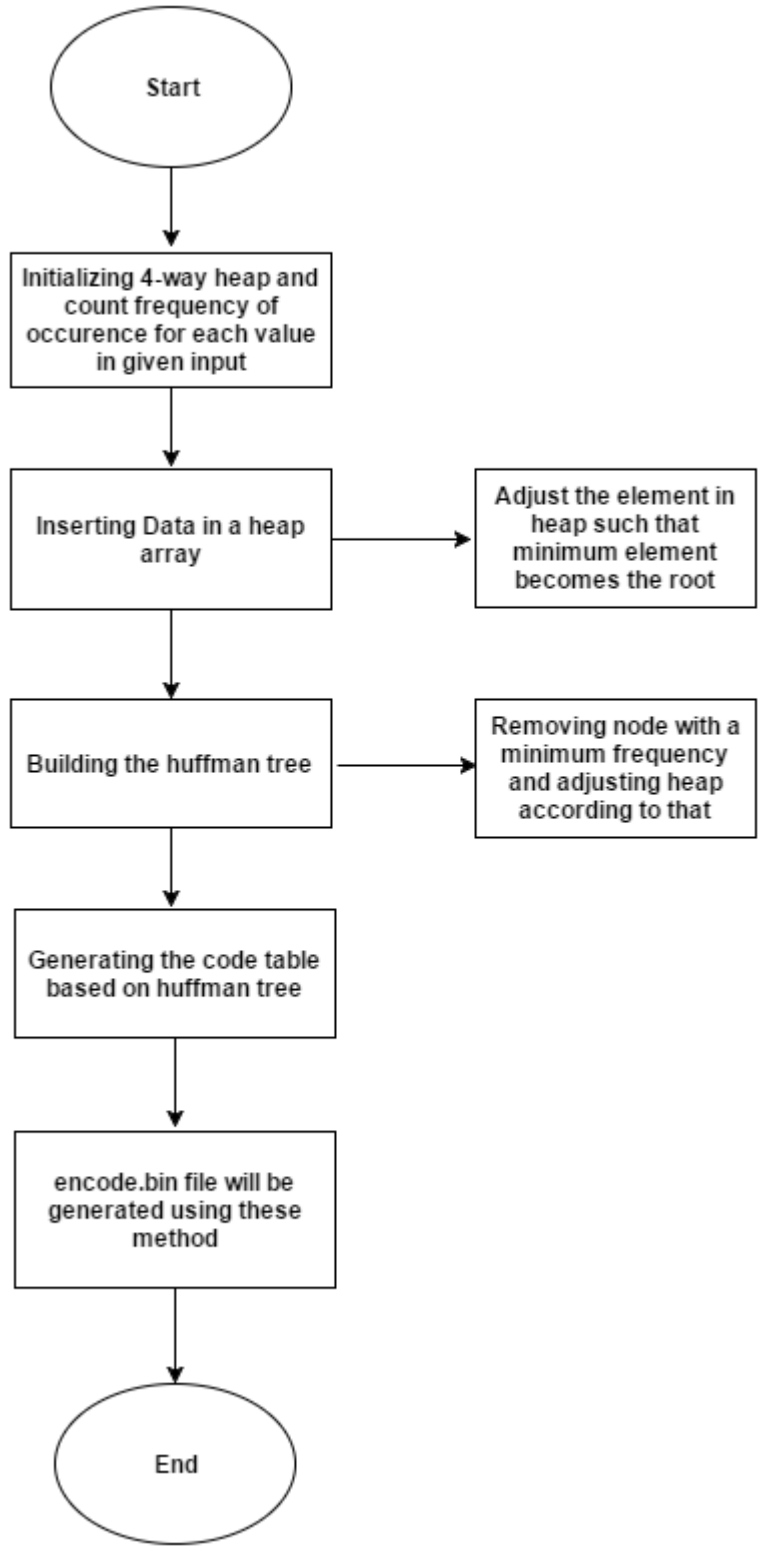
3.4 Class Decoder

Data Variables		
hm1	HashMap	Store data for a code table
root_node	Node1*	Intialize root-node for huffan tree
FileName1	String	Stores path to the encoded.bin file
FileName	String	Path for the decoded.txt file
bFile	Byte Array	Store data in a byte array form

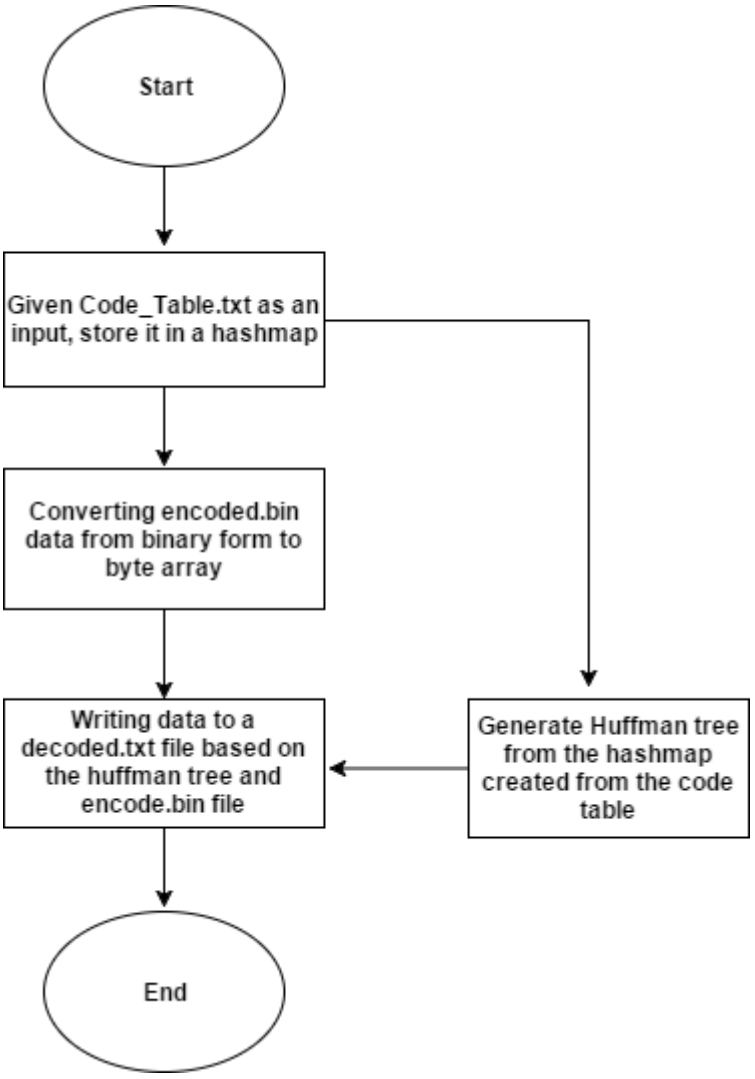
Public functions:

1	**Boolean isLeaf()**
	Checks whether node is leaf or not
2	**Void tree_for_decoder(Map<String,String> hm, Node1 root_node)**
	Generate Huffman tree for decoding purpose
3	**Void insert1(Node1 x,char c)**
	Inset a node in a huffman tree based on a character of a string that is retrieved from an encoded.bin file

4. Program Structure



1.1 Encoder.java



1.2 Decoder.java

5. Performance Analysis

Below is average time given for building a Huffman tree for a large input using three different data structures.

	Binary heap	**4-way heap**	**Min-pairing heap**
Time (Seconds)	4.54405	4.374795	5.3299

We see that 4-way cache utilization heap is faster than any other heap. Its removeMin() operation can perform 4-compares at a level rather than two comparision which decreases the number of levels in a tree to half which makes it more efficient than binary tree. Now in the min pairing heap, we don't know the number of levels and there might be a lot of comparision at a level after removeMin() operation which makes it less efficient than 4-way cache utilization heap.
That is why I have implemented 4-way cache optimization heap which seems to be more efficient than other data structures.

6. Decoding Algorithm

Given an input of a code table we can easily generate Huffman tree which can later help us to decode encoder.bin file.
Below is the piece of code used to generate Huffman tree.

```
for(String k :hm.keySet())
{
                        String tmp = hm.get(k);
                        int i=0;
                        Node1 temp = root_node;
                        while(i<tmp.length())
                        {
                                temp = insert1(temp,tmp.charAt(i));
                                i++;
                        }
                        temp.value = k;
}
public static Node1 insert1(Node1 x,char c)
{
                if(c=='0')
        {
                if(x.left==null)
                {
                        x.left = new Node1("0");
                }
                x = x.left;
        }
```

```
else
{
if(x.right==null)
        {
                x.right = new Node1("0");
        }
        x = x.right;
}
        return x;
}
```

Below is a piece of code that can be used to write decoder.txt file.

```
S3.append(file); // file is encoded.bin file
while(j<s3.length())
{
Node1 temp1 = root_node;
while(!(temp1.left==null && temp1.right==null))
{
                if(s3.charAt(j)=='0')
{
temp1 = temp1.left;
}
                else
               {
                 temp1 = temp1.right;
          }
          j++;
}
bw2.write(temp1.value);
bw2.newLine();

}
```

Time Complexity can be **O(nlogn)** where n is number of unique inputs and log n is maximum number of levels that tree has to traverse to reach to its leaf node.