COP 5536 Advanced Data Structures

Spring 2017

Programming Project Report

Huffman Coding and Decoding

Sandeep Basavaraju

UFID: 28862242

Contents

1. Goals of the project

- Implement the generation of Huffman codes using the priority queue structures: Binary Heap, 4-way cache optimized heap and PairingHeap. Evaluate the performances for each of those and select the best among those to encode and construct the Huffman tree and code table.

- Implement a decoder algorithm to decode the encoded input using the code table.

2. Associated Files

- Encoder
- Decoder
- Make file

3. Programming Environment

Operating System: Windows 8

Complier/IDE: Eclipse Neon.

Programming Language: Java 1.8.

Testing Environment: thunder.ufl.cse.edu.

4. Evaluation of different priority queue structures

The following table gives the performance of the Binary Heap, 4-way cache optimized heap and PairingHeap for building a Huffman tree. The evaluation was done on the thunder.ufl.cise.edu using java as the programming language for the file “sample_input_large.txt”.

Priority queue Structure	Time taken to build Huffman tree for 10 iterations (T seconds)	Average Time taken to build Huffman tree per iterations (T/10 seconds)
Binary Heap	13.436	1.3436
4-way cache optimized heap	11.204	1.1204
ParingHeap	21.381	2.1381

As it can be seen **4-way cache optimized heap** has the best performance for building the Huffman tree. Hence 4-way cache optimized heap is selected for generation of Huffman codes.

Screen shots :

```
stormx:16% java HuffmanCodingBinaryHeap sample_input_large.txt
13.435643021
10000000
Enterd the file creation stage
Done with flushing
```

Execution of Binary tree

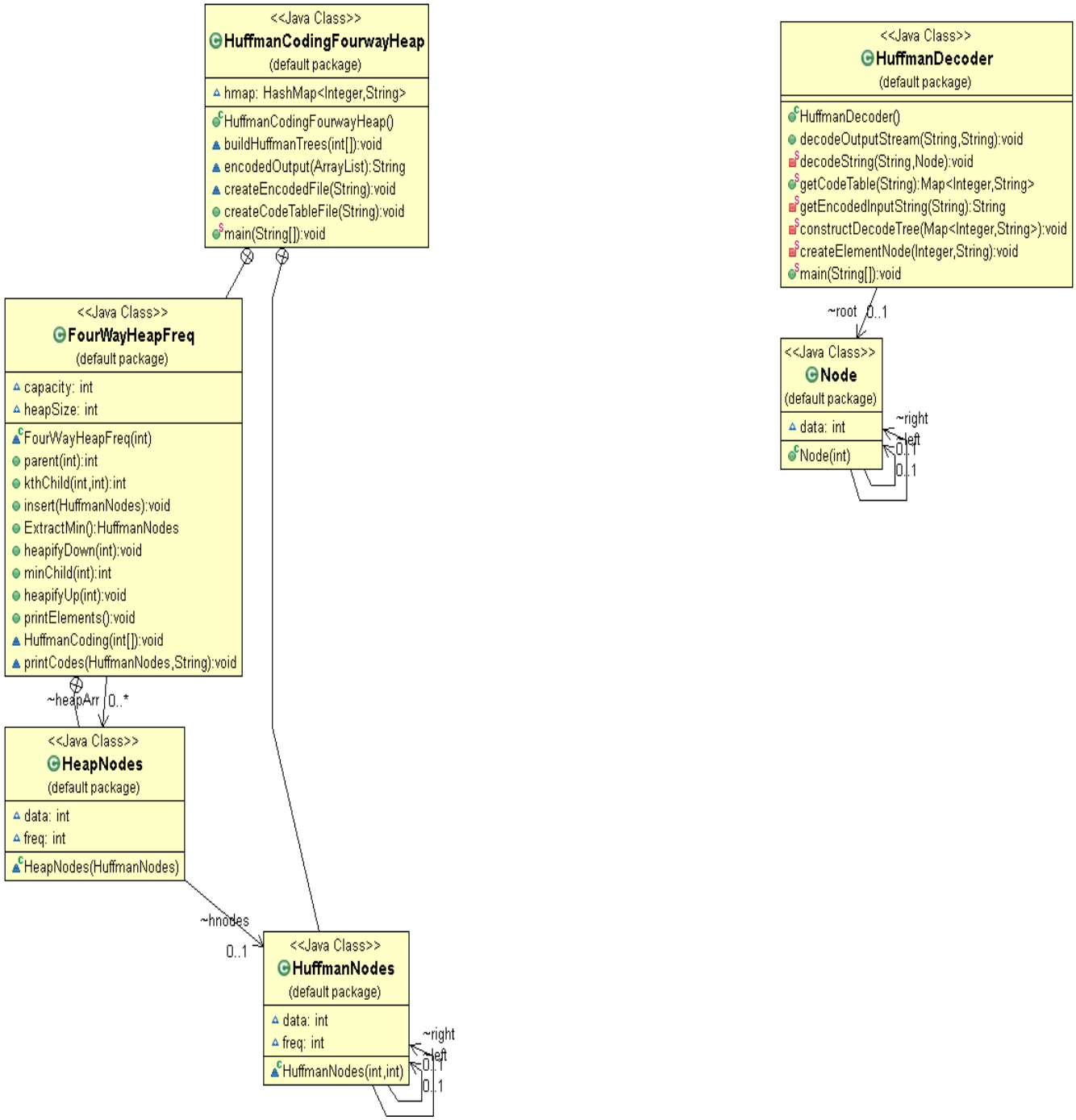
```
stormx:18% java HuffmanCodingFourwayHeap sample_input_large.txt
11.203748248
10000000
Enterd the file creation stage
```

Execution of 4-way cache optimized heap

```
thunderx:16% java HuffmanCodingWithPairingHeap sample_input_large.txt
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
21.381555841
```

Execution of PairingHeap

5. Class Diagram



6. Function Prototypes

Class FourwayHeapFreq:

1. parent (int)

Parameters: index of the node.

Return value: int.

Description: Function to get index of parent.

2. kthChild (int, int)

Parameters: index of the node and the kth child.

Return: int

Description: Function to get index of k th child of i.

3. insert (HuffmanNodes x)

Parameters: The node to be inserted.

Return: void.

Description: This method is intended to insert the nodes to Huffman tree.

4.ExtractMin ()

Parameters: None.

Return: HuffmanNodes.

Description: This methods removes the minimum value node in the Huffman tree.

5.heapifyDown(int)

Parameters: index of the node.

Return: void.

Description: After the extract min is done this method is called in order to maintain the heap structure.

6.minChild(int)

Parameters: index of the node.

Return: int

Description: Description: After the extract min is done , this method gives the minimum valued child.

7. heapifyUp(int)

Parameters: index of the node

Return: void.

Description: After the insertion is done this method is called to maintain the heap structure.

8.HuffmanCoding (int [])

Parameters: input array.

Return: void

Description: This method to merge the frequency of the left and right child.

Class: HuffmanCodingFourwayHeap

1. buildHuffmanTrees(int[])

Parameters: input array.

Return: void.

Description: Method calls HuffmanCoding () to build the tree.

2. encodedOutput (ArrayList)

Parameters: Arraylist of fileContentarr.

Return: String.

Description: Method return the encoded string.

3.createEncodedFile (String)

Parameters: Encoded string

Return: void.

Description: This method creates the endoded.bin file

4. createCodeTableFile(String)

Parameters: Input file

Return: void.

Description : The method to create the code table.

Class: HuffmanDecoder

1.decodeOutputStream (Codetable, Encode)

Parameters: code table and encodebin.

Return: void

Description: This method calls the appropriate methods to decode the encoded bin using the code table.

2. decodeString (String, Node)

Parameters: encoded string and the root

Return: void.

Description: This method helps in creation of the decoded file.

3. getCodeTable(String)

Parameters: code table as input

Return: code table;

Description: gets the code table file.

4. constructDecodeTree(Codetable)

Parameters: The code table;

Return: void.

Description: Construction of the decode tree.

5. createElementNode (Int, String)

Parameters: Key and value.

Return: void.

Description: Create an element of the tree with the key value pair.

7. Decoder Algorithm

7.1 Decode Tree Construction Algorithm:

- Read the code table text file line by line.
- Get the key and the corresponding code word from the file for a given line.
- Start traversing down the path from the root.
- While traversing down check if the path exists, if no path then creates a dummy node along the path with least integer value in the environment.
- Once you reach the end of the code word create a node with data in it. For E.g., if the code is “0100” create nodes root->left->right->left->left.
- Data is stored only on the leaf nodes, remaining all non-leaf nodes contain dummy data and are used for traversing to the leaf.

Considering that there are d elements in code table. For each element, we are traversing from the root all the way to the leaf of the tree. If on an average the codeword length is n for each of the d elements then the time complexity for construction of the Decode Tree is **d log2n**.

8. Conclusion

Based on the evaluation for the large input it can be concluded that the 4-way cache optimized heap performs better when compared to the binary heap and the pairingHeap