COP5536: Advanced Data Structures, Spring 2017
Project Report

Karthik Narayanan
UFID: 9019 0260 | Email: karthikn@ufl.edu

Compiling Instructions:

This project has been written in Java and can be compiled using standard JDK with javac compiler.

Unzip the submitted compressed file. It contains program files (.java) and the makefile. The input files must be placed in the same directory as the extracted files before running the program.

To build the code in the Unix environment, run the following command after changing the directory to where the extracted files are.

$ make

Run the encoder and decoder using the following commands:

$ java encoder <input_file_name>
$ java decoder <encoded_file_name> <code_table_file_name>

Class Definitions and Function Prototypes:

The classes and their respective function prototypes are as mentioned below:

1. **Node.java**
   This class contains the structure of a node in the Huffman tree.
   Variables:
   a. Data (Integer) – Contains the data specifically of a leaf, else it is -1.
   b. Freq (Integer) – Contains the frequency of its subtree.
   Pointers:
   a. Left (Node) – Pointer to the left sibling.
   b. Right (Node) – Pointer to the right sibling.
   This class contains a parameterized constructor which sets the values of the appropriate fields in a node. It also contains a compareTo() function which compares two nodes based on their frequency. It overrides the default comparable class function.

2. **MyPriorityQueue.java**
   This file contains an interface for implementing multiple types of priority queue's. It contains the following four basic function declarations needed by any priority queue.

public void add (Node newNode)

Parameters: Node

Return type: void

Description: A function to insert a new node into the priority queue.

public Node extractMin ()

Parameters: void

Return type: Node

Description: A function to extract the node with the least frequency which will be the root node in case of a tree and return the address of that node.

public int size ()

Parameters: void

Return type: Integer

Description: A function to return the number of nodes present in the priority queue.

public Node peek ()

Parameters: void

Return type: Node

Description: A function to retrieve and return the head of the priority queue.

3. BinaryHeapPQ.java

This class implements the MyPriorityQueue interface by using min-binary heap as a data-structure. This class contains an ArrayList used to store the nodes of the tree in level order. Apart from the functions listed and explained in the interface, this class contains:

public void clear ()

Parameters: void

Return type: void

Description: A function to clear the ArrayList used to store the nodes.

public int parent (int i)

Parameters: Integer

Return type: Integer

Description: A function to return the index of the parent node in the ArrayList of a node whose index is provided as a parameter.

public int left (int i)

Parameters: Integer

Return type: Integer

Description: A function to return the index of the left sibling node in the ArrayList of a node whose index is provided as a parameter.

public int right (int i)

Parameters: Integer

Return type: Integer

Description: A function to return the index of the right sibling node in the ArrayList of a node whose index is provided as a parameter.

public void bubbleDown (int p)

Parameters: Integer

Return type: void

Description: A function to restore the min tree order whenever extract-min replaces the root node by the last node of the tree. Order is restored by swapping a node with minimum of its child's and calling the same function recursively on the child.

public void bubbleUp (int i)

Parameters: Integer

Return type: void

Description: A function to restore the min tree order whenever a new node is inserted at the bottom of the tree. Order is restored by swapping the node with its parent if the node is smaller than its parent and calling the same function recursively on the parent after the swap.

4. CacheOptimizedFourWayHeapPQ.java

This class implements the MyPriorityQueue interface by using min-four way heap as a data- structure. This class contains an ArrayList used to store the nodes of the tree in level order. The first three positions in the ArrayList are left empty to align the nodes with cache. All the functions in this class are same as in BinaryHeapPQ.java except for the fact that comparison is done with four nodes instead of two in the bubbleDown() function.

5. PairingHeapPQ.java

This class implements the MyPriorityQueue interface by using pairing heap as a data-structure. This class contains a class PairNode which is a structure used to store the nodes of the tree with appropriate pointers. Apart from the functions listed and explained in the interface, this class contains:

public void clear ()

Parameters: void

Return type: void

Description: A function to clear all nodes basically by setting root to point null.

public PairNode compareAndLink(PairNode one, PairNode two)

Parameters: PairNode one, PairNode two

Return type: PairNode

Description: A function to compare two trees and link the tree with the larger root as the left subtree of the other along with maintaining all pointers.

public PairNode combineSiblings(PairNode firstSibling)

Parameters: PairNode

Return type: PairNode

Description: A function to meld all subtrees using multi-pass scheme whenever extract-min removes the root of the tree.

6. HuffmanTree.java

This class builds a Huffman tree from the priority queue provided as an input and has a pointer to the root of the tree. The tree is built using the following constructor:

HuffmanTree(MyPriorityQueue pq)

Parameters: Object of a class implementing the interface MyPriorityQueue.

Return type: Nil

Description: This constructor takes an object of a priority queue as an input and extracts two minimum frequency nodes from the queue and reinserts a new tree into the queue formed using those nodes until a single tree structure is formed. The pointer variable root points to the root of this Huffman tree.

7. encoder.java

This class contains the main function that takes in the input file name as a command line argument and processes the file for encoding using the following functions:

public static void buildFrequencyTable(String fileName)

Parameters: String

Return type: void

Description: A function that takes as input the filename of the file to read. It generates a frequency table i.e. the number of times an element is present in the file in the form of a HashMap.

public static void buildHuffmanTree()

Parameters: void

Return type: void

Description: A function that iterates through the frequency table and creates a node from each key-value pair and inserts it into the priority queue used. After

the complete HashMap is iterated and populated into the queue, a Huffman tree is built by calling the Huffman tree class constructor.

public static void generateCodeTable()

Parameters: void

Return type: void

Description: A function that calls another function named InOrderTree() which is basically traversing the tree in depth first search format along with appending '0' and '1' to a string as we go left and right respectively. Whenever a leaf node is encountered, the string generated on the path is the Huffman code for the element that is present in the data part of that node. This entry is placed in the code table HashMap.

public static void encode(String fileName)

Parameters: String

Return type: void

Description: A function that takes in the filename of the input file, reads it, finds each elements corresponding code from the code table and writes that code in a bit set. This bit set is then converted to a byte array and written to the binary file "encoded.bin".

8. decoder.java

This class contains the main function that takes in the encoded.bin filename and code-table filename as a command line arguments and processes the files for building the decode tree and and decoding the encoded file using the following functions:

public static Node buildDecodeTree(String fileName)

Parameters: String

Return type: Node

Description: A function that takes as input the code-table filename. It builds a decode tree using an algorithm which is listed later in this document. This function returns the address of the root of the decoded tree built.

public static void decode(Node root, String fileName)

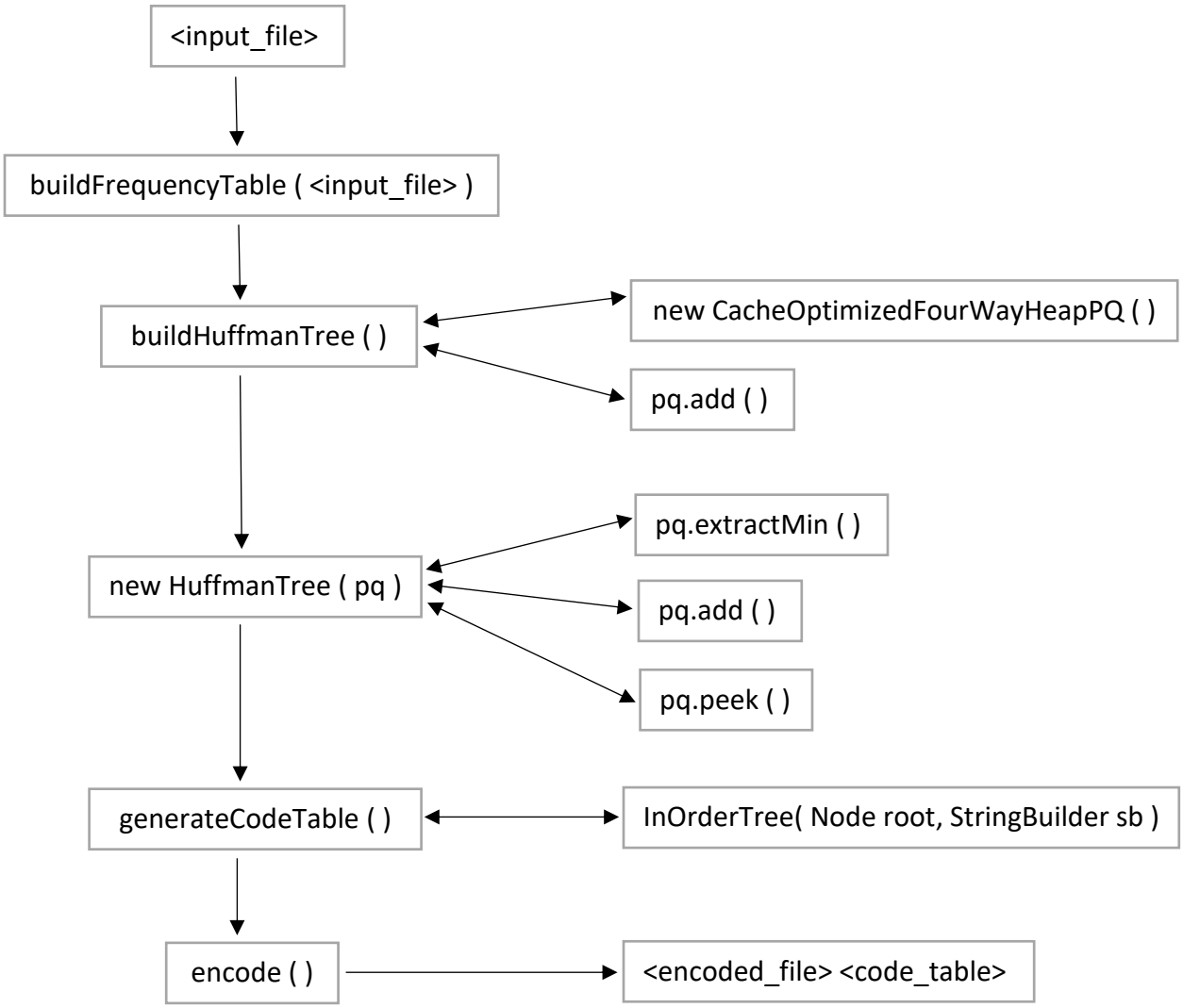
Parameters: Node root, String fileName

Return type: void

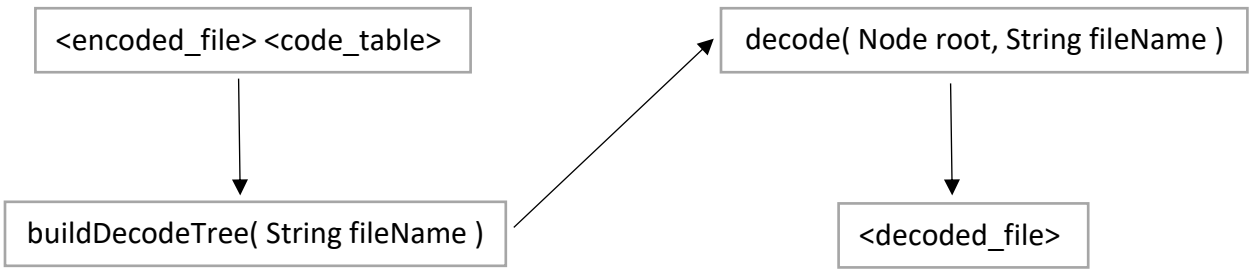
Description: A function that takes in as input, the filename of the encoded file and root of the decode tree built. A loop iterates through the bit set generated by reading the encoded file and traverses through the decode tree. Each time a leaf node is encountered, the data part of that node contains the decoded element which is written into the "decoded.txt" file and the pointer is set back to the root.

Program Structure:

1. Encoder



2. Decoder



Performance Analysis:

Testing the data structures for sample_input_large.txt file, averaging over 10 iterations of populating the priority queue and building Huffman tree gave the following result:

```
run:
Binary Heap: 0.7413 s
4-way Heap: 0.5426 s
Pairing Heap: 1.5984 s
BUILD SUCCESSFUL
```

Time taken by 4-way heap is lesser than binary heap as for each insert( ) and extractMin( ) function call, bubbleUp( ) and bubbleDown( ) goes $\log_2 n$ levels up/down for binary heap whereas it goes $\log_4 n$ levels up/down for 4-way heap. Hence 4-way heap is faster than binary heap since $\log_4 n$ is asymptotically faster than $\log_2 n$ neglecting unit time to compare two/four elements in the next level for binary/4-way heap respectively.

Pairing heap is the slowest of all since for a huge data set, it may be possible that the level below root has too many nodes and hence extract min takes linear time in the order of number of nodes in that level. Hence, the overhead of selecting the new minimum node by melding too many trees, causes pairing heap to be quite inefficient as compared to binary heap/4-way heap each of whom take logarithmic time to find the new minimum.

Decoding Algorithm:

1. Take code table file name as input argument.
2. Read and iterate through the file line by line.
3. For each line, split element and its Huffman code by space.
4. If root is null, create a root node with dummy values.
5. Iterate through each Huffman code with respect to its length starting from the root.
6. For each bit in the code, if bit is set/not set i.e. '1'/'0' respectively, check if a node exists on the current node's right/left respectively.
7. If it does exist, move to that node and if not, create a node with dummy values.
8. When each bit of the code is traversed through, a connected path from root to the leaf is built. After the traversal, set the last node's data to the element obtained from the code table.
9. Repeat steps 5-7 for the whole file.
10. Return the root of the built decode tree.

The time taken by this algorithm would be **linear in the order of the number of lines in code table** (O(n)). Although, we are iterating through each code's bits, the size of each code is much smaller than size of the whole code table. Traversing the whole file is necessary since each element has a unique code and hence the decode tree is incomplete without traversing for all codes.