Advanced Data Structures (COP 5536)
Spring 2017
Programming Project Report

Raghav Ravishankar
UFID: 1999-5874
raghav29@ufl.edu

Table of Contents

Project Description

The project mainly deals with the compression of a given input file using Huffman Encoding and Decoding techniques while implementing a Binary heap, 4-way heap and a Pairing heap.

We initially read from the input file and generate a frequency table using an array. The frequency table is fed into the Huffman Tree algorithm which generates a hash map containing all the input values and their respective Huffman Codes. The encoder uses the hash map and the Huffman Tree generated from the previous step and outputs two files - a compressed input binary file (encoded.bin) and a text file (code_table.txt) containing all the input values with their respective Huffman Codes.

Using the contents of code_table.txt, a Huffman Decode tree is generated. The bits of the binary file (encoded.bin) are traversed along the Huffman Decode tree and the data values (input values) are written into a text file (decoded.txt).

Assumptions:

1. Input file contains values only within the range 0-999999.
2. No space in the input file.
3. Each line in the input file has only one integer.
4. Input values with similar frequencies can have different Huffman Codes

Compiling Instructions

Project has been compiled and tested on thunder.cise.ufl.edu and Eclipse Neon.1a

To execute the program on thunder:

Remotely access the server using ssh

username@thunder.cise.ufl.edu

run the Makefile using **make** command which gives the **enocoder and decoder** executable files which takes input file/files as an argument.

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

Output can be checked for consistency with **decoded.txt**.

Structure of the Program

Classes : ‘HeapNode.java’ , ’PairNode.java’ and ‘FourNode.java’,’BinaryHeap.java’, ‘PairingHeap.java’,’FourAryHeap.java’, ‘HuffmanTree.java’, ‘encoder.java’ , ‘decoder.java’.

HeapNode.java

Class describes the structure of each node in the Min Priority Binary Heap.

Variables:

frequency

Type : Integer

Stores the number of times the value has been repeated in the input file.

data

Type : Integer

Stores the input integer value.

left & right

Type : HeapNode

Holds a pointer to the left and right of the specified HeapNode.

Note: Setters and Getters are generated for all the variables.

FourNode.java

Class describes the structure of each node in the Min Priority Four-Ary Heap.

Variables:

frequency

Type : Integer

Stores the number of times the value has been repeated in the input file.

data

Type : Integer

Stores the input integer value.

left & right

Type : HeapNode

Holds a pointer to the left and right of the specified FourNode.

Note: Setters and Getters are generated for all the variables.

PairNode.java

Class describes the structure of each node in the Min Priority Pair Heap.

Variables:

frequency

Type : Integer

Stores the number of times the value has been repeated in the input file.

data

Type : Integer

Stores the input integer value.

left & right

Type : PairNode

Holds a pointer to the left and right of the specified PairNode.

leftSibling & rightSibling

Type : PairNode

Holds the pointer to the left and right sibling of the specified PairNode.

child

Type : PairNode

Points to one of the children of the PairNode.

Note: Setters and Getters are generated for all the variables.

BinaryHeap.java

Class describes all the operations handled by the Binary Heap.

Variables:

binarr[]

Type : Class HeapNode(Array)

Contains an array of heap nodes.

hSize

Type : integer

Size of the Min Priority Binary heap

capacity

Type : integer

Initial capacity of the binarr[].

Procedures:

boolean isEmpty()

Returns true, if the binary heap is not empty; else returns false.

Int size()

Returns the size of the binary heap.

HeapNode findMin()

Returns the HeapNode with the minimum value.

void Insert(HeapNode x)

Inserts the specified HeapNode into the heap.

HeapNode delMin()

Deletes the HeapNode with the minimum value from the heap.

void moveUp()

Heapification of the binary heap upon the insertion of a HeapNode.

void moveDown()

Heapification of the binary heap upon the deletion of a HeapNode.

HeapNode resize()

Doubles the size of the array.

void swap(Integer, Integer)

Swaps the HeapNodes of the specified indices.

boolean hasParent(Integer), hasLeftChild(Integer) & hasRightChild(Integer)

Returns true if the specified HeapNode has a parent, left child or a right child respectively, else, returns false.

Integer leftChildIndex(Integer), rightChildIndex(Integer) & parentIndex(Integer)

Returns the indices of the leftchild, rightchild and the parent of the specified HeapNode respectively.

HeapNode parentelem(Integer)

Returns the parent of the specified HeapNode.

FourAryHeap.java

Class describes all the operations handled by the Four-Ary Heap.

Variables:

fourarr[]

Type : Class FourNode(Array)

Contains an array of four nodes.

fSize

Type : integer

Size of the Min Priority Four-ary heap

capacity

Type : integer

Initial capacity of the fourarr[].

Procedures:

boolean isEmpty()

Returns true, if the Four-ary heap is not empty; else returns false.

Int size()

Returns the size of the Four-ary heap.

FourNode findMin()

Returns the FourNode with the minimum value.

void Insert(FourNode x)

Inserts the specified FourNode into the heap.

FourNode delMin()

Deletes the FourNode with the minimum value from the heap.

void moveUp()

Heapification of the Four-ary heap upon the insertion of a FourNode.

void moveDown()

Heapification of the Four-ary heap upon the deletion of a FourNode.

FourNode resize()

Doubles the size of the array.

void swap(Integer, Integer)

Swaps the FourNodes of the specified indices.

boolean hasParent(Integer), hasFirstChild(Integer) , hasSecondChild(Integer), hasThirdChild(Integer), hasFourthChild(Integer)

Returns true if the specified FourNode has a parent, first child, second child, third child or a fourth child respectively, else, returns false.

Integer firstChildIndex(Integer), parentIndex(Integer)

Returns the indices of the firstchild, secondchild, thirdchild, fourthchild and the parent of the specified FourNode respectively.

FourNode parentelem(Integer)

Returns the parent of the specified FourNode.

Integer childNum(Integer)

Returns the number of children for a given FourNode.

Integer fetchMin(**Integer**, **Integer**)

Returns the child of the specified FourNode containing the minimum value.

PairHeap.java

Class describes all the operations handled by the Pairing Heap.

Variables:

Pairarr[]

Type : Class PairNode(Array)

Contains an array of pair nodes.

pSize

Type : integer

Size of the Min Priority Pairing heap

capacity

Type : integer

Initial capacity of the pairarr[].

Procedures:

boolean isEmpty()

Returns true, if the Pairing heap is not empty; else returns false.

Int size()

Returns the size of the Pairing heap.

PairNode findMin()

Returns the PairNode with the minimum value.

void Insert(PairNode x)

Inserts the specified PairNode into the heap.

PairNode delMin()

Deletes the PairNode with the minimum value from the heap.

PairNode compareAndLink(PairNode firstNode, PairNode secondNode)

Compares and links the new node that is inserted into the pairing heap.

PairNode combineSiblings(PairNode firstSibling)

Performs two-pass scheme melding.

PairNode resize()

Doubles the size of the array.

HuffmanTree.java

Class describes the Huffman Tree generation operations for all the three different data structures.

Variables:

BinHuffCode

Type : HashMap<Integer,String>

Contains the data of the Heapnode and it's corresponding HufCode.

PairHuffCode

Type : HashMap<Integer,String>

Contains the data of the PairNode and it's corresponding HuffCode.

FourHuffCode

Type : HashMap<Integer,String>

Contains the data of the FourNode and it's corresponding HuffCode.

S

Type: String Builder

Used to generate HuffCode while traversing the HuffmanTree.

Procedures:

HeapNode buildHuffmanTreeUsingBinHeaps (HeapNode)

Returns a HeapNode containing the HuffmanTree.

FourNode buildHuffmanTreeUsingFourHeaps (FourNode)

Returns a FourNode containing the HuffmanTree.

PairNode buildHuffmanTreeUsingPairHeaps (PairNode)

Returns a PairNode containing the HuffmanTree.

Map getBinHuffCode(HeapNode node, StringBuilder s)

Returns a hashmap containing all the HeapNode data values with their respective HuffCodes.

Map getPairHuffCode(HeapNode node, StringBuilder s)

Returns a hashmap containing all the PairNode data values with their respective HuffCodes.

Map getFourHuffCode(HeapNode node, StringBuilder s)

Returns a hashmap containing all the FourNode data values with their respective HuffCodes.

Encoder.java

Class describes all the operations performed for encoding the data.

Variables:

***freq_table*[]**

Type: Integer (Array)

Contains all the input values and their respective frequencies.

CODEFILENAME

Type: String

Contains the file directory of the file code_table.txt

INPFILENAME

Type: String

Contains the file directory of the input file.

ENCFILENAME

Type: String

Contains the file directory of the encoded.bin file

Procedures:

readFromFile(String inputfile)

Reads from the input file.

writeCodeTable(Map<Integer,String>hmap)

Creates a new file code_table.txt and writes all the Huffman codes into the file.

encoderbin(Map<Integer,String>hmap, String inputfile)

Prints all the values in binary into the file encoded.bin.

Decoder.java

Class describes all the operations performed for decoding the data.

Variables:

***BYTE_FORMAT* = "%8s", Character CHAR_BLANK = ' ' and Character CHAR_ZERO = '0';**

Type: String

Used to convert Binary values to String.

DECODEDFILENAME

Type: String

Contains the directory of the file decoded.txt

Decmap

Type: HashMap

Contains the code table values.

Procedures:

String getEncodeBinMessage(String ENCFILENAME)

Returns a String after the conversion of the binary file.

Map getDecMap(String codetbfile)

Returns a hashmap containing the code table values.

HeapNode getDecodeTree(String codetbfile)

Returns a decode Huffman Tree.

void getDecodedMessage(String encoding,String codetbfile)

Decodes and returns the file decoded.txt

Performance Analysis

The time required to generate Huffman Trees using the three data structures have been tabulated below. The above tests were performed on thunder.cise.ufl.edu.

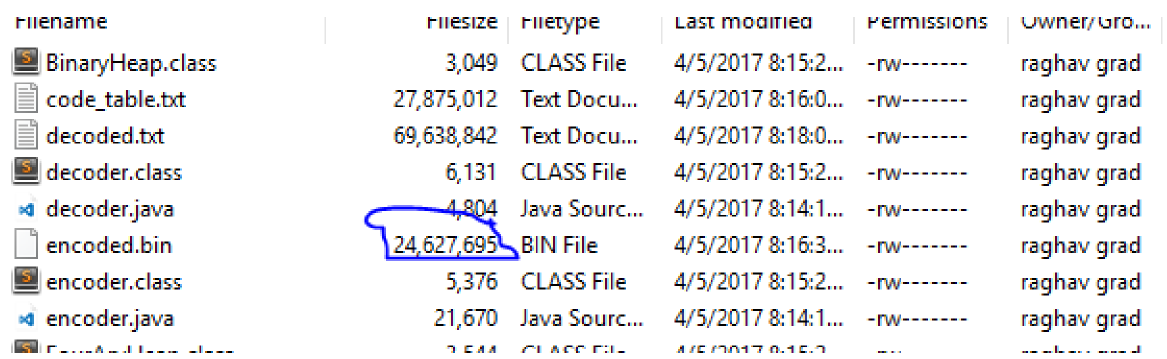
Binary Heap (secs)	
Run 1	0.575
Run 2	0.538
Run 3	0.513
Run 4	0.533
Run 5	0.445
Run 6	0.623
Run 7	0.503
Run 8	0.569
Run 9	0.556
Run 10	0.498
Average	**0.535**

Pairing heap (secs)	
Run 1	0.739
Run 2	0.722
Run 3	1.127
Run 4	0.829
Run 5	0.893
Run 6	0.769
Run 7	0.739
Run 8	0.779
Run 9	0.818
Run 10	0.783
Average	**0.819**

Four Way Heap (secs)	
Run 1	0.543
Run 2	0.485
Run 3	0.459
Run 4	0.414
Run 5	0.444
Run 6	0.447
Run 7	0.631
Run 8	0.437
Run 9	0.434
Run 10	0.439
Average	**0.473**

From the tests conducted above, it can be seen that the average run time of Huffman Tree creation using Four-Way Heaps > Binary heaps > Pairing Heaps. The combined time required for the process of encoding and decoding using Four-Way heaps was also found to be significantly better than using Binary Heaps or Pairing Heaps.

The number of nodes in each layer of a d-ary heap grows exponentially by a factor of d at each step. The height of a d-ary heap is $O(\log_d n) = O(\log n / \log d)$. The height of the four-way heap is nearly half of the binary heap. As the height decreases, the insertion and decrease key operations also take lesser time and fewer calls are made to the heapify operation.



Filename	Filesize	Filetype	Last modified	Permissions	Owner/Gro...
BinaryHeap.class	3,049	CLASS File	4/5/2017 8:15:2...	-rw-------	raghav grad
code_table.txt	27,875,012	Text Docu...	4/5/2017 8:16:0...	-rw-------	raghav grad
decoded.txt	69,638,842	Text Docu...	4/5/2017 8:18:0...	-rw-------	raghav grad
decoder.class	6,131	CLASS File	4/5/2017 8:15:2...	-rw-------	raghav grad
decoder.java	4,804	Java Sourc...	4/5/2017 8:14:1...	-rw-------	raghav grad
encoded.bin	24,627,695	BIN File	4/5/2017 8:16:3...	-rw-------	raghav grad
encoder.class	5,376	CLASS File	4/5/2017 8:15:2...	-rw-------	raghav grad
encoder.java	21,670	Java Sourc...	4/5/2017 8:14:1...	-rw-------	raghav grad

The Huffman Encoding and Decoding Algorithms using Four-Way heaps were run on thunder.cise.ufl edu. The encoder program takes sample_input_large.txt as the input and generates code_table.txt and encoded.bin. The decoder program takes code_table.txt and encoded.bin as the input and outputs decoded.txt. The size of the files encoded.bin, code_table.txt and decoded.txt were found to be the same as the ones provided in the Canvas website.

Decoding Algorithm

The pseudocode for the decoding algorithm is as follows:

```
For every element in the hashmap
 For each character ∈ HuffCode.length-1
    temp = root
    If character is a ‘0’
        If temp.left = null
            Create a new node on the left
        else
            temp = temp.left
    If character is a ‘1’
        If temp.right = null
            Create a new node on the right
        else
            temp = temp.right
If character is a ‘0’
    Create a new node with its corresponding data and insert it on temp.left
else if the character is a ‘1’
    Create a new node with its corresponding data and insert it on temp.right
```

The above algorithm reads the HuffCode until the last character and creates a new node on the left if it’s a ‘0’ and a new node on the right if it’s a ‘1’. If the last character is a ‘0’ it inserts a new node on the left with it’s corresponding data or on the right if it’s a ‘1’. The complexity of the above algorithm is O(no. of bits * no. of keys) i.e O(Total no. of bits).

Conclusion

Project has been successfully implemented to generate an encoded binary file, code table file and a decoded file from the given input file with the help of min Priority Four-Way heap.

Project has helped understand the following concepts:

- Complexity and usage of Binary heaps, Pairing heaps and Four-Way heaps.
- Huffman Tree generation
- Encoding and Decoding of bits.
- File handling in java.