

Advanced Data Structures
Spring 2017
Programming Project Report

Deepak Addepalli

UFID: 06200926

Email id: deepak246@ufl.edu

PROJECT DESCRIPTION:

The main goal of this project is to develop Huffman encoder and decoder. The initial part of the project is to evaluate which of the following priority queue structures gives best performance: Binary Heap, Four-way cache optimized heap and Pairing Heap. The data structure with the best performance is used for the construction of Huffman tree and the encoder uses this Huffman Tree. The encoder reads an input file that is to be compressed and generates two output files – the compressed version of the input file i.e encoded.bin and the code_table.txt. The decoder reads two input files – encoded.bin, code_table.txt and then it constructs the decode tree using the code table. Then the encoded message is used to generate the decode message using the decode tree.

To achieve this goal the following data structures were implemented:

- Binary heap
- Four Way heap
- Pairing heap
- Huffman Tree.

COMPILING INSTRUCTIONS:

The project folder contains the files analysis.cpp, encoder.cpp and decoder.cpp. The project has been compiled and tested on thunder.cise.ufl.edu. To execute the program, You can remotely access the server using ssh username@thunder.cise.ufl.edu

For running encoder and decoder on an input files, type:

To compile: make

To execute:

```
./encoder <input_file_name>
```

```
./decoder <encoded_file_name> <code_table_file_name>
```

Code Implementation and Structure :

The code structure consists of three files. The first file named as Comparision.cpp contains of Huffman Tree using three heap structures namely, Binary Heap, Four-way Heap and Pairing Heap. Upon construction of Huffman tree using the above-mentioned heap structures the time taken by each heap structure is calculated, compared and the Heap structure with the lowest time taken is chosen to proceed for later stages i.e Encoding and Decoding.

The following are the variables and functions in the below mentioned classes and functions,

Class Huffman_tree() :

Huffman_tree_node()	Constructor to without any arguments
Huffman_tree()	Constructor
getRoot()	Function that returns root node
sz	Variable that indicates the no.of nodes tree
combine()	Function that combines to nodes and forms a new parent node
delete_subTree()	Function that deletes the subtree at a particular node
delete_tree()	Function to delete entire tree
traverse()	Function that traverses to every node on the tree and assigns 0 to left node and 1 to right node

Class heap_element:

This is the actual object stored in the heap structures.

int freq	Sum of frequencies of all the nodes in tree
Huffman_tree* tree	Represents the Huffman tree

Class Binary_heap() :

Insert()	Inserts a new element into the Binary heap
extract_Min()	Returns the minimum value from the Binary heap
Parent()	Returns the index of parent node
Left()	Returns the index of left child node
Size()	Returns the size of the vector used to store heap elements
getHuffmanTree	Returns completely built Huffman Tree

Class Four_way_optimised_heap() :

Insert()	Inserts a new element into the Four way heap
extract_Min()	Returns the minimum value from the Four way heap
Parent()	Returns the index of parent node
Left()	Returns the index of left child node
Size()	Returns the size of the vector used to store heap elements
four_way_optimised_heap()	Constructor used to append 0's at the beginning of vector to make it Four way optimized heap

Class Pairing_heap() :

Insert()	Inserts a new element into the Pairing heap
extract_Min()	Returns the minimum value from the Pairing heap
Size()	Returns the size of the vector used to store heap elements

Methods to build Huffman Tree :

build_tree_using_four_way_optimised_heap()	Function used to build Huffman tree using Four way optimised heap
build_tree_using_binary_heap()	Function used to build Huffman tree using Binary heap
build_tree_using_pairing_heap()	Function used to build Huffman tree using Pairing heap
firstElement	Variable that stores the first minimum frequency out of all.
secondElement	Variable that stores the second minimum frequency out of all.

storm.cise.ufl.edu - PuTTY

Session Special Command Window Logging Files Transfer Hangup ?

```
login as: dedepak
Authorized Access only
```

```
UNAUTHORIZED ACCESS TO THIS DEVICE IS PROHIBITED.
You must have explicit permission to access this
device. All activities performed on this device
are logged. Any violations of access policy can
result in disciplinary action.
```

```
dedepak@storm.cise.ufl.edu's password:
Last login: Wed Apr  5 16:45:09 2017 from ip70-185-114-43.ga.at.cox.net
stormx:1% cd /cise/homes/dedepak/Desktop
stormx:2% g++ Comparision.cpp
stormx:3% ./a.out
Time using 4-way heap (microsecond): 2895376
Time using binary heap (microsecond): 1744735
Time using pairing heap (microsecond): 11569328
stormx:4% █
```

Since the time taken by Binary Heap to construct the Huffman Tree was the smallest out of all the three heap structures, I have chosen Binary Heap to proceed further with encoding of input data.

Encoding Phase:

The second file encoder.cpp has the code for encoder. Initially, the input data is read from the input file i.e sample_input_large.txt and the frequency of each input data element is stored in the array freq_table[]. The Huffman tree is then constructed with the frequencies in the array freq_table[] using the 'binary_heap'. Then the constructed Huffman tree is traversed and every left child of a node is assigned 0 and every right child is assigned 1. The code corresponding to each input data is stored in the array code_table[]. Once the code_table[] is filled, the input file is read again and for each data read from input file, the corresponding binary bits string from code_table[] is written to encoded.bin file.

storm.cise.ufl.edu - PuTTY

Session Special Command Window Logging Files Transfer Hangup ?

device. All activities performed on this device are logged. Any violations of access policy can result in disciplinary action.

```
dedepak@storm.cise.ufl.edu's password:
Last login: Wed Apr  5 16:44:01 2017 from ip70-185-114-43.ga.at.cox.net
stormx:1% cd /cise/homes/dedepak/Desktop
stormx:2% make
g++ encoder.cpp -o encoder
g++ decoder.cpp -o decoder
g++ Comparision.cpp -o Comparision
stormx:3% ./encoder sample_input_large.txt
stormx:4% ls -la
total 119935
drwxr-xr-x+  2 dedepak grad      13 Apr  5 16:46 ./
drwx--x--x+ 21 dedepak grad      39 Apr  5 13:22 ../
-rw-----+  1 dedepak grad 27875167 Apr  5 16:46 code_table.txt
-rwx-----+  1 dedepak grad   48916 Apr  5 16:45 Comparision*
-rw-----+  1 dedepak grad    8013 Apr  5 13:26 Comparision.cpp
-rwx-----+  1 dedepak grad   20117 Apr  5 16:45 decoder*
-rw-----+  1 dedepak grad    2872 Apr  5 12:57 decoder.cpp
-rw-----+  1 dedepak grad 24627695 Apr  5 16:46 encoded.bin
-rwx-----+  1 dedepak grad   35067 Apr  5 16:45 encoder*
-rw-----+  1 dedepak grad    4162 Apr  5 12:55 encoder.cpp
-rw-----+  1 dedepak grad     200 Apr  5 13:00 Makefile
-rw-----+  1 dedepak grad 69638842 Apr  5 13:05 sample_input_large.txt
-rw-----+  1 dedepak grad      62 Apr  5 13:05 sample_input_small.txt
```

Decoding Phase:

The third file, i.e decoder.cpp has the decoder logic. The codes assigned for each input data are read from code_table.txt and stored in code_table[] array. Each code is inserted into the decoder tree and while inserting a code into decoder tree, we start at the root node and for each bit in code we move to right child of the current node if it's a '1' and for '0' we move to the left child. When there are no more bits left in the code, the value of the current node is set to the data represented by the code. While moving from parent to child, a new node is created if child node doesn't exist.

Complexity analysis of decoder construction algorithm: Codes for all the data are inserted into the decoder tree. Time taken for inserting each code into the tree is the number of bits in the code. Time for inserting all the codes into the tree is equal to sum of number of bits in all the codes.


Total time $T = l_1 + l_2 + l_3 + \dots + l_n$.

where l_i be number of bits in the code for data d_i .

Therefore, complexity of the algorithm used for constructing the decoder tree is $O(T)$ where

$$T = l_1 + l_2 + l_3 + \dots + l_n.$$

After constructing the decoder tree, it is used for decoding the encoded.bin file. We start from root and for each bit in the encoded.bin file we move to right child if it's a one or else we move to left child. Whenever a leaf node is reached, the data present in the leaf node is written to decoded.txt file and we once again start traversing the decoder tree from the root. Time complexity of this algorithm is $O(\text{totals bits in the binary file})$.

 storm.cise.ufl.edu - PuTTY

Session Special Command Window Logging Files Transfer Hangup ?

```
dedepak@storm.cise.ufl.edu's password:
Last login: Wed Apr  5 16:44:01 2017 from ip70-185-114-43.ga.at.cox.net
stormx:1% cd /cise/homes/dedepak/Desktop
stormx:2% make
g++ encoder.cpp -o encoder
g++ decoder.cpp -o decoder
g++ Comparision.cpp -o Comparision
stormx:3% ./encoder sample_input_large.txt
stormx:4% ls -la
total 119935
drwxr-xr-x+  2 dedepak grad      13 Apr  5 16:46 ./
drwx--x--x+ 21 dedepak grad      39 Apr  5 13:22 ../
-rw-----+ 1 dedepak grad 27875167 Apr  5 16:46 code_table.txt
-rwx-----+ 1 dedepak grad  48916 Apr  5 16:45 Comparision*
-rw-----+ 1 dedepak grad   8013 Apr  5 13:26 Comparision.cpp
-rwx-----+ 1 dedepak grad  20117 Apr  5 16:45 decoder*
-rw-----+ 1 dedepak grad   2872 Apr  5 12:57 decoder.cpp
-rw-----+ 1 dedepak grad 24627695 Apr  5 16:46 encoded.bin
-rwx-----+ 1 dedepak grad   35067 Apr  5 16:45 encoder*
-rw-----+ 1 dedepak grad   4162 Apr  5 12:55 encoder.cpp
-rw-----+ 1 dedepak grad    200 Apr  5 13:00 Makefile
-rw-----+ 1 dedepak grad 69638842 Apr  5 13:05 sample_input_large.txt
-rw-----+ 1 dedepak grad    62 Apr  5 13:05 sample_input_small.txt
stormx:5% ./decoder encoded.bin code_table.txt
stormx:6% ls -la
total 188266
drwxr-xr-x+  2 dedepak grad      14 Apr  5 16:46 ./
drwx--x--x+ 21 dedepak grad      39 Apr  5 13:22 ../
-rw-----+ 1 dedepak grad 27875167 Apr  5 16:46 code_table.txt
-rwx-----+ 1 dedepak grad  48916 Apr  5 16:45 Comparision*
-rw-----+ 1 dedepak grad   8013 Apr  5 13:26 Comparision.cpp
-rw-----+ 1 dedepak grad 69638842 Apr  5 16:47 decoded.txt
-rwx-----+ 1 dedepak grad  20117 Apr  5 16:45 decoder*
-rw-----+ 1 dedepak grad   2872 Apr  5 12:57 decoder.cpp
-rw-----+ 1 dedepak grad 24627695 Apr  5 16:46 encoded.bin
-rwx-----+ 1 dedepak grad   35067 Apr  5 16:45 encoder*
-rw-----+ 1 dedepak grad   4162 Apr  5 12:55 encoder.cpp
-rw-----+ 1 dedepak grad    200 Apr  5 13:00 Makefile
-rw-----+ 1 dedepak grad 69638842 Apr  5 13:05 sample_input_large.txt
-rw-----+ 1 dedepak grad    62 Apr  5 13:05 sample_input_small.txt
```

Conclusion:

Since pairing heap has an amortized performance of $O(\log n)$ for extract minimum operation, four-way heap and binary heap gives better performance than pairing heap.