ADVANCED DATA STRUCTURES Spring 2017

Programming Project Report

Abhinav Reddy Podduturi

UFID: 15992633

Email id: abhinavpodduturi@ufl.edu

Project Description

The goal of this project is to develop Huffman encoder and decoder. The first part of the project is to evaluate which of the following priority queue structures gives best performance: Binary Heap, 4-way cache optimized heap and Pairing Heap. The data structure with the best performance is used for building the Huffman tree. The Huffman tree is used by encoder. The encoder reads an input file that is to be compressed and generates two output files – the compressed version of the input file and the code table. The decoder reads two input files - encoded message, code table and then constructs the decode tree using the code table. Then the decoded message is generated from the encoded message using the decode tree.

To achieve this the following data structures were implemented:

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

Working Environment

Hardware requirement:

Hard Disk space: 4GB minimum

Memory: 512 Mb minimum

CPU: 32bit

Operating System: Ubuntu 14.04

Compiler: g++

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 CLASSES

The code structure consists of three files analysis.cpp, encoder.cpp and decoder.cpp. analysis.cpp contains the code for three structures and was used for comparing the performance of the structures. encoder.cpp contains the code for encoder. decoder.cpp contains the code for decoder.

Class descriptions

Class huffman_tree_node:

Represents each node in an huffman tree.

`huffman_tree_node* left`	Pointer to left child
`huffman_tree_node* right`	Pointer to right child
int val	Value represented by this node. Its set to -1 for non leaf nodes and set to a data value for leaf node.
`huffman_tree_node()`	Constructor without any arguments.
`huffman_tree_node(int v)`	Constructor with value as a parameter. val is set to v.

Class huffman_tree:

Represents the Huffman tree.

`huffman_tree_node* root`	Root of the Huffman tree
int sz	Used for storing the number of nodes in the tree.
`huffman_tree()`	constructor.
getRoot()	Returns the root of the tree.
`insert_root( huffman_tree_node *)`	Sets the root of the tree to the passed argument
`delete_subTree (huffman_tree_node*)`	Deletes subtree rooted at the passed node
`delete_tree()`	Deletes the tree.
`combine(huffman_tree *)`	Combines two trees by making the tree with larger root value as child to the tree of smaller root value.
size()	Returns the number of nodes in the tree.

Class heap_element:

This is the actual object stored in the heap structures.

int freq	Sum of frequencies of all the nodes in the huffman tree
huffman_tree* tree	represents the huffman tree

Class binary_heap:

Represents binary heap

vector<heap_element> arr	Vector used for storing heap elements
insert(heap_element)	Inserts element into the heap
get_min()	Removes the minimum element from the heap and returns it.

size()	Returns the number of elements in the heap.
delete_heap()	Deletes the heap.

Class four_way_heap:

Represents four-way heap

vector<heap_element> arr	Vector used as heap
insert()	Inserts an element into the heap
get_min()	Removes the minimum element from the heap and returns it.
size()	Returns the number of elements in the heap.
delete_heap()	Deletes the heap.
`getHuffmanTree()`	Returns the Huffman tree of the smallest object in the heap
`four_way_heap()`	Constructor with no input arguments. Three dummy values are inserted into the arr vector for cache optimization.

Class pairing_heap_node:

Represents each node in pairing heap.

`Vector <pairing_heap_node*>`	Vector used for storing the children of the node.
`heap_element elem`	This is the heap element represented by this node.

Class pairing_heap:

Represents the pairing heap structure.

`pairing_heap_node* root`	Root of the heap structure
insert()	Inserts an element into the heap
get_min()	Removes the minimum element from the heap and returns it. Used two pass scheme.
size()	Returns the number of elements in the heap.
delete_heap()	Deletes the heap.
sz	Number of elements in the pairing heap
pairing_heap()	Constructor. Initializes root to NULL and sz to 0.

encoder.cpp:

This file has all the encoder code. First, data is read from the input file and frequency of each data is stored in freq_table[]. Huffman tree is built with frequencies in freq_table[] using the four_way_heap and huffma_tree classes. Then the constructed Huffman tree is traversed and code of each data is stored in code_table[]. Once the code_table[] is filled, the input file is read again and and for each data read from input file, the corresponding code from code_table[] is written to encoded.bin file. Finally contents of code_table[] is written to code_table.txt file.

decoder.cpp

This file has all the decoder logic. Codes from code_table.txt are read and stored in code_table[]. Now each code is inserted into the decoder tree. While inserting a code into decoder tree, we start at the root 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 = $l_1 + l_2 + l_3 \ldots\ldots + l_n$.

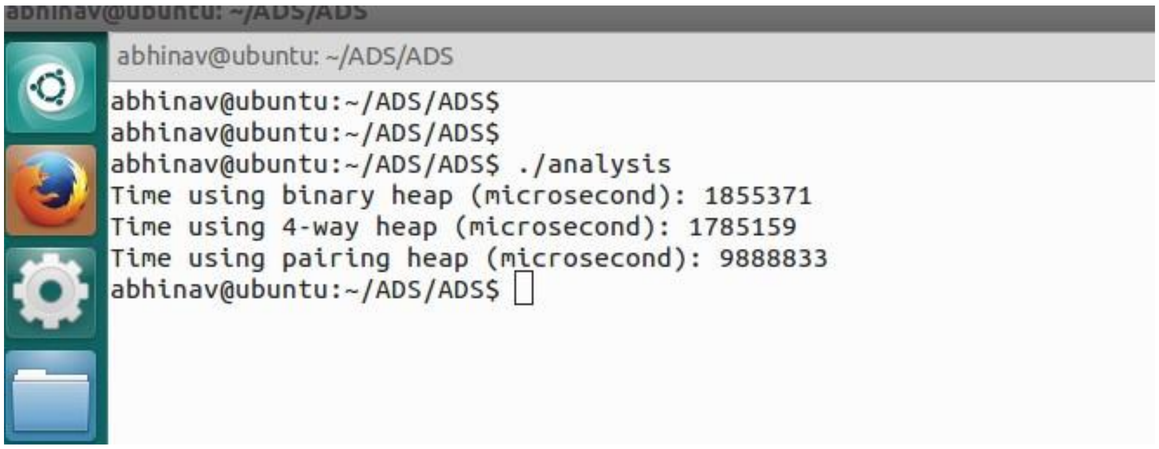
where l_i be number of bits in the code for data d_i.

Therefore, complexity of the algorithm used for constructing the decoder tree is O(T) where T = $l_1 + l_2 + l_3 \ldots\ldots + l_n$.

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 (totals bits in the binary file).

Results

The screenshot comparing the run times of three structures.



```
abhinav@ubuntu:~/ADS/ADS$
abhinav@ubuntu:~/ADS/ADS$
abhinav@ubuntu:~/ADS/ADS$ ./analysis
Time using binary heap (microsecond): 1855371
Time using 4-way heap (microsecond): 1785159
Time using pairing heap (microsecond): 9888833
abhinav@ubuntu:~/ADS/ADS$
```

The results are as expected. Since pairing heap has an amortized performance of O(logn) for extract minimum operation, 4-way heap and binary heap gives better performance than pairing heap. Since 4-way heap is cache optimized it performs better than the binary heap.