Huffman Coding Programming Project

Advanced Data Structures (COP 5536) Spring 2017

Chang Xu

Email: theoneneo@ufl.com UFID: 94101882

Table of Contents

1. Introduction

This report gives detailed description how the Huffman Coding Program is implemented. The Program is developed in C++ and it comprises two major executable parts: Encoder and Decoder. Besides the two, it also contains a Benchmark part to examine the performance of 3 types of Priority Queue.

Section 2 Priority Queue Implementation provides details about the 3 types of Priority Queue – Binary Heap, Four-Way Heap, and Pairing Heap – and their performance analysis.

Section 3 Huffman Encoding provides details about the algorithm, data structure, program structure, function prototypes and encoding result of the Encoder.

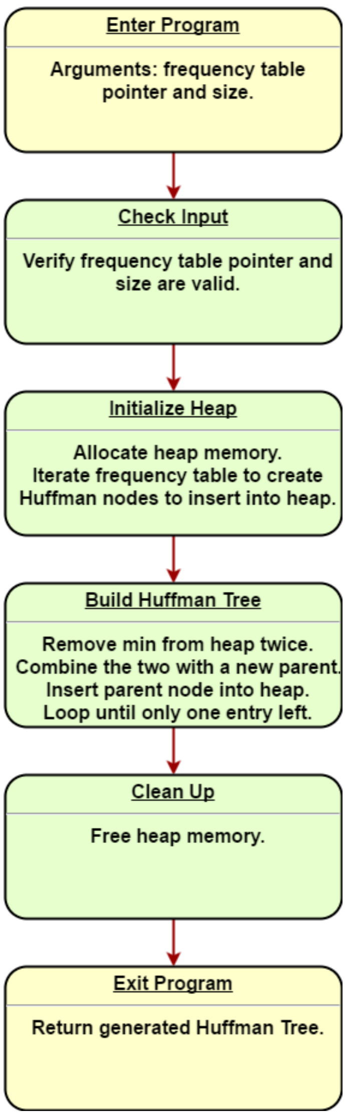
Section 4 Huffman Decoding provides details about the algorithm, data structure, program structure, function prototypes and encoding result of the Decoder.

Section 5 Performance in Other Environments gives more program performance results in environments other than CISE Server.

2. Priority Queue Implementation

Priority Queue is needed to build the Encoder Huffman Tree using Greedy Algorithm. There are 3 types of Priority Queue implemented in this project: Binary Heap, Four-Way Heap, and Pairing Heap.

2.1. Program Structure



Heap Program Structure

The program structure of all 3 Heaps is the same as shown on the left. It takes frequency table along with its size as input and gives generated Huffman Tree as output.

2.2. Binary Heap

Binary Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array. It is easy to access node parent and children using array index. The index of root is set to 1 instead of 0. And the index of parent is (current_index / 2); the index of left child is (current_index * 2); the index of right child is (current_index * 2 + 1).

The complexity of Heap Initialization is O(n log n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Binary Heap is O(n log n).

2.2.1. Data Structure

Structure	binary_entry
Attribute	int key void* value_pointer
Description	Binary Heap entry.

2.2.2. Function Prototype

Function	insert
Argument	int key void* value_pointer
Return Type	int
Return Value	0 - success
Description	Insert entry into binary heap.
Complexity	O(log n)

Function	remove_min
Argument	binary_entry* output
Return Type	int
Return Value	0 - success -1 - heap is empty
Description	Remove min from binary heap. And calculate new root after removal.
Complexity	O(log n)

Function	build_huffman_tree_using_binary_heap
Argument	int* frequency_table int size_of_frequency_table
Return Type	huffman_node*
Return Value	Pointer to generated Huffman Tree.
Description	Build Huffman Tree using binary heap.
Complexity	O(n log n)

2.3. Four-Way Heap

Four-Way Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array. It is easy to access node parent and children using array index. The index of root is set to 3 instead of 0. And the index of parent is (current_index / 4 + 2); the index of the first child is (current_index * 4); the index of other children can be calculated based on the first child.

The complexity of Heap Initialization is O(n log n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Four-Way Heap is O(n log n).

2.3.1. Data Structure

Structure	four_way_entry
Attribute	int key void* value_pointer
Description	Four-Way Heap entry.

2.3.2. Function Prototype

Function	insert
Argument	int key void* value_pointer
Return Type	int
Return Value	0 - success
Description	Insert entry into four-way heap.
Complexity	O(log n)

Function	remove_min
Argument	four_way_entry* output
Return Type	int
Return Value	0 - success -1 - heap is empty
Description	Remove min from four-way heap. And calculate new root after removal.
Complexity	O(log n)

Function	build_huffman_tree_using_four_way_heap
Argument	int* frequency_table int size_of_frequency_table
Return Type	huffman_node*
Return Value	Pointer to generated Huffman Tree.
Description	Build Huffman Tree using four-way heap.
Complexity	O(n log n)

2.4. Pairing Heap

Pairing Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array but node is accessed via pointer instead of array index. Insert function uses the meld function directly; remove_min uses meld function to implement two-pass melding.

The complexity of Heap Initialization is O(n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Four-Way Heap is O(n log n).

2.4.1. Data Structure

Structure	pairing_node
Attribute	int key void* value_pointer pairing_node* left_sibling pairing_node* right_sibling pairing_node* child
Description	Pairing Heap node.

2.4.2. Function Prototype

Function	meld
Argument	pairing_node* root1 pairing_node* root2
Return Type	pairing_node*
Return Value	Pointer to result tree.
Description	Meld two trees into one.
Complexity	O(1)

Function	insert
Argument	pairing_node* node
Return Type	void
Return Value	NA
Description	Insert node into pairing heap using meld function.
Complexity	O(1)

Function	remove_min
Argument	void
Return Type	pairing_node*
Return Value	Pointer to min node in pairing heap.
Description	Remove min from pairing heap. And meld subtrees using two-pass scheme.
Complexity	O(log n)

Function	build_huffman_tree_using_pairing_heap
Argument	int* frequency_table int size_of_frequency_table
Return Type	huffman_node*
Return Value	Pointer to generated Huffman Tree.
Description	Build Huffman Tree using pairing heap.
Complexity	O(n log n)

2.5. Performance Analysis

The benchmark reuslt of 3 types of Heap is shown below. The input file is a 66.4MB sample file containing 10 million records; and for each heap the iteration time is 10.

Though the time consumed to build Huffman Tree varies on different plantform, the relative speed is consistent: Binary Heap has the best performance, then is Four-Way Heap, and Pairing Heap is the slowest. However, considering the size of input frequency table is nearly 1 million, all the 3 Heaps are giving quite excellent performance. The building of Huffman Tree takes less than 1 second to complete.

```
stormx:20% ./benchmark sample_input_large.txt
Benchmark started.
Huffman tree average construction time using binary heap (ms): 481
Huffman tree average construction time using four way heap (ms): 499
Huffman tree average construction time using pairing heap (ms): 725
Benchmark completed.
```

Benchmark Result on CISE Server

```
Benchmark started.
Huffman tree average construction time using binary heap (ms): 198
Huffman tree average construction time using four way heap (ms): 207
Huffman tree average construction time using pairing heap (ms): 443
Benchmark completed.
```

Benchmark Result on Local Windows

Theoretically, Four-Way Heap is supposed to have the best performance, but the benchmark reuslt shows different. The discrepancy is probrably due to the optimized implementation of Binary Heap: storage array index starts at 1 rather than 0 makes the children of every heap node lie at the same cache line. Hence, the Binary Heap is cache optimized.

	root	c1	c2	c11	c12	c21	c22	c111	c112	c121	c122	c211	c212	c221	c222

Binary Heap Array

3. Huffman Encoding

3.1. Encoding Algorithm

The algorithm used to encode file is:

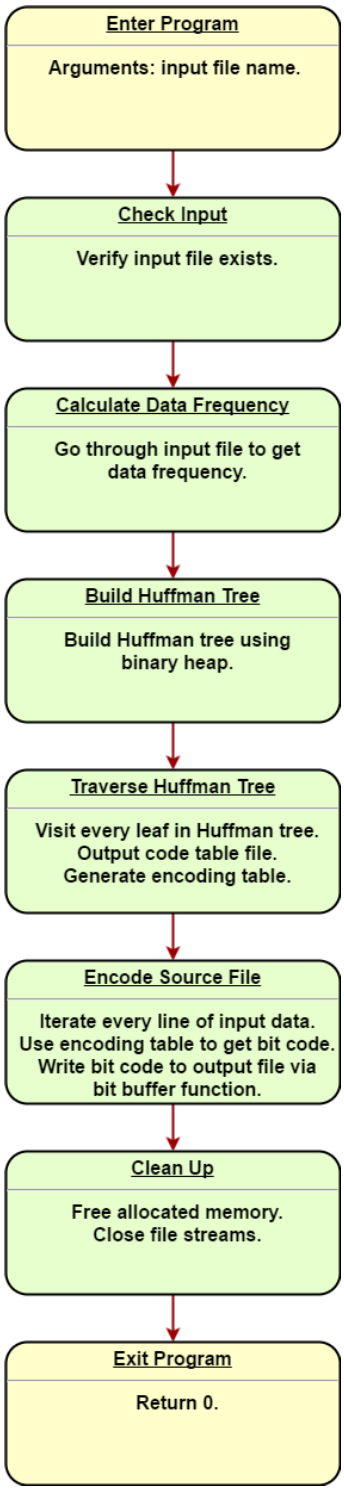
1. Go through input file to get frequency of each value.
2. Build encoding Huffman Tree based on frequency data:
   a. Use Binary Heap as Priority Queue (because it has the best performance).
   b. Use Greedy Algorithm to build up Huffman Tree.
3. Traverse generated Huffman Tree to output code table file and build internal coding table.
4. Iterate every line of input file again to translate into coded bit.
5. Write bit code to output file via bit buffer function.

```
//build huffman tree
new_huffman_memory(num_of_values);
huffman_tree = build_huffman_tree(frequency_table, num_of_values);
delete[] frequency_table;
frequency_table = NULL;
//build and output code table
code_table = new code_entry[num_of_values];
output_file.open(output_code_table_file_name, ios::binary);
traverse_tree(huffman_tree);    //traverse huffman tree to initialize and output code table
output_file.close();
delete_huffman_memory();
//encode and output source file
input_file.clear();
input_file.seekg(0, ios::beg);
output_file.open(output_encoded_file_name, ios::binary);
while (input_file >> input_string) {
        code_index = string_to_int(input_string);
        write_bit(code_table[code_index].bit_code, code_table[code_index].bit_size);
}
```

Code Snippet of Encoding File

The complexity to get frequency data is O(n); the complexity to encoding input file is O(n). Hence, the overall complexity of encoder is O(n).

3.2. Program Structure



Encoder Program Structure

Encoder takes input file name as input and gives "encoded.bin" and "code_table.txt" as output.

3.3. Data Structure

Structure	huffman_node
Attribute	int numeric char string[SIZE_OF_STRING_VALUE] huffman_node* left_child huffman_node* right_child
Description	Huffman Tree node.

3.4. Function Prototype

Function	write_bit
Argument	unsigned long long bit_code int bit_size
Return Type	void
Return Value	NA
Description	Write bit code to output file
Complexity	O(1)

Function	traverse_node
Argument	huffman_node* node string string_code
Return Type	void
Return Value	NA
Description	Traverse Huffman Tree node to set and output code.
Complexity	O(n)

Function	traverse_tree
Argument	uffman_node* root
Return Type	void
Return Value	NA
Description	Calling function of traverse_node.
Complexity	O(n)

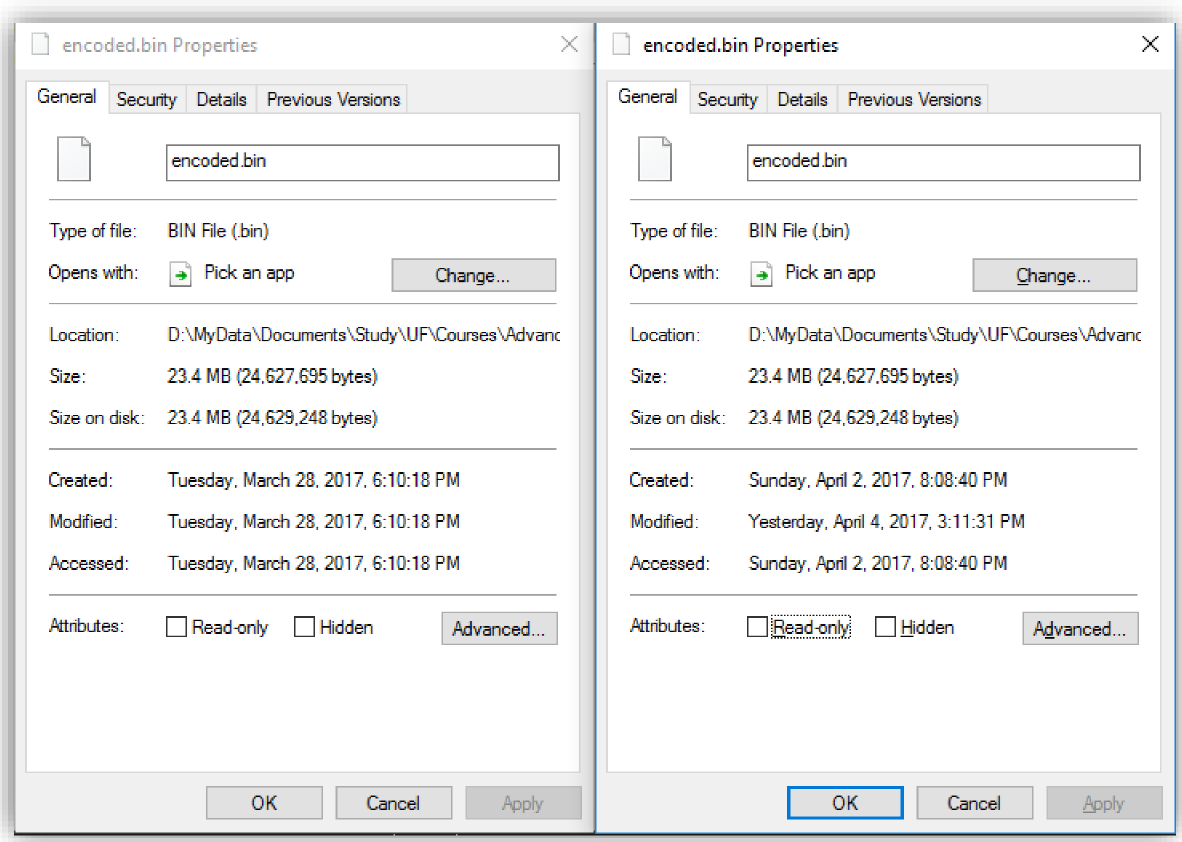
Function	encoder
Argument	char* input_source_file_name char* output_encoded_file_name char* output_code_table_file_name
Return Type	int
Return Value	0 - success -1 - error
Description	Encode input file.

Complexity	O(n)

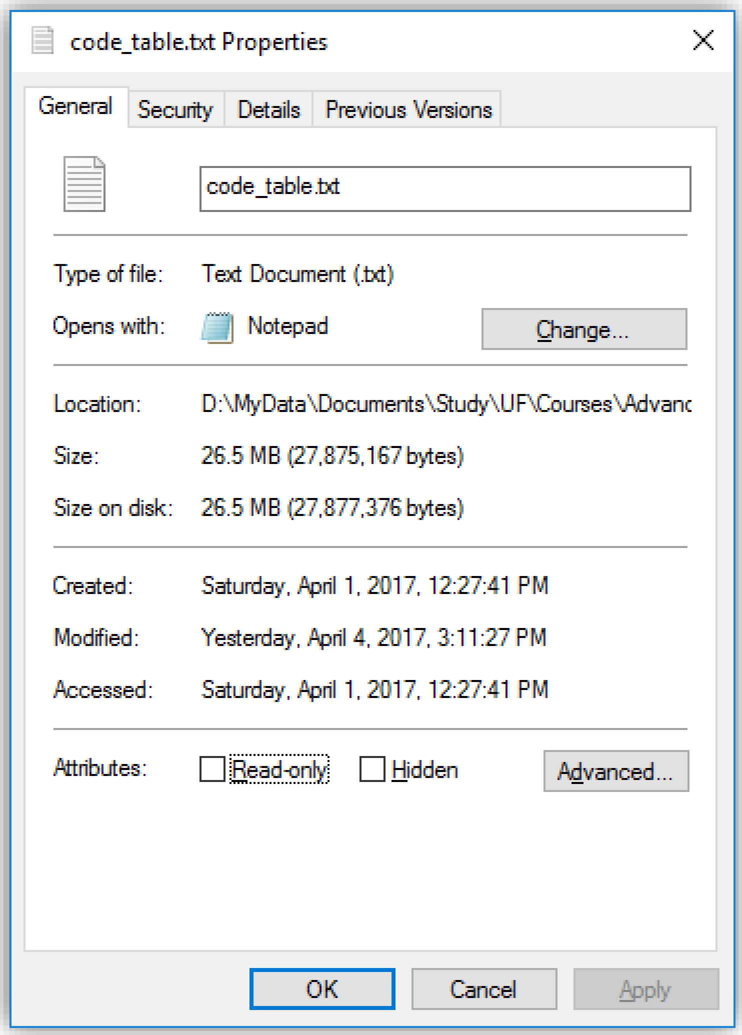
Function	main
Argument	int argc char* argv[]
Return Type	int
Return Value	0 - success
Description	Calling function encoder.
Complexity	O(n)

3.5. Encoding Result

The encoding processing gives a quite excellent performance. It took only 4410 ms on CISE Server to encoder the 66.4MB size 10 million record file. The encoded file has the same size as the sample encoded file. And encoder also outputs a code table file as required. The Huffman Encoding is successful.



File Size Comparison



Output Code Table File

```
stormx:18% ./encoder sample_input_large.txt
Encoding started.
Building huffman tree using binary heap.
Encoding completed. Encoding time (ms): 4410
```

Time Used on CISE Server

4. Huffman Decoding

4.1. Decoding Algorithm

The algorithm used to decode file is:

1. Build decoding Huffman Tree from input code table file.
2. While iterating the input code table file, find out the minimal length of bit code.
3. Traverse the generated Huffman Tree to every node whose degree is the minimal length of bit code. Store the node pointers as the starting node table.
4. Go through the encoded file to decode.
    a. Read bit code of minimal length from the file via a bit buffer function.
    b. User starting node table to get access to a node which could be very close to the leaf.
    c. Read 1 bit at a time to find the corresponding leaf.
    d. Write decoded string to output file.
    e. Loop until end of input file.

```
//decode and output file
output_file.open(output_decoded_file_name, ios::binary);
while (read_bit(&bit_code, min_bit_size) == 0) { //performance is 30% better than starting at root and
reading 1 bit each time
	current_node = starting_node_table[bit_code];  //direct access to node very close to leaf
	if (current_node->left_child != NULL) {
		while (read_bit(&bit_code, 1) == 0) {
			current_node = bit_code == 0 ? current_node->left_child :
current_node->right_child;
			if (current_node->left_child == NULL) {
				break;
			}
		}
	}
	output_file << current_node->string << "\x0A";
}
```

Code Snippet of Decoding File

To build the decoding Huffman Tree, following processing is made:

1. Read one line of string from the input code table file.
2. Start traverse from the root of Huffman Tree.
3. Iterate every byte of the string.
4. Create needed node if not exist.
5. Copy decoding string to leaf.
6. Recording minimal bit size of code.
7. Loop the processing until end of the code table file.

```
//build huffman tree from code table file
huffman_tree = new_huffman_node();
while (code_table_file >> string_value >> string_code) {
	current_node = huffman_tree;
	for (i = 0; string_code[i] != '\0'; i++) {
		if (string_code[i] == '0') {
			if (current_node->left_child == NULL) {
				current_node->left_child = new_huffman_node();
			}
			current_node = current_node->left_child;
		} else {
			if (current_node->right_child == NULL) {
				current_node->right_child = new_huffman_node();
			}
			current_node = current_node->right_child;
		}
	}
	copy_string(current_node->string, string_value);
	if (strlen(string_code) < min_bit_size) {
		min_bit_size = strlen(string_code);
	}
}
```

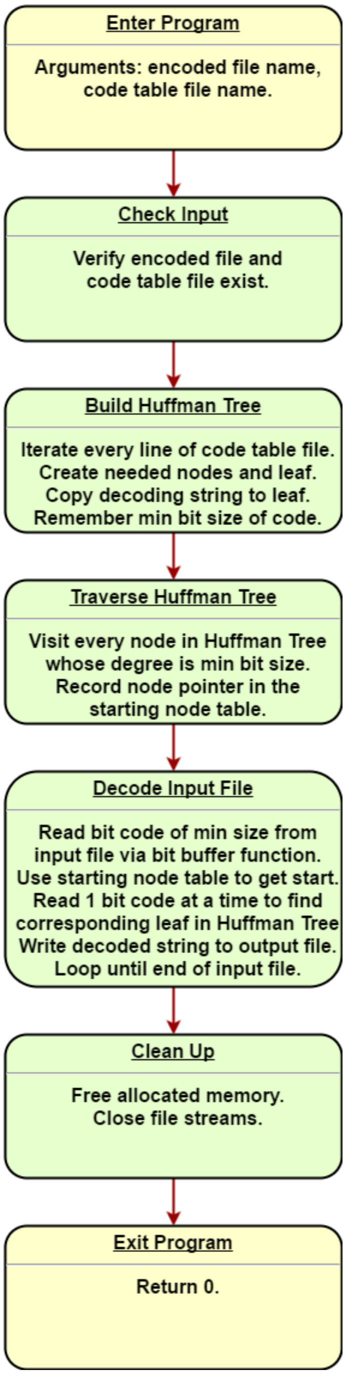
Code Snippet of Building Decoding Huffman Tree

The complexity of building Huffman Tree is O(n) in terms of the total length of all code bit (not including value string part) in the code table file.

Since every bit in the encoded file is read only once and trigger at most one changing of node in the Huffman Tree, the overall complexity of decoder is O(n).

4.2. Program Structure

Decoder takes encoded file name and code table file name as input and gives “decoded.txt” as output.



Decoder Program Structure

4.3. Data Structure

Structure	huffman_node
Attribute	int numeric char string[SIZE_OF_STRING_VALUE] huffman_node* left_child huffman_node* right_child
Description	Huffman Tree node.

4.4. Function Prototype

Function	read_bit
Argument	unsigned long long* bit_code int bit_size
Return Type	int
Return Value	Bit code value in integer format.
Description	Read bit code from encoded file
Complexity	O(1)

Function	traverse_node
Argument	huffman_node* node string string_code
Return Type	void
Return Value	NA
Description	Traverse Huffman Tree node to set starting node table. Recursion function.
Complexity	O(n)

Function	traverse_tree
Argument	uffman_node* root
Return Type	void
Return Value	NA
Description	Calling function of traverse_node.
Complexity	O(n)

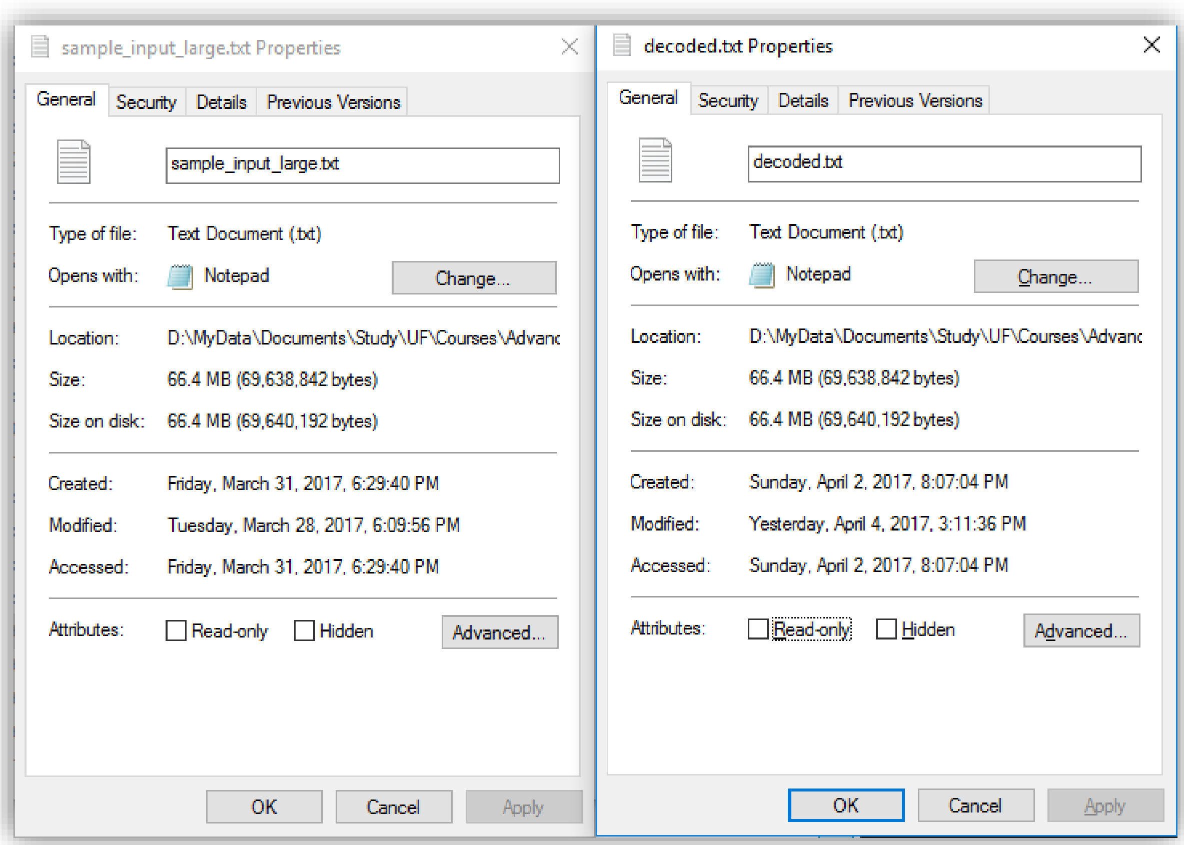
Function	decoder
Argument	char* input_encoded_file_name char* input_code_table_file_name char* output_decoded_file_name
Return Type	int
Return Value	0 - success -1 - error
Description	Decode input file.

Complexity	O(n)

Function	main
Argument	int argc char* argv[]
Return Type	int
Return Value	0 - success
Description	Calling function decoder.
Complexity	O(n)

4.5. Decoding Result

Thanks to the starting node table, the decoding processing took even less time than encoding. The time used is only 4096 milliseconds on CISE Server. The decoded file has both the same size and the same content as the source sample file. The Huffman Decoding is successful.



File Size Comparison

sample_input_large.txt	decoded.txt
674012	674012
487899	487899
619617	619617
489110	489110
683914	683914
769365	769365
432337	432337
526283	526283
279177	279177
893989	893989
430052	430052
404903	404903
680821	680821
637787	637787
629860	629860
297619	297619
198464	198464
445182	445182
66858	66858
809737	809737
660030	660030
466248	466248
726452	726452
473210	473210
324327	324327
457871	457871
191156	191156
49642	49642
254166	254166
799976	799976

File Content Comparison

```
stormx:19% ./decoder encoded.bin code_table.txt
Decoding started.
Decoding completed. Decoding time (ms): 4096
```

Time Used on CISE Server

5. Performance in Other Environments

The performance result of Encoder, Decoder and Benchmark on Windows and Ubuntu are shown below.

Processor:	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 2.60 GHz
Installed memory (RAM):	16.0 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	Touch Support with 5 Touch Points

System Information

5.1. Windows

```
Benchmark started.
Huffman tree average construction time using binary heap (ms): 198
Huffman tree average construction time using four way heap (ms): 207
Huffman tree average construction time using pairing heap (ms): 443
Benchmark completed.
Encoding started.
Building huffman tree using binary heap.
Encoding completed. Encoding time (ms): 6799
Decoding started.
Decoding completed. Decoding time (ms): 5441
Processing completed.
```

Performance Result on Windows 10 64-bit

5.2. Ubuntu

```
neo@NeoUbuntu64bit:~/Code/ADS/Programming Project$ ./autorun
Benchmark started.
Huffman tree average construction time using binary heap (ms): 449
Huffman tree average construction time using four way heap (ms): 482
Huffman tree average construction time using pairing heap (ms): 634
Benchmark completed.
Encoding started.
Building huffman tree using binary heap.
Encoding completed. Encoding time (ms): 4324
Decoding started.
Decoding completed. Decoding time (ms): 3470
Processing completed.
```

Performance Result on Ubuntu 16.04 64-bit (Virtual Machine)