Deflate compression

1.0

Generated by Doxygen 1.9.7

Chapter 1

README

Place your project here

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 huffman Class Reference

Public Member Functions

- huffman (string inFileName, string outFileName)
- void compress ()
- void decompress ()

4.1.1 Detailed Description

Definition at line 45 of file functions.h.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 huffman()

```
huffman::huffman (
            string inFileName,
            string outFileName )  [inline]
```

Definition at line 97 of file functions.h.

```
00098         {
00099             this->inFileName = inFileName;
00100             this->outFileName = outFileName;
00101             createArr();
00102         }
```

4.1.3 Member Function Documentation

4.1.3.1 compress()

```
void huffman::compress ( )
```

Definition at line 254 of file deflate.cpp.

```
00254                          {
00255     createMinHeap();
00256     createTree();
00257     createCodes();
00258     saveEncodedFile();
00259 }
```

4.1.3.2 decompress()

```
void huffman::decompress ( )
```

Definition at line 262 of file deflate.cpp.

```
00262                            {
00263     getTree();
00264     saveDecodedFile();
00265 }
```

The documentation for this class was generated from the following files:

- functions.h
- deflate.cpp

4.2 Node Struct Reference

Public Attributes

- char data
- unsigned freq
- std::string code
- Node * left
- Node * right

4.2.1 Detailed Description

Definition at line 32 of file functions.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Node()

```
Node::Node ( )  [inline]
```

Definition at line 38 of file functions.h.

```
00038           {
00039         left = right = NULL;
00040     }
```

4.2.3 Member Data Documentation

4.2.3.1 code

```
std::string Node::code
```

Definition at line 35 of file functions.h.

4.2.3.2 data

```
char Node::data
```

Definition at line 33 of file functions.h.

4.2.3.3 freq

```
unsigned Node::freq
```

Definition at line 34 of file functions.h.

4.2.3.4 left

```
Node* Node::left
```

Definition at line 36 of file functions.h.

4.2.3.5 right

```
Node * Node::right
```

Definition at line 36 of file functions.h.

The documentation for this struct was generated from the following file:

- functions.h

Chapter 5

File Documentation

5.1 decode.cpp File Reference

```
#include <iostream>
#include "functions.h"
```

Functions

- int main (int argc, char *argv[])

5.1.1 Detailed Description

Author

Yassine Bendimerad (yb308985@student.polsl.pl)

Version

0.1

Date

2023-01-13

Copyright

CPP file to run the decompression(decode)(unzip) the ENCODED OUTPUT FILE to OUTPUT FILE similar to the first input file.

Definition in file decode.cpp.

5.1.2 Function Documentation

5.1.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 15 of file decode.cpp.

```
00015                                   {
00016     //error message if failed to detect file
00017     if(argc != 3){
00018         std::cout«"Failed to detect Files";
00019         exit(1);
00020     }
00021
00022     //starting the decompression
00023     huffman f(argv[1], argv[2]);
00024     f.decompress();
00025     std::cout«"Decompressed successfully"«std::endl;
00026
00027     return 0;
00028
00029 }
```

5.2 decode.cpp

Go to the documentation of this file.

```
00001
00012 #include <iostream>
00013 #include "functions.h"
00014
00015 int main(int argc, char* argv[]) {
00016     //error message if failed to detect file
00017     if(argc != 3){
00018         std::cout«"Failed to detect Files";
00019         exit(1);
00020     }
00021
00022     //starting the decompression
00023     huffman f(argv[1], argv[2]);
00024     f.decompress();
00025     std::cout«"Decompressed successfully"«std::endl;
00026
00027     return 0;
00028
00029 }
```

5.3 deflate.cpp

```
00001
00013 #include "functions.h"
00014
00015 void huffman::createArr() {
00016     //defining and setting the huffman array that stores all characters and their frequency
00017     for (int i = 0; i < 128; i++) {
00018         arr.push_back(new Node());
00019         arr[i]->data = i;
00020         arr[i]->freq = 0;
00021     }
00022 }
00023
00024 void huffman::traverse(Node* r, string str) {
00025     //traversing to the right +='1' , traversing to the left +='0'
00026     if (r->left == NULL && r->right == NULL) {
00027         r->code = str;
00028         return;
```

```
00029     }
00030
00031     traverse(r->left, str + '0');
00032     traverse(r->right, str + '1');
00033 }
00034
00035 int huffman::binToDec(string inStr) {
00036     //to convert binary code to decimal
00037     int res = 0;
00038     for (auto c : inStr) {
00039         res = res * 2 + c - '0';
00040     }
00041     return res;
00042 }
00043
00044 string huffman::decToBin(int inNum) {
00045     //to convert decimal to binary
00046     string temp = "", res = "";
00047     while (inNum > 0) {
00048         temp += (inNum % 2 + '0');
00049         inNum /= 2;
00050     }
00051     res.append(8 - temp.length(), '0');
00052     for (int i = temp.length() - 1; i >= 0; i--) {
00053         res += temp[i];
00054     }
00055     return res;
00056 }
00057
00058 void huffman::createMinHeap() {
00059     char id;
00060     inFile.open(inFileName, ios::in);
00061     inFile.get(id);
00062     //Incrementing frequency of characters that appear in the input file
00063     while (!inFile.eof()) {
00064         arr[id]->freq++;
00065         inFile.get(id);
00066     }
00067     inFile.close();
00068     //Pushing the Nodes which appear in the file into the priority queue (Min Heap)
00069     for (int i = 0; i < 128; i++) {
00070         if (arr[i]->freq > 0) {
00071             minHeap.push(arr[i]);
00072         }
00073     }
00074 }
00075
00076 void huffman::buildTree(char a_code, string& path) {
00077     //building the huffman tree with the paths and new nodes
00078     Node* curr = root;
00079     for (int i = 0; i < path.length(); i++) {
00080         if (path[i] == '0') {
00081             if (curr->left == NULL) {
00082                 curr->left = new Node();
00083             }
00084             curr = curr->left;
00085         }
00086         else if (path[i] == '1') {
00087             if (curr->right == NULL) {
00088                 curr->right = new Node();
00089             }
00090             curr = curr->right;
00091         }
00092     }
00093     curr->data = a_code;
00094 }
00095
00096 void huffman::createTree() {
00097     //Creating Huffman Tree with the Min Heap created earlier
00098     Node *left, *right;
00099     priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
00100     while (tempPQ.size() != 1)
00101     {
00102         left = tempPQ.top();
00103         tempPQ.pop();
00104
00105         right = tempPQ.top();
00106         tempPQ.pop();
00107
00108         root = new Node();
00109         root->freq = left->freq + right->freq;
00110
00111         root->left = left;
00112         root->right = right;
00113         tempPQ.push(root);
00114     }
00115 }
```

```
00116
00117 void huffman::createCodes() {
00118     //Traversing the Huffman Tree and assigning specific codes to each character
00119     traverse(root, "");
00120 }
00121
00122 void huffman::saveEncodedFile() {
00123     //Saving encoded (.huf) file
00124     inFile.open(inFileName, ios::in);
00125     outFile.open(outFileName, ios::out | ios::binary);
00126     string in = "";
00127     string s = "";
00128     char id;
00129
00130     //Saving the meta data (huffman tree)
00131     in += (char)minHeap.size();
00132     priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
00133     while (!tempPQ.empty()) {
00134         Node* curr = tempPQ.top();
00135         in += curr->data;
00136         //Saving 16 decimal values representing code of curr->data
00137         s.assign(127 - curr->code.length(), '0');
00138         s += '1';
00139         s += curr->code;
00140         //Saving decimal values of every 8-bit binary code
00141         in += (char)binToDec(s.substr(0, 8));
00142         for (int i = 0; i < 15; i++) {
00143             s = s.substr(8);
00144             in += (char)binToDec(s.substr(0, 8));
00145         }
00146         tempPQ.pop();
00147     }
00148     s.clear();
00149
00150     //Saving codes of every character appearing in the input file
00151     inFile.get(id);
00152     while (!inFile.eof()) {
00153         s += arr[id]->code;
00154         //Saving decimal values of every 8-bit binary code
00155         while (s.length() > 8) {
00156             in += (char)binToDec(s.substr(0, 8));
00157             s = s.substr(8);
00158         }
00159         inFile.get(id);
00160     }
00161
00162     //Finally if bits remaining are less than 8, append 0's
00163     int count = 8 - s.length();
00164     if (s.length() < 8) {
00165         s.append(count, '0');
00166     }
00167     in += (char)binToDec(s);
00168     //append count of appended 0's
00169     in += (char)count;
00170
00171     //write the in string to the output file
00172     outFile.write(in.c_str(), in.size());
00173     inFile.close();
00174     outFile.close();
00175 }
00176
00177 void huffman::saveDecodedFile() {
00178     inFile.open(inFileName, ios::in | ios::binary);
00179     outFile.open(outFileName, ios::out);
00180     unsigned char size;
00181     inFile.read(reinterpret_cast<char*>(&size), 1);
00182     //Reading count at the end of the file which is number of bits appended to make final value 8-bit
00183     inFile.seekg(-1, ios::end);
00184     char count0;
00185     inFile.read(&count0, 1);
00186     //Ignoring the meta data (huffman tree) (1 + 17 * size) and reading remaining file
00187     inFile.seekg(1 + 17 * size, ios::beg);
00188
00189     vector<unsigned char> text;
00190     unsigned char textseg;
00191     inFile.read(reinterpret_cast<char*>(&textseg), 1);
00192     while (!inFile.eof()) {
00193         text.push_back(textseg);
00194         inFile.read(reinterpret_cast<char*>(&textseg), 1);
00195     }
00196
00197     Node *curr = root;
00198     string path;
00199     for (int i = 0; i < text.size() - 1; i++) {
00200         //Converting decimal number to its equivalent 8-bit binary code
00201         path = decToBin(text[i]);
00202         if (i == text.size() - 2) {
```

```
00203             path = path.substr(0, 8 - count0);
00204         }
00205         //Traversing huffman tree and appending resultant data to the file
00206         for (int j = 0; j < path.size(); j++) {
00207             if (path[j] == '0') {
00208                 curr = curr->left;
00209             }
00210             else {
00211                 curr = curr->right;
00212             }
00213
00214             if (curr->left == NULL && curr->right == NULL) {
00215                 outFile.put(curr->data);
00216                 curr = root;
00217             }
00218         }
00219     }
00220     inFile.close();
00221     outFile.close();
00222 }
00223
00224 void huffman::getTree() {
00225     inFile.open(inFileName, ios::in | ios::binary);
00226     //Reading size of MinHeap
00227     unsigned char size;
00228     inFile.read(reinterpret_cast<char*>(&size), 1);
00229     root = new Node();
00230     //next size * (1 + 16) characters contain (char)data and (string)code[in decimal]
00231     for(int i = 0; i < size; i++) {
00232         char aCode;
00233         unsigned char hCodeC[16];
00234         inFile.read(&aCode, 1);
00235         inFile.read(reinterpret_cast<char*>(hCodeC), 16);
00236         //converting decimal characters into their binary equivalent to obtain code
00237         string hCodeStr = "";
00238         for (int i = 0; i < 16; i++) {
00239             hCodeStr += decToBin(hCodeC[i]);
00240         }
00241         //Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1' character
00242         int j = 0;
00243         while (hCodeStr[j] == '0') {
00244             j++;
00245         }
00246         hCodeStr = hCodeStr.substr(j+1);
00247         //Adding node with aCode data and hCodeStr string to the huffman tree
00248         buildTree(aCode, hCodeStr);
00249     }
00250     inFile.close();
00251 }
00252
00253 // Compressing order of functions
00254 void huffman::compress() {
00255     createMinHeap();
00256     createTree();
00257     createCodes();
00258     saveEncodedFile();
00259 }
00260
00261 // Decompressing order of functions
00262 void huffman::decompress() {
00263     getTree();
00264     saveDecodedFile();
00265 }
```

5.4 encode.cpp File Reference

```
#include <iostream>
#include "functions.h"
```

Functions

- int main (int argc, char *argv[])

5.4.1 Detailed Description

Author

Yassine Bendimerad (yb308985@student.polsl.pl)

Version

0.1

Date

2023-01-13

Copyright

A lossless compression using Huffman algorithm

CPP FILE to run the compression(zip)(encode) an INPUT TEXT FILE to a compressed (.huf) file.

Definition in file encode.cpp.

5.4.2 Function Documentation

5.4.2.1 main()

```
int main (
             int argc,
             char * argv[] )
```

Definition at line 17 of file encode.cpp.

```
00017                                   {
00018     if (argc != 3) {
00019         //Error message if failed to detect file
00020         std::cout«"Failed to detect Files";
00021         exit(1);
00022     }
00023
00024     // starting the compression
00025     huffman f(argv[1], argv[2]);
00026     f.compress();
00027     cout«"Compressed successfully"«std::endl;
00028
00029     return 0;
00030
00031 }
```

5.5 encode.cpp

Go to the documentation of this file.

```
00001
00014 #include <iostream>
00015 #include "functions.h"
00016
00017 int main(int argc, char* argv[]) {
00018     if (argc != 3) {
00019         //Error message if failed to detect file
00020         std::cout«"Failed to detect Files";
00021         exit(1);
00022     }
00023
00024     // starting the compression
00025     huffman f(argv[1], argv[2]);
00026     f.compress();
00027     cout«"Compressed successfully"«std::endl;
00028
00029     return 0;
00030
00031 }
```

5.6 functions.h File Reference

```
#include <string>
#include <vector>
#include <queue>
#include <fstream>
```

Classes

- struct Node
- class huffman

5.6.1 Detailed Description

Fundamentals of Computer Programing Final Project, 1st Semester, Silesian University of Technology (Gliwice Poland), Informatics in english major. Program that allows zipping and unzipping files. Program should implement DEFLATE algorithm by itself (no external library). Program should be able to pack folder structure. Parameters for DEFLATE algorithm should be kept in config file. Program takes two parameters: -i input file to pack -o output where to store zipped file

Author

Yassine Bendimerad (yb308985@student.polsl.pl)

Version

0.1

Date

2023-01-16

Copyright

Copyright (c) 2023

Header file to define the functions for all CPP files

Definition in file functions.h.

5.7 functions.h

Go to the documentation of this file.

```
00001
00022 //Header Guards to prevent header files from being included multiple times
00023 #ifndef HUFFMAN_HPP
00024 #define HUFFMAN_HPP
00025 #include <string>
00026 #include <vector>
00027 #include <queue>
00028 #include <fstream>
00029
00030
00031 //Defining Huffman Tree Node
00032 struct Node {
00033     char data;
00034     unsigned freq;
00035     std::string code;
00036     Node *left, *right;
00037
00038     Node(){
00039         left = right = NULL;
00040     }
00041 };
00042
00043 using namespace std;
00044
00045 class huffman {
00046     private:
00047         vector  <Node*> arr;
00048         fstream inFile, outFile;
00049         string inFileName, outFileName;
00050         Node *root;
00051
00052         class Compare {
00053             public:
00054                 bool operator() (Node* l, Node* r){
00055                     return l->freq > r->freq;
00056                 }
00057         };
00058
00059
00060         priority_queue <Node*, vector<Node*>, Compare> minHeap;
00061
00062         //Initializing a vector of tree nodes representing character's ascii value and initializing
    its frequency with 0
00063         void createArr();
00064
00065         //Traversing the constructed tree to generate huffman codes of each present character
00066         void traverse(Node*, string);
00067
00068         //Function to convert binary string to its equivalent decimal value
00069         int binToDec(string);
00070
00071         //Function to convert a decimal number to its equivalent binary string
00072         string decToBin(int);
00073
00074         //Reconstructing the Huffman tree while Decoding the file
00075         void buildTree(char, string&);
00076
00077         //Creating Min Heap of Nodes by frequency of characters in the input file
00078         void createMinHeap();
00079
00080         //Constructing the Huffman tree
00081         void createTree();
00082
00083         //Generating Huffman codes
00084         void createCodes();
00085
00086         //Saving Huffman Encoded File
00087         void saveEncodedFile();
00088
00089         //Saving Decoded File to obtain the original File
00090         void saveDecodedFile();
00091
00092         //Reading the file to reconstruct the Huffman tree
00093         void getTree();
00094
00095     public:
00096         //Constructor
00097         huffman(string inFileName, string outFileName)
00098         {
00099             this->inFileName = inFileName;
00100             this->outFileName = outFileName;
00101             createArr();
```

```
00102         }
00103         //Compressing input file
00104         void compress();
00105         //Decrompressing input file
00106         void decompress();
00107 };
00108 #endif
```

Index