

Text Compressor and Decompressor



Jaypee Institute of Information Technology

Odd Semester 2020

Course name - Data Structures

Course faculty - Dr. Parul Agarwal and Ms. Sarishty Gupta

By

Manu Rajput 19103190

Arjun Sharma 19103205

Batch - B6

Problem Statement:

We have achieved Text File compression and decompression with our program. Compression is required when you need to transfer big files over the internet. Once transferred the user on the other end can decompress the file again to get back the original data.

Project Brief :

In this project we have implemented the Huffman Algorithm for encoding and decoding. The Algorithm was developed by David A. Huffman while he was a student at MIT. The Algorithm works on the idea to store the most frequent characters in a smaller size and less frequent characters could be accommodated with a bigger size. The Algorithm encodes the input text files into bit streams of 0's and 1's.

THE ALGORITHM:

Let me demonstrate the Algorithm with an example.

Consider a string “ **MISSISSIPPI RIVER**”

The string has got **17** characters in it. If we were to save this string we know that each character would take one byte equivalent to 8 bits and the resulting file size would be **17*8= 136 bits**.

Lets see the extent to which we can compress this if we save the encoded string in to the memory.

Step 1:

Count the unique characters and their frequencies.

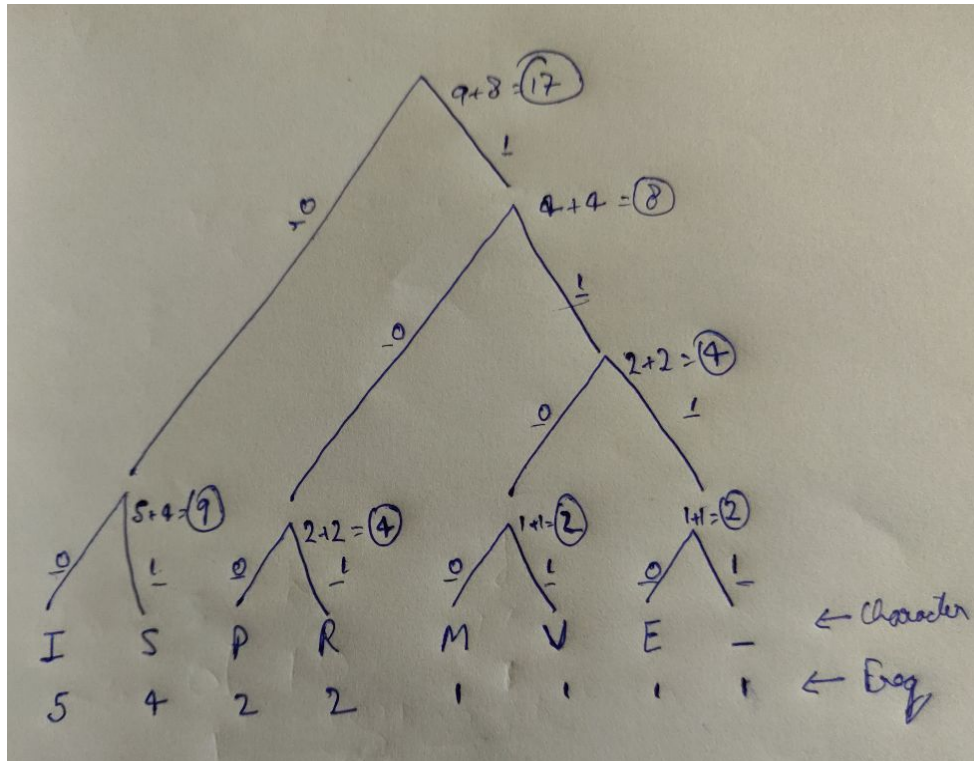
Character Frequency	
• M	1
• I	5
• S	4
• P	2
• R	2
• V	1
• E	1
• Space	1

$$1+5+4+2+2+1+1+1=17(\text{Total Characters})$$

Sort the unique characters according to their frequency

I(5) S(4) P(2) R(2) M(1) V(1) E(1) SPACE(1)

Lets generate the Huffman tree for this input. The characters are present at the leaves of the tree. This way it ensures that the path to each of the characters is unique thus ensuring the uniqueness of the codes.



To generate the code for each character start from the root node and append 0 when to travers left and append 1 when you go right.
The Huffman encoding for the above tree will be

- I - 00
- S - 01
- P - 100
- R - 101
- M - 1100
- V - 1101
- E - 1110
- SPACE - 1111

You the most frequent character I has the least bits of codes i.e. 00 and the least frequent character SPACE has the most bits 1111

The encoding string for "MISSISSIPPI RIVER" will be -

1100000101000101001001000011111010011011110101 i.e. 46 bits

**46 bits is way less than 136 bits of the original price. $46/136 = 33\%$
67% reduction of the original data.**

The Algorithm is an optimal prefix code which means that bit sequence or codes are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how the algorithm makes sure that there is no ambiguity when decoding the generated bitstream.

To understand the prefix codes let us consider four characters w, x, y and z and their corresponding codes be 00,01,0 and 1. This will lead to clashes while decoding as codes assigned to y is the prefix of codes assigned w and x. If the encoded bit streams for some string is 0001, the output may be "yyyz","yyx" etc.

Our program takes the input in the form of text file and saves the compressed string along with characters and their codes in a binary file(.bin).

List of data structure used in the program:

1. Tree
2. Priority Queue
3. Vectors

Project Design:

Brief Flow of the program:

1. Take the input from the input_text.txt file
2. Read the file and generate their codes
3. Write the huffman codes and encoded string to a new binary file(this will be our compressed file)

In you wish to decompress the file

1. Read the compressed binary file
2. Decode the file
3. Save the decoded test to new decoded_text.txt file

Class Models:

1. _node()

```
class _node{
public:
char data;
int freq;
_node* left;
```

```

_node* right;
_node(char data,int freq){
    this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(int freq){
    //this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(){
    this->left=NULL;
    this->right=NULL;
}
};

```

2. _singlechar()

```

class _singlechar{
public:
    char data;
    int freq;
    _singlechar(char data){
        this->data=data;
        this->freq=1;
    }
    void increment(){
        this->freq++;
    }
};

```

3. _code()

```

class _code{
public:
    char character;
    string code;
    _code(char c,string s){
        this->character=c;
        this->code=s;
    }
};

```

Main Functions

1. **compress()** - to write the encoded huffman code of the input string, individual characters along with their codes and size(the number of unique characters in the input) in the following format
 - a. size
Character code
Character code
.
Encoded String

NOTE - The format is really important as we will be reading the compressed file back in the same format to uncompress it.

2. **decode()** - This function reads the compressed binary file decodes the stored bits to back and the original data is restored.

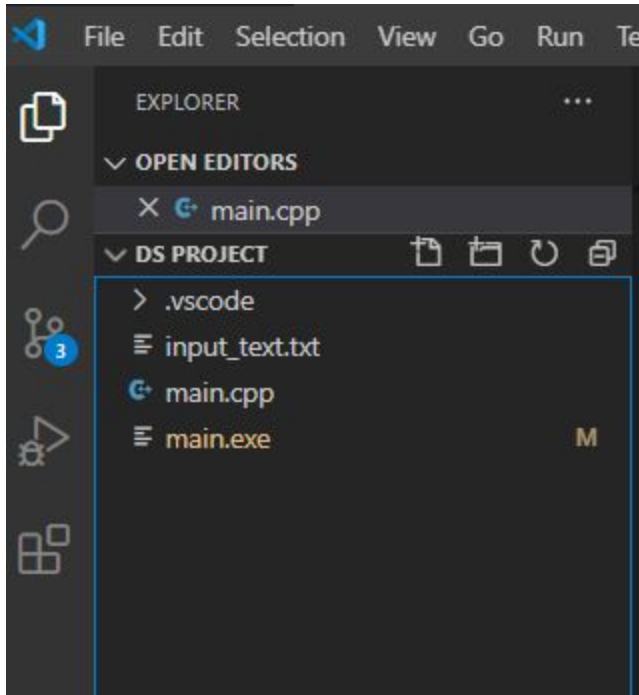
Helper functions:

1. `binary_to_decimal` - takes the input in binary and returns its decimal value
2. `decimal_to_binary` - takes decimal and returns binary of that
3. `Compare` - This is a FUNCTOR, loaded to compare two nodes by their frequency
4. `huffmantree()` and `printtree()` to generate the huffman coding of the unique characters
5. `ins()` - to get the unique characters count their frequency

Working Demo

Snippets of the working code

1. Directory in before running the program



The input_text contains the random string to be compressed

2. Lets build and run the code

```
PROBLEMS  TERMINAL  ...  2: Task - C/C++: g++.exe  +  [ ]  [X]  ^  X

> Executing task: C/C++: g++.exe build active file <

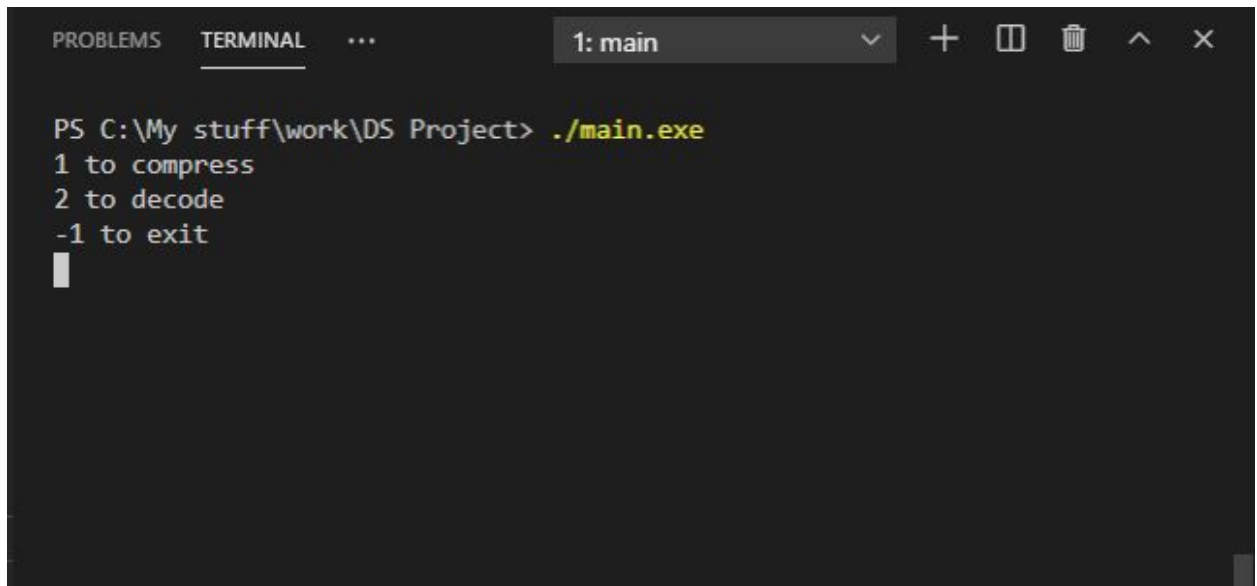
Starting build...
Build finished with errors(s):
C:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/bin/.
./lib/gcc/x86_64-w64-mingw32/8.1.0/../../../../x86_64-w64-mingw32/bin/ld.e
xe: cannot open output file C:\My stuff\work\DS Project\main.exe: Permissi
on denied
collect2.exe: error: ld returned 1 exit status

The terminal process terminated with exit code: -1.

Terminal will be reused by tasks, press any key to close it.
```

Build successful

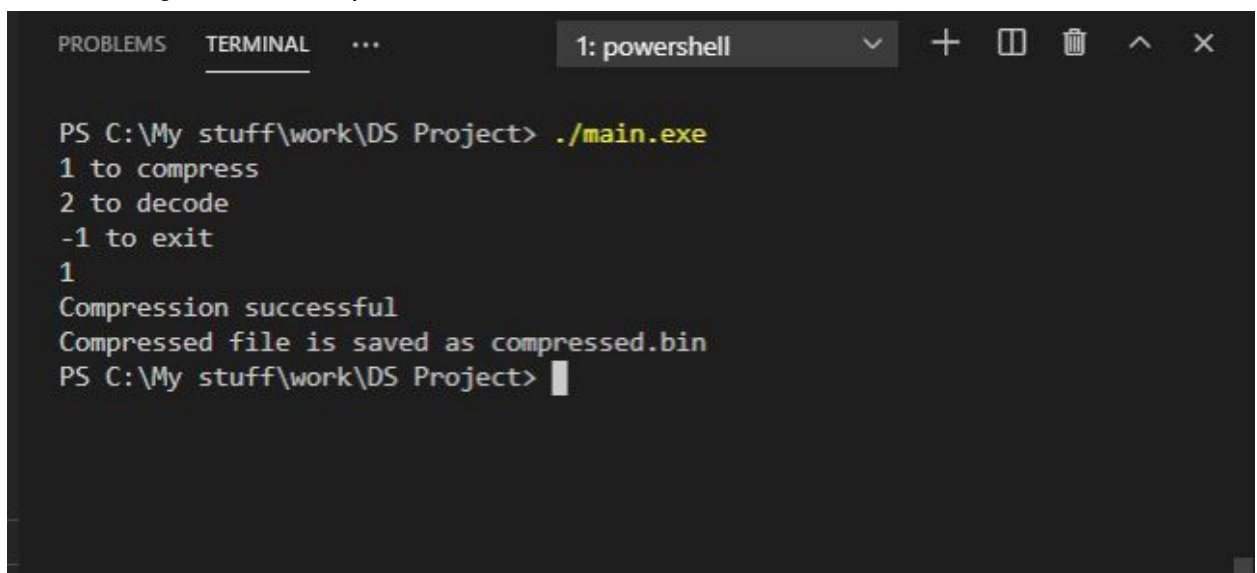
3. ./main.exe to run the executable file



```
PROBLEMS  TERMINAL  ...  1: main  +  [ ]  [ ]  ^  X

PS C:\My stuff\work\DS Project> ./main.exe
1 to compress
2 to decode
-1 to exit
█
```

4. I am entering 1 one to compress the file

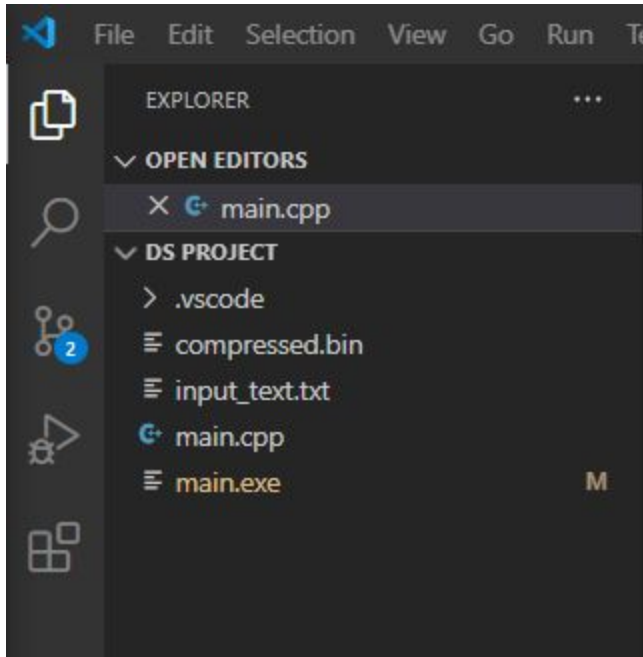


```
PROBLEMS  TERMINAL  ...  1: powershell  +  [ ]  [ ]  ^  X

PS C:\My stuff\work\DS Project> ./main.exe
1 to compress
2 to decode
-1 to exit
1
Compression successful
Compressed file is saved as compressed.bin
PS C:\My stuff\work\DS Project> █
```

Compression successful. The compressed file is saved as compressed.bin .

5. Lets check the working directory and compare the size of the compressed.bin file with the input_text.txt file.



A new compressed.bin file is created.
If we compare the size of the two files.

A screenshot of a Windows File Explorer window showing the contents of the 'DS Project' folder. The files and their sizes are listed in a table. The sizes for 'compressed.bin' (164 KB) and 'input_text.txt' (200 KB) are circled in red.

Name	Date modified	Type	Size
.vscode	08-12-2020 19:01	File folder	
compressed.bin	08-12-2020 19:33	BIN File	164 KB
input_text	08-12-2020 19:01	Text Document	200 KB
main.cpp	08-12-2020 19:09	C++ Source	12 KB
main	08-12-2020 19:02	Application	439 KB

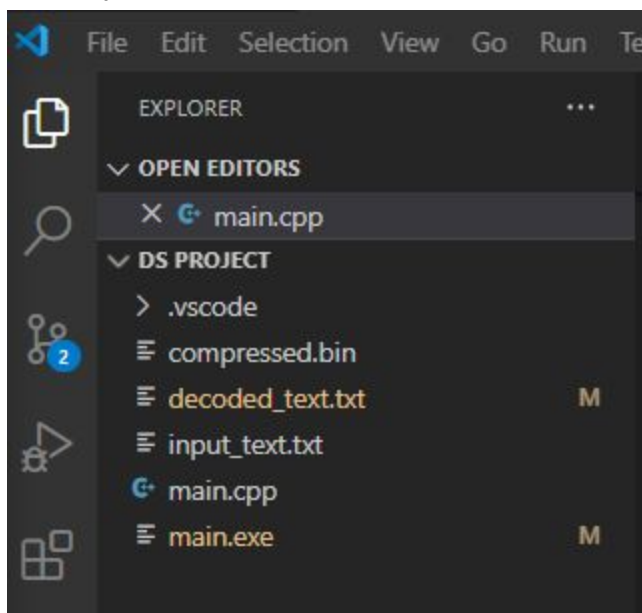
The file size of the compressed file is 164 kbs as compared to 200 kbs of the input_text file.

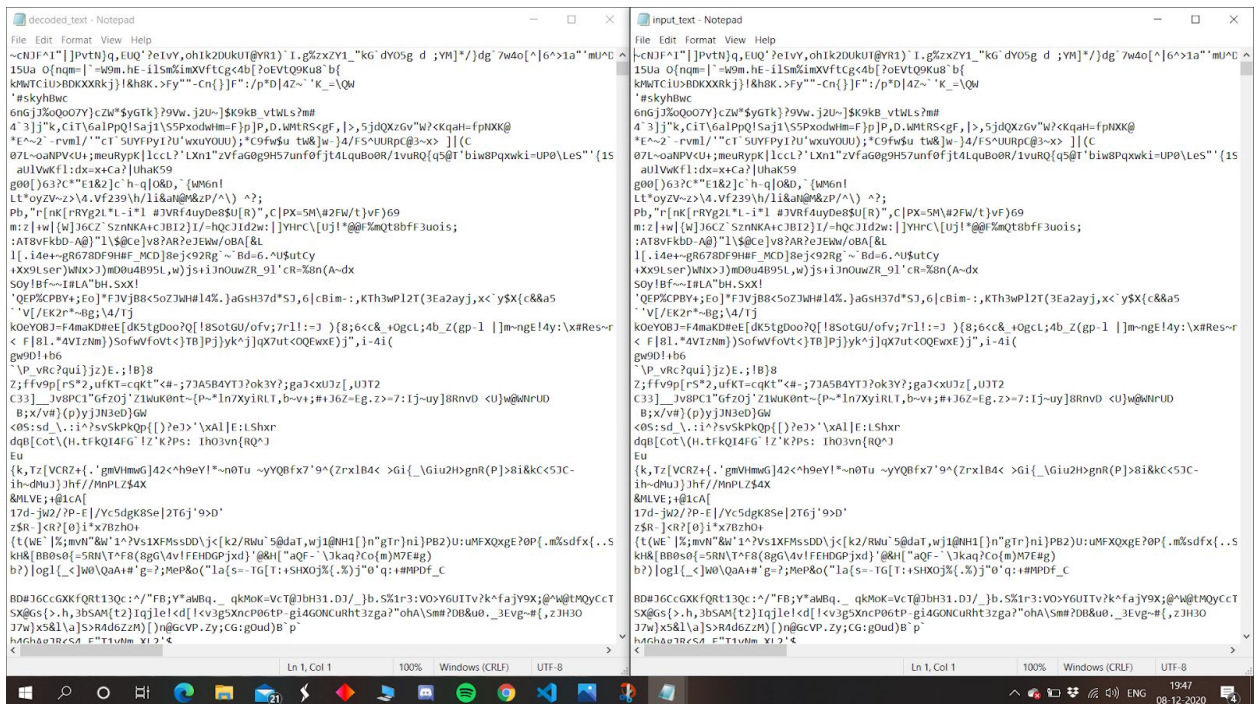
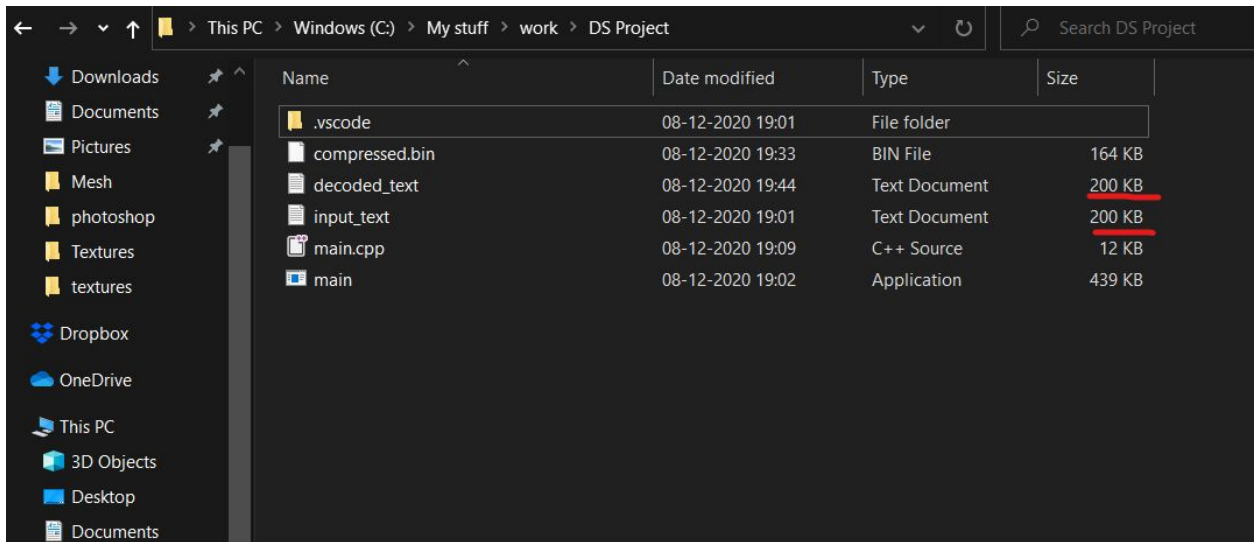
6. Now lets try to decompress "compressed.bin" file and obtain our data back
Run the ./main.exe command to run the program and enter 2 to decompress file. The decompression takes a bit longer as compared to compression.

```
PROBLEMS  TERMINAL  ...  1: powershell  +  [ ]  [ ]  ^  X

PS C:\My stuff\work\DS Project> ./main.exe
1 to compress
2 to decode
-1 to exit
1
Compression successful
Compressed file is saved as compressed.bin
PS C:\My stuff\work\DS Project> ./main.exe
1 to compress
2 to decode
-1 to exit
2
Decompression successful
File saved as decoded_text.txt
PS C:\My stuff\work\DS Project> [ ]
```

7. Decompression successful the decoded file is saved as decoded_text.txt. Lets check the directory now and compare the size and content of the two files.





The content of the two files is exactly the same.

The entire code(421 lines):
Content of main.cpp-

```

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

```

```
using namespace std;
```

```
int binary_to_decimal(string in)
```

```
{
    int result = 0;
    for (int i = 0; i < in.size(); i++)
        result = result * 2 + in[i] - '0';
    return result;
}
```

```
string decimal_to_binary(int in)
```

```
{
    string temp = "";
    string result = "";
    while (in)
    {
        temp += ('0' + in % 2);
        in /= 2;
    }
    result.append(8 - temp.size(), '0');
    //append '0' ahead to let the result become fixed length of 8
    for (int i = temp.size() - 1; i >= 0; i--)
    {
        result += temp[i];
    }
    return result;
}
```

```
class _code{
```

```
public:
    char character;
    string code;
    _code(char c,string s){
        this->character=c;
        this->code=s;
    }
};
```

```
class _singlechar{
```

```
public:
    char data;
    int freq;
    _singlechar(char data){
        this->data=data;
        this->freq=1;
    }
    void increment(){
        this->freq++;
    }
};
```

```
class _node{
```

```
public:
    char data;
```

```

int freq;
_node* left;
_node* right;
_node(char data,int freq){
    this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(int freq){
    //this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(){
    this->left=NULL;
    this->right=NULL;
}
};

struct compare{
    bool operator()(_node*l,_node*r){
        return(l->freq > r->freq);
    }
};

void printtree(_node* root,string code,vector<_code*> &coding){
    if(root==NULL){
        return;
    }
    if(root->left==NULL && root->right==NULL){
        //cout<<root->data<<" : " << code <<endl;
        coding.push_back(new _code(root->data,code));
    }
    printtree(root->left, code+'0',coding);
    printtree(root->right,code+'1',coding);
}

void huffmantree(vector<_code*> &coding,vector<_singlechar*> &a,int size){
    //class _node{ char data; int freq; _node* left; _node* right;}
    _node* left,*right,*top;
    priority_queue<_node*, vector<_node*>, compare> q ;
    for(int i=0;i<size;i++){
        q.push(new _node(a[i]->data,a[i]->freq));
    }

    while(q.size()!=1){
        left=q.top();
        q.pop();
        right=q.top();
        q.pop();
        top=new _node(left->freq+right->freq);
    }
}

```

```

        top->left=left;
        top->right=right;
        q.push(top);
    }
    printtree(q.top(),"",coding);
}

void ins(vector<_singlechar*> &a,char b){
    if(a.empty()){
        a.push_back(new _singlechar(b));
    }
    else{
        auto i=a.begin();
        int flag=0;
        while(i!=a.end()){
            if((*i)->data>=b){
                if((*i)->data==b){
                    ((*i)->freq)++;
                }
                else{
                    a.insert(i,new _singlechar(b));
                }
                flag=1;
                break;
            }
            i++;
        }
        if(flag==0){
            a.push_back(new _singlechar(b));
        }
    }
}

void compress(){
    //class _singlechar{public:char data;int freq;_singlechar(char data){this->data=data;this->freq=1;}void
increment(){this->freq++;}}
    //vector to store unique characters along with their frequency
    vector<_singlechar*> a;
    //string str
    //cin>>str;
    //int size = str.size();

    char c;
    ifstream read;
    read.open("input_text.txt");
    if(read.fail()){
        cout<<"couldn't open source text file"<<endl;
        exit(1);
    }
    else{
        while(read.get(c)){
            ins(a,c);
        }
    }
}

```

```

}
read.close();
//class _code{public:char character;string code;_code(char c,string s){this->character=c;this->code=s;}};
//vector to store characters along with their coding
vector<_code*> coding;
huffmantree(coding,a,a.size());
/*
for(auto i=coding.begin();i!=coding.end();i++){
    cout<<(*i)->character<<"writing"<<(*i)->code<<endl;
}
*/
ofstream out_file;
ifstream in_file;
in_file.open("input_text.txt",ios::in);
out_file.open("compressed.bin",ios::binary|ios::out);
string in = "",s="";
//the first byte saves the size of the vector
//in+=(char)coding.size();
in+=coding.size();
for(auto i=coding.begin();i!=coding.end();i++){
    in+=(int)(*i)->character;
    //set the codes with a fixed 128-bit string form[000i1 + real code]
    //'1' indicates the start of huffman code
    s.assign(127-(((*i)->code).size(),'0');
    s+='1';
    s.append((*i)->code);
    in+=(char)binary_to_decimal(s.substr(0,8));
    for (int i = 0; i < 15; i++)
        { //cut into 8-bit binary codes that can convert into saving char needed for binary file
            s = s.substr(8);
            in += (char)binary_to_decimal(s.substr(0, 8));
        }
}
s.clear();
char b;
in_file.get(b);
while(!in_file.eof()){
    //convert char by char
    auto i=coding.begin();
    while((*i)->character!=b){
        i++;
    }
    s+=((*i)->code);
    while(s.size()>8){
        //chop into 8 bits
        in+=(char)binary_to_decimal(s.substr(0,8));
        s=s.substr(8);
    }
    in_file.get(b);
}
int count = 8-s.size();
if(s.size()<8){

```

```

        s.append(count,'0');
    }
    in+=(char)binary_to_decimal(s);
    in+=(char)count;
    //cout<<in<<endl;
    //cout<<in.size()<<endl;
    out_file.write(in.c_str(),in.size());
    in_file.close();
    out_file.close();
    cout<<"Compression successful"<<endl<<"Compressed file is saved as compressed.bin"<<endl;

}

char search(vector<_code*> &coding,string binary){
    auto i=coding.begin();
    while(i!=coding.end()){
        if((*i)->code==binary){
            break;
        }
        i++;
    }
    return (*i)->character;
}

void decode(){
    //class _code{ char character; string code }
    ifstream in_file;
    in_file.open("compressed.bin",ios::in|ios::binary);
    unsigned char size;
    in_file.read(reinterpret_cast<char*>(&size),1);

    //vector to store characters along with their decoding
    vector<_code*> decode;
    for(int i=0;i<size;i++){
        char a_code;
        unsigned char h_code_c[16];
        in_file.read(&a_code,1);

        in_file.read(reinterpret_cast<char*>(h_code_c),16);
        string h_code_s="";
        for(int i=0;i<16;i++){
            //obtain original 128 bit-binary string
            h_code_s+=decimal_to_binary(h_code_c[i]);
        }
        int j=0;
        while(h_code_s[j]!='0'){
            //delete the added '0000iii1' to get the real huffman code
            j++;
        }
        h_code_s=h_code_s.substr(j+1);
        //cout<<a_code<<"reading"<<h_code_s<<endl;
        decode.push_back(new _code(a_code,h_code_s));
    }
}

```



```

/*
class _node{
public:
char data;
int freq;
_node* left;
_node* right;
_node(char data,int freq){
    this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(int freq){
    //this->data=data;
    this->freq=freq;
    this->left=NULL;
    this->right=NULL;
}
_node(){
    this->left==NULL;
    this->right==NULL;
}
};
*/
//create a new huffman tree for decoding
_node * root=new _node();
_node * current;
auto i=decode.begin();
while(i!=decode.end()){
    current=root;
    char charac=(*)->character;
    string code=(*)->code;
    while(code.size()!=0){
        if(code[0]=='0'){
            if(!current->left){
                current->left=new _node();
                current=current->left;
                code=code.substr(1);
            }
            else{
                current=current->left;
                code=code.substr(1);
            }
        }
        else{
            if(!current->right){
                current->right=new _node();
                current=current->right;
                code=code.substr(1);
            }
            else{

```

```

        current=current->right;
        code=code.substr(1);
    }
}
current->data=charac;
i++;
}

in_file.close();
in_file.open("compressed.bin",ios::in|ios::binary);
fstream out_file;
out_file.open("decoded_text.txt",ios::out);
//unsigned char size;
//get number of codings
in_file.read(reinterpret_cast<char*>(&size),1);
//jump to the last one byte to get the number of '0' append to the string at last
in_file.seekg(-1,ios::end);
char count0;
in_file.read(&count0,1);
//jump to the position where text starts
in_file.seekg(1+17*size,ios::beg);
vector<unsigned char> text;
unsigned char textseg;
in_file.read(reinterpret_cast<char*>(&textseg),1);
while(!in_file.eof()){
    //get the text byte by byte using unsigned char
    text.push_back(textseg);
    in_file.read(reinterpret_cast<char*>(&textseg),1);
}
string path;
string complete="";
for(int i=0;i<text.size()-1;i++){
    //translate the huffman code
    path=decimal_to_binary(text[i]);
    if(i==text.size()-2){
        path = path.substr(0,8-count0);
    }
    complete=complete+path;
}
//cout<<complete<<endl;
string to_write="";
current=root;
while(complete.size()>0){
    if(complete[0]=='0'){
        current=current->left;
        complete=complete.substr(1);
    }
    else{
        current=current->right;
        complete=complete.substr(1);
    }
}

```

```

        if(current->left==NULL && current->right==NULL){
            to_write+=current->data;
            current=root;
        }
    }
    /*
while(complete.size()>0){
    temp+=complete[0];
    if(complete.size()==1){
        complete="";
    }
    else{
        complete=complete.substr(1);
    }
    int flag=0;
    auto i=decode.begin();
    while(i!=decode.end()){
        if(temp==(i->code)){
            flag=1;
            break;
        }
        i++;
    }
    if(flag==1){
        to_write+=(i->character);
        temp="";
    }

}
*/
//cout<<to_write<<endl;
out_file<<to_write;
in_file.close();
out_file.close();
cout<<"Decompression successful"<<endl<<"File saved as decoded_text.txt";

}

int main(){
    int choice=0;
    cout<<"1 to compress"<<endl<<"2 to decode"<<endl<<"-1 to exit"<<endl;;
    cin>>choice;
    switch(choice){
        case 1:
            compress();
            break;
        case 2:
            decode();
            break;
        default:
            cout<<"Enter a choice"<<endl;
    }
}

```

```
    return 0;  
}
```

End of report.

Thank you.