Huffman Algorithm

Проектът ми е изграден на основата на двоичното хъфманово дърво. Това представлява дърво в което върховете съдържат число което показва общата сума на срещаните символи на поддървото с корен съответният връх. Колкото по-често е срещан даден символ толкова по-близко до корена се намир върхът който съответсва на символа(тоест листото). Кодирането на информацията след това се осъществяв като всеки символ се замества с код който се получава след преминаване през дървото – ако сме тръгнали наляво съответният бит от кода е 0, а в противен случай 1.Имам специална структура **Char** която съдържа символ и съответният му битов код. След като имаме таблица от кодовете на символите кодирането става тривиално.

Процесът на кодиране е както следва. Създава се таблица от елементи които са структурата ми **Occurance** което представлява символ и брой срещания в текста. След това за всеки символ се създава хъфманово дърво които се вкарват в контейнер. След това сортирам контейнера в низходящ ред по брой срещания. След това обединявам последните две дървета в предпоследното(**merge**). След това пак сортирам, пак обединявам и така докато не остане само едно дърво което е крайното хъфманово дърво за даденият текст. След това правя търсене в дълбочина на дървото и създавам таблицата от кодове на символите. Имам функция което по символ ми търси в тази таблица и връща кодът. Обхождам текстът символ по символ и заместват символите с кодовете им. След това обхождам контейнера с кодовете и за всеки осем бита създавам структура **bitset<8>** като след това я записвам в променлива **usigned char**. Всичките получени така осем битови числа ги записвам двойчно във файла.

Поради нуждата от наличие на хъфманово дърво за декодиране на компресираните файлове, организацията на изходните файлове е както следва:

Първите 32 бита са определени за броят на записаните във файла битове – това ми е нужно за да определя при последният комплект от 8 бита колко да взема.

Следващите 32 бита са за големината на таблица със срещанията на символите.

Следва самата таблица и след нея е кодираната информация. По таблицата със срещанията създавам хъфманово дърво и декодирам текста.

За работа с файловите потоци както и за оценка на аргументите на програмата съм създал специален клас – **CompressionFunctionality.**

За реализирането на проекта съм използвал редица малки структури:

```
struct Char {
	char symbol;
	vector<bool> code;
}; - структура която представлява кодиран символ – от нея е изградена таблицата на кодираните символи чрез която кодирам и декодирам текстовете.

struct Occurance {
	char symbol;
	int count;
```

} – представлява обект който определя броя на срещанията на даден символ. От тази структура конструирам таблицата с информация за броя на срещанията на даден символ, а пък чрез такава таблица построявам дървото на хъфман.

```
struct Node {
	int occurances;
	int symbol;
	Node* left;
	Node* right;
```

} – това е структурата която представлява връх в дървото на хъфман. Ако даден връх не е листо то тогава стойността на променливата **symbol** е -1 което е невалидно при положение че използвам само **unsigned char** като символен тип.

Всяко хъфманово дърво е изградено от указател към корена на дървото и таблица със срещанията на символите. Функциите които представлява този клас съм раздели главно на два типа: такива които са рекурсивни – те са защитени чрез тип **private** и всички останали. Например функцията за принтиране ми извиква рекурсивната функция **_print(Node*)**;

Private:

`void _print(Node* node)` – обхожда рекурсивно корена, лявото и дясното поддърво като изпечатва стойностите им на конзолата.

`void _buildTree(vector<Occurance>& frequency_table)` – построява дървото по описаният по-горе алгоритъм – за всяка буква създава дърво и започва да ги слива.

`void _make_table(string text, vector<Occurance>& frequency_table)` – попълвам таблицата със срещанията. Търся дали даден символ се е срещал и ако да – увеличавам му **counter**-а.

`void _destroy(Node* node)` – рекурсивна функция за изчисване на паметта

Public:

constructors:

`HuffmanTree(int o, int s = -1)` – използвам за построяване на дърво само с един връх – за отделните букви.

`HuffmanTree(string text)` – цялостен workflow за построяване на дървото. Първо популвам таблицата за срещанията, после извиквам **_buildTree**.

`HuffmanTree(vector<Occurance>& frequency_table)` – извиквам директно **_buildTree**. Използвам този конструктор при декомпресиране, когато вече имам готовата таблица.

destructor: *извиквам _destroy за корена на дървото*

Помощни:

`void merge(HuffmanTree*&)` – деструктивна функция която слива дървото подадено като параметър с текущото дърво. След операцията подаденото като параметър дърво не съществува вече.

`bool operator>(HuffmanTree&)` и `bool operator<(HuffmanTree&)` – функции реализиращи определянето на наредба за хъфмановите дървета – ползвам ги при сортирането.

`vector<Char> make_alphabet()` – функция която създава таблица(азбука) от кодираните символи използвайки

```
void add_to_alphabet_recursively(
        vector<bool> container,
        vector<Char>& alphabet,
        Node* node)
```

която рекурсивно обхожда върховете като пази текущият път във container.

`vector<unsigned char> compress(string text, int& totalCount)` – създава таблицата за кодиране и обхожда текстът символ по символ. Преди да върне резултата извиква помощната функция `vector<unsigned char> make_ints(vector<bool> compressed_bits)` – тя приема контейнера от преведените символи и ги групира в числа по 8 бита готови за запис във файл.Също така задава стойност на променливата **totalCount** за да знаем колко бита сме записали – използвам го при декомпресирането за да знам колко бита от последното число трябва да не прочета.

`string decompress(vector<unsigned char> compressed_bits,int intsCount)` – създавам първоначалният контейнер от компресирани битове. След това го обхождам бит по бит и според стойността се движа наляво или надясно във дървото. Ако стигна до листо записвам символа и се връщам във корена.

`vector<Occurance>& getTable()` – getter за списъка на срещанията – ползвам го при записване на таблицата в компресираният файл.

Следва класът който ми отговаря за достъпването на цялата функционалност на дървото по user-friendly начин – **CompressionFunctionality**. Тук имам само три функции: за компресиране, за декомпресиране и за прочитане на параметрите от конзолата.

```
void compress(string filename_in, string filename_out)
void decompress(string filename_in, string filename_out)
```

На ниско ниво първите две фукции първо четата от файл след това извършват главаната функционалност построявайки дръвото на хъфман и след това записват резултата във файл. Съответно първите параметри са за файлът от който четем, а вторите за файлът където пишем. И двата файла се отварят двойно.

`void parameters_parse(int argc, char** argv)` – позволява употребата на цялата функционалност чрез зададени параметри на конзолният ред.Позволените параметри са както следва:

-c[ompress] задава режим на работа за компресия след него трябва да се зададът имената на файловете чрез параметрите –i за входящият файл и –o за изходящият.

-d[ompress] аналогично, като задава режим на работа за декомпресиране на входният файл.