***Wojskowa Akademia Techniczna***

***im. Jarosława Dąbrowskiego***

Laboratorium z przedmiotu:

Interfejsy komputerów cyfrowych

Sprawozdanie nr 1:

**Kodowanie huffmana**

Prowadząca:

mgr inż. Małgorzata Pisula

**Wykonał:** Radosław Relidzyński

**Grupa:** WCY20IY4S1

**Data laboratoriów**: 03.06.2022 r.

Spis treści

[Kompletny program 2](#_Toc105197847)

[Przykład działania (plik: „smalltest.txt”) 2](#_Toc105197848)

[Przykład działania (plik „test.txt”) 3](#_Toc105197849)

[Użyte struktury 5](#_Toc105197850)

[Analiza tworzenia modelu 6](#_Toc105197851)

[Analiza tworzenia drzewa kodowania 7](#_Toc105197852)

[Analiza tworzenia tablicy kodowej 9](#_Toc105197853)

[Analiza kompresji 11](#_Toc105197854)

[Analiza pobierania tablicy kodowej 12](#_Toc105197855)

[Analiza dekompresji 12](#_Toc105197856)

# Kompletny program

#include<iostream>

#include<algorithm>

#include<fstream>

#include<bitset>

#include<map>

using namespace std;

struct HuffmanNode{

    int s;

    int f;

};

typedef struct HuffmanNode HuffmanNode;

vector<HuffmanNode> huffmanModelNodes;

struct HuffmanNodeTree{

    string s;

    int f;

    string code;

    HuffmanNodeTree \*parent;

    HuffmanNodeTree \*l\_child;

    HuffmanNodeTree \*r\_child;

};

typedef struct HuffmanNodeTree HuffmanNodeTree;

HuffmanNodeTree \*root;

int readedBytesCount = 0;

map<string, string> codes;

/////////////

//Kompresja//

/////////////

int CompareHuffmanNodes(HuffmanNode n1, HuffmanNode n2){

    // Funkcja pomocnicza do sortowania modelu

    if(n1.f == n2.f){

        return n1.s > n2.s;

    }

    else{

        return n1.f > n2.f;

    }

}

void GenerateModelFromFile(char\* inputTextFile, vector<HuffmanNode> \*huffmanModel, int \*readedBytesCount){

    // Wyznacza model dla zadanego pliku i zapisuje go w odpowiedniej strukturze

    FILE \*textfile = fopen(inputTextFile, "r");

    if(!textfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    int readCount = 0;

    unsigned char buffer[1];

    int readBytesLength = 1;

    int countBytesReaded = 0;

    bool found;

    while (readCount = fread(buffer, sizeof(unsigned char), readBytesLength, textfile)){

        //Obsługa odczytanego bajtu

        found = false;

        // Szukanie w aktualnym zbiorze symboli

        for(HuffmanNode &x : \*huffmanModel){

            if(x.s==int(buffer[0])){

                x.f++;

                found = true;

            }

        }

        // Jesli jest to nowy symbol

        if(!found){

            HuffmanNode huffmanNode;

            huffmanNode.s=int(buffer[0]);

            huffmanNode.f=1;

            huffmanModel->push\_back(huffmanNode);

        }

        countBytesReaded++;

    }

    \*readedBytesCount = countBytesReaded;

    fclose(textfile);

    sort(huffmanModelNodes.begin(), huffmanModelNodes.end(), CompareHuffmanNodes);

}

void WriteModelToFile(char\* outputModelFile, vector<HuffmanNode> huffmanModelNodes){

    // Zapisuje wyznaczony model do pliku tekstowego (z odpowiednim formatowaniem)

    ofstream modelfile;

    modelfile.open(outputModelFile);

    if(!modelfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    modelfile << huffmanModelNodes.size() << endl;

    for(HuffmanNode x : huffmanModelNodes){

        modelfile << x.s << ":" << x.f << endl;

    }

    modelfile.close();

}

int CompareHuffmanTreeNodes(HuffmanNodeTree \*n1, HuffmanNodeTree \*n2){

    // Funkcja pomocnicza do sortowania drzewa

    return n1->f > n2->f;

}

void GenerateHuffmanTreeFromModel(vector<HuffmanNode> huffmanModelNodes){

    // Generuje drzewo kodowania

    vector<HuffmanNodeTree\*> huffmanNodeTreeVector;

    for (HuffmanNode x : huffmanModelNodes){

        HuffmanNodeTree \*temp = new HuffmanNodeTree;

        temp->s=to\_string(x.s);

        temp->f=x.f;

        temp->parent=NULL;

        temp->l\_child=NULL;

        temp->r\_child=NULL;

        huffmanNodeTreeVector.push\_back(temp);

    }

    for(int i=1; huffmanNodeTreeVector.size()!=1; i++){

        HuffmanNodeTree \*temp = new HuffmanNodeTree;

        temp->parent=NULL;

        temp->s="#"+ to\_string(i);

        temp->l\_child = new HuffmanNodeTree;

        temp->l\_child=huffmanNodeTreeVector[huffmanNodeTreeVector.size()-1];

        temp->l\_child->parent=temp;

        temp->r\_child = new HuffmanNodeTree;

        temp->r\_child=huffmanNodeTreeVector[huffmanNodeTreeVector.size()-2];

        temp->r\_child->parent=temp;

        temp->f = temp->l\_child->f + temp->r\_child->f;

        huffmanNodeTreeVector.pop\_back();

        huffmanNodeTreeVector.pop\_back();

        huffmanNodeTreeVector.push\_back(temp);

        sort(huffmanNodeTreeVector.begin(), huffmanNodeTreeVector.end(), CompareHuffmanTreeNodes);

    }

    root = huffmanNodeTreeVector[0];

}

void WriteHuffmanTreeToFile\_preorder(HuffmanNodeTree \*curr, FILE \*treefile) {

    // Funkcja pomocnicza dla zapisywania drzewa - przeszukuje w kolejności preorder

    if(curr){

        // Wezel

        fprintf(treefile, "%s ", curr->s.c\_str());

        // Czestotliwosc

        fprintf(treefile, "%s ", to\_string(curr->f).c\_str());

        // Lewy potomek

        if(curr->l\_child)

            fprintf(treefile, "%s ", curr->l\_child->s.c\_str());

        else

            fprintf(treefile, "nn ");

        // Prawy potomek

        if(curr->r\_child)

            fprintf(treefile, "%s ", curr->r\_child->s.c\_str());

        else

            fprintf(treefile, "nn ");

        // Rodzic

        if(curr->parent)

            fprintf(treefile, "%s ", curr->parent->s.c\_str());

        else

            fprintf(treefile, "nn ");

        fprintf(treefile, "\n");

        if(curr->l\_child != NULL) WriteHuffmanTreeToFile\_preorder(curr->l\_child, treefile);

        if(curr->r\_child != NULL) WriteHuffmanTreeToFile\_preorder(curr->r\_child, treefile);

    }

}

void WriteHuffmanTreeToFile(char\* outputTreeFile, HuffmanNodeTree \*root){

    // Zapisuje drzewo kodowania do pliku

    FILE \*treefile = fopen(outputTreeFile, "w");

    if(!treefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    fprintf(treefile, "wezel czestosc l\_child r\_child parent\n");

    WriteHuffmanTreeToFile\_preorder(root, treefile);

    fclose(treefile);

}

void GenerateCodeTableFromTree(HuffmanNodeTree \*curr, string code){

    // Generuje kody znakow - przeszukuje w kolejności preorder

    if(curr){

            curr->code=code;

        if(curr->l\_child) GenerateCodeTableFromTree(curr->l\_child, code+"0");

        if(curr->r\_child) GenerateCodeTableFromTree(curr->r\_child, code+"1");

    }

}

void WriteCodeTableToFile\_preorder(HuffmanNodeTree \*curr, FILE \*codefile) {

    // Funkcja pomocnicza dla zapisywania kodów - przeszukuje w kolejności preorder

    if(curr==NULL) return;

    if(curr->s[0] != '#'){

        fprintf(codefile, "%s %s\n", curr->s.c\_str(), curr->code.c\_str());

    }

    if(curr->l\_child != NULL) WriteCodeTableToFile\_preorder(curr->l\_child, codefile);

    if(curr->r\_child != NULL) WriteCodeTableToFile\_preorder(curr->r\_child, codefile);

}

void WriteCodeTableToFile(char\* outputCodeFile, HuffmanNodeTree \*root){

    // Zapisuje kody znakow do pliku

    FILE \*codefile;

    codefile = fopen(outputCodeFile, "w");

    if(!codefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    fprintf(codefile, "znak kod\n");

    WriteCodeTableToFile\_preorder(root, codefile);

    fclose(codefile);

}

string GetSymbolCodeFromTree(HuffmanNodeTree \*curr, string symbol){

    // Funkcja zwracajaca kod znakow na podstawie symbolu

    if(curr==NULL) return "";

    // Znaleziony

    if(curr->s==symbol) return curr->code;

    // Szukaj

    if(curr->l\_child != NULL)

        if(GetSymbolCodeFromTree(curr->l\_child, symbol)!="")

            return GetSymbolCodeFromTree(curr->l\_child, symbol);

    if(curr->r\_child != NULL)

        if(GetSymbolCodeFromTree(curr->r\_child, symbol)!="")

            return GetSymbolCodeFromTree(curr->r\_child, symbol);

    // Koncowy element drzewa, nie ten znak

    return "";

}

void WriteCompressedFile(char\* input\_text\_file, char\* output\_compressed\_file, HuffmanNodeTree \*root){

    // Funkcja dokonujaca kompresji pliku

    FILE \*textfile = fopen(input\_text\_file, "r");

    if(!textfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    ofstream compressedfile(output\_compressed\_file, std::ios\_base::out | std::ios::binary);

    if(!compressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string binary;

    unsigned char buffer[1];

    if(textfile){

        while(fread(buffer, sizeof(unsigned char), 1, textfile)){

            binary += GetSymbolCodeFromTree(root, to\_string(int(buffer[0])));

            if(binary.size()>=8){

                compressedfile << char(stoi(binary.substr(0,8), 0, 2));

                binary.erase(0, 8);

            }

        }

    }

    fclose(textfile);

    if(binary.size()) compressedfile << char(stoi(binary, 0, 2));

    compressedfile.close();

}

///////////////

//Dekompresja//

///////////////

void ImportAndMapCodesFromFile(char\* output\_code\_file, map<string, string> \*codes){

    // Funkcja pobierająca tablicę kodową do programu

    ifstream codefile;

    codefile.open(output\_code\_file);

    if(!codefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string code, symbol, temp;

    codefile >> symbol >> code;

    while(!codefile.eof()){

        codefile >> symbol >> code;

        codes->insert(pair<string, string>(code, symbol));

    }

    codefile.close();

}

string DecompressCurrentBitset(string \*code, map<string, string> codes){

    string result, binary;

    int i=0; // Indeks dla niezdekompresowanych bitow

    for(char c : \*code){

        binary+=c;

        if(codes.find(binary) != codes.end()){

            result+=char(stoi(codes.find(binary)->second));

            i+=binary.size();

            binary="";

        }

    }

    \*code=code->erase(0, i);

    return result;

}

void WriteDecompressedFile(char\* input\_compressed\_file, char\* output\_decompressed\_file, map<string, string> codes){

    // Dekompresuje plik do pierwotnej postaci

    ifstream compressedfile(input\_compressed\_file, ios\_base::in | ios::binary);

    if(!compressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    ofstream decompressedfile;

    decompressedfile.open(output\_decompressed\_file);

    if(!decompressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string binary, decomp, t;

    unsigned char temp='\0', buffer='\0';

    compressedfile.read((char \*) &buffer, 1);

    temp=buffer;

    while(compressedfile.read((char \*) &buffer, 1)) {

        binary+=bitset<8>(temp).to\_string();

        decomp=DecompressCurrentBitset(&binary, codes);

        if(decomp != "") decompressedfile << decomp.c\_str();

        temp=buffer;

    }

    // Uzupelnij ostatnim bajtem

    int i = int(temp);

    while(i>0){

        t += to\_string(i%2);

        i /= 2;

    }

    reverse(t.begin(), t.end());

    binary+= t;

    decompressedfile << DecompressCurrentBitset(&binary, codes).c\_str();

    compressedfile.close();

    decompressedfile.close();

}

int main(int argc, char\* argv[]){

    char inputTextFile[] = "smalltest.txt";

    char outputModelFile[] = "output\_model.txt";

    char outputTreeFile[] = "output\_tree.txt";

    char outputCodeFile[] = "output\_code.txt";

    char outputCompressedTextFile[] = "output\_compressed.txt";

    char outputDecompressedTextFile[] = "output\_decompressed.txt";

    // Compress

    //if(argv[1] == "1"){

        cout << endl << "START kompresji" << endl << endl;

        GenerateModelFromFile(inputTextFile, &huffmanModelNodes, &readedBytesCount);

        WriteModelToFile(outputModelFile, huffmanModelNodes);

        cout << "Zapisano model" << endl;

        GenerateHuffmanTreeFromModel(huffmanModelNodes);

        WriteHuffmanTreeToFile(outputTreeFile, root);

        cout << "Zapisano drzewo" << endl;

        GenerateCodeTableFromTree(root, "");

        WriteCodeTableToFile(outputCodeFile, root);

        cout << "Zapisano kody znakow" << endl;

        cout << endl << "Kodowanie pliku ..." << endl;

        WriteCompressedFile(inputTextFile, outputCompressedTextFile, root);

        cout << "Zapisano zakodowany plik" << endl;

        cout << endl << "KONIEC kompresji" << endl;

    //}

    // Decompress

    //if(argv[1] == "2"){

        cout << endl << "START dekompresji" << endl << endl;

        map<string, string> codes;

        ImportAndMapCodesFromFile(outputCodeFile, &codes);

        cout << "Pobrano kody znakow" << endl;

        cout << endl << "Odkodowanie pliku ..." << endl;

        WriteDecompressedFile(outputCompressedTextFile, outputDecompressedTextFile, codes);

        cout << "Zapisano odkodowany plik" << endl;

        cout << endl << "KONIEC dekompresji" << endl;

    //}

    return 0;

}

# Przykład działania (plik: „smalltest.txt”)

Zawartość pliku:

Obraz zawierający tekst, stół

Opis wygenerowany automatycznie

Konsola:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Porównanie plików (kolejno: wejściowy, skompresowany, zdekompresowany):

Obraz zawierający tekst, wewnątrz, zrzut ekranu

Opis wygenerowany automatycznie

# Przykład działania (plik „test.txt”)

Zawartość pliku:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Konsola - kompresja:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Konsola - dekompresja:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Porównanie plików (kolejno: wejściowy, skompresowany, zdekompresowany):

Obraz zawierający tekst, zrzut ekranu, wewnątrz

Opis wygenerowany automatycznie

# Użyte struktury

1. Dla modelu.

struct HuffmanNode{

    int s;

    int f;

};

typedef struct HuffmanNode HuffmanNode;

vector<HuffmanNode> huffmanModelNodes;

int readedBytesCount = 0;

Struktura do modelu zawierająca informacje o symbolu i jego częstości. Przechowywane w wektorze. Dodatkowo zmienna przechowująca wszystkie pobrane znaki.

1. Dla drzewa.

struct HuffmanNodeTree{

    string s;

    int f;

    string code;

    HuffmanNodeTree \*parent;

    HuffmanNodeTree \*l\_child;

    HuffmanNodeTree \*r\_child;

};

typedef struct HuffmanNodeTree HuffmanNodeTree;

HuffmanNodeTree \*root;

Struktura do drzewa zawierająca jego symbol, częstość, kod i pokrewieństwo z innymi symbolami. Wskaźnik „root” pokazuje korzeń drzewa.

1. Dla tablicy kodowej (pobranej na nowo).

map<string, string> codes;

Mapuje dla symbolu jego kod. Jest to mapa, ponieważ jest ona znacznie szybsza obliczeniowo przy przeszukiwaniu.

# Analiza tworzenia modelu

1. Pobieram kolejne bajty z pliku (do zmiennej „unsigned char”).
2. Jeśli znak już wystąpił, zwiększam częstość.
3. Jeśli jest to nowy znak, dodaję nowy element do wektora.
4. Odczytuję sumę odczytanych bajtów.
5. Sortuję wszystkie symbole (najpierw po częstości, potem po wartości symbolu).
6. Zapisuję do pliku ilość bajtów.
7. Zapisuję do pliku każdy element modelu (symbol:częstość).

void GenerateModelFromFile(char\* inputTextFile, vector<HuffmanNode> \*huffmanModel, int \*readedBytesCount){

    // Wyznacza model dla zadanego pliku i zapisuje go w odpowiedniej strukturze

    FILE \*textfile = fopen(inputTextFile, "r");

    if(!textfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    int readCount = 0;

    unsigned char buffer[1];

    int readBytesLength = 1;

    int countBytesReaded = 0;

    bool found;

    while (readCount = fread(buffer, sizeof(unsigned char), readBytesLength, textfile)){

        //Obsługa odczytanego bajtu

        found = false;

        // Szukanie w aktualnym zbiorze symboli

        for(HuffmanNode &x : \*huffmanModel){

            if(x.s==int(buffer[0])){

                x.f++;

                found = true;

            }

        }

        // Jesli jest to nowy symbol

        if(!found){

            HuffmanNode huffmanNode;

            huffmanNode.s=int(buffer[0]);

            huffmanNode.f=1;

            huffmanModel->push\_back(huffmanNode);

        }

        countBytesReaded++;

    }

    \*readedBytesCount = countBytesReaded;

    fclose(textfile);

    sort(huffmanModelNodes.begin(), huffmanModelNodes.end(), CompareHuffmanNodes);

}

void WriteModelToFile(char\* outputModelFile, vector<HuffmanNode> huffmanModelNodes){

    // Zapisuje wyznaczony model do pliku tekstowego (z odpowiednim formatowaniem)

    ofstream modelfile;

    modelfile.open(outputModelFile);

    if(!modelfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    modelfile << huffmanModelNodes.size() << endl;

    for(HuffmanNode x : huffmanModelNodes){

        modelfile << x.s << ":" << x.f << endl;

    }

    modelfile.close();

}

# Analiza tworzenia drzewa kodowania

1. Tworzę wektor struktur elementów drzewa.
2. Każdy element modelu wpisuję do wektora drzewa (nadal są posortowane).
3. Uzupełniam drzewo (dopóki nie zostanie tylko 1 element - korzeń):
   1. Tworzę nowy element „#n” jako suma 2 elementów o najmniejszej częstości.
   2. Uzupełniam wskaźniki potomków wpisując adresy tych 2 elementów.
   3. Dla potomków ustawiam wartość rodzica jako adres nowego elementu.
   4. Z wektora usuwam 2 elementy o najmniejszej częstości i dodaję nowy element.
   5. Sortuję wektor (po częstości).
4. W kolejności „preorder” wpisuję kolejne elementy drzewa (symbol, czestosc, pokrewienstwo).
   1. Wpisz.
   2. Wpisz lewy.
   3. Wpisz prawy.

int CompareHuffmanTreeNodes(HuffmanNodeTree \*n1, HuffmanNodeTree \*n2){

    // Funkcja pomocnicza do sortowania drzewa

    return n1->f > n2->f;

}

void GenerateHuffmanTreeFromModel(vector<HuffmanNode> huffmanModelNodes){

    // Generuje drzewo kodowania

    vector<HuffmanNodeTree\*> huffmanNodeTreeVector;

    for (HuffmanNode x : huffmanModelNodes){

        HuffmanNodeTree \*temp = new HuffmanNodeTree;

        temp->s=to\_string(x.s);

        temp->f=x.f;

        temp->parent=NULL;

        temp->l\_child=NULL;

        temp->r\_child=NULL;

        huffmanNodeTreeVector.push\_back(temp);

    }

    for(int i=1; huffmanNodeTreeVector.size()!=1; i++){

        HuffmanNodeTree \*temp = new HuffmanNodeTree;

        temp->parent=NULL;

        temp->s="#"+ to\_string(i);

        temp->l\_child = new HuffmanNodeTree;

        temp->l\_child=huffmanNodeTreeVector[huffmanNodeTreeVector.size()-1];

        temp->l\_child->parent=temp;

        temp->r\_child = new HuffmanNodeTree;

        temp->r\_child=huffmanNodeTreeVector[huffmanNodeTreeVector.size()-2];

        temp->r\_child->parent=temp;

        temp->f = temp->l\_child->f + temp->r\_child->f;

        huffmanNodeTreeVector.pop\_back();

        huffmanNodeTreeVector.pop\_back();

        huffmanNodeTreeVector.push\_back(temp);

        sort(huffmanNodeTreeVector.begin(), huffmanNodeTreeVector.end(), CompareHuffmanTreeNodes);

    }

    root = huffmanNodeTreeVector[0];

}

void WriteHuffmanTreeToFile\_preorder(HuffmanNodeTree \*curr, FILE \*treefile) {

    // Funkcja pomocnicza dla zapisywania drzewa - przeszukuje w kolejności preorder

    if(curr){

        // Wezel

        fprintf(treefile, "%s ", curr->s.c\_str());

        // Czestotliwosc

        fprintf(treefile, "%s ", to\_string(curr->f).c\_str());

        // Lewy potomek

        if(curr->l\_child)

            fprintf(treefile, "%s ", curr->l\_child->s.c\_str());

        else

            fprintf(treefile, "nn ");

        // Prawy potomek

        if(curr->r\_child)

            fprintf(treefile, "%s ", curr->r\_child->s.c\_str());

        else

            fprintf(treefile, "nn ");

        // Rodzic

        if(curr->parent)

            fprintf(treefile, "%s ", curr->parent->s.c\_str());

        else

            fprintf(treefile, "nn ");

        fprintf(treefile, "\n");

        if(curr->l\_child != NULL) WriteHuffmanTreeToFile\_preorder(curr->l\_child, treefile);

        if(curr->r\_child != NULL) WriteHuffmanTreeToFile\_preorder(curr->r\_child, treefile);

    }

}

void WriteHuffmanTreeToFile(char\* outputTreeFile, HuffmanNodeTree \*root){

    // Zapisuje drzewo kodowania do pliku

    FILE \*treefile = fopen(outputTreeFile, "w");

    if(!treefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    fprintf(treefile, "wezel czestosc l\_child r\_child parent\n");

    WriteHuffmanTreeToFile\_preorder(root, treefile);

    fclose(treefile);

}

# Analiza tworzenia tablicy kodowej

1. W kolejności „preorder” generuje kod kolejnych symboli.
   1. Podaj kod.
   2. Podaj kod lewego jako kod bieżącego + „0”.
   3. Podaj kod prawego jako kod bieżącego + „1”.
2. W kolejności „preorder” wpisuję kolejne kody symboli (symbol, kod).
   1. Wpisz.
   2. Wpisz lewy.
   3. Wpisz prawy.

void GenerateCodeTableFromTree(HuffmanNodeTree \*curr, string code){

    // Generuje kody znakow - przeszukuje w kolejności preorder

    if(curr){

            curr->code=code;

        if(curr->l\_child) GenerateCodeTableFromTree(curr->l\_child, code+"0");

        if(curr->r\_child) GenerateCodeTableFromTree(curr->r\_child, code+"1");

    }

}

void WriteCodeTableToFile\_preorder(HuffmanNodeTree \*curr, FILE \*codefile) {

    // Funkcja pomocnicza dla zapisywania kodów - przeszukuje w kolejności preorder

    if(curr==NULL) return;

    if(curr->s[0] != '#'){

        fprintf(codefile, "%s %s\n", curr->s.c\_str(), curr->code.c\_str());

    }

    if(curr->l\_child != NULL) WriteCodeTableToFile\_preorder(curr->l\_child, codefile);

    if(curr->r\_child != NULL) WriteCodeTableToFile\_preorder(curr->r\_child, codefile);

}

void WriteCodeTableToFile(char\* outputCodeFile, HuffmanNodeTree \*root){

    // Zapisuje kody znakow do pliku

    FILE \*codefile;

    codefile = fopen(outputCodeFile, "w");

    if(!codefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    fprintf(codefile, "znak kod\n");

    WriteCodeTableToFile\_preorder(root, codefile);

    fclose(codefile);

}

string GetSymbolCodeFromTree(HuffmanNodeTree \*curr, string symbol){

    // Funkcja zwracajaca kod znakow na podstawie symbolu

    if(curr==NULL) return "";

    // Znaleziony

    if(curr->s==symbol) return curr->code;

    // Szukaj

    if(curr->l\_child != NULL)

        if(GetSymbolCodeFromTree(curr->l\_child, symbol)!="")

            return GetSymbolCodeFromTree(curr->l\_child, symbol);

    if(curr->r\_child != NULL)

        if(GetSymbolCodeFromTree(curr->r\_child, symbol)!="")

            return GetSymbolCodeFromTree(curr->r\_child, symbol);

    // Koncowy element drzewa, nie ten znak

    return "";

}

# Analiza kompresji

1. Odczytuję kolejne bajty pliku.
2. Dla każdego bajtu:
   1. Pobieram pobieram z drzewa jego kod.
   2. Dodaje odczytany kod do zmiennej „binary”.
   3. Jeśli zmienna zawiera ponad bajt, to zapisuje go do pliku.
   4. Usuwam odkodowane wartości ze zmiennej.
3. Jeśli zostaną jakieś niezapisane zakodowane wartości to dopisuję je do pliku.

void WriteCompressedFile(char\* input\_text\_file, char\* output\_compressed\_file, HuffmanNodeTree \*root){

    // Funkcja dokonujaca kompresji pliku

    FILE \*textfile = fopen(input\_text\_file, "r");

    if(!textfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    ofstream compressedfile(output\_compressed\_file, std::ios\_base::out | std::ios::binary);

    if(!compressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string binary;

    unsigned char buffer[1];

    if(textfile){

        while(fread(buffer, sizeof(unsigned char), 1, textfile)){

            binary += GetSymbolCodeFromTree(root, to\_string(int(buffer[0])));

            if(binary.size()>=8){

                compressedfile << char(stoi(binary.substr(0,8), 0, 2));

                binary.erase(0, 8);

            }

        }

    }

    fclose(textfile);

    if(binary.size()) compressedfile << char(stoi(binary, 0, 2));

    compressedfile.close();

}

# Analiza pobierania tablicy kodowej

1. Pobieram kolejne kody z pliku:
   1. Symbol
   2. Kod
   3. Dodaję do mapy wartości <kod, symbol>

void ImportAndMapCodesFromFile(char\* output\_code\_file, map<string, string> \*codes){

    // Funkcja pobierająca tablicę kodową do programu

    ifstream codefile;

    codefile.open(output\_code\_file);

    if(!codefile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string code, symbol, temp;

    codefile >> symbol >> code;

    while(!codefile.eof()){

        codefile >> symbol >> code;

        codes->insert(pair<string, string>(code, symbol));

    }

    codefile.close();

}

# Analiza dekompresji

1. Pobieram kolejne bajty z pliku:
   1. Dopisuję je do zmiennej „binary” jako bitset (umożliwia to wzięcie pod uwagę zer na początku rozwinięcia binarnego).
   2. Dekompresuję chwilowo odczytany kod binarny
      1. Sprawdzam do skutku coraz dłuższe fragmenty rozwinięcia binarnego.
      2. Jeśli znajdę, to konwertuję je z powrotem do symbolu.
      3. Usuwam odkodowane bity.
      4. Jeśli już nie da się na ten moment więcej zdekodować, zakończ działanie i zwróć bieżąco odkowane symbole.
   3. Dopisuję odkodowane symbole do pliku.
2. Na koniec, ostatnie nieodkodowane wartości koduję je i dopisuję do pliku.

void WriteDecompressedFile(char\* input\_compressed\_file, char\* output\_decompressed\_file, map<string, string> codes){

    // Dekompresuje plik do pierwotnej postaci

    ifstream compressedfile(input\_compressed\_file, ios\_base::in | ios::binary);

    if(!compressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    ofstream decompressedfile;

    decompressedfile.open(output\_decompressed\_file);

    if(!decompressedfile){cout<<"Nie mozna otworzyc pliku!"; exit(0);}

    string binary, decomp, t;

    unsigned char temp='\0', buffer='\0';

    compressedfile.read((char \*) &buffer, 1);

    temp=buffer;

    while(compressedfile.read((char \*) &buffer, 1)) {

        binary+=bitset<8>(temp).to\_string();

        decomp=DecompressCurrentBitset(&binary, codes);

        if(decomp != "") decompressedfile << decomp.c\_str();

        temp=buffer;

    }

    // Uzupelnij ostatnim bajtem

    int i = int(temp);

    while(i>0){

        t += to\_string(i%2);

        i /= 2;

    }

    reverse(t.begin(), t.end());

    binary+= t;

    decompressedfile << DecompressCurrentBitset(&binary, codes).c\_str();

    compressedfile.close();

    decompressedfile.close();

}