

WOJSKOWA AKADEMIA TECHNICZNA



Sprawozdanie z przedmiotu Teoria Informacji i Kodowania

Prowadzący: Pisula Małgorzata

Wykonawca: Kawka Maciej, 76797, WCY20IY2S1

1. Opis zadania.

1.1. Cel zadania.

Celem zadania jest zaimplementowanie tzw. Kodowania Huffmana, czyli metody kompresji bezstratnej w celu kompresji oraz dekompresji plików.

1.2. Opis teoretyczny.

Kodowanie Huffmana składa się z 5 etapów:

1. Obliczanie ilości występowania każdego symbolu z pliku.
2. Posortowanie otrzymanego zbioru symboli i ich częstotliwości.
3. Utworzenie drzewa kodowego.
4. Zapisanie nowych kodów znaków na podstawie drzewa kodowego do tablicy kodowej.
5. Zapisanie skompresowanego pliku za pomocą odczytanych znaków.

2. Opis wykonanego programu.

2.1. Obliczenie ilości występowania każdego symbolu z pliku.

Pierwszym punktem omawianej metody jest utworzenie modelu źródła informacji, czyli zestawienia każdego symbolu oraz ilości jego występowania w pliku. W prezentowanym programie dane te będą trzymane w tablicy, której identyfikator istnieje pod nazwą `huffmanModelArray`.

```
struct HuffmanNode* huffmanModelArray = (HuffmanNode*)malloc(sizeof(HuffmanNode) * 1);
```

Rysunek 1 .Alokacja pamięci dla tablicy kodowej

W celu wypełnienie tablicy potrzeba odczytać i policzyć wszystkie symbole w pliku. Program wykonuję to za pomocą funkcji `GenerateModelFromFile`.

```
huffmanModelArray = GenerateModelFromFile(huffmanModelArray, &iter, &checksum);
```

Rysunek 2. Wywołanie funkcji GenerateModelFromFile

```

struct HuffmanNode *GenerateModelFromFile(struct HuffmanNode *huffmanModelArray, int* it, int* ch) {
    FILE* inputFileHandle = fopen(inputFile, "rb");
    int readCount = 0;
    unsigned char buffer[1];
    int readBytesLength = 1;
    int iter = 0;
    int temp = 0, checksum = 0;
    //huffmanModelArray = (HuffmanNode*)malloc(sizeof(HuffmanNode) * 1);
    while (readCount = fread(buffer, sizeof(unsigned char), readBytesLength, inputFileHandle)) {
        readCount += 1;
        temp = IsItInArray(huffmanModelArray, buffer[0], iter);
        if (temp != 0) {
            huffmanModelArray[temp - 1].frequency += 1;
        }
        else {
            huffmanModelArray = (HuffmanNode*)realloc(huffmanModelArray, sizeof(HuffmanNode) * ((iter)+1));
            huffmanModelArray[iter].symbol = (int)buffer[0];
            huffmanModelArray[iter].parent = -1;
            huffmanModelArray[iter].left = -1;
            huffmanModelArray[iter].right = -1;
            huffmanModelArray[iter].frequency = 1;
            iter++;
        }
        checksum += 1;
    }
    fclose(inputFileHandle);
    //printf("Przed sortowaniem:\n");
    int i;
    //for (i = 0; i < iter; i++) {
        // printf("%d:%d\n", huffmanModelArray[i].symbol, huffmanModelArray[i].frequency);
    //}
    *it = iter;
    *ch = checksum;
    return huffmanModelArray;
}

```

Rysunek 3. Funkcja GenerateModelFromFile.

Powyższa funkcja w pętli while, odczytuje kolejne bajty z pliku i zapisuje je do zmiennej typu unsigned char o nazwie buffer. W pętli jest wywoływana funkcja "IsItInArray", która sprawdza czy w tablicy huffmanModelArray został zapisany już symbol o danym kodzie i zwraca liczbę jeśli tak, a -1 jeśli nie.

```

// This functions returns 0 if there is no symbol like this in buffor and if there is some like that it returns it position in array
int IsItInArray(HuffmanNode huffmanModelArray[MODEL_ARR_LEN], int buffer, int iter) {
    int i;
    for (i = 0; i < iter; i++) {
        if (huffmanModelArray[i].symbol == buffer) {
            return i + 1;
        }
    }
    return 0;
}

```

Rysunek 4. Funkcja IsItInArray

Zwracana wartość jest zapisywana do zmiennej temp i na jej podstawie dokonywany jest wybór. Jeżeli temp ma wartość 0 to znaczy, że dany symbol jeszcze nie został wczytany i w związku z tym

należy zaalokować dodatkową pamięć dla tablic znaków i zapisać tam dany znak. Jeżeli temp ma wartość inną niż 0 to oznacza, że temp-1 to jest to indeks w tablicy pod którym znajduje się wczytany znak, a więc należy zwiększyć o jeden licznik jego wystąpień.

Warto wspomnieć o strukturze, z której zbudowana jest tablica huffmanModelArray.

```
typedef struct HuffmanNode {
    int symbol;
    int frequency;
    //struct HuffmanTreeNode *left, *right, *parent;
    int left, right, parent;
}HuffmanNode;
```

Rysunek 5. Struktura HuffmanNode, z której stworzona jest tablica huffmanModelArray

Zawiera ona dodatkowe parametry typu o nazwach left, right i parent w celu dalszego przekształcenia tablicy w drzewo kodowe o konstrukcji takiej, że zmienne left, right i parent będą indeksami w tablicy.

2.2. Posortowanie otrzymanego zbioru symboli i ich częstotliwości.

Sortowanie tablicy symboli odbywa się za pomocą funkcji SortHuffmanModel.

```
SortHuffmanModel(iter, huffmanModelArray);
```

Rysunek 6. Wywołanie funkcji SortHuffmanModel

```
int CompareHuffmanNodes(const void* item1, const void* item2) {
    HuffmanNode* node1 = (HuffmanNode*)item1;
    HuffmanNode* node2 = (HuffmanNode*)item2;
    //printf("Node 1: %d : %d \n", (*node1).symbol, (*node1).frequency);
    //printf("Node 2: %d : %d \n", node2->symbol, node2->frequency);
    int compareResult = (node1->frequency - node2->frequency);
    if (compareResult == 0) {
        compareResult = (node1->symbol - node2->symbol);
    }
    return -compareResult;
}

void SortHuffmanModel(int modelArrayLength, struct HuffmanNode *huffmanModelArray) {
    qsort(huffmanModelArray, modelArrayLength, sizeof(HuffmanNode), CompareHuffmanNodes);
}
```

Rysunek 7. Funkcja SortHuffmanModel

Jest to funkcja, która wykorzystuje algorytm quicksort do posortowania tablicy.

2.3. Utworzenie drzewa kodowego.

Drzewo kodowe tworzone jest na bazie tablicy huffmanModelArray za pomocą funkcji GenerateHuffmanTreeFromModel.

```
huffmanModelArray = GenerateHuffmanTreeFromModel(huffmanModelArray, iter, checksum, &root);
```

Rysunek 8. Wywołanie funkcji *GenerateHuffmanTreeFromModel*

```
struct HuffmanNode *GenerateHuffmanTreeFromModel(struct HuffmanNode *huffmanModelArray, int iter, int checksum, int *root) {
    int lastNode = iter-1, leftNode, rightNode, nodeSpecial = 1001;
    while(huffmanModelArray[iter-1].frequency != checksum)
    {
        FindSmallestNode(&leftNode, &rightNode, huffmanModelArray, iter);
        huffmanModelArray = (struct HuffmanNode*)realloc(huffmanModelArray, sizeof(HuffmanNode) * ((iter)+1));
        huffmanModelArray[iter].symbol = nodeSpecial;
        huffmanModelArray[iter].left = leftNode;
        huffmanModelArray[iter].right = rightNode;
        huffmanModelArray[iter].frequency = huffmanModelArray[leftNode].frequency + huffmanModelArray[rightNode].frequency;
        huffmanModelArray[iter].parent = -1;
        huffmanModelArray[leftNode].parent = iter;
        huffmanModelArray[rightNode].parent = iter;
        //printf("w funkcji %d\n", iter/*huffmanModelArray[iter].frequency*/);
        iter++;
        nodeSpecial++;
    }
    //printf("pytanie %d ", huffmanModelArray[iter-1].right);
    iter = iter-1;
    *root = iter;
    //printf("w funkcji %d\n", huffmanModelArray[iter].frequency);
    return huffmanModelArray;
}
```

Rysunek 9. Funkcja *GenerateHuffmanTreeFromModel*

Działanie funkcji jest następujące. W pętli znajdowane są 2 najmniejsze wiersze tablicy kodowej. Następnie tworzony jest nowy wiersz tablicy, który staje się rodzicem dwóch mniejszych, a jego częstotliwość jest sumą częstotliwości dwóch ówczśnie znalezionych wierszy. Jest to wykonywane jedynie dla tych wierszy, w których parent jest równy -1, ponieważ kiedy ta zmienna przyjmują taką wartość to znaczy, że element nie ma jeszcze rodzica, a to oznacza, że dalej bierze udział w tworzeniu drzewa. Pętla wykonuje się aż do czasu, kiedy badany element nie będzie miał frequency odpowiadającego liczbie wszystkich znaków w tekście. Warto omówić to na podstawie przykładu.

W tablicy kodowej zapisane jest 5 elementów drzewa kodowego.

Index	1	2	3	4	5
Symbol	a	b	c	d	e
Częstotliwość	3	5	2	6	10
Lewy id	-1	-1	-1	-1	-1
Prawy id	-1	-1	-1	-1	-1
Rodzic	-1	-1	-1	-1	-1

Wartości w wszystkich kolumnach w wierszach Lewy id, Prawy id, Rodzic są ustawione na -1 już w pierwszym kroku implementacji. Dzięki temu można rozróżnić, które symbole w tablicy odpowiadające za magazynowanie symboli. Symbole zawsze będą miały wartości -1 na indeksie lewego oraz prawego dziecka. Kiedy rodzic jest ustawiony na -1 oznacza to, że są brane pod uwagę w wybieraniu kolejnych elementów dla drzewa.

Index	1	2	3	4	5	6
Symbol	a	b	c	d	e	a+b
Częstotliwość	3	5	2	6	10	5
Lewy id	-1	-1	-1	-1	-1	3
Prawy id	-1	-1	-1	-1	-1	1

Rodzic	6	-1	6	-1	-1	-1
--------	---	----	---	----	----	----

Po pierwszej iteracji został dodany nowy element tablicy o indeksie 6. Nie ma on rodzica, ponieważ jest to nowy element, a jego potomkami są elementy o indeksach 1 i 3. Można zauważyć, że zmienił się rodzic w nowo połączonych elementach i jest on ustawiony na 6. Przez to w następnej iteracji pętli nie będą już one analizowane.

Index	1	2	3	4	5	6	7
Symbol	a	b	c	d	e	a+b	B+id 6
Częstotliwość	3	5	2	6	10	5	10
Lewy id	-1	-1	-1	-1	-1	3	2
Prawy id	-1	-1	-1	-1	-1	1	6
Rodzic	6	7	6	-1	-1	7	-1

Jak widać w powyższej tabeli, został utworzony nowy element o indeksie 7, który połączył elementy 2 i nowy element 6. Jak widać rodzicem w 2 i 6 jest teraz 7, a więc nie będą one brane więcej pod uwagę.

To postępowanie algorytm wykonuję aż do momentu, kiedy ostatni element będzie miał częstotliwość, równą ilości znaków w każdym tekście. Warto zauważyć, że ostatnim elementem tablicy będzie korzeń.

Poniżej zamieszczam jeszcze funkcję odpowiedzialną za wybór 2 kolejnych elementów tablicy, w każdej iteracji.

```
void FindSmallestNode(int *lN, int *rN, struct HuffmanNode *huffmanModelArray, int iter) {
    int i, leftNodeMin = 2147483647, rightNodeMin = 2147483647, leftNode, rightNode;
    for (i = 0; i < iter; i++) {
        if (huffmanModelArray[i].frequency <= leftNodeMin && huffmanModelArray[i].parent == -1) //patrze tylko tam gdzie nie ma rodzica
        {
            leftNode = i;
            leftNodeMin = huffmanModelArray[i].frequency;
        }
    }
    for (i = 0; i < iter; i++) {
        if (huffmanModelArray[i].frequency <= rightNodeMin && huffmanModelArray[i].parent == -1 && i != leftNode)
        {
            rightNode = i;
            rightNodeMin = huffmanModelArray[i].frequency;
        }
    }
    //printf("\n",i);
    if (leftNode == rightNode)
        printf("\n-----Bład-----\n");
    *lN = leftNode;
    *rN = rightNode;
}
```

Rysunek 10. Funkcja FindSmallestNode.

Został zastosowany tutaj sposób na zwracanie więcej niż jednej wartości za pomocą odpowiedniego manipulowania wskaźnikami.

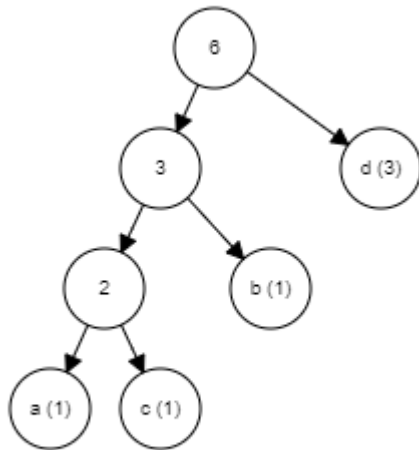
2.4. Zapisanie nowych kodów znaków na podstawie drzewa kodowego do tablicy kodowej.

Po wygenerowaniu drzewa należy utworzyć tablice kodową w celu późniejszego wykorzystania jej do zapisania tablicy kodowej. Poniżej zamieszczam linijkę kodu, która uruchamia funkcję odpowiedzialną za to zadanie.

```
GenerateCodeTableFromTree(root, arr, top, huffmanModelArray, fileForCode);
```

Rysunek 11. Wywołanie funkcji *GenerateCodeTableFromTree*.

W kodowaniu Huffmana symbol otrzymuje swój znak na podstawie drzewa kodowego. W celu wytłumaczenia zasady działania algorytmu posłużę się przykładem ze strony <https://cmps-people.ok.ubc.ca/ylucet/DS/Huffman.html>.



Rysunek 12. Przykład drzewa kodowego.

Założmy że plik pozwolił wygenerować takie drzewo Huffmana jak na rysunku powyżej. Aby od korzenia dostać się do znaku “a” należy wybrać 3-krotnie drogę w lewo. W kodowaniu Huffmana dogra w lewo oznacza 0, a w prawo 1, więc kodem znaku “a” jest “000”. Na tej samej zasadzie możemy określić kod znaku “c”, należy pójść 2-krotnie w lewo i na koniec w prawo, a więc kod znaku “c” to “001”.

Z powyższego rozumowania wynikają, następujące wnioski: aby określić kod każdego znaku potrzeba znać jego drogę od korzenia aż do liścia, w którym zapisany jest dany znak. Taką możliwość daje algorytm przeszukiwania drzewa “post-order”. Tak właśnie działa funkcja *GenerateCodeTableFromTree*.

```

void GenerateCodeTableFromTree(int root, int arr[], int top, struct HuffmanNode *huffmanModelArray, FILE *outFileH) {
    if (huffmanModelArray[root].left != -1) {
        arr[top] = 0;
        GenerateCodeTableFromTree(huffmanModelArray[root].left, arr, top + 1, huffmanModelArray, outFileH);
    }

    if (huffmanModelArray[root].right != -1) {
        arr[top] = 1;
        GenerateCodeTableFromTree(huffmanModelArray[root].right, arr, top + 1, huffmanModelArray, outFileH);
    }

    if (huffmanModelArray[root].left == -1 && huffmanModelArray[root].right == -1) {
        //fprintf(outputFileCodeTable, "%d-", root->symbol);
        //printf("%d-", huffmanModelArray[root].symbol);
        fprintf(outFileH, "%d-", huffmanModelArray[root].symbol);
        int i;
        for (i = 0; i < top; i++) {
            //printf("%d", arr[i]);
            fprintf(outFileH, "%d", arr[i]);
        }
        //printf("\n");
        fprintf(outFileH, "\n");
    }
}

```

Rysunek 13. Funkcja GenerateCodeTableFromTree.

Funkcja przeszukuje rekurencyjnie drzewo w kierunku post-order, przekazując do następnych wywołań funkcji dotychczas odczytany kod. Po każdym przejściu dopisuje to tablicy arr kolejne 0 lub 1 zależnie od tego czy “poszło” w lewo lub w prawo. Tablice przekazuje do następnych wywołań, tak aby wywołanie funkcji, które na trafi na liść (wierzchołek którego lewy i prawy potomek są równe -1 czyli nie istnieje ich potomstwo) wypisało znak i jego kod do pliku.

W tym punkcie dobrze też poruszyć temat funkcji zapisującej drzewo do pliku. Funkcja nosi nazwę WriteHuffmanTreeToFile.


```

void WriteHuffmanTreeToFile(struct HuffmanNode *huffmanModelArray, int size) {
    FILE* outputFileH;
    if ((outputFileH = fopen(outputFileTreeF, "w")) == NULL) {
        printf("Otwieranie pliku do zapisu %s nie powiodlo sie\n", outputFile);
    }
    int i;
    for (i = 0; i <= size; i++)
    {
        if (huffmanModelArray[i].symbol >= 1000) {
            //printf("#%d", huffmanModelArray[i].symbol - 1000);
            fprintf(outputFileH, "#%d", huffmanModelArray[i].symbol - 1000);
        }
        else {
            //printf("%d", huffmanModelArray[i].symbol);
            fprintf(outputFileH, "%d", huffmanModelArray[i].symbol);
        }
        //printf(":%d", huffmanModelArray[i].frequency);
        fprintf(outputFileH, ":%d", huffmanModelArray[i].frequency);

        if (huffmanModelArray[huffmanModelArray[i].left].symbol >= 1000 && huffmanModelArray[i].left != -1) {
            //printf("\tchildLeft:%d", huffmanModelArray[huffmanModelArray[i].left].symbol - 1000);
            fprintf(outputFileH, "\tchildLeft:%d", huffmanModelArray[huffmanModelArray[i].left].symbol - 1000);
        }
        else if (huffmanModelArray[huffmanModelArray[i].left].symbol == 0 || huffmanModelArray[i].left == -1) {
            //printf("\tchildLeft:-");
            fprintf(outputFileH, "\tchildLeft:-");
        }
        else {
            //printf("\tchildLeft:%d", huffmanModelArray[huffmanModelArray[i].left].symbol);
            fprintf(outputFileH, "\tchildLeft:%d", huffmanModelArray[huffmanModelArray[i].left].symbol);
        }

        if (huffmanModelArray[huffmanModelArray[i].right].symbol >= 1000 && huffmanModelArray[i].right != -1) {
            //printf("\tchildRight:%d", huffmanModelArray[huffmanModelArray[i].right].symbol - 1000);
            fprintf(outputFileH, "\tchildRight:%d", huffmanModelArray[huffmanModelArray[i].right].symbol - 1000);
        }
        else if (huffmanModelArray[huffmanModelArray[i].right].symbol == 0 || huffmanModelArray[i].right == -1) {
            //printf("\tchildRight:-");
            fprintf(outputFileH, "\tchildRight:-");
        }
        else {
            //printf("\tchildRight:%d", huffmanModelArray[huffmanModelArray[i].right].symbol);
            fprintf(outputFileH, "\tchildRight:%d", huffmanModelArray[huffmanModelArray[i].right].symbol);
        }

        if (i == size) {
            //printf("\tParent:-");
            fprintf(outputFileH, "\tParent:-");
        }
        else {
            if (huffmanModelArray[huffmanModelArray[i].parent].symbol >= 1000) {
                //printf("\tParent:%d", huffmanModelArray[huffmanModelArray[i].parent].symbol - 1000);
                fprintf(outputFileH, "\tParent:%d", huffmanModelArray[huffmanModelArray[i].parent].symbol - 1000);
            }
            else if (huffmanModelArray[huffmanModelArray[i].parent].symbol == -1) {
                //printf("\tParent:-");
                fprintf(outputFileH, "\tParent:-");
            }
        }
    }
}

```

```

    }
    else {
        //printf("\tParent:%d", huffmanModelArray[huffmanModelArray[i].parent].symbol);
        fprintf(outputFileH, "\tParent:%d", huffmanModelArray[huffmanModelArray[i].parent].symbol);
    }
}
//printf("\n");
fprintf(outputFileH, "\n");
}
fclose(outputFileH);
}

```

Rysunek 14. Funkcja *GenerateCodeTableFromTree*.

Jest to bardzo prosta funkcja, ponieważ całe drzewo jest zapisane w tablicy, więc funkcja po prostu wykonuje pętlę for tak długo, aż wpisze całą tablicę do pliku. Wszystkie konstrukcje if są po to, aby drzewo zostało zapisane w odpowiednim formacie.

Poniżej zamieszczam również funkcję, która zapisuje model źródła danych do pliku. Funkcja ta również wypisuje elementy tablicy w pętli jednak w momencie, w którym funkcja ta się wykonuje to jeszcze nie ma utworzonego drzewa na podstawie tablicy, a więc wypiszą się tylko te elementy tablicy, które są znakami.

```

void WriteModelToFile(int iter, struct HuffmanNode *huffmanModelArray) {
    FILE* outputFileHandle;
    if ((outputFileHandle = fopen(outputFileModel, "w")) == NULL) {
        printf("Otwieranie pliku do zapisu %s nie powiodło się\n", outputFile);
    }
    else {
        fprintf(outputFileHandle, "%d\n", iter);
        int i;
        for (i = 0; i < iter; i++) {
            fprintf(outputFileHandle, "%d:%d\n", huffmanModelArray[i].symbol, huffmanModelArray[i].frequency);
        }
    }
    fclose(outputFileHandle);
}

```

Rysunek 15. Funkcja *WriteModelToFile*.

2.5. Zapisać skompresowany plik z pomocą odczytanych znaków.

Sekwencje zapisu skompresowanego pliku rozpoczyna wczytanie tablicy kodowej z pliku utworzonego w poprzednim punkcie. Tablica kodowa zostanie zapisana do tablicy, której typem jest struktura o nazwie *CodeDock*.

```

typedef struct CodeDock {
    int sign;
    int current;
    unsigned int code;
}CodeDock;

```

Rysunek 16. Struktura *CodeDock*.

Tablica kodowa zostaje zapisana do tablicy w programie za pomocą funkcji *CodeDockCreate*.

```

struct CodeDock *CodeDockCreate(int *toPass) {
    FILE *inFileH;
    CodeDock *cDock;
    unsigned char bufor[1];
    int iter = 0, mode = 1, count = 0, first = 1;
    if((inFileH = fopen(outputFileCodeTab, "r")) == NULL) {
        printf("Nie znaleziono pliku\n");
        return NULL;
    }
    cDock = (CodeDock*)malloc(sizeof(CodeDock));
    cDock[iter].sign = 0;
    while (fread(bufor, sizeof(unsigned char), 1, inFileH)) {
        //printf("%c", bufor[0]);
        if (bufor[0] == '-') {
            first = 0;
            cDock[iter].code = 0x00;
            cDock[iter].current = 0;
            continue;
        }
        if (bufor[0] == '\n') {
            //printf("Code Table: %d 0x%X\n", cDock[iter].sign, cDock[iter].code);
            iter++;
            cDock = (CodeDock*)realloc(cDock, sizeof(CodeDock)*(iter+1));
            cDock[iter].sign = 0;
            first = 1;
            continue;
        }
        if (first) {
            cDock[iter].sign = cDock[iter].sign*10 + bufor[0] - 48;
            //printf("buffor: %c to to: %d ", bufor[0], cDock[iter].sign);
        }
        else {
            cDock[iter].code = cDock[iter].code << 1;
            cDock[iter].code += bufor[0] - 48;
            cDock[iter].current += 1;
        }
    }
    *toPass = iter;
    fclose(inFileH);
    return cDock;
}

```

Rysunek 17. Funkcja CodeDockCreate.

Funkcja działa na podstawie pętli while, w której każda jej iteracja wczytuje kolejny bajt, aż do końca pliku. Program zapisuje do tablicy cDock kolejne znaki. Kiedy zmienna first jest ustawiona na 1 to znaczy, że zapisuję symbol tablicy kodowej a więc jakąś liczbę typu int. W programie wczytujemy kolejne znaki tej tablicy, a więc żeby wczytać całą liczbę oznaczającą znak należy następną wczytaną cyfrę pomnożyć dodać do poprzedniej wartości razy 10. Cyfry w alfabecie ASCII są reprezentowane od indexu 48, a więc żeby zapisać sczytaną cyfrę należy odjąć od znaku ASCII liczbę 48. Jeżeli program natrafi w pliku na znak "-" to zmienna first zmieni się na 0 i program zacznie wczytywać kod znaku. Kod znaku jest czytany w ten sposób że początkowo ma wartość 0x00. Kiedy zostanie wczytany znak zwiększa się current o 1 oraz sam kod zostanie przesunięty binarnie o 1 w lewo i

zostanie do niego dodany znak ASCII –48 czyli 0 albo 1. Warto zwrócić uwagę na zwiększanie `current` z każdą iteracją. Dzięki temu w dalszej części programu będzie wiadomo, ile bitów, patrząc od lewej, z kodu reprezentują kod. To działanie jest wykonywane aż do znalezienia znaku nowej linii i wtedy zostanie zaalokowane więcej pamięci na kolejny wiersz tablicy `cDock` oraz zmienna `first` znowu przyjmie wartość 1 co oznacza że znowu program będzie wczytywał liczbę, która jest reprezentacją znaku w ASCII.

Kolejną funkcją, która uczestniczy bezpośrednio w zapisaniu skompresowanego pliku jest `WriteCompressedFile`.

```

void WriteCompressedFile(CodeDock *codeTable, int limes) {
    FILE *input, *output;
    if((input = fopen(inputFile, "rb")) == NULL) {
        printf("Nie znaleziono pliku z danymi!\n");
    }
    if((output = fopen(outputFile, "wb")) == NULL) {
        printf("Nie udało sie utworzyc pliku!\n");
    }
    unsigned char bufor[1], bit, out = 0x00, mask;
    //unsigned int temp;
    int i,j, shift=8, tableShift;
    while(fread(bufor, sizeof(unsigned char), 1, input)) {
        for (i = 0; i < limes; i++) {
            if (codeTable[i].sign == bufor[0]) {
                tableShift = codeTable[i].current - 1;
                for (j = 0; j < codeTable[i].current; j++) {
                    shift--;
                    if ( ((codeTable[i].code >> (tableShift - j)) & 0x00000001) != 0x00) {
                        //printf("1");
                        mask = 0x01;
                    }
                    else {
                        mask = 0x00;
                        //printf("0");
                    }
                    mask = mask << shift;
                    //printf("shift: %d", shift);
                    out = out ^ mask;
                    //tableShift--;
                    if (shift == 0) {
                        shift = 8;
                        fprintf(output, "%c", out);
                        //printf(" %X ",out);
                        out = 0x00;
                    }
                }
            }
        }
    }
    if (shift != 0) {
        fprintf(output, "%c", out);
    }
    fclose(input);
    fclose(output);
}

```

Rysunek 18. Funkcja WriteCompressedFile.

Głównym elementem tej funkcji jest pętla while, która czyta kolejne bajty wejściowego, nieskompresowanego pliku do zmiennej bufor. W pętli jest kolejna pętla for, która wykonuje się tyle razy, ile wierszy ma tablica codeTable, która jest tablicą ze znakami i ich kodami, wczytana w poprzedniej funkcji. Zmienna bufor jest porównywana z tablicą kodową i kiedy symbol jest taki sam jak w tablicy to funkcja przystępuje do utworzenia bajtu do zapisania do pliku. Niedogodnością języka c w tym przypadku jest to, że minimalna ilość danych, którą możemy zapisać do pliku jest jeden bajt. Dlatego należy go najpierw uzupełnić i potem zapisać. Zmienna out reprezentuje bajt do zapisania do pliku. Na początku ustawiana jest wartość tableShift która jest o jeden mniejsza niż ilość

bitów którą trzeba wczytać. W pętli for, która wykona się tyle razy, ile ma wielkość w bitach kod znaku wpisywane jest po jednym bicie do zmiennej out. Aby wyłuskać kolejne bity będziemy sprawdzać w if wartość codeTable[i].code przesuwając ją o wartość tableShift – j (w ten sposób co iteracje pętli będziemy przesuwac o inną wartość tak że wartość do wpisania będzie zawsze na najmłodszej pozycji) i wykonywać operację and z wartością 1. Wtedy, jeżeli uzyskana wartość będzie równa 1, to oznacza to że powinniśmy do zmiennej out wpisać 1 a jeżeli 0 to powinniśmy dopisać 0. Oczywiście zależnie od uzyskanej jedynki lub zera zapisujemy do zmiennej mask odpowiednio 0x01 albo 0x00. Następnie mask jest przesuwane binarnie tak aby odpowiednio wypozyjonować 1 i jest xorowana z out. Dzięki temu uzyskamy na odpowiedniej pozycji w out bit 1 lub 0. Shift w każdej iteracji się zmienia, ponieważ musimy zapisywać bity na odpowiednich pozycjach. Kiedy shift przyjmie wartość 0 oznacza to, że zmienna out została w całości zapisana, a więc zapisujemy out do pliku oraz ustawiamy shift z powrotem na 8. Na koniec funkcji zapisujemy jeszcze zmienną out w momencie, gdy shift nie jest równy 0, ponieważ wtedy oznacza to że nie zapisaliśmy ostatniego bajtu, który ówczśnie już został wypełniony.

3. Dekompresja.

3.1. Odczytanie drzewa kodowego.

Dekompresja zaczyna się od odczytania drzewa kodowego. Pierwszą funkcją wykonywaną w tym celu jest funkcja CreateNotConfiguredList.

```

struct HuffmanNode *CreateNotConfiguredList(FILE *inFileH, int *ret) {
    unsigned char bufor[1]; //, *fullFile = (unsigned char*)malloc(sizeof(unsigned char));
    int mode = 1, iter = 0, temp = 0, offset = 0;
    struct HuffmanNode *code = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    while (fread(bufor, sizeof(unsigned char), 1, inFileH)) {
        //printf("%c", bufor[0]);
        if (mode == 1) {
            if (bufor[0] == ':') {
                code[iter].symbol = temp + offset;
                //printf("symbol: %d",code[iter].symbol);
                //printf("symbol: %d",code[iter].symbol);
                mode = 2; temp = 0; offset = 0;
                continue;
            }
            else if (bufor[0] == '#') {
                offset = 1000;
            }
            else {
                temp = temp*10 + bufor[0] - 48;
                //printf("symbol: %d",temp);
            }
        }
        else if (mode == 2) {
            if (bufor[0] == '\t') {
                code[iter].frequency = temp;
                //printf(" frequency: %d",code[iter].frequency);
                mode = 3; temp = 0;
                continue;
            }
            else {
                temp = temp*10 + bufor[0] - 48;
            }
        }
        else if (mode == 3) {
            if (bufor[0] == '\t') {
                if (temp == -3) {
                    code[iter].left = -1;
                }
                else {
                    code[iter].left = temp + offset;
                }
                //printf(" left: %d",code[iter].left);
                mode = 4; temp = 0; offset = 0;
                continue;
            }
            else if ((bufor[0] >= 65 && bufor[0] <= 122) || bufor[0] == ':') {
                continue;
            }
            else if (bufor[0] == '#') {
                offset = 1000;
            }
            else {

```

```

else if (mode == 4) {
    if (bufor[0] == '\t') {
        if (temp == -3) {
            code[iter].right = -1;
        }
        else {
            code[iter].right = temp + offset;
        }
        //printf(" right: %d",code[iter].right);
        mode = 5; temp = 0; offset = 0;
        continue;
    }
    else if ((bufor[0] >= 65 && bufor[0] <= 122) || bufor[0] == ':') {
        continue;
    }
    else if (bufor[0] == '#') {
        offset = 1000;
    }
    else {
        temp = temp*10 + bufor[0] - 48;
    }
}
else if (mode == 5) {
    if ((int)bufor[0] == 10) {
        if (temp == -3) {
            code[iter].parent = -1;
        }
        else {
            code[iter].parent = temp + offset;
        }
        //printf(" parent: %d\n",code[iter].parent);
        mode = 1; temp = 0; offset = 0;
        iter++;
        code = (HuffmanNode*)realloc(code, sizeof(HuffmanNode)*(iter+2));
        continue;
    }
    else if ((bufor[0] >= 65 && bufor[0] <= 122) || bufor[0] == ':') {
        continue;
    }
    else if (bufor[0] == '#') {
        //printf(" jest hasz ");
        offset = 1000;
        continue;
    }
    else {
        temp = temp*10 + bufor[0] - 48;
        //printf(" temp: %d", temp);
    }
}
}
}
*ret = iter;
return code;
}

```


Rysunek 19. Funkcja `CreateNotConfiguredList`.

Funkcja ta wczytuje kolejne wiersze pliku z drzewem do tablicy `code`, która jest taką samą tablicą jak ta, która zawierała drzewo w trakcie kompresji. W wyniku tego wczytywania uzyskujemy tablicę, która jest delikatnie wypaczona. Jest ona taka z tego powodu, że nie ma po przypisywanych odpowiednich danych pod `left`, `right` i `parent`, a są wstawione zamiast indexów symbole znaków lub symbole innych wierzchołków, które w celu dalszej interpretacji są powiększone o 1000. W celu poprawnego skonfigurowania listy powstała funkcja `ConfigureListToTree`.

```
struct HuffmanNode *ConfigureListToTree(struct HuffmanNode *code, int size) {
    int i, index;
    for (i = 0; i < size; i++) {
        if (code[i].left > 0) {
            index = getNodeIndex(code, size, code[i].left);
            code[i].left = index;
        }
        if (code[i].right > 0) {
            index = getNodeIndex(code, size, code[i].right);
            code[i].right = index;
        }
        if (code[i].parent > 0) {
            index = getNodeIndex(code, size, code[i].parent);
            code[i].parent = index;
        }
    }
    return code;
}
```

Rysunek 20. Funkcja `ConfigureListToTree`.

Funkcja ta iteruje się po każdym wierszu z tablicy reprezentującej drzewo kodowe i tam gdzie wartość `left`, `right` lub `parent`, są większe od 0 (ponieważ kiedy te nie ma tych wartości to w tamte miejsca wpisane jest -1) wywołuje funkcję `getNodeIndex`, która zwraca odpowiedni index do zapisania w `left`, `right` lub `parent`.

```

int getNodeIndex(struct HuffmanNode *code, int size,int symbol) {
    int i;
    for (i = 0; i < size; i++) {
        if (symbol == code[i].symbol) {
            return i;
        }
    }
    return 0;
}

```

Rysunek 21. Funkcja `getNodeIndex`.

Powyższa funkcja iteruję się po całej tablicy, w której zapisane jest drzewo binarne, a następnie zwraca odpowiedni index z tabeli, kiedy natrafi na zadany znak. W innym przypadku zwraca 0.

Ostatnią funkcją, która bierze udział w dekompresji jest `WriteDecompressedFile`.

```

void WriteDecompressedFile(HuffmanNode *code, int limes, int root) {
    FILE *inFileH;
    if ((inFileH = fopen(inputFile, "rb")) == NULL) {
        printf("Wystapil blad - niektore pliki sa niedostepne\n");
    }
    FILE *outFileH;
    if ((outFileH = fopen(outputFile, "wb")) == NULL) {
        printf("Wystapil blad - niektore pliki sa niedostepne\n", outputFileCodeTab);
    }
    unsigned char bufor[1], mask = 0x00, temp = 0x00;
    int iter = 0, node = root;
    while (iter < limes) {
        if (mask == 0x00) {
            fread(bufor, sizeof(unsigned char), 1, inFileH);
            mask = 0x80;
        }
        temp = bufor[0] & mask;
        if (temp) {
            node = code[node].right;
        }
        else {
            node = code[node].left;
        }
        mask = mask >> 1;
        if (code[node].left == -1 && code[node].right == -1) {
            fprintf(outFileH, "%c", code[node].symbol);
            node = root;
            iter++;
        }
    }
    //printf("%d ",iter);
    fclose(inFileH);
    fclose(outFileH);
}

```

Rysunek 22. Funkcja WriteDecompressedFile.

Funkcja ta działa w oparciu o pętlę while, która będzie wykonywać się tak długo aż zmienna iter nie przyjmie wartości równej ilości wszystkich bajtów w pliku. Ilość ta została wcześniej odczytana z częstotliwości zapisanej w korzeniu drzewa.

Funkcja ta odczytuje znaki w następujący sposób. Po wczytaniu całego bajtu z skompresowanego pliku, "wyłuskuję" z niego jeden bit w każdej iteracji pętli. Wyłuskiwanie jest uzyskiwane za pomocą maski. Maska na początku ma wartość 0x00 co oznacza, że wykona się to co jest w pierwszym if czyli zostanie sczytany bajt z pliku skompresowanego oraz mask zostanie ustawione na 0x80. Oznacza to że mask ma jedynkę na pierwszym bicie. Dzięki temu po wykonaniu operacji and z bajtem sczytanym z pliku, w zmiennej temp zostanie zapisana wartość 0 lub inna w zależności od tego czy ten pierwszy bit to 0 lub 1. Mask z każdą iteracją jest poddawane operacji przesunięciu binarnego w prawo dzięki

czemu 1 będzie przechodziło od pierwszej do ostatniej pozycji w bajcie i po operacji and będziemy wiedzieć, czy odczytaliśmy 0 czy 1. W zależności od tego czy odczytaliśmy 0 czy 1, zmieniamy indeks, który reprezentuje nam miejsce w drzewie kodowym. W takim razie, kiedy jest 0 to zmienna node przyjmuję wartość indeksu lewego potomka, a jeżeli 1 to przyjmuje wartość prawego potomka. Kiedy dojdziemy tymi iteracjami do liścia (czyli nie ma potomków co reprezentują wartość -1), to zapisujemy odczytany znak, zwiększamy iter, który jest licznikiem zapisanych znaków oraz ustawiamy node na korzeń drzewa, aby przystąpić do wczytywania kolejnego znaku. Ma to związek z teorią, ponieważ tablica kodowa jest wykonywana w bardzo podobny sposób na podstawie drzewa.

4. Funkcja main.

Funkcja main jest odpowiedzialna za uruchamianie kolejnych funkcji w poprawnej sekwencji oraz kontrolowaniu interfejsu dla użytkownika. Ważne jest to, przy opcji kompresji, że do nazwy tworzy inne nazwy pliku na podstawie pliku wyjściowego dopisując do niego ".code" ".graf" ".model" za pomocą funkcji z biblioteki string o nazwie strcat, a w dekompresji na podstawie pliku wejściowego tworzy nazwę pliku z drzewem dopisując ".graf" do jego nazwy i robi to również za pomocą funkcji strcat.

```

int main() {
    int option = 1;
    while (1)
    {
        printf("Tik Laboratorium nr 3\n1. Kompresja\n2. Dekompresja\n3. Wyjscie\n-->");
        scanf("%d", &option);
        if (option == 1) {
            printf("Plik wejscowy: ");
            scanf("%s", inputFile);
            printf("Plik wyjscowy: ");
            scanf("%s", outputFile);
            strcpy(outputFileCodeTab, outputFile);
            strcat(outputFileCodeTab, ".code");
            strcpy(outputFileTreeF, outputFile);
            strcat(outputFileTreeF, ".graf");
            strcpy(outputFileModel, outputFile);
            strcat(outputFileModel, ".model");

            int iter, checksum, root;
            struct HuffmanNode* huffmanModelArray = (HuffmanNode*)malloc(sizeof(HuffmanNode) * 1);
            huffmanModelArray = GenerateModelFromFile(huffmanModelArray, &iter, &checksum);
            //printf("%d", iter);
            SortHuffmanModel(iter, huffmanModelArray);
            //printf("Po sortowaniu: checksum: %d\n", checksum);
            int i;
            //for (i = 0; i < iter; i++) {
                // printf("%d:%d\n", huffmanModelArray[i].symbol, huffmanModelArray[i].frequency);
            //}
            WriteModelToFile(iter, huffmanModelArray);
            printf("\n\nModel zrodla zostal zapisany do pliku.\n");
            huffmanModelArray = GenerateHuffmanTreeFromModel(huffmanModelArray, iter, checksum, &root);
            int* arr;
            arr = (int*)malloc(sizeof(int) * iter);
            int top = 0;
            //for (i = 0; i <= root; i++) {
                // printf("%d parent: %d, symbol: %d, lewy: %d, prawy: %d\n", i, huffmanModelArray[i].frequency, huffmanModelArray[i].symbol, huffmanModelArray[i].left, huffmanModelArray[i].right);
            //}
            WriteHuffmanTreeToFile(huffmanModelArray, root);
            printf("Drzewo kodowania + ");
            FILE *fileForCode;
            if ((fileForCode = fopen(outputFileCodeTab, "w")) == NULL) {
                printf("Otwieranie pliku do zapisu %s nie powiodlo sie\n", outputFileCodeTab);
            }
            GenerateCodeTableFromTree(root, arr, top, huffmanModelArray, fileForCode);
            printf("tablica kodowa zostaly zapisane do pliku.\n");
            //printf("ok root: %d ", root);
            //WriteDecompressedFile(huffmanModelArray, huffmanModelArray[root].frequency, root);
            free(huffmanModelArray);
            fclose(fileForCode);
            int sizeCodeTable;
            struct CodeDock *code = CodeDockCreate(&sizeCodeTable);
            int j, calc = 0;
            /*for(i = 0; i < sizeCodeTable; i++) {
                printf("%c-0x%X %d", code[i].sign, code[i].code, code[i].current);
                putc('\n', stdout);
            }*/
            //printf("ok2");
            WriteCompressedFile(code, sizeCodeTable);
            printf("Kompresja zakonczona\n\n\n");
            //printf("ok4");
            free(code);
        }
    }
}

```

```

else if (option == 2){
    printf("Plik wejsciowy: ");
    scanf("%s", inputFile);
    printf("Plik wyjsciowy: ");
    scanf("%s", outputFile);
    strcpy(outputFileTreeF, "");
    strcpy(outputFileTreeF, inputFile);
    strcat(outputFileTreeF, ".graf");

    struct HuffmanNode* huffmanModelArray = (HuffmanNode*)malloc(sizeof(HuffmanNode) * 1);
    int size;
    huffmanModelArray = ReadHuffmanTreeFromFile(&size);
    int i;
    /*for (i = 0; i < size; i++) {
        printf("%d:%d lewy: %d, prawy: %d parent: %d\n", huffmanModelArray[i].symbol, huffmanModelArray[i].frequency, huffmanModelArray[i].left, huffmanModelArray[i].right, huffmanModelArray[i].parent);
    }*/
    size -= 1;
    WriteHuffmanTreeToFileTemp(huffmanModelArray, size);
    //struct HuffmanNode *code, FILE *inFileH, FILE *outFileH, int limes, int root
    WriteDecompressedFile(huffmanModelArray, huffmanModelArray[size].frequency, size);
    //WriteDecompressedFile(huffmanModelArray, huffmanModelArray[size].frequency, size);
    //printf("%d", huffmanModelArray[size].parent);
    printf("\n\nDekompresja zakonczona\n\n\n");
    free(huffmanModelArray);
}
else {
    return 0;
}
}
return 0;
}

```

Rysunek 23. Funkcja main.

5. Podsumowanie.

5.1. Przykład działania programu.

Poniżej przedstawiam zrzuty ekranu z działania programu dla pliku test.txt, który był wykorzystany do prezentacji działania programu na zajęciach.









Ekran terminalu dla wyboru opcji 1. Kompresja:

```
C:\Users\mkawk\OneDrive\Dokumenty\TeoriaInformacjiKodowania\kodowanie_huffmana\kodowanie_huffmana.exe
Tik Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->1
Plik wejściowy: test.txt
Plik wyjściowy: tree

Model źródła został zapisany do pliku.
Drzewo kodowania + tablica kodowa zostały zapisane do pliku.
Kompresja zakończona

Tik Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->
```

Utworzone pliki:

 tree		12.06.2022 00:06	Plik	1 579 KB
 tree.code		12.06.2022 00:06	Plik CODE	1 KB
 tree.graf		12.06.2022 00:06	Plik GRAF	7 KB
 tree.model		12.06.2022 00:06	Plik MODEL	1 KB

Ekran terminalu dla wyboru opcji 2. Dekompresja:

```
C:\Users\mkawk\OneDrive\Dokumenty\TeoriaInformacjiKodowania\kodowanie_huffmana\kodowanie_huffmana.exe
1. Kompresja
2. Dekompresja
3. Wyjście
-->1
Plik wejściowy: test.txt
Plik wyjściowy: tree











Model źródła został zapisany do pliku.
Drzewo kodowania + tablica kodowa zostały zapisane do pliku.
Kompresja zakończona

Tik Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->2
Plik wejściowy: tree
Plik wyjściowy: test.txt.decompressed

Dekompresja zakończona

Tik Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->
```

Utworzone pliki:

 test.txt.decompressed		12.06.2022 00:08	Plik DECOMPRESS...	2 693 KB
 tree		12.06.2022 00:06	Plik	1 579 KB
 tree.code		12.06.2022 00:06	Plik CODE	1 KB
 tree.graf		12.06.2022 00:06	Plik GRAF	7 KB
 tree.model		12.06.2022 00:06	Plik MODEL	1 KB

Plik z modelem źródła informacji:

```

tree.model — Notatnik
Plik  Edytuj  Wyświetl

65
32: 409500
105: 224640
101: 202020
97: 177060
111: 138060
110: 127530
115: 124020
114: 109980
116: 104520
109: 93210
99: 88920
117: 83850
122: 78000
13: 70590
10: 70590
100: 65910
45: 59280
119: 54210
108: 54210
112: 49530
121: 45630
107: 34710
98: 34710
106: 28860
103: 28470
179: 23610
44: 22620
104: 19890
118: 18330
234: 16380
185: 16380
156: 13260
230: 10530
191: 10140
102: 8190
243: 6630
120: 6240
113: 5460
241: 2730
67: 1950
66: 1950
83: 1170
77: 1170
71: 1170
68: 1170
33: 1170
159: 780
86: 780
85: 780
72: 780
65: 780

Wiersz 1, kolumna 1    100%    Windows (CRLF)    UTF-8

```

Plik z tablicą kodową:

tree.code — Notatnik

PlikEdytujWyswietl

115-0000
45-00010
100-00011
110-0010
65-00110000000
72-00110000001
46-001100000100
58-001100000101
63-00110000011
159-001100000100
84-001100001010
81-0011000010110
82-0011000010111
66-0011000011
102-00110001
185-0011001
234-0011010
118-0011011
98-001110
107-001111
111-0100
10-01010
13-01011
122-01100
67-0110100000
71-01101000010
77-01101000011
33-011010000100
68-011010000101
83-011010000110
85-0110100001110
86-0110100001111
191-01101001
104-0110101
230-01101100
113-011011010
241-0110110110
73-011011011000
74-0110110111001
69-0110110111010
70-0110110111011
79-0110110111100
80-0110110111101
76-0110110111110
78-0110110111111
44-0110111
117-011110
99-01111
97-1000
109-10010
121-100110
179-1001110

Wiersz 1, kolumna 1100%Windows (CRLF)UTF-8

Plik z drzewem kodowania:

tree.graf — Notatnik

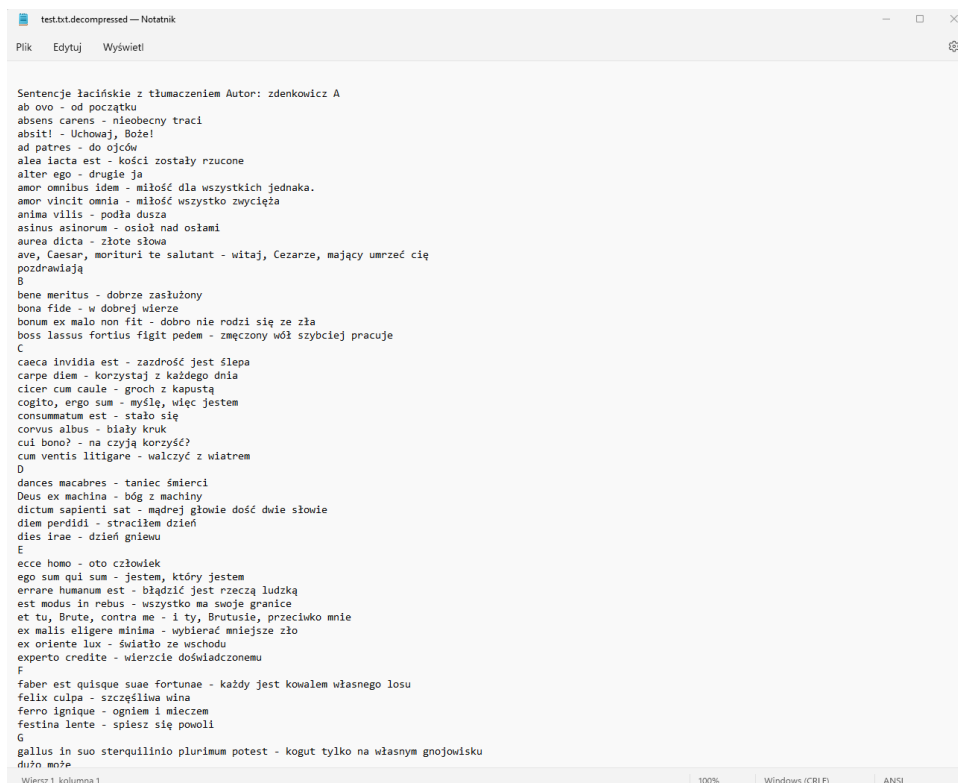
PlikEdytujWyswietl

32:409500 childLeft:- childRight:- Parent:#60
105:224640 childLeft:- childRight:- Parent:#57
101:202020 childLeft:- childRight:- Parent:#56
97:177060 childLeft:- childRight:- Parent:#55
111:138060 childLeft:- childRight:- Parent:#53
110:127530 childLeft:- childRight:- Parent:#52
115:124020 childLeft:- childRight:- Parent:#51
114:109980 childLeft:- childRight:- Parent:#50
116:104520 childLeft:- childRight:- Parent:#49
109:93210 childLeft:- childRight:- Parent:#48
99:88920 childLeft:- childRight:- Parent:#47
117:83850 childLeft:- childRight:- Parent:#47
122:78000 childLeft:- childRight:- Parent:#46
13:70590 childLeft:- childRight:- Parent:#45
10:70590 childLeft:- childRight:- Parent:#45
100:65910 childLeft:- childRight:- Parent:#43
45:59280 childLeft:- childRight:- Parent:#43
119:54210 childLeft:- childRight:- Parent:#42
108:54210 childLeft:- childRight:- Parent:#41
112:49530 childLeft:- childRight:- Parent:#41
121:45630 childLeft:- childRight:- Parent:#40
107:34710 childLeft:- childRight:- Parent:#38
98:34710 childLeft:- childRight:- Parent:#38
106:28860 childLeft:- childRight:- Parent:#36
103:28470 childLeft:- childRight:- Parent:#36
179:23010 childLeft:- childRight:- Parent:#35
44:22620 childLeft:- childRight:- Parent:#34
104:19890 childLeft:- childRight:- Parent:#33
118:18330 childLeft:- childRight:- Parent:#32
234:16380 childLeft:- childRight:- Parent:#32
185:16380 childLeft:- childRight:- Parent:#31
156:13260 childLeft:- childRight:- Parent:#30
230:10530 childLeft:- childRight:- Parent:#29
191:10140 childLeft:- childRight:- Parent:#28
102:8190 childLeft:- childRight:- Parent:#27
243:6630 childLeft:- childRight:- Parent:#26
120:6240 childLeft:- childRight:- Parent:#26
113:5460 childLeft:- childRight:- Parent:#25
241:2730 childLeft:- childRight:- Parent:#22
67:1950 childLeft:- childRight:- Parent:#20
66:1950 childLeft:- childRight:- Parent:#19
83:1170 childLeft:- childRight:- Parent:#16
77:1170 childLeft:- childRight:- Parent:#15
71:1170 childLeft:- childRight:- Parent:#15
68:1170 childLeft:- childRight:- Parent:#14
33:1170 childLeft:- childRight:- Parent:#14
159:780 childLeft:- childRight:- Parent:#13
86:780 childLeft:- childRight:- Parent:#12
85:780 childLeft:- childRight:- Parent:#12
72:780 childLeft:- childRight:- Parent:#11
65:780 childLeft:- childRight:- Parent:#11
63:780 childLeft:- childRight:- Parent:#10

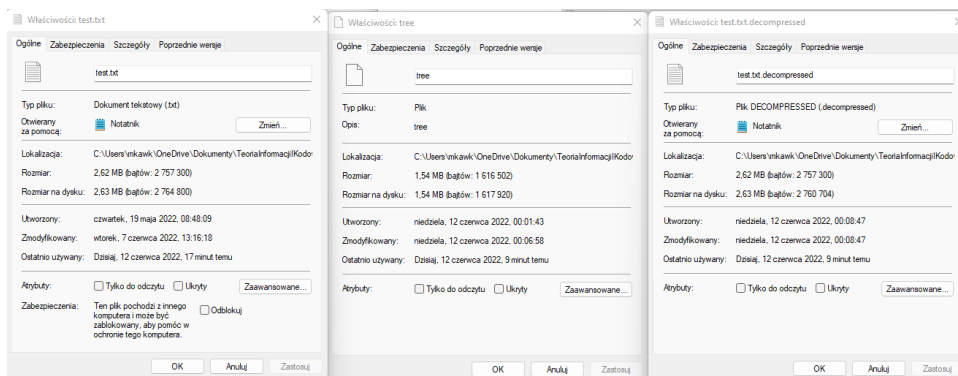
Wiersz 1, kolumna 1100%Windows (CRLF)UTF-8

Plik do kompresji:

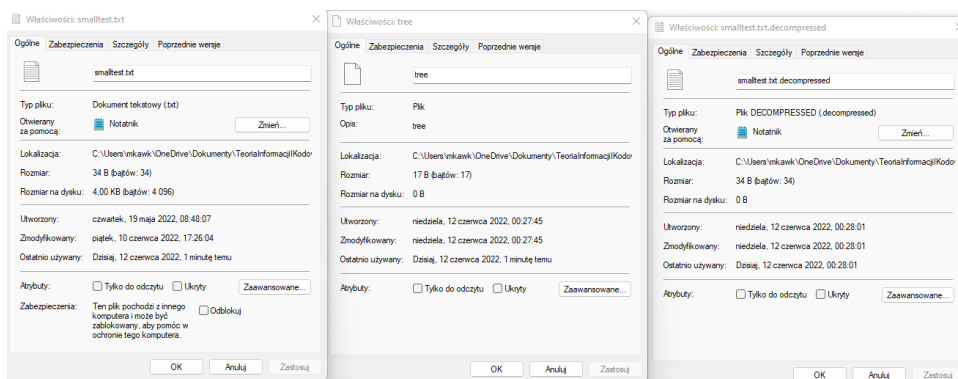
Plik zdekompresowany:



Porównanie wielkości pliku przed kompresją, skompresowanego oraz zdekompresowanego:



Porównanie wielkości pliku przed kompresją, skompresowanego oraz zdekompresowanego dla małego pliku:



5.2. Wnioski.

Jak widać na powyższym przykładzie, kompresja jest efektywna zarówno dla małych i dużych plików. Największy wpływ na jakość kompresji ma ilość różnych znaków, ponieważ im ich mniej tym mniejsze drzewo kodowe, a co za tym idzie, krótsze kody znaków.

Ważną kwestią przy implementacji kodowania Huffmana jest to, że kody znaków mogą być dłuższe od oryginalnych kodów. W takim przypadku należy zastosować "unsigned int" zamiast "unsigned char".

Przedstawiony program działa poprawnie dla dużych oraz małych plików, a po dekompresji plik jest dokładnie taki sam jak przed kompresją.