Dokumentacja techniczna projektu CCompressor

Kompresor danych wykorzystujący algorytm Huffmana, wykonany w języku C

Skład zespołu

- Mikołaj Pątkowski
- Marcin Wardak
- Paweł Wróblewski

Opis projektu

Celem projektu jest wytworzenie programu w języku C, który kompresuje bezstratnie plik wejściowy za pomocą algorytmu Huffmana. Plik wejściowy jest dowolnym plikiem binarnym. Plik wyjściowy jest plikiem zawierającym skompresowane dane za pomocą nowoutworzonych kodów, zapisanych w formie słownika na początku pliku wyjściowego. Dekompresja pliku polega na odkodowaniu danych przy pomocy tabeli kodów.

Opis flag

`-i <nazwa_pliku>`	nazwa pliku wejściowego (jeżeli plik nie jest możliwy do czytania program kończy działanie z odpowiednim statusem)
`-b`	zmiana rozmiaru bufforu (ilość bajtów wczytywana w jednej pętli wczytywania, default=50)
`-h`	wyświetlenie instrukcji obsługi programu

Sposób działania

Program zaczyna działanie od otworzenia pliku wejściowego. Jeżeli jest to niemożliwe, kończy działanie. Program tworzy plik wyjściowy (gdy plik wejściowy ma rozszerzenie zmienia jego rozszerzenie na .cps, gdy plik wejściowy nie ma rozszerzenia, program dodaje rozszerzenie .cps). Program wczytuje treść pliku, stosując buffor wielkości 50B (lub innym jeżeli zostało to zdefiniowane przy pomocy flagi –b), oraz liczy podczas tego częstotliwość występowania poszczególnych znaków i zapisuje ją do tablicy.

```
18      while (read = fread ( DstBuf: input.data, ElementSize: sizeof (*input.data), Count: input.BUFFER_SIZE, File: in))
19      {
20          input.length += read;
21          for(int i = 0; i < read; i++) {
22              input.num[input.data[i]]++;     // Zlicz wystepowanie znaków
23          }
```

Następnie program dla każdego odczytanego znaku tworzy osobne drzewo binarne. Wszystkie znaki zapisane jako osobne węzły drzewa zostają umieszczone w kolejce priorytetowej. Kolejka ma formę listy liniowej z priorytetami, będącymi ilością wystąpień danego znaku

```
32      queue *q = initQue();
33      for (int i = 0; i < 256; ++i)
34          if (input.num[i]) {
35              ++code_num;
36              // Stwórz treeNode i dodaj je do kolejki
37              addToQue(q,  tree: makeTreeNode(i,input.num[i]));
38          }
39
40      // Tworzenie drzewa binarnego i słownika
41      treeNode *root = makeTree(q);
42  #ifdef DEBUG
43      ...
44  #endif
45      unsigned char*** dict = make_dict(root, code_num);
```

```
typedef struct queNode{     // Struktura pojedynczego elementu w kolejce
    struct queNode *next;
    struct treeNode *tree; // Kolejka zawiera poszczególne drzewa, w których są zapisane symbole
    int prior;             // Priorytet = liczba wystąpień danego znaku
} queNode;

typedef struct queue{       // Struktura "kolejki" - ma wskaźniki na początek i koniec
    queNode *head;
    queNode *tail;
    int size;              // Liczba unikalnych znaków (liści drzewa)
} queue;
```

```
queue *initQue(){
    queue *newQue = malloc( Size: sizeof *newQue);
    if (newQue != NULL){
        newQue->head = NULL;
        newQue->tail = NULL;
        newQue->size = 0;
    }
    return newQue;
}
```

```
typedef struct treeNode{
    unsigned char c;
    int is_leaf;
    int count;
    struct treeNode *left;
    struct treeNode *right;
} treeNode;
```

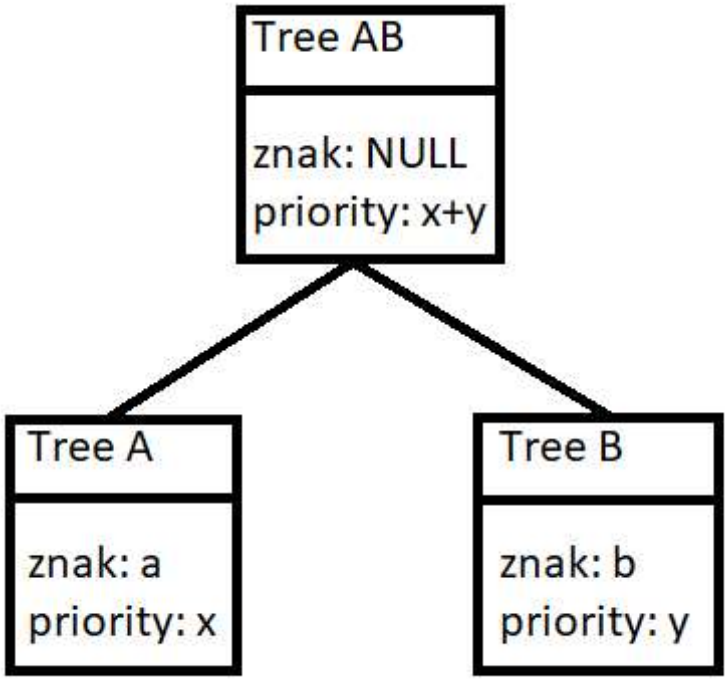
```
treeNode *makeTree(struct queue *q){

    treeNode *lowestPrior1 = NULL;
    treeNode *lowestPrior2 = NULL;

    while(q->head->next != NULL){   // Dopóki w kolejce są min 2 elementy
        lowestPrior1 = q->head->tree;
        lowestPrior2 = q->head->next->tree;
        // Połącz 2 elementy kolejki o najmniejszym priorytecie
        treeNode *newTree = joinNodes( first: lowestPrior1,  second: lowestPrior2);
        moveQueue(q);
        addToQue(q,  tree: newTree);
    }

    return q->head->tree;
}
```

Pierwsze dwa elementy kolejki (czyli te, które mają najmniejszy priorytet) zostają połączone w jedno drzewo w taki sposób, że synami nowego drzewa są korzenie dwóch starych drzew, a jego priorytet jest równy sumie priorytetów jego synów. Algorytm powtarza się, dopóki w kolejce nie zostanie tylko jeden element.



W taki sposób powstaje pełne drzewo binarne, którego liśćmi są konkretne znaki. Na jego podstawie program tworzy nowe kody dla znaków. Przechodząc po drzewie, każde przejście do lewego potomka odkłada na stos znak '0', a każde przejście w prawo '1'. Gdy program natrafi na liść (czyli węzeł, którego zmienna `is_leaf == 1`), to temu znakowi zostaje podana aktualna wartość przetrzymywana na stosie. Po przejściu przez zarówno lewego i prawego potomka, ze stosu zostaje usunięty jeden znak.

```c
unsigned char*** make_dict(treeNode *root, int code_num){
    int index = 0;  // Licznik kodów
    // Utwórz słownik
    unsigned char*** dict = malloc( Size: 2 * sizeof(char**));
    dict[0] = malloc( Size: code_num * sizeof(char*));
    dict[1] = malloc( Size: code_num * sizeof(char*));

    // Zainicjuj stos
    stack_t *stack = initialize_stack();
    if(stack == NULL){
        fprintf( stream: stderr,  format: "Error: no memory for making codes\n");
        return NULL;
    }

    traverse(root, stack, dict, &index);
    free_stack(stack);
    return dict;
}
```

```c
// Funkcja przechodzi rekurencyjnie po drzewie celem stworzenia kodów
static void traverse(treeNode *root, stack_t* stack, unsigned char*** dict, int* index){
    if(root->left != NULL){
        put(stack, '0');
        traverse( root: root->left, stack, dict, index);
//          free(root->left);
    }
    if(root->right != NULL){
        put(stack, '1');
        traverse( root: root->right, stack, dict, index);
//          free(root->right);
    }
    if(root->is_leaf){  // Jeżeli wierzchołek jest liściem, dodaj symbol i kod do słownika
        get_char(root, dict, *index);
        get_code(stack, dict, *index);
        (*index)++;
    }
    pop(stack);
}
```

Do przechowywania słownika używana jest zmienna `char ***dict`. W polu `dict[0][i]` znajduje się i-ty znak, a w polu `dict[1][i]` jego nowy kod. Tak więc znak ‘a’ o kodzie “0101” ma swoją reprezentację jako:

`dict[0][0] == “a”`

`dict[1][0] == “0101”`

Słownik jest następnie przekładany do struktury, która przechowuje kody w formie binarnej oraz ich długość w bitach. Wywołana zostaje także funkcja kodująca słownik do ciągu bajtów w celu umieszczenia w pliku wyjściowym.

Następuje teraz zakodowanie pliku wejściowego. Dla każdej wczytanej partii wywoływana jest funkcja kodująca.

Funkcja kodująca otrzymuje strukturę ze słownikiem oraz treść pliku. Dla każdego znaku znajduje jego kod, po czym, posługując się operatorem ‘ |= ‘, wstawia go po fragmencie do danych wyjściowych.

Po zakończeniu działania funkcji, program wysyła skompresowane dane do pliku wyjściowego.

Przykład

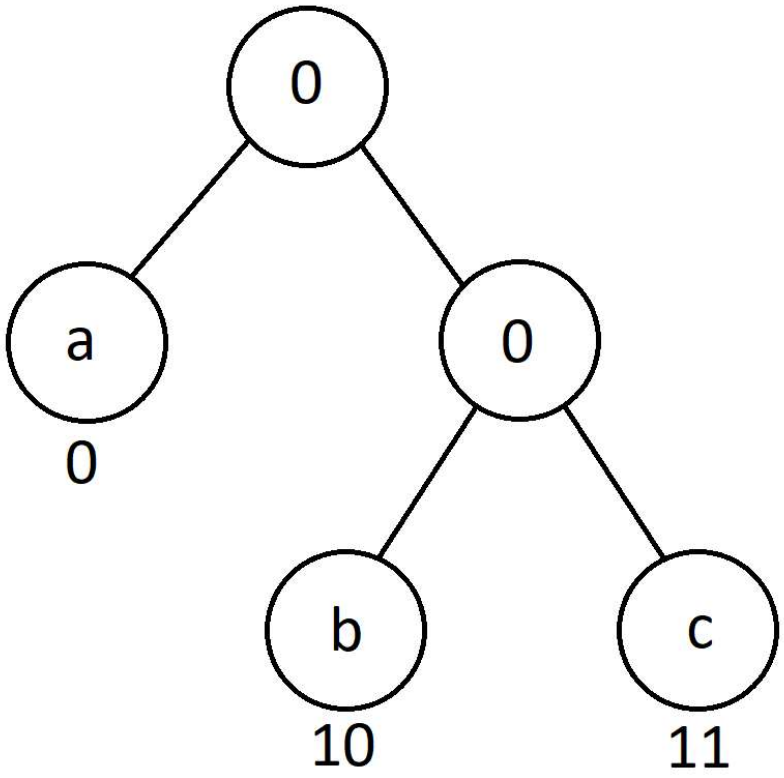
Program na wejściu dostaje plik zawierający przykładowy ciąg znaków:

lorem ipsum dolor sit amet, consectetur adipiscing elit. proin elementum, justo eget fermentum pulvinar, urna nunc pharetra ligula, at convallis diam mauris vitae est. vivamus vitae dui at velit consectetur accumsan. nullam auctor risus in neque vestibulum, vel semper erat faucibus. ut hendrerit ultrices odio vitae consequat. phasellus tincidunt orci.

Program zlicza występowanie poszczególnych znaków, tworzy z nich drzewa i ustawia je w kolejności od najrzadszych w strukturze Priority Queue.

j	q	f	b	h	g	,	.	p	d	v	o	m	c	l	r	n	s	a	u	t	i	e	_
1	2	2	2	3	3	5	6	7	7	20	13	14	15	17	19	29	20	24	27	28	28	32	49

Każde najmniej popularne znaki łączone są ze sobą w formie drzewa binarnego. Powstaje jedno przykładowe drzewo:



Węzły w drzewie, które reprezentują połączenie (czyli nie mają przypisanego znaku) są oznaczone wartością ‘0’. Przechodząc w dół po drzewie powstają nowe kody dla znaków będącymi liśćmi. Przejście do lewego syna dopisuje do kodu znak 0, a do prawego syna 1. Przykładowo:

Korzeń -[lewy]-> 0 (a)

Korzeń -[prawy]-> 1 -[lewy]-> 01 (b)

Korzeń -[prawy]-> 1 -[prawy]-> 11 (c)

W pliku wyjściowym w pierwszych dwóch bajtach podane są kolejno: ostatni bit zajęty przez kod w ostatnim bajcie pliku wyjściowego oraz długość słownika (należy brać [00000000] == 256, ponieważ max liczba w bajcie to 255), treść słownika, a następnie zakodowana treść pliku wejściowego. Słownik zakodowany jest w taki sposób, że najpierw występuje znak, następnie długość odpowiadającego mu kodu w bitach, po czym sam kod w postaci binarnej. Kod wyrównany jest do prawej, bity z lewej strony niebędące częścią kodu są ustawione na 0. Przykładowo:

(W przykładzie treść pojedynczego bajta podana będzie w [] nawiasach kwadratowych, np. 'a' <=> [01100001]).

Znak	Kod	Długość	Długość w postaci binarnej
A	101	3	00000011
B	111001000	9	00001001
C	0011001100	10	00001010

A[00000011][00000101]B[00001001][00000001][11001000]C[00001010][00000000][11001100]

Po słowniku będzie bezpośrednio dołączona skompresowana treść pliku.

Treść pliku wyjściowego:

(ilość wolnych bitów w ostatnim bajcie)[00000011]A[00000011][00000101]B[00001001][00000001][11001000]C[00001010][00000000][11001100]<treść>

Specyfikacja Makefile

`make kompresja`	kompilacja całego programu
`make kompresjad`	kompilacja programu w trybie DEBUG (wyświetlanie informacji co program robi)
`make cleantst`	usunięcie pozostałości po testach
`make test1`	wywołanie testu 1
`make test2`	wywołanie testu 2
`make test3`	wywołanie testu 3
`make test4`	wywołanie testu 4
`make clean`	usunięcie skompilowanych plików kompresora (kompresja, kompresjad)

Sposób wywołania programu

Aby wywołać program należy w folderze, w którym został wykonany makefile użyć komendy:

```
./kompresja | kompresjad    [-i <plik_wejsciowy> | -t  | -b <rozmiar_buffora> | -h]
```

W przypadku wywołania programu jednocześnie używając flag –i oraz –t program skorzysta z tej flagi, która została wywołana później.

Testowanie oprogramowania

W celu przetestowania poprawności działania kompresora, został zbudowany prosty dekompresor w języku C.

Testy:

1)
   a) Utworzenie pliku zawierającego predefiniowany tekst o długości 20
   b) Kompresja (z flagą -debug)
   c) Wyświetlenie kodu wyjścia programu
2)
   a) Utworzenie pliku zawierającego predefiniowany skompresowany tekst
   b) Dekompresja
   c) Wyświetlenie kodu wyjścia programu
3)
   a) Utworzenie pliku zawierającego losowe znaki o długości 100
   b) Kompresja (z flagą -debug)
   c) Wyświetlenie kodu wyjścia programu
4)
   a) Utworzenie pliku zawierającego losowe znaki o długości 300
   b) Kompresja
   c) Dekompresja
   d) Porównanie treści pliku wejściowego z dekompresowanym plikiem wyjściowym

Obsługa błędów

Kod	Opis
0	Poprawne wykonanie
3	Nieprawidłowy odczyt pliku (nie istnieje)
4	Nieprawidłowy zapis pliku (brak uprawnień)