Wojciech Pratkowiecki

Projekt: kodowanie znaków

Specyfikacja:

Wersja kodowanie:

Wejście: niepusty plik tekstowy w formacie ‘.txt’. Program uruchomiony z terminala za pomocą przekazania danych:

./projekt (nazwa programu) c (koduj) in.txt (przykładowa nazwa pliku do zakodowania) out.txt (przykładowa nazwa pliku do zakodowania)

Wyjście: plik tekstowy w formacie ‘.txt’ zawierający zakodowany na wejściu tekst algorytmem Huffmana wraz z niezbędnymi danymi do odkodowania.

Wersja dekodowanie:

Wejście: plik tekstowy w formacie ‘.txt’ zawierający budowę drzewa Huffmana oraz zakodowany plik tekstowy. Program uruchomiony z terminala za pomocą komend:

./projekt u (dekoduj) code.txt text.txt

Wyjście: plik tekstowy w formacie ‘.txt’ zawierający odkodowany teskt.

Przykładowe kodowanie tekstu:

Weźmy tekst: „zaraz a raz!” i zakodujmy go zgodnie ze specyfikacją programu. Za początkową fazę kodowania odpowiedzialny jest moduł get_code. Na początek należy zliczyć wystąpienia poszczególnych znaków z powyższego tekstu. Zostanie do tego użyta tablica *struktur letter*. Każda z 128 początkowych komórek odpowiada dokładnie jednemu znakowi ASCII. Po zliczeniu znaków otrzymamy dane:

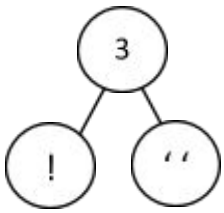
Znak:	‘ ‘	!	a	r	z
Wystąpienia:	2	1	4	2	3

Następnym etapem jest posortowanie według liczby wystąpień powyższej tabeli. Oczywiście tabela ta będzie zawierała jeszcze 123 komórki wypełnione zerami. Po posortowaniu powyższa tabela będzie wyglądać następująco:

Znak:	!	‘ ‘	r	z	a
Wystąpienia:	1	2	2	3	4

Aby pozbyć się komórek zawierających zera zostaje stworzona lista struktur huff_list. Struktura ta może przechowywać element w postaci struktury letter lub huff_tree. Wszystkie struktury używane w całym programie są zdefiniowane w module lib.

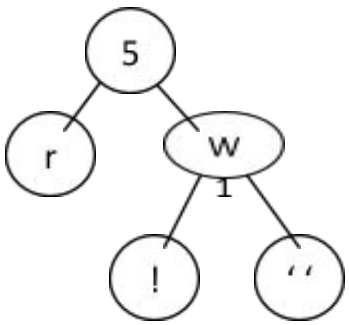
Kolejnym krokiem jest budowanie drzewa Huffmana. Ta część programu ma miejsce w module make_code. Funkcja tworząca drzewo to get_tree. W tym celu bierzemy dwa pierwsze elementy z listy i łączymy je węzłem. Następnie nowo powstały węzeł umieszczamy w powyższej liście w odpowiednim miejscu. Za znalezienie pozycji odpowiedzialna jest funkcja add_elem. Wartością występowania tego węzła będzie suma wartości występowania znaków znajdujących się w jego liściach.



Nazwijmy ten węzeł W1. Nasza lista wygląda teraz tak:

Znak:	!	‘ ‘	r	W1	z	a
Wystąpienia:	1	2	2	3	3	4

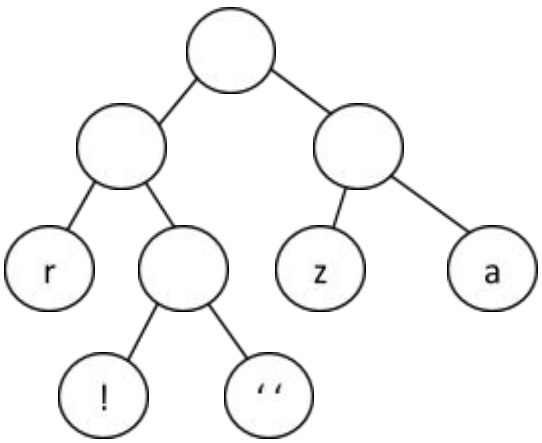
Zgodnie z algorytmem bierzemy dwa kolejne elementy z tabeli (kolorem szarym zaznaczone są elementy już zakodowane) i kodujemy je tak jak poprzednio. Tymi elementami są litera ‘r’ oraz węzeł W1. Powstaje więc kolejny węzeł:



Nazwijmy ten węzeł W2 i wstawmy w odpowiednie miejsce w liście. Po tym kroku lista będzie występować następująco:

Znak:	!	‘ ‘	r	W1	z	a	W2
Wystąpienia:	1	2	2	3	3	4	5

Następne węzły powstają analogicznie. Po stworzeniu ostatniego węzła nasze drzewo będzie wyglądać następująco:



Kolejnym etapem jest wyznaczenie każdemu znakowi odpowiedniego kodu. Zauważmy, że znaki znajdują się tylko w liściach drzewa. Algorytm przyporządkowujący kod jest następujący: startujemy od korzenia drzewa. Kiedy idziemy w lewą jego stronę do znaku dopisujemy ‘0’, kiedy w prawo ‘1’. Kody te wyznaczane są w funkcji code_tree i zapisane w tablicy dwuwymiarowej code_arr. Kody powyższych znaków będą występowały następująco:

Znak:	!	‘ ‘	r	z	a
Kod:	010	011	00	10	11

Możemy więc zakodować tekst. W tym celu każdemu znakowi z tekstu na wejściu przyporządkowujemy jego kod. Algorytm ten jest zaimplementowany w funkcji text2code Po zakodowaniu tekstu początkowego otrzymamy następujący ciąg bitów:

10110011100111101100111 0010

Dzielimy zakodowane bity na „ósemki” dzięki czemu będziemy mogli zapisać je jako ciąg liczb typu *char*. Musimy też wiedzieć ile bitów z ostatniej liczby należy do zakodowanego tekstu. W tym celu tworzymy zmienną while_eof która jest równa długości powyższego ciągu modulo 8. Mamy więc:

10110011 10011110 11001110 010 while_eof = 3

Zamieniamy powyższe bity na liczby w funkcji bin2dec. Otrzymujemy:

179 158 206 2

W celu zmniejszenia rozmiaru pliku wyjściowego zamiast liczb zapisywać będziemy znaki char odpowiadające poszczególnym wartościom. W ten sposób otrzymaliśmy zakodowany tekst wejściowy, wypisywany w funkcji text2code. Dysponujemy również danymi potrzebnymi do jego odkodowania. Nasz plik wyjściowy będzie wyglądał następująco:

ZAKODOWANY TEKST:

3 (wartość while_eof)

5 (ilość zakodowanych znaków)

! – 010 (kody znaków)

‘ ‘ – 011

r – 00

z – 10

a – 11

179 158 206 2 (w rzeczywistości znaki odpowiadające tym liczbom)

Dekodowanie tekstu:

Dekodowanie tekstu rozpoczyna się od zbudowania drzewa Huffmana. Wczytanie i przeanalizowanie zakodowanego tekstu odbywa się w module get_uncode, tworzenie drzewa w funkcji get_tree. Dane do drzewa wyglądają tak jak te podane w pliku wyjściowym kodowania. Każdy znak uaktualnia drzewo o kolejne gałęzie tworzone na podstawie wartości binarnej znaku. Ścieżki dla każdego znaku powstają w funkcji create_path. Znaki te pobierane są w funkcji numbers która tworzy również listę liczb odpowiadających znakom.

Po przeanalizowaniu kodu ostatniego znaku otrzymamy drzewo dokładnie takie samo jak to utworzone podczas kodowania. Znamy również ilość bitów oznaczających znaki w ostatnim znaku podanym na wejściu. Kolejna część programu odbywa się w module make_uncode. Algorytm dekodowania będzie wyglądał więc następująco:

Z wartości znaków podanych na wejściu brane będą bity od lewej strony. Dla przykładu pierwsza wartość znaku to 179. Tak więc jej skrajnie lewy bit to 1. Na naszym drzewie idziemy więc w prawą stronę. Sprawdzamy, czy węzeł w którym właśnie się znaleźliśmy ma w obu gałęziach wartości NULL. W naszym przypadku nie ma więc bierzemy kolejny bit, tym razem 0. W naszym drzewie poruszamy się więc w lewo. Węzeł, w którym się znaleźliśmy na w gałęziach wartości NULL. Jest on więc liściem drzewa. W takim razie wiemy, że pierwszym znakiem naszego tekstu jest ‘z’, gdyż to ta litera odpowiada kodowi 10. Bierzemy kolejne bity i kolejne liczby postępując analogicznie. Kiedy dojdziemy do ostatniej liczby sprawdzamy ilość bitów wskazaną przez while_eof. W ten sposób jesteśmy w stanie utworzyć plik wyjściowy, który będzie wyglądał następująco:

zaraz a raz!

Podsumowanie:

Głównymi zadaniami w trakcie kodowania tekstu będzie:

1. Wczytanie pliku
2. Zliczanie wystąpień
3. Sortowanie tablicy i utworzenie listy
4. Tworzenie drzewa
5. Przypisanie kodu każdemu znakowi
6. Przedstawienie tekstu w postaci ciągu bitów
7. Przedstawienie ciągu bitów w postaci liczb dziesiętnych a następnie znaków
8. Wypisanie pliku wyjściowego

Natomiast dekodowanie będzie wymagało następujących czynności:

1. Wczytanie pliku
2. Tworzenie drzewa na podstawie kodów znaków
3. Odczytywanie kojonych znaków w tekście
4. Wypisanie pliku wyjściowego

Moduły:

1. main
2. lib - moduł z deklaracjami struktur oraz funkcjami je tworzącymi
3. get_code - moduł odpowiedzialny za analizowanie pliku przeznaczonego do kodowania
4. make_uncode - odpowiada za kodowanie pliku
5. get_uncode - pobiera i analizuje zakodowany tekst
6. make_uncode - posiadając wszystkie konieczne informacje dekoduje plik tekstowy

Użyte struktury:

1.Kodowanie:

- *letter* - zawiera znak typu char oraz liczbę wystąpień tego znaku w tekście
- *huff_list* - zawiera strutkurę letter lub huff_tree, wartość struktury i wskaźnik na następną
- *huff_tree* - zawiera wartość struktury, element huff_list oraz dwa wskaźniki

2. Dekodowanie:

- *uncode_tree* - zawiera znak typu char oraz dwa wskaźniki
- *list* - podstawowa lista (wartość int oraz wskaźnik na następny element)

Główne funkcje:

1. Kodowanie:
    a. huff_list *read_f - wczytywanie danych z pliku, zwraca listę posortowanych według wystąpień znaków z pliku.
    b. huff_tree *make_tree - tworzy drzewo binarne konieczne do generowania kodów binarnych znaków. Zwraca wskaźnik na korzeń.
    c. huff_list *add_elem - wstawia nowy węzeł w odpowiednie miejsce w liście, zwraca wskaźnik na początek listy.
    d. void code_tree - do dwuwymiarowej tablicy wstawia kody binarne poszczególnych znaków.
    e. void text2code - wypisuje plik wyjściowy.
2. Dekodowanie:
    a. list *numbers - wczytuje znaki z pliku wyjściowego i tworzy listę z ich wartościami. Zwraca wskaźnik na początek listy.
    b. uncode_tree *get_tree - ze wczytanych wcześniej znaków odtwarza drzewo kodowania. Zwraca wskaźnik na korzeń.
        i. uncode_tree *create_path - dla pojedynczego znaku odtwarza ścieżkę w drzewie i tworzy brakujące węzły.

c. void get_sign - mając listę wartości znaków oraz odtworzone drzewo za pomocą wartości binarnych znaków rozkodowuje kolejne zakodowane znaki i wypisuje je do pliku wyjściowego