Algorytm Huffmana w języku C++

Źródło: Michael Dziedzic, domena publiczna.

Poznaliśmy już podstawowe informacje dotyczące algorytmu Huffmana. W tym e-materiale zaimplementujemy go w języku *C++*.

Ciekawi cię, jak wyglądają implementacje algorytmu Huffmana w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych e-materiałach z tej serii:

- Algorytm Huffmana w języku Java,
    - Algorytm Huffmana w języku Python.

Więcej zadań? Przejdź do Algorytm Huffmana – zadania maturalne.

> **Twoje cele**
>
> - Prześledzisz zrealizowany w języku C++ algorytm Huffmana.
> - Zaimplementujesz program w języku C++ kodujący ciąg znaków za pomocą algorytmu Huffmana.
> - Zakodujesz ciągi znaków w celu zmniejszenia zajmowanej przez nie pamięci.

Przeczytaj

Już wiesz

W roku 1952 David Huffman opracował prefiksową metodę kompresji bezstratnej.

Mówimy, że kodowanie jest prefiksowe, gdy żadne słowo kodowe nie jest początkiem innego słowa.

Słowem kodowym w kontekście kodowania Huffmana określamy ciągi bitów, które jednoznacznie interpretujemy jako znak. Uwaga – w implementacji dla uproszczenia będziemy kodować każdy znak osobno, bez wcześniejszego grupowania.
Problem 1

Załóżmy, że w alfabecie, którym się posługujemy, są trzy znaki zgrupowane w następującej książce kodowej:

- A – reprezentowane przez 1,
- B – reprezentowane przez 01,
- C – reprezentowane przez 00.

Za ich pomocą zakodowaliśmy pewną wiadomość i otrzymaliśmy taki oto wynik: 1010000011.

Spróbuj odkodować samodzielnie ten napis.

> **Problem 2**
>
> Załóżmy, że w alfabecie, którym się posługujemy, są trzy znaki zgrupowane w następującej książce kodowej:
>
> - A – reprezentowane przez 0,
> - B – reprezentowane przez 01,
> - C – reprezentowane przez 1.
>
> Za ich pomocą zakodowaliśmy pewną wiadomość i otrzymaliśmy taki oto wynik: 00111010.
>
> Spróbuj odkodować samodzielnie ten napis:

Kodowanie prefiksowe umożliwia odkodowanie wiadomości metodą zachłanną, tzn. odkodowujemy najwcześniejsze odnalezione słowo kodowe.

Słowa kodowe używane w algorytmie Huffmana mają postać binarną, a w języku C++ domyślnie posługujemy się kodem ASCII.

> **Problem 3**
>
> Używając słów kodowych z problemu pierwszego, udało nam się zakodować 6 znaków (ABCCBA) na 10 bitach. Należy pamiętać, że każdy z tych znaków zajmuje w pamięci programu po 8 bitów, czyli po 1 bajcie. Udało nam się więc skompresować 48 bitów do 10 bitów, czyli 6 bajtów do niecałych 2 bajtów.
>
> Jaki jest stopień kompresji?

Implementacja w języku C++

> **Problem 4**
>
> Zaimplementujmy w języku C++ program kodujący podany ciąg znaków za pomocą algorytmu Huffmana.

W celu uproszczenia implementacji do przechowywania zakodowanej wiadomości w postaci bitowej posłużymy się łańcuchem znaków, w którym wykorzystamy znaki ASCII

odpowiadające cyfrom 0 oraz 1. Należy pamiętać, że taka forma przechowywania danych w pamięci nie powoduje zmniejszenia liczby wykorzystanych bitów, lecz dodatkowo ją zwiększa.

Specyfikacja problemu:

Dane:

- `wiadomosc` – łańcuch znaków do zakodowania

Wynik:

- `zakodowana_wiadomosc` – łańcuch znaków (zer lub jedynek) reprezentujący w postaci bitowej zakodowany według kodowania Huffmana ciąg znaków `wiadomosc`

Algorytm zaimplementujemy w kilku etapach. Najpierw zaimportujemy potrzebne biblioteki oraz wskażemy użycie standardowej przestrzeni nazw. Wykorzystamy bibliotekę `<iostream>` do pobierania i wypisywania danych ze standardowego wejścia oraz wyjścia, a także bibliotekę `<string>` do wykonywania operacji na łańcuchach znaków. Potrzebna będzie nam również implementacja listy w języku C++, dostępna w bibliotece `<list>`, która w przeciwieństwie do zwykłych tablic pozwala na szybkie usuwanie oraz dodawanie elementów.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;
```

Następnie utworzymy funkcję, której zadaniem będzie wpisanie do tablicy informacji, ile razy w ciągu znaków `zdanie` wystąpiły poszczególne znaki kodowane za pomocą standardu ASCII. Dodatkowo funkcja zliczy oraz zwróci liczbę unikalnych znaków, która wskazuje, jak dużo liści będzie występować w drzewie kodowym używanym do kodowania Huffmana.

```
int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
```

```
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}
```

W tym momencie możemy przetestować działanie utworzonej funkcji. Poniższy kod wypisze wszystkie znaki z łańcucha znaków `wiadomosc` oraz ich zliczenia:

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadodadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

```

```
    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    return 0;
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
```

Program wyświetlił zliczenia kolejnych znaków. Jak widzisz, jeśli dany znak się powtarza, przy każdym jego wystąpieniu w łańcuchu znaków pojawia się informacja, ile razy dany znak się pojawił w całym łańcuchu (w tym przypadku zwróć uwagę na literę d; po pierwsze osobno jest liczone jej wystąpienie jako wielkiej litery, po drugie – cały łańcuch znaków jest wyświetlony w formie kolumny, więc powtarzające się wystąpienia litery d również się pojawiają).

Utworzymy teraz definicję struktury drzewiastej, którą wykorzystamy do przechowywania drzewa kodowego Huffmana. Głównym elementem struktury będzie `wierzcholek`, który posiada pola takie jak: znak, częstość występowania tego znaku w ciągu znaków `wiadomosc` oraz wskaźniki na lewy i prawy wierzchołek w strukturze drzewa.

Domyślne wartości pól `lewy` oraz `prawy` zostały ustawione na `nullptr`, co w języku C++ oznacza wskaźnik pusty. Taki zabieg umożliwi rozpoznanie wierzchołków, które nie posiadają wierzchołków podrzędnych.

```
struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};
```

W kolejnym kroku utworzymy listę unikalnych znaków wraz z ich częstościami. Korzystając z zaimplementowanej funkcji, zliczymy unikalne znaki, a następnie dla każdego z nich, jeśli jego częstość wynosi więcej niż 0, utworzymy nowy wierzchołek i dodamy go do listy wierzchołków. Lista ta posłuży nam później do utworzenia drzewa kodowego. Fragment kodu umieścimy w funkcji `main`.

> **Ważne!**
>
> Poniższy fragment kodu, jak i niektóre z kolejnych fragmentów nie zwalniają pamięci rezerwowanej z wykorzystaniem słowa kluczowego `new`. Dlatego po zakończeniu algorytmu należy zwolnić każdy rezerwowany wcześniej fragment pamięci, wykorzystując słowo kluczowe `delete`. Zwalnianie pamięci zostanie przez nas zaimplementowane w ostatecznym kodzie.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
```

```
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    // tworzenie listy wierzchołków
    list<wierzcholek*> lista_wierzcholkow;
    int j = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            lista_wierzcholkow.push_back(new wierzcholek());
            lista_wierzcholkow.back()->znak = i;
            lista_wierzcholkow.back()->czestosc = tablica_zliczen
            ++j;
            if (j == liczba_znakow) {
                break;
            }
        }
    }

```

```
    // wypisywanie listy wierzchołków
    cout << "Lista wierzcholkow:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    return 0;
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
d: 5
i: 1
k: 1
o: 3
t: 1
```

Poza zliczeniami znaków pojawiła się lista wierzchołków. Każdy znak z łańcucha znaków jest już w niej unikalny.

Moglibyśmy teraz przejść do tworzenia drzewa kodowego Huffmana, które polegałoby na pobraniu z listy dwóch wierzchołków o najmniejszej częstości, usunięciu ich z listy oraz połączeniu w nowy wierzchołek, a następnie umieszczeniu na liście. Tak skonstruowany algorytm wymagałby jednak przeszukiwania listy w każdym kroku. Znacznie prostszym rozwiązaniem może okazać się zadbanie o to, aby lista w każdym momencie była posortowana.

Dzięki wykorzystaniu implementacji listy ze standardowej biblioteki języka C++ możemy skorzystać z metody `sort()`, która sortuje elementy listy według przekazanej do niej funkcji porównującej. Napiszmy zatem potrzebną funkcję:

```
bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}
```

Warto wiedzieć, że metoda `sort()` korzysta z algorytmu sortowania introspektywnego (IntroSort), który jest pochodną sortowania szybkiego, kopcowego oraz przez wstawianie. Jego złożoność obliczeniowa wynosi: $O = (n \cdot \log_2(n))$

Tak skonstruowana funkcja umożliwi wykorzystanie metody `sort()` do posortowania listy wierzchołków nierosnąco według ich częstości. Przetestujmy jej działanie, wykorzystując poniższy kod:

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
```

```
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadodadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    // tworzenie listy wierzchołków
    list<wierzcholek*> lista_wierzcholkow;
    int j = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            lista_wierzcholkow.push_back(new wierzcholek());
            lista_wierzcholkow.back()->znak = i;
            lista_wierzcholkow.back()->czestosc = tablica_zliczen
            ++j;
            if (j == liczba_znakow) {
                break;
            }
        }
```

```
    }

    // wypisywanie listy wierzchołków
    cout << "Lista wierzcholkow:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    // sortowanie listy wierzchołków
    lista_wierzcholkow.sort(funkcja_porownujaca);

    // wypisywanie listy wierzchołków po posortowaniu
    cout << "Lista wierzcholkow po posortowaniu:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    return 0;
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
```

```
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
```

Lista wierzchołków została posortowana. Ułatwi to tworzenie drzewa Huffmana.

W tym momencie możemy przejść do budowy drzewa kodowego. Dopóki na liście jest więcej niż jeden wierzchołek, zdejmujemy z listy dwa ostatnie wierzchołki (o najmniejszej częstości), łączymy je w nowy wierzchołek o częstości będącej sumą częstości obu wierzchołków, a następnie dodajemy go do listy w odpowiednie miejsce. Gdy na liście zostanie jeden wierzchołek, mamy gotowe drzewo kodowe.

> **Ważne!**
>
> W celu uproszczenia implementacji, zamiast wyszukiwać odpowiednie miejsce dla nowopowstałego wierzchołka, odłożymy go na koniec listy, a następnie ją posortujemy. Należy pamiętać, że takie rozwiązanie zwiększa złożoność obliczeniową algorytmu.

```cpp
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
```

```
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    // tworzenie listy wierzchołków
    list<wierzcholek*> lista_wierzcholkow;
    int j = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            lista_wierzcholkow.push_back(new wierzcholek());
            lista_wierzcholkow.back()->znak = i;
            lista_wierzcholkow.back()->czestosc = tablica_zlicze
            ++j;
```

```
            if (j == liczba_znakow) {
                break;
            }
        }
    }

    // wypisywanie listy wierzchołków
    cout << "Lista wierzcholkow:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    // sortowanie listy wierzchołków
    lista_wierzcholkow.sort(funkcja_porownujaca);

    // wypisywanie listy wierzchołków po posortowaniu
    cout << "Lista wierzcholkow po posortowaniu:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    // tworzenie drzewa kodowego
    while (lista_wierzcholkow.size() > 1) {
        // zdejmujemy z listy 2 wierzchołki w1 i w2
        wierzcholek* w1 = lista_wierzcholkow.back();
        lista_wierzcholkow.pop_back();
        wierzcholek* w2 = lista_wierzcholkow.back();
        lista_wierzcholkow.pop_back();

        // łączymy wierzchołki w1 oraz w2, tworząc wierzchołek w
        wierzcholek* w3 = new wierzcholek();
        w3->czestosc = w1->czestosc + w2->czestosc;
        w3->lewy = w1;
        w3->prawy = w2;

        // umieszczamy wierzchołek w3 na końcu listy, a następni
        lista_wierzcholkow.push_back(w3);
        lista_wierzcholkow.sort(funkcja_porownujaca);
    }

    // gotowe drzewo kodowe
    wierzcholek* drzewo_kodowe = lista_wierzcholkow.back();
```

```
    return 0;
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
```

W tym momencie możemy napisać funkcję, która rekurencyjnie przeszuka drzewo kodowe w celu utworzenia książki kodowej. Utwórzmy prostą strukturę, która będzie zawierała znak oraz odpowiadający mu kod:

```
struct kod {
    char znak;
    string slowo_kodowe;
};
```

Rekurencyjne przeszukiwanie drzewa kodowego zaczniemy od sprawdzenia, czy obecny wierzchołek jest liściem — nie posiada potomka. Jeśli trafiliśmy na taki wierzchołek, spełniony jest warunek rekurencji. W książce kodowej dodany zostanie nowy wpis, po czym nastąpi powrót. Jeśli jednak wierzchołek nie jest liściem, przechodzimy rekurencyjnie do obu jego potomków, powiększając dotychczasowy kod o cyfrę 0 lub 1.

> **Ważne!**
>
> Kodowanie Huffmana nie jest jednoznaczne. W przypadku poniższego kodu przejściu do lewego potomka odpowiada cyfra 0, a do prawego cyfra 1. Jeśli sytuację odwrócimy, otrzymamy inny kod, ale o takiej samej długości.

```
void utworz_ksiazke_kodowa(wierzcholek* drzewo_kodowe, kod* ksiaz
    static int i = 0;
    if (drzewo_kodowe->lewy == nullptr && drzewo_kodowe->prawy ==
        ksiazka_kodowa[i].znak = drzewo_kodowe->znak;
        ksiazka_kodowa[i].slowo_kodowe = dotychczasowy_kod;
        ++i;
    }
    else {
        utworz_ksiazke_kodowa(drzewo_kodowe->lewy, ksiazka_kodowa
        utworz_ksiazke_kodowa(drzewo_kodowe->prawy, ksiazka_kodow
    }
}
```

Wewnątrz funkcji `main` umieszczamy fragment odpowiadający za rezerwację pamięci dla tablicy o długości równej liczbie znaków tak, aby każdy otrzymał swoje słowo kodowe, a następnie wywołujemy rekurencyjnie funkcję wypełniającą książkę kodową. Na koniec, aby sprawdzić poprawność, możemy wypisać książkę kodową.

```
#include <iostream>
```

```cpp
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

struct kod {
    char znak;
    string slowo_kodowe;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

void utworz_ksiazke_kodowa(wierzcholek* drzewo_kodowe, kod* ksia
    static int i = 0;
    if (drzewo_kodowe->lewy == nullptr && drzewo_kodowe->prawy =
        ksiazka_kodowa[i].znak = drzewo_kodowe->znak;
        ksiazka_kodowa[i].slowo_kodowe = dotychczasowy_kod;
```

```
        ++i;
    }
    else {
        utworz_ksiazke_kodowa(drzewo_kodowe->lewy, ksiazka_kodow
        utworz_ksiazke_kodowa(drzewo_kodowe->prawy, ksiazka_kodo
    }
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    // tworzenie listy wierzchołków
    list<wierzcholek*> lista_wierzcholkow;
    int j = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            lista_wierzcholkow.push_back(new wierzcholek());
            lista_wierzcholkow.back()->znak = i;
            lista_wierzcholkow.back()->czestosc = tablica_zlicze
            ++j;
            if (j == liczba_znakow) {
                break;
            }
        }
    }

    // wypisywanie listy wierzchołków
    cout << "Lista wierzcholkow:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }
```

```
86
87      // sortowanie listy wierzchołków
88      lista_wierzcholkow.sort(funkcja_porownujaca);
89
90      // wypisywanie listy wierzchołków po posortowaniu
91      cout << "Lista wierzcholkow po posortowaniu:" << endl;
92      for (wierzcholek* w : lista_wierzcholkow) {
93          cout << w->znak << ": " << w->czestosc << endl;
94      }
95
96      // tworzenie drzewa kodowego
97      while (lista_wierzcholkow.size() > 1) {
98          // zdejmujemy z listy 2 wierzchołki w1 i w2
99          wierzcholek* w1 = lista_wierzcholkow.back();
100         lista_wierzcholkow.pop_back();
101         wierzcholek* w2 = lista_wierzcholkow.back();
102         lista_wierzcholkow.pop_back();
103
104         // łączymy wierzchołki w1 oraz w2, tworząc wierzchołek w
105         wierzcholek* w3 = new wierzcholek();
106         w3->czestosc = w1->czestosc + w2->czestosc;
107         w3->lewy = w1;
108         w3->prawy = w2;
109
110         // umieszczamy wierzchołek w3 na końcu listy, a następni
111         lista_wierzcholkow.push_back(w3);
112         lista_wierzcholkow.sort(funkcja_porownujaca);
113     }
114
115     // gotowe drzewo kodowe
116     wierzcholek* drzewo_kodowe = lista_wierzcholkow.back();
117
118     // tworzenie książki kodowej
119     kod* ksiazka_kodowa = new kod[liczba_znakow];
120     utworz_ksiazke_kodowa(drzewo_kodowe, ksiazka_kodowa);
121
122     // wypisywanie książki kodowej
123     cout << "Ksiazka kodowa:" << endl;
124     for (int i = 0; i < liczba_znakow; ++i) {
125         cout << ksiazka_kodowa[i].znak << ": " << ksiazka_kodowa
126     }
127
```

```
    return 0;
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
Ksiazka kodowa:
o: 00
a: 01
i: 1000
```

```
D: 1001
t: 1010
k: 1011
d: 11
```

Każdy zakodowany znak ma teraz przypisany właściwy łańcuch znaków.

Teraz możemy przejść do kodowania wiadomości. W tym celu utworzymy dwie funkcje. Pierwsza z nich będzie odpowiadać za odnalezienie w książce kodowej danego znaku oraz zwrócenie jego słowa kodowego. Druga zaś dla każdego znaku w wiadomości wywoła pierwszą funkcję, a następnie połączy słowa kodowe oraz zwróci zakodowaną wiadomość.

```
string znajdz_slowo_kodowe(char znak, kod* ksiazka_kodowa, int dl
    for (int i = 0; i < dlugosc_ksiazki_kodowej; ++i) {
        if (ksiazka_kodowa[i].znak == znak) {
            return ksiazka_kodowa[i].slowo_kodowe;
        }
    }
}

string zakoduj_wiadomosc(string wiadomosc, kod* ksiazka_kodowa, i
    string zakodowana_wiadomosc = "";
    for (int i = 0; i < wiadomosc.length(); ++i) {
        zakodowana_wiadomosc += znajdz_slowo_kodowe(wiadomosc[i],
    }
    return zakodowana_wiadomosc;
}
```

W funkcji `main` możemy teraz dodać kod odpowiadający za wywołanie funkcji kodującej oraz wypisywanie zakodowanej wiadomości:

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
```

```
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

struct kod {
    char znak;
    string slowo_kodowe;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

void utworz_ksiazke_kodowa(wierzcholek* drzewo_kodowe, kod* ksia
    static int i = 0;
    if (drzewo_kodowe->lewy == nullptr && drzewo_kodowe->prawy =
        ksiazka_kodowa[i].znak = drzewo_kodowe->znak;
        ksiazka_kodowa[i].slowo_kodowe = dotychczasowy_kod;
        ++i;
    }
    else {
        utworz_ksiazke_kodowa(drzewo_kodowe->lewy, ksiazka_kodow
        utworz_ksiazke_kodowa(drzewo_kodowe->prawy, ksiazka_kodo
    }
}

```

```
string znajdz_slowo_kodowe(char znak, kod* ksiazka_kodowa, int d
    for (int i = 0; i < dlugosc_ksiazki_kodowej; ++i) {
        if (ksiazka_kodowa[i].znak == znak) {
            return ksiazka_kodowa[i].slowo_kodowe;
        }
    }
}

string zakoduj_wiadomosc(string wiadomosc, kod* ksiazka_kodowa,
    string zakodowana_wiadomosc = "";
    for (int i = 0; i < wiadomosc.length(); ++i) {
        zakodowana_wiadomosc += znajdz_slowo_kodowe(wiadomosc[i]
    }
    return zakodowana_wiadomosc;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
        cout << znak << ": " << tablica_zliczen[znak] << endl;
    }

    // tworzenie listy wierzchołków
    list<wierzcholek*> lista_wierzcholkow;
    int j = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            lista_wierzcholkow.push_back(new wierzcholek());
            lista_wierzcholkow.back()->znak = i;
            lista_wierzcholkow.back()->czestosc = tablica_zlicze
            ++j;
            if (j == liczba_znakow) {
                break;
            }
```

```
        }
    }

    // wypisywanie listy wierzchołków
    cout << "Lista wierzcholkow:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    // sortowanie listy wierzchołków
    lista_wierzcholkow.sort(funkcja_porownujaca);

    // wypisywanie listy wierzchołków po posortowaniu
    cout << "Lista wierzcholkow po posortowaniu:" << endl;
    for (wierzcholek* w : lista_wierzcholkow) {
        cout << w->znak << ": " << w->czestosc << endl;
    }

    // tworzenie drzewa kodowego
    while (lista_wierzcholkow.size() > 1) {
        // zdejmujemy z listy 2 wierzchołki w1 i w2
        wierzcholek* w1 = lista_wierzcholkow.back();
        lista_wierzcholkow.pop_back();
        wierzcholek* w2 = lista_wierzcholkow.back();
        lista_wierzcholkow.pop_back();

        // łączymy wierzchołki w1 oraz w2, tworząc wierzchołek w
        wierzcholek* w3 = new wierzcholek();
        w3->czestosc = w1->czestosc + w2->czestosc;
        w3->lewy = w1;
        w3->prawy = w2;

        // umieszczamy wierzchołek w3 na końcu listy, a następni
        lista_wierzcholkow.push_back(w3);
        lista_wierzcholkow.sort(funkcja_porownujaca);
    }

    // gotowe drzewo kodowe
    wierzcholek* drzewo_kodowe = lista_wierzcholkow.back();

    // tworzenie książki kodowej
    kod* ksiazka_kodowa = new kod[liczba_znakow];
```

```
136     utworz_ksiazke_kodowa(drzewo_kodowe, ksiazka_kodowa);
137
138     // wypisywanie książki kodowej
139     cout << "Ksiazka kodowa:" << endl;
140     for (int i = 0; i < liczba_znakow; ++i) {
141         cout << ksiazka_kodowa[i].znak << ": " << ksiazka_kodowa
142     }
143
144     // kodowanie wiadomości
145     string zakodowana_wiadomosc = zakoduj_wiadomosc(wiadomosc, k
146
147     // wypisywanie niezakodowanej i zakodowanej wiadomości
148     cout << "Niezakodowana wiadomosc:" << endl;
149     cout << wiadomosc << endl;
150     cout << "Zakodowana wiadomosc:" << endl;
151     cout << zakodowana_wiadomosc << endl;
152
153     return 0;
154 }
```

Wynik działania programu:

```
1  Zliczenia kazdego znaku:
2  D: 1
3  o: 3
4  d: 5
5  a: 3
6  d: 5
7  o: 3
8  d: 5
9  a: 3
10 d: 5
11 o: 3
12 d: 5
13 a: 3
14 t: 1
15 k: 1
16 i: 1
17 Lista wierzcholkow:
18 D: 1
19 a: 3
```

```
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
Ksiazka kodowa:
o: 00
a: 01
i: 1000
D: 1001
t: 1010
k: 1011
d: 11
Niezakodowana wiadomosc:
Dodadodadodatki
Zakodowana wiadomosc:
100100110111001101110011011010111000
```

Pozostało już tylko uzupełnić kod o zwalnianie pamięci dynamicznej. Ponieważ wszystkie wskaźniki na istniejące w pamięci struktury typu `wierzcholek` zawierają się w drzewie kodowym, napiszemy funkcję, która rekurencyjnie zwalnia dla nich wszystkich pamięć.

```
void zwolnij_pamiec(wierzcholek* drzewo_kodowe) {
    if (drzewo_kodowe->lewy != nullptr && drzewo_kodowe->prawy !=
        zwolnij_pamiec(drzewo_kodowe->lewy);
        zwolnij_pamiec(drzewo_kodowe->prawy);
    }
    delete drzewo_kodowe;
}
```

Wewnątrz funkcji `main` zawrzemy wywołanie napisanej funkcji, a także usunięcie pamięci przeznaczonej na książkę kodową, która jest tablicą dynamiczną. Ostateczny kod

prezentuje się następująco:

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

struct kod {
    char znak;
    string slowo_kodowe;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

void utworz_ksiazke_kodowa(wierzcholek* drzewo_kodowe, kod* ksia
```

```
    static int i = 0;
    if (drzewo_kodowe->lewy == nullptr && drzewo_kodowe->prawy =
        ksiazka_kodowa[i].znak = drzewo_kodowe->znak;
        ksiazka_kodowa[i].slowo_kodowe = dotychczasowy_kod;
        ++i;
    }
    else {
        utworz_ksiazke_kodowa(drzewo_kodowe->lewy, ksiazka_kodow
        utworz_ksiazke_kodowa(drzewo_kodowe->prawy, ksiazka_kodo
    }
}

string znajdz_slowo_kodowe(char znak, kod* ksiazka_kodowa, int d
    for (int i = 0; i < dlugosc_ksiazki_kodowej; ++i) {
        if (ksiazka_kodowa[i].znak == znak) {
            return ksiazka_kodowa[i].slowo_kodowe;
        }
    }
}

string zakoduj_wiadomosc(string wiadomosc, kod* ksiazka_kodowa,
    string zakodowana_wiadomosc = "";
    for (int i = 0; i < wiadomosc.length(); ++i) {
        zakodowana_wiadomosc += znajdz_slowo_kodowe(wiadomosc[i]
    }
    return zakodowana_wiadomosc;
}

void zwolnij_pamiec(wierzcholek* drzewo_kodowe) {
    if (drzewo_kodowe->lewy != nullptr && drzewo_kodowe->prawy !
        zwolnij_pamiec(drzewo_kodowe->lewy);
        zwolnij_pamiec(drzewo_kodowe->prawy);
    }
    delete drzewo_kodowe;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);
```

```
82
83      // wypisywanie zliczeń dla każdego znaku z wiadomości
84      cout << "Zliczenia kazdego znaku:" << endl;
85      for (int i = 0; i < wiadomosc.length(); ++i) {
86          char znak = wiadomosc[i];
87          cout << znak << ": " << tablica_zliczen[znak] << endl;
88      }
89
90      // tworzenie listy wierzchołków
91      list<wierzcholek*> lista_wierzcholkow;
92      int j = 0;
93      for (int i = 0; i <= 255; ++i) {
94          if (tablica_zliczen[i] > 0) {
95              lista_wierzcholkow.push_back(new wierzcholek());
96              lista_wierzcholkow.back()->znak = i;
97              lista_wierzcholkow.back()->czestosc = tablica_zlicze
98              ++j;
99              if (j == liczba_znakow) {
100                 break;
101             }
102         }
103     }
104
105     // wypisywanie listy wierzchołków
106     cout << "Lista wierzcholkow:" << endl;
107     for (wierzcholek* w : lista_wierzcholkow) {
108         cout << w->znak << ": " << w->czestosc << endl;
109     }
110
111     // sortowanie listy wierzchołków
112     lista_wierzcholkow.sort(funkcja_porownujaca);
113
114     // wypisywanie listy wierzchołków po posortowaniu
115     cout << "Lista wierzcholkow po posortowaniu:" << endl;
116     for (wierzcholek* w : lista_wierzcholkow) {
117         cout << w->znak << ": " << w->czestosc << endl;
118     }
119
120     // tworzenie drzewa kodowego
121     while (lista_wierzcholkow.size() > 1) {
122         // zdejmujemy z listy 2 wierzchołki w1 i w2
123         wierzcholek* w1 = lista_wierzcholkow.back();
```

```
124         lista_wierzcholkow.pop_back();
125         wierzcholek* w2 = lista_wierzcholkow.back();
126         lista_wierzcholkow.pop_back();
127 
128         // łączymy wierzchołki w1 oraz w2, tworząc wierzchołek w
129         wierzcholek* w3 = new wierzcholek();
130         w3->czestosc = w1->czestosc + w2->czestosc;
131         w3->lewy = w1;
132         w3->prawy = w2;
133 
134         // umieszczamy wierzchołek w3 na końcu listy, a następni
135         lista_wierzcholkow.push_back(w3);
136         lista_wierzcholkow.sort(funkcja_porownujaca);
137     }
138 
139     // gotowe drzewo kodowe
140     wierzcholek* drzewo_kodowe = lista_wierzcholkow.back();
141 
142     // tworzenie książki kodowej
143     kod* ksiazka_kodowa = new kod[liczba_znakow];
144     utworz_ksiazke_kodowa(drzewo_kodowe, ksiazka_kodowa);
145 
146     // wypisywanie książki kodowej
147     cout << "Ksiazka kodowa:" << endl;
148     for (int i = 0; i < liczba_znakow; ++i) {
149         cout << ksiazka_kodowa[i].znak << ": " << ksiazka_kodowa
150     }
151 
152     // kodowanie wiadomości
153     string zakodowana_wiadomosc = zakoduj_wiadomosc(wiadomosc, k
154 
155     // wypisywanie niezakodowanej i zakodowanej wiadomości
156     cout << "Niezakodowana wiadomosc:" << endl;
157     cout << wiadomosc << endl;
158     cout << "Zakodowana wiadomosc:" << endl;
159     cout << zakodowana_wiadomosc << endl;
160 
161     // zwolnienie pamięci
162     zwolnij_pamiec(drzewo_kodowe);
163     delete[] ksiazka_kodowa;
164 
165     return 0;
```

```
}
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
Ksiazka kodowa:
o: 00
a: 01
i: 1000
D: 1001
```

```
t: 1010
k: 1011
d: 11
Niezakodowana wiadomosc:
Dodadodadodatki
Zakodowana wiadomosc:
10010011011100110111001101101010111000
```

> **Problem 5**
>
> Zaimplementujmy w języku C++ program, który odkoduje wcześniej zakodowany za pomocą algorytmu Huffmana ciąg znaków.

Zakodowana wiadomość może zostać jednoznacznie zdekodowana pod warunkiem, że znana jest książka kodowa lub drzewo kodowe. Napiszmy funkcję dekodującą wiadomość za pomocą wcześniej utworzonego drzewa kodowego. Z uwagi na trudność w zawarciu drzewa kodowego jako dane programu, przedstawiona implementacja będzie polegała na odkodowaniu wcześniej zakodowanej wiadomości według już zawartego w programie drzewa kodowego.

Specyfikacja problemu:

Dane:

- `zakodowana_wiadomosc` – łańcuch znaków (zer lub jedynek) do odkodowania
- `drzewo_kodowe` – drzewo Huffmana utworzone w poprzednim programie

Wynik:

- `odkodowana_wiadomosc` – łańcuch znaków reprezentujący odkodowany łańcuch znaków `zakodowana_wiadomosc`

Odkodowywanie wiadomości polega na poruszaniu się po drzewie kodowym według podanych w zakodowanej wiadomości zer lub jedynek. Jeśli aktualny znak to 0, schodzimy po drzewie w lewą stronę, natomiast w przypadku znaku 1 – w prawą. Gdy dojdziemy do wierzchołka, który jest liściem, możemy dodać do odkodowanej wiadomości znak zawarty w danym wierzchołku, a następnie rozpocząć poruszanie się po drzewie od początku. Kończymy w momencie, kiedy wyczerpiemy znaki z zakodowanej wiadomości. Proces ten przedstawmy w formie funkcji.

```
string odkodujWiadomosc(string zakodowanaWiadomosc, wierzcholek*
    string odkodowanaWiadomosc = "";
```

```
    wierzcholek* aktualny_wierzcholek = drzewo_kodowe;
    for (int i = 0; i < zakodowanaWiadomosc.length(); ++i) {
        if (aktualny_wierzcholek->lewy == nullptr && aktualny_wie
            odkodowanaWiadomosc += aktualny_wierzcholek->znak;
            aktualny_wierzcholek = drzewo_kodowe;
        }
        if (zakodowanaWiadomosc[i] == '0') {
            aktualny_wierzcholek = aktualny_wierzcholek->lewy;
        }
        else {
            aktualny_wierzcholek = aktualny_wierzcholek->prawy;
        }
    }
    odkodowanaWiadomosc += aktualny_wierzcholek->znak;
    return odkodowanaWiadomosc;
}
```

Cały działający kod – stworzony w oparciu o używanie ciągów znaków zamiast ciągów bitów do kodowania – prezentuje się następująco:

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct wierzcholek {
    char znak;
    int czestosc;
    wierzcholek* lewy = nullptr;
    wierzcholek* prawy = nullptr;
};

struct kod {
    char znak;
    string slowo_kodowe;
};

int zlicz_znaki(string wiadomosc, int* tablica_zliczen) {
    for (int i = 0; i <= 255; ++i) {
        tablica_zliczen[i] = 0;
```

```
    }
    for (int i = 0; i < wiadomosc.length(); ++i) {
        ++tablica_zliczen[wiadomosc[i]];
    }
    int liczba = 0;
    for (int i = 0; i <= 255; ++i) {
        if (tablica_zliczen[i] > 0) {
            ++liczba;
        }
    }
    return liczba;
}

bool funkcja_porownujaca(wierzcholek* w1, wierzcholek* w2) {
    return w1->czestosc > w2->czestosc;
}

void utworz_ksiazke_kodowa(wierzcholek* drzewo_kodowe, kod* ksia
    static int i = 0;
    if (drzewo_kodowe->lewy == nullptr && drzewo_kodowe->prawy =
        ksiazka_kodowa[i].znak = drzewo_kodowe->znak;
        ksiazka_kodowa[i].slowo_kodowe = dotychczasowy_kod;
        ++i;
    }
    else {
        utworz_ksiazke_kodowa(drzewo_kodowe->lewy, ksiazka_kodow
        utworz_ksiazke_kodowa(drzewo_kodowe->prawy, ksiazka_kodo
    }
}

string znajdz_slowo_kodowe(char znak, kod* ksiazka_kodowa, int d
    for (int i = 0; i < dlugosc_ksiazki_kodowej; ++i) {
        if (ksiazka_kodowa[i].znak == znak) {
            return ksiazka_kodowa[i].slowo_kodowe;
        }
    }
}

string zakoduj_wiadomosc(string wiadomosc, kod* ksiazka_kodowa,
    string zakodowana_wiadomosc = "";
    for (int i = 0; i < wiadomosc.length(); ++i) {
        zakodowana_wiadomosc += znajdz_slowo_kodowe(wiadomosc[i]
```

```
    }
    return zakodowana_wiadomosc;
}

void zwolnij_pamiec(wierzcholek* drzewo_kodowe) {
    if (drzewo_kodowe->lewy != nullptr && drzewo_kodowe->prawy !
        zwolnij_pamiec(drzewo_kodowe->lewy);
        zwolnij_pamiec(drzewo_kodowe->prawy);
    }
    delete drzewo_kodowe;
}

string odkodujWiadomosc(string zakodowanaWiadomosc, wierzcholek*
    string odkodowanaWiadomosc = "";
    wierzcholek* aktualny_wierzcholek = drzewo_kodowe;
    for (int i = 0; i < zakodowanaWiadomosc.length(); ++i) {
        if (aktualny_wierzcholek->lewy == nullptr && aktualny_wi
            odkodowanaWiadomosc += aktualny_wierzcholek->znak;
            aktualny_wierzcholek = drzewo_kodowe;
        }
        if (zakodowanaWiadomosc[i] == '0') {
            aktualny_wierzcholek = aktualny_wierzcholek->lewy;
        }
        else {
            aktualny_wierzcholek = aktualny_wierzcholek->prawy;
        }
    }
    odkodowanaWiadomosc += aktualny_wierzcholek->znak;
    return odkodowanaWiadomosc;
}

int main() {
    int tablica_zliczen[256];
    string wiadomosc = "Dodadododadodatki";

    // zliczanie znaków w wiadomości
    int liczba_znakow = zlicz_znaki(wiadomosc, tablica_zliczen);

    // wypisywanie zliczeń dla każdego znaku z wiadomości
    cout << "Zliczenia kazdego znaku:" << endl;
    for (int i = 0; i < wiadomosc.length(); ++i) {
        char znak = wiadomosc[i];
```

```
106         cout << znak << ": " << tablica_zliczen[znak] << endl;
107     }
108 
109     // tworzenie listy wierzchołków
110     list<wierzcholek*> lista_wierzcholkow;
111     int j = 0;
112     for (int i = 0; i <= 255; ++i) {
113         if (tablica_zliczen[i] > 0) {
114             lista_wierzcholkow.push_back(new wierzcholek());
115             lista_wierzcholkow.back()->znak = i;
116             lista_wierzcholkow.back()->czestosc = tablica_zlicze
117             ++j;
118             if (j == liczba_znakow) {
119                 break;
120             }
121         }
122     }
123 
124     // wypisywanie listy wierzchołków
125     cout << "Lista wierzcholkow:" << endl;
126     for (wierzcholek* w : lista_wierzcholkow) {
127         cout << w->znak << ": " << w->czestosc << endl;
128     }
129 
130     // sortowanie listy wierzchołków
131     lista_wierzcholkow.sort(funkcja_porownujaca);
132 
133     // wypisywanie listy wierzchołków po posortowaniu
134     cout << "Lista wierzcholkow po posortowaniu:" << endl;
135     for (wierzcholek* w : lista_wierzcholkow) {
136         cout << w->znak << ": " << w->czestosc << endl;
137     }
138 
139     // tworzenie drzewa kodowego
140     while (lista_wierzcholkow.size() > 1) {
141         // zdejmujemy z listy 2 wierzchołki w1 i w2
142         wierzcholek* w1 = lista_wierzcholkow.back();
143         lista_wierzcholkow.pop_back();
144         wierzcholek* w2 = lista_wierzcholkow.back();
145         lista_wierzcholkow.pop_back();
146 
147         // łączymy wierzchołki w1 oraz w2, tworząc wierzchołek w
```

```
148         wierzcholek* w3 = new wierzcholek();
149         w3->czestosc = w1->czestosc + w2->czestosc;
150         w3->lewy = w1;
151         w3->prawy = w2;
152 
153         // umieszczamy wierzchołek w3 na końcu listy, a następni
154         lista_wierzcholkow.push_back(w3);
155         lista_wierzcholkow.sort(funkcja_porownujaca);
156     }
157 
158     // gotowe drzewo kodowe
159     wierzcholek* drzewo_kodowe = lista_wierzcholkow.back();
160 
161     // tworzenie książki kodowej
162     kod* ksiazka_kodowa = new kod[liczba_znakow];
163     utworz_ksiazke_kodowa(drzewo_kodowe, ksiazka_kodowa);
164 
165     // wypisywanie książki kodowej
166     cout << "Ksiazka kodowa:" << endl;
167     for (int i = 0; i < liczba_znakow; ++i) {
168         cout << ksiazka_kodowa[i].znak << ": " << ksiazka_kodowa
169     }
170 
171     // kodowanie wiadomości
172     string zakodowana_wiadomosc = zakoduj_wiadomosc(wiadomosc, k
173 
174     // wypisywanie niezakodowanej i zakodowanej wiadomości
175     cout << "Niezakodowana wiadomosc:" << endl;
176     cout << wiadomosc << endl;
177     cout << "Zakodowana wiadomosc:" << endl;
178     cout << zakodowana_wiadomosc << endl;
179 
180     // odkodowanie zakodowanej wiadomości
181     string odkodowana_wiadomosc = odkodujWiadomosc(zakodowana_wi
182 
183     // wypisanie odkodowanej wiadomości
184     cout << "Odkodowana wiadomosc:" << endl;
185     cout << odkodowana_wiadomosc << endl;
186 
187     // zwolnienie pamięci
188     zwolnij_pamiec(drzewo_kodowe);
189     delete[] ksiazka_kodowa;
```

```
190
191     return 0;
192 }
```

Wynik działania programu:

```
Zliczenia kazdego znaku:
D: 1
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
d: 5
o: 3
d: 5
a: 3
t: 1
k: 1
i: 1
Lista wierzcholkow:
D: 1
a: 3
d: 5
i: 1
k: 1
o: 3
t: 1
Lista wierzcholkow po posortowaniu:
d: 5
a: 3
o: 3
D: 1
i: 1
k: 1
t: 1
Ksiazka kodowa:
o: 00
a: 01
```

```
i: 1000
D: 1001
t: 1010
k: 1011
d: 11
Niezakodowana wiadomosc:
Dodadodadodatki
Zakodowana wiadomosc:
10010011011100110111001101101010111000
Odkodowana wiadomosc:
Dodadodadodatki
```

> **Już wiesz**
>
> Podsumujmy najważniejsze informacje:
>
> - algorytm Huffmana generuje kod zero-jedynkowy;
> - kod każdego znaku nie jest początkowym fragmentem kodu innego znaku;
> - generowany kod jest tzw. kodem prefiksowym (żaden kod nie jest prefiksem innej litery), który pozwala na jednoznaczne dekodowanie zapisanego znaku;
> - kod jest tworzony w taki sposób, aby średnia długość kodu znaku była najkrótsza.

Słownik

> **bezstratna kompresja danych**
>
> kompresja, która polega na tym, że informacja po skompresowaniu jest dekompresowalna do takiej samej postaci, z której początkowo wyszliśmy
> **drzewo binarne**
>
> struktura drzewiasta, w której każdy wierzchołek ma co najwyżej dwóch potomków; drzewa w informatyce są jednym z rodzajów struktur danych
> **książka kodowa**
>
> narzędzie ułatwiające kodowanie i wprowadzanie danych do komputera, zawiera instrukcje dotyczące postępowania przy kodowaniu wszystkich odpowiedzi na pytania zamknięte i otwarte, zawiera zbiór kodów z wyjaśnieniem, które odpowiedzi mają być przyporządkowane danemu kodowi
> **słowo kodowe**
>
> słowo, wyrażenie, znak lub ciąg bitów odpowiadające słowu, które chcemy zakodować; we wczesnych etapach rozwoju kryptografii słowa kodowe były układane w ramach książek kodowych i wykorzystywane do szyfrowania wiadomości
> **stopień kompresji**

stosunek wielkości danych przed przeprowadzaniem kompresji do wielkości danych po kompresji

wierzchołek drzewa binarnego

element drzewa binarnego; każdy wierzchołek w drzewie binarnym ma co najwyżej dwóch potomków; wierzchołek może być:

- liściem – jeśli nie ma żadnych potomków,
- korzeniem – jeśli nie ma żadnego rodzica (jest pierwszym elementem drzewa)

wskaźnik pusty

specjalna wartość liczbowa przeznaczona na celowe reprezentowanie wskaźnika, który nie wskazuje na żaden element w pamięci

Infografika

> **Polecenie 1**
>
> Zapoznaj się z infografiką przedstawiającą przebieg działania algorytmu Huffmana dla przykładowego ciągu znaków. Zdekoduj podaną wiadomość.
>
> Nad infografiką znajdują się przyciski odnoszące się do kolejnych kroków.

Algorytm Huffmana dla przykładowego ciągu znaków

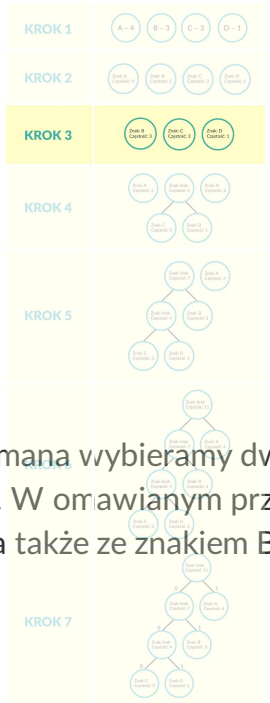
Krok 1

Zliczamy wystąpienia każdego znaku dla ciągu AAABBCABCDC.

Krok 2

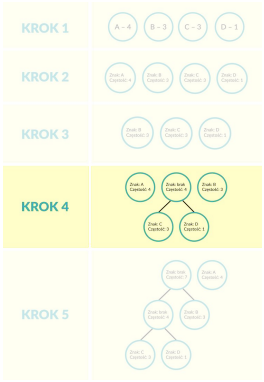
Tworzymy cztery wierzchołki – każdy zawierający znak i częstość.

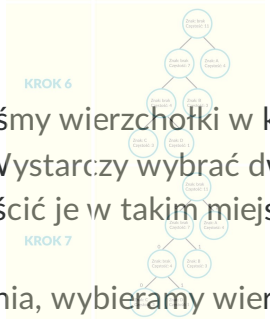
Krok 3



Zgodnie z algorytmem Huffmana wybieramy dwa wierzchołki o najmniejszych częstościach. W omawianym przypadku będzie to wierzchołek ze znakiem D, a także ze znakiem B lub C.

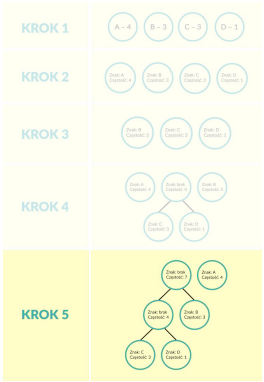
Krok 4

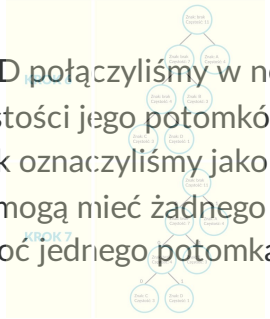




Nie bez przyczyny umieściliśmy wierzchołki w kolejności od największej częstości do najmniejszej. Wystarczy wybrać dwa wierzchołki znajdujące się na końcu, a potem umieścić je w takim miejscu listy, by częstości wciąż były malejące.
Idąc tym tokiem rozumowania, wybieramy wierzchołki z C i D.

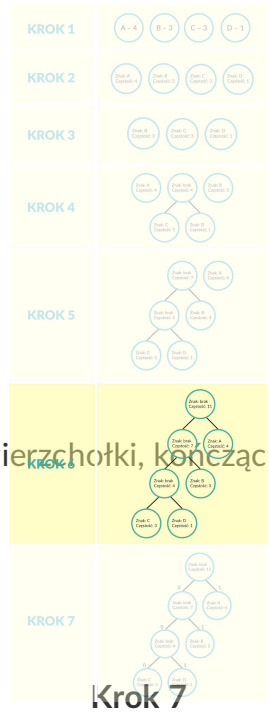
Krok 5





Wierzchołki ze znakiem C i D połączyliśmy w nowy wierzchołek, którego częstość stanowi sumę częstości jego potomków.
W nowym wierzchołku znak oznaczyliśmy jako „brak” (zauważmy, że wierzchołki ze znakami nie mogą mieć żadnego potomka, zatem każdy wierzchołek posiadający choć jednego potomka nie będzie miał swojego znaku).
Kontynuując algorytm, wybieramy kolejne dwa wierzchołki z najmniejszymi częstościami znajdującymi się w liście. Warto zwrócić uwagę, że wierzchołek o większej częstości umieszczamy z lewej strony (można to zrobić również z prawej, jednak należy postępować w sposób konsekwentny, tzn. umieszczać wierzchołki z większą częstością po konkretnej stronie).

Krok 6



Krok 7

Wybieramy dwa ostatnie wierzchołki, kończąc tym samym algorytm.

Na podstawie tego drzewa tworzymy słownik kodowy. Każdą odnogę w lewo oznaczymy jako 0, a każdą odnogę w prawo jako 1.
Tym samym otrzymamy graf.
Książka kodowa:
A: 1
B: 01
C: 000
D: 001

Spróbuj zdekodować następującą wiadomość: 01000001011, odczytując kolejne znaki prosto z grafu. Zacznij od korzenia grafu z częstością 11. Następnie, jeśli napotkasz 0, przejdź do wierzchołka z lewej, a jeśli 1 – do wierzchołka z prawej.
Jeżeli wierzchołek jest korzeniem, odczytaj z niego literę i wróć do korzenia. Jeżeli nim nie jest, odczytaj kolejną liczbę i kontynuuj zgodnie ze sposobem opisanym w poprzednich krokach.

Ćwiczenie 1

Zaimplementuj w języku C++ algorytm Huffmana służący do kodowania ciągu znaków. Podczas łączenia wierzchołków upewnij się, że prawy potomek będzie wierzchołkiem z mniejszą częstością niż lewy. Jeśli wierzchołki mają tę samą częstość, za mniejszy uznajemy ten o znaku z niższą wartością w kodowaniu ASCII. Nie musisz uwzględniać sytuacji, w której porównywane wierzchołki mają takie same częstości, a nie mają znaku (nie są liśćmi) – dane testowe ją wykluczają. Podczas poruszania się po drzewie kodowym zakładamy, że przejściu na lewo odpowiada cyfra 1, a przejściu na prawo cyfra 0. Do napisania programu możesz wykorzystać zmodyfikowany algorytm przedstawiony w sekcji „Przeczytaj". Działanie programu przetestuj dla podanej wiadomości:

```
wiadomosc = "AAAADBABCABCDAC"
```

Specyfikacja problemu:

Dane:

- `wiadomosc` – łańcuch znaków do zakodowania

Wynik:

- `zakodowanaWiadomosc` – łańcuch znaków (zer i jedynek) reprezentujący zakodowany do postaci binarnej zgodnie z kodowaniem Huffmana łańcuch znaków `wiadomosc`

Twoje zadania

1. Zaimplementuj w języku C++ algorytm Huffmana kodujący ciąg znaków. Przetestuj jego działanie dla łańcucha znaków `wiadomosc`.

```cpp
#include <iostream>

using namespace std;

int main() {
    string wiadomosc = "AAABBCABCDC";
    //tu dodaj kod

```

```
    return 0;
}
```

```
1
```

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Napisz program, który dla ciągu znaków `wiadomosc` zliczy wystąpienia każdego znaku z tablicy ASCII, a następnie wyświetli, ile razy dany znak pojawił się w tym ciągu znaków. Program powinien pomijać te znaki, które nie pojawiły się ani razu. Swój program przetestuj dla podanego ciągu znaków:

```
wiadomosc = "Ardua prima via est"
```

Specyfikacja problemu:

Dane:

- `wiadomosc` – łańcuch znaków, w którym należy zliczyć wystąpienia znaków

Wynik:

- liczba odpowiadająca każdemu znakowi z tablicy ASCII występującemu w łańcuchu znaków `wiadomosc` oraz liczba wskazująca, ile razy dany znak pojawił się w łańcuchu; informacja o każdym znaku powinna znaleźć się w oddzielnej linijce, a pomiędzy obiema liczbami powinien znaleźć się dwukropek

Przykładowe wyjście:

```
32:3
65:1
97:3
100:1
101:1
105:2
109:1
112:1
114:2
115:1
116:1
117:1
118:1
```

Twoje zadania

1. Program zlicza i wypisuje litery oraz odpowiadające im częstości dla zadanego ciągu znaków.

Ćwiczenie 2

Napisz program, który zakoduje ciąg znaków `wiadomosc`, a następnie wypisze zakodowany łańcuch znaków na standardowe wyjście. Ważne, by wykorzystana została książka kodowa. Przetestuj swój program dla podanego ciągu znaków:

```
wiadomosc = "Jestem przykladowym tekstem i nalezy mnie zakodowa
```

Specyfikacja problemu:

Dane:

- `wiadomosc` – łańcuch znaków
- `ksiazka_kodowa` – tablica zawierająca struktury `kod` z kodowanym znakiem i odpowiadającym mu słowem kodowym
- `DLUGOSC_KSIAZKI_KODOWEJ` – stała całkowita zawarta w programie; długość tablicy `ksiazka_kodowa`

Wynik:

- `zakodowana_wiadomosc` – zakodowany łańcuch znaków

Twoje zadania

1. Program koduje wiadomość.

Ćwiczenie 3

Napisz program, który korzystając z podanej książki kodowej, odkoduje ciąg znaków `wiadomosc_zakodowana` zakodowany za pomocą algorytmu Huffmana. W celu zdekodowania wiadomości możesz utworzyć drzewo kodowe na podstawie książki lub wykorzystać fakt, że kodowanie Huffmana jest kodowaniem prefiksowym. Zdekodowaną wiadomość wypisz na standardowe wyjście. Swój program przetestuj dla ciągu znaków:

```
wiadomosc_zakodowana = "11101010010010110110011111010011010010000
```

Specyfikacja problemu:

Dane:

- `wiadomosc_zakodowana` – łańcuch znaków (zer i jedynek) do odkodowania
- `ksiazka_kodowa` – tablica zawierająca struktury `kod` z kodowanym znakiem i odpowiadającym mu słowem kodowym
- `DLUGOSC_KSIAZKI_KODOWEJ` – stała całkowita zawarta w programie; długość tablicy `ksiazka_kodowa`

Wynik:

- `wiadomosc_odkodowana` – odkodowany łańcuch znaków

Twoje zadania

1. Program dekoduje wiadomość.

Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Algorytm Huffmana w języku C++

Grupa docelowa:

Liceum ogólnokształcące i technikum, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

III. Posługiwanie się komputerem, urządzeniami cyfrowymi i sieciami komputerowymi, w tym: znajomość zasad działania urządzeń cyfrowych i sieci komputerowych oraz wykonywania obliczeń i programów.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) do realizacji rozwiązania problemu dobiera odpowiednią metodę lub technikę algorytmiczną i struktury danych;

3) objaśnia dobrany algorytm, uzasadnia poprawność rozwiązania na wybranych przykładach danych i ocenia jego efektywność;

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres podstawowy. Uczeń:

1) projektuje i programuje rozwiązania problemów z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje warunkowe, instrukcje iteracyjne, funkcje z parametrami i bez parametrów, testuje poprawność programów dla różnych danych; w szczególności programuje algorytmy z punktu I.2);

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

III. Posługiwanie się komputerem, urządzeniami cyfrowymi i sieciami komputerowymi.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) dokonuje kompresji informacji, objaśnia różnice między kompresją stratną i bezstratną tekstów, obrazów, dźwięków, filmów;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Prześledzisz zrealizowany w języku C++ algorytm Huffmana.
- Zaimplementujesz program w języku C++ kodujący ciąg znaków za pomocą algorytmu Huffmana.
- Zakodujesz ciągi znaków w celu zmniejszenia zajmowanej przez nie pamięci.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka C++, w tym kompilator GCC/G++ 4.5 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Algorytm Huffmana w języku C++". Uczniowie mają zapoznać się z treściami w sekcji „Przeczytaj" i wykonać obliczenia na podstawie dołączonych danych.
2. Chętny lub wybrany uczeń przygotowuje rozwiązanie polecenia nr 1 z sekcji „Infografika". Będzie pełnił rolę eksperta podczas zajęć.

Faza wstępna:

1. Nauczyciel wprowadza uczniów szczegółowo w temat lekcji i jej cele. Może posłużyć się wyświetloną na tablicy zawartością sekcji „Wprowadzenie".
2. **Rozpoznanie wiedzy uczniów.** Nauczyciel prosi wybranego ucznia lub uczniów o przedstawienie sytuacji problemowej związanej z tematem lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeżeli jest ono niewystarczające, prosi wybraną osobę o przedstawienie najważniejszych informacji z sekcji „Przeczytaj". Na forum klasy uczniowie analizują problemy 1-5. Następnie pracując indywidualnie, powtarzają i testują zaprezentowaną implementację algorytmu na swoich komputerach.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Infografika". Osoba, która pełni funkcję eksperta, omawia treść infografiki. W kolejnym kroku uczniowie w parach piszą program wykorzystujący algorytm Huffmana, a następnie chętne osoby odczytują słowa kodowe.
3. **Ćwiczenie umiejętności.** Uczniowie wykonują indywidualnie ćwiczenie nr 1 z sekcji „Sprawdź się", a następnie porównują swoje odpowiedzi z kolegą lub koleżanką.

4. Uczniowie, pracując w parach, wykonują ćwiczenie nr 2 z sekcji „Sprawdź się”, a następnie dzielą się swoimi wynikami przez porównanie napisanego kodu z inną grupą, która również zakończyła zadanie.

Faza podsumowująca:

1. Nauczyciel wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie”. W kontekście ich realizacji podsumowuje przebieg zajęć, a także wskazuje mocne i słabe strony pracy uczniów.
2. Na koniec zajęć z programowania w C++ nauczyciel prosi uczniów o rozwinięcie zdania: „Na dzisiejszych zajęciach nauczyłam/łem się jak…”.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 3 z sekcji „Sprawdź się”.

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka C++.
- Oficjalna dokumentacja techniczna dla kompilatora GCC/G++ 4.5 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj” można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.