Algorytmy 2
Laboratorium: tablica mieszająca

Przemysław Klęsk

21 listopada 2019

1 Cel

Celem zadania jest wykonanie implementacji struktury danych nazywanej *tablicą mieszającą* lub *tablicą z haszowaniem* (ang. *hash table*). Struktura ta stanowi pamięć asocjacyjną, co oznacza, że przechowuje ona pary *(klucz, wartość)* i w niektórych językach programowania bywa w związku z tym nazywana bezpośrednio słownikiem (ang. *dictionary*) np. w Pythonie i C#.

Ogólna zasada działania tablic mieszających powoduje, że są one szybkimi strukturami danych kosztem pamięci. Złożoność obliczeniowa operacji wyszukiwania w tablicy mieszającej jest stała — $O(1)$ — czyli nie zależy od liczby przechowywanych elementów (par). Dodawanie nowych elementów do tablicy mieszającej jest także operacją stałoczasową, przy czym ściśle rzecz biorąc tylko w sensie zamortyzowanym, co wynika z własności tablicy dynamicznej będącej strukturą podstawową, na której buduje się tablicę mieszającą.

Głównym zabiegiem pozwalającym na uzyskanie powyższej szybkości działania tablicy mieszającej jest używanie (podczas wyszukiwania i dodawania) pewnej ustalonej funkcji, która odwzorowuje klucze w indeksy całkowite — to jest tzw. *funkcji mieszającej* (ang. *hash function*). Dzięki niej, możliwe jest bezpośrednie sięganie pod opowiedni indeks (adres) w tablicy, bez konieczności przechodzenia po wielu indeksach. A zatem koszty powyższych operacji sprowadzają się w dużej mierze do samego kosztu obliczenia wartości funkcji mieszającej. W powszechnym użyciu jest wiele różnych funkcji mieszających. Zwyczajowo są one pewnymi sumami zbudowanymi z wykorzystaniem liczb pierwszych i dzielenia modulo. Dobra funkcja mieszająca powinna dobrze rozpraszać — czyli generować rozkład wynikowych wartości bliski równomiernemu. W praktyce nieuniknionym problemem występującym w tablicach mieszających są *kolizje*. Są to sytuacje, w których dwa lub więcej kluczy zostaje odwzorowanych w ten sam indeks całkowity. Nieuniknioność tego problemu wynika po części z zasady szufladkowej Dirichleta (gdy moc zbioru kluczy jest większa niż moc zbioru indeksów), a po części z paradoksu dnia urodzin (w przeciwnym przypadku). Podstawową techniką radzenia sobie z kolizjami jest *łańcuchowanie* (ang. *chaining*), czyli możliwość przechowywania wielu par (klucz, wartość) dla każdego indeksu tablicy. Oczywiście ważnym jest, aby takie kolekcja par (klucz, wartość) były odpowiednio krótkie.

Na potrzeby niniejszego zadania laboratoryjnego, łańcuchowanie zostanie zrealizowane z wykorzystaniem list z dowiązaniami. A zatem wykonana tablica mieszająca będzie w sensie programistycznym tablicą

list. A dopiero elementami tychże list będą pary (klucz, wartość). Tradycyjnie, dodatkowym celem będzie wykonanie odpowiednich pomiarów czasowych w celu sprawdzenia teoretycznej złożoności obliczeniowej.

2 Instrukcje, wskazówki, podpowiedzi

1. W ogólności w tablicach mieszających praktykuje się dopuszczanie dowolnych typów zarówno dla zbioru kluczy jak i wartości. A zatem, mając na uwadze mechanizm szablonów (`template`) języka C++, można by myśleć o stworzeniu klasy (lub struktury) parametryzowanej pewnymi dwoma typami np.: `hash_table<K, V>`, gdzie `K` oznaczałby typ dla zbioru kluczy, a `V` typ dla zbioru wartości. Dla uproszczenia realizacji tego zadania proszę przyjąć, że typ dla zbioru kluczy jest ustalony i jest to typ napisowy (np. `std::string`). A zatem jedynym parametrem szablonowym będzie typ dla zbioru wartości.

2. Wygodnym zabiegiem może być stworzenie prostej pomocniczej struktury reprezentującej pojedynczy wpis w tablicę mieszającą, czyli parę (klucz, wartość).

3. Implementacja może wykorzystywać wcześniej wykonane własne kontenery listy oraz tablicy dynamicznej (np. poprzez dołączenie odpowiednich plików nagłówkowych `.h`). Pozwoli to zredukować czas pracy.

4. Tablica mieszająca powinna obserwować swój aktualny stopień napełnienia. Można wprowadzić pewną stałą (np. 0.75) reprezentującą graniczne napełnienie, po przekroczeniu którego tablica powinna się rozszerzyć i przemieszać (tzn. zrealizować ponownie wszystkie wpisy, obliczając dla każdego z nich nową wartość funkcji mieszającej) — zabieg analogiczny do rozszerzenia tablicy dynamicznej, powodujący chwilowy koszt liniowy, ale redukujący się do kosztu stałego w amortyzacji.

5. Na potrzeby tego zadania **interfejs tablicy mieszającej** powinien udostępniać następujące funkcje / metody:

   (a) dodanie nowego elementu (argumenty: klucz oraz wartość — czyli inaczej dane skojarzone z kluczem),

   (b) wyszukanie elementu (argument: klucz; wynik: wskaźnik na znaleziony element tj. parę (klucz, wartość) lub `NULL` w przypadku niepowodzenia),

   (c) usunięcie elementu (argument: klucz; wynik: flaga logiczna sygnalizująca powodzenie lub niepowodzenie),

   (d) czyszczenie tablicy mieszającej tj. usunięcie wszystkich elementów,

   (e) zwrócenie napisowej reprezentacji tablicy mieszającej — np. funkcja / metoda `to_string(...)` (wynikowy napis powinien przedstawiać krótkie niepuste listy par (klucz, wartość) występujące pod poszczególnymi indeksami tablicy),

   (f) obliczenie wartości funkcji mieszającej (argument: klucz; wynik: typu `int` — więcej szczegółów dalej),

(g) rozszerzenie i przemieszanie tablicy — tzw. operacja *rehash* (ta funkcja powinna być wywoływana z wnętrza funkcji dodającej nowy element).

6. W programie można wykorzystać ogólne wskazówki z poprzednich zadań dotyczące:
   - dynamicznego zarządania pamięcią (`new`, `delete`) — w szczególności przemyślenia miejsc odpowiedzialnych za uwalnianie pamięci danych,
   - wydzielenia implementacji interfejsu tablicy mieszającej do odrębnego pliku `.h`,
   - pracy z napisami (użycie typu `std::string`),
   - pomiaru czasu (funkcja `clock()` po dołączeniu `#include <time.h>`),
   - użycia wskaźników na funkcje,
   - generowania losowych danych (funkcje `rand()` i `srand(...)`).

7. Uwaga: operacja dodawania powinna w szczególnym przypadku realizować podmianę wartości (zamiast dodania nowego wpisu), wtedy gdy jako argument podany zostanie istniejący już w strukturze klucz. Czyli musi tu mieć miejsce dodatkowe porównanie napisowe dwóch kluczy (czy są sobie dokładnie równe), a nie tylko samo wykrycie kolizji indeksów.

8. W programie należy wykorzystać poniższą **funkcję mieszającą** będącą rozwinięciem podanego jako argument napisu $s = (s_0, s_1, \ldots, s_{q-1})$ w bazie 31:

$$h(s) = \left(s_0 31^{q-1} + s_1 31^{q-2} + \cdots + s_{q-1} 31^0\right) \bmod N, \tag{1}$$

   gdzie q oznacza długość napisu, a N aktualny maksymalny rozmiar tablicy dynamicznej (będącej podstawą tablicy mieszającej). Uwaga: proszę zapoznać się z dokumentacją dzielenia modulo w języku C++ i zagwarantować nieujemny wynik takiego dzielenia (co pozwoli na użycie go jako indeksu w tablicy).

9. W celu sprawdzenia prawidłowego zachowania się funkcji mieszającej i jej własności rozpraszających, należy dodać do implementacji funkcję / metodę pełniącą rolę informacyjną, która zliczy proste **statystyki długości list** obecnych wewnątrz struktury. Mogą to być np.: liczba niepustych list, minimalna i maksymalna długość listy, średnia długość listy. Dzięki temu programista dowie się, czy powstające listy nie stają się zbyt długie wraz z dodawaniem dużej liczby elementów. Oczywiście uruchomienie tej funkcji będzie wymagało przebiegnięcia wszystkich elementów tablicy dynamicznej znajdującej się „pod spodem".

10. Poniżej przedstawiono **przykład napisowej reprezentacji** tablicy mieszającej w formie skrótowej (10 pierwszych elementów) wraz ze statystykami długości list. Przykład ten otrzymano w wyniku próby dodania miliona losowych elementów — par `<std::string, int>` — do tablicy mieszającej. W przykładzie warto zwrócić uwagę, że faktyczna liczba dodanych elementów (998 422) była mniejsza niż milion, ponieważ pewna liczba dodań zadziała jako podmiana w związku z powtórzeniem się losowych kluczy sześcioznakowych.

```
hash table:
  current_size: 998422
  max_size: 2097152
  table:
  {
        2: mudwrl -> 425811;
        5: fixxie -> 854961;
        8: cixxih -> 108209, dnjqyl -> 761203;
        9: hwmdzu -> 493654;
        13: kudwrw -> 273624;
        18: nnjqyv -> 392255;
        23: ludwsb -> 257412, nlbeqy -> 721784, yudwsb -> 891592;
        ...
  }
  stats:
    list min size: 1
    list max size: 7
    non-null lists: 787487
    list avg size: 1.267858
```

11. Na potrzeby głównego eksperymentu przydatne będzie stworzenie funkcji (poza klasą / strukturą tablicy mieszającej), która będzie zwracała **losowy napis stanowiący klucz** (przynajmniej 6-znakowy).

3 Zawartość funkcji `main()`

Główny eksperyment zawarty w funkcji `main()` ma polegać na wielokrotnym dodawaniu coraz większej liczby elementów (danych) do tablicy mieszającej (rzędy wielkości od 10^1 aż do 10^7), a następnie wykonaniu pewnej liczby prób wyszukiwania (np. 10^4 prób). Należy raportować czasy dodawania i wyszukiwania (całkowite i średnie).

Poniższy listing pokazuje schemat eksperymentu (proszę traktować go jako poglądowy przykład):

```
int main()
{
    ...
    const int MAX_ORDER = 7; // maksymalny rzad wielkosci dodawanych danych

    hash_table<int>* ht = new hash_table<int>(); // w tym przykladzie tablica mieszajaca par <string,
        int> jako <klucz, wartosc>
    for (int o = 1; o <= MAX_ORDER; o++)
    {
        const int n = pow(10, o); // rozmiar danych

        // dodawanie do tablicy mieszajacej
        clock_t t1 = clock();
```

```
        for (int i = 0; i < n; i++)
            ht->put(random_key(6), i); // klucze losowe 6-znakowe, a jako wartosci indeks petli
        clock_t t2 = clock();
        ... // wypis na ekran aktualnej postaci tablicy mieszajacej (skrotowej) oraz pomiarow
             czasowych

        // wyszukiwanie w tablicy mieszajacej
        const int m = pow(10, 4);
        int hits = 0;
        t1 = clock();
        for (int i = 0; i < m; i++)
        {
            hash_table_entry<int>* entry = ht->get(random_key(6)); // wyszukiwanie wg losowego klucza
            if (entry != NULL)
                hits++;
        }
        t2 = clock();
        ... // wypis na ekran pomiarow czasowych i liczby trafien

        // wypis statystyk (dlugosci list kryjacych sie w tablicy mieszajacej)
        ht->print_stats();

        ht->clear(); // czyszczenie tablicy mieszajacej
    }
    delete ht;
    return 0;
}
```

4 Sprawdzenie antyplagiatowe — przygotowanie wiadomości e-mail do wysłania

1. Kod źródłowy programu po sprawdzeniu przez prowadzącego zajęcia laboratoryjne musi zostać przesłany na adres *algo2@zut.edu.pl*.

2. Plik z kodem źródłowym musi mieć nazwę wg schematu: `nr_albumu.algo2.nr_lab.main.c` (plik może mieć rozszerzenie `.c` lub `.cpp`). Przykład: `123456.algo2.lab06.main.c` (szóste zadanie laboratoryjne studenta o numerze albumu 123456). Jeżeli kod źródłowy programu składa się z wielu plików, to należy stworzyć jeden plik, umieszczając w nim kody wszystkich plików składowych.

3. Plik musi zostać wysłany z poczty ZUT (zut.edu.pl).

4. Temat maila musi mieć postać: `ALGO2 IS1 XXXY LAB06`, gdzie `XXXY` to numer grupy (np. `ALGO2 IS1 210C LAB06`).

5. W pierwszych trzech liniach pliku z kodem źródłowym w komentarzach muszą znaleźć się:

   - informacja identyczna z zamieszczoną w temacie maila (linia 1),
   - imię i nazwisko autora (linia 2),
   - adres e-mail (linia 3).

6. Mail nie może zawierać żadnej treści (tylko załącznik).

7. W razie wykrycia plagiatu, wszytkie uwikłane osoby otrzymają za dane zadanie ocenę 0 punktów (co jest gorsze niż ocena 2 w skali $\{2, 3, 3.5, 4, 4.5, 5\}$).