Algorytmy i złożoności

Wykład 5. Haszowanie (hashowanie, mieszanie)

Wprowadzenie

Haszowanie jest to pewna technika rozwiązywania ogólnego problemu słownika. Przez problem słownika rozumiemy tutaj takie zorganizowanie struktur danych i algorytmów, aby można było w miarę efektywnie przechowywać i wyszukiwać elementy należące do pewnego dużego zbioru danych (uniwersum). Przykładem takiego uniwersum mogą być liczby lub napisy (wyrazy) zbudowane z liter jakiegoś alfabetu. Innym rozwiązaniem, które poznamy na dalszych zajęciach są tzw. *drzewa BST*.

Wyobraźmy sobie, że musimy w jakiś sposób przechowywać w programie słowa. Jeżeli chcielibyśmy użyć do tego celu tablicy, to powstanie oczywiście problem, w jaki sposób zamienić (odwzorować) poszczególne słowa na liczby będące indeksami w tej tablicy. Gdybyśmy chcieli ponumerować wszystkie słowa, które występują w języku polskim i następnie używać tych numerów jako indeksów, to tablica musiałaby mieć ogromny rozmiar. Tutaj właśnie rozpoczyna się *haszowanie*. Polega ono na odwzorowaniu zbioru wszystkich możliwych słów S przy pomocy tzw. funkcji haszującej w stosunkowo niewielką tablicę:

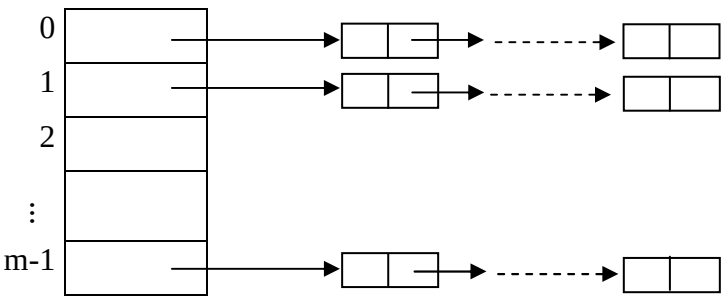
$$H : S \to \{0,\ldots,m-1\}$$

Oczywiście nie ma cudów: nie da się znaleźć takiej funkcji H, która mogła by odwzorować duży zbiór S w znacznie mniejszy zbiór $\{0,\ldots,m-1\}$ w sposób różnowartościowy. Ale jeżeli dobierze się odpowiednią funkcją H, to sytuacje, w których występują kolizje powinny się zdarzać stosunkowo rzadko. Oczywiście problem kolizji należy jednak uwzględnić.

Rozwiązywanie problemu kolizji

Metoda I

Jest to tzw. **metoda łańcuchowa**. Dla każdego indeksu $i \in \{0,\ldots,m-1\}$ przechowujemy w tablicy *HashTab* początek do listy elementów, które mają tę samą wartość funkcji haszującej. Tak więc nasza struktura może wyglądać tak:



Metoda II
Jest to tzw. **metoda adresowania liniowego**. Jeżeli dana pozycja H(k) w tablicy *HashTab* jest już zajęta, to szukamy nowego miejsca pod adresami H(k)+1, H(k)+2, ... Gdy dojdziemy do końca tablicy *HashTab* (czyli przekroczymy ostatni indeks równy *m-1*), to „zawijamy" się na początek tablicy. Można wiec to opisać pseudokodem tak:

```
i = 1;
while (((HashTab[(H(k) + i) mod m]) jest zajęta)
     i++;
wstaw element w miejsce HashTab[(H(k) + i) mod m]
```

Wadą tej metody jest tendencja do grupowania się obiektów wokół pierwotnych kluczy.

Metoda III
Jest to tzw. **metoda adresowania kwadratowego**. Metoda ta jest prostą modyfikacją adresowania liniowego. Zamiast dodawać kolejne wartości *i* dodajemy ich kwadraty i^2:

```
i = 1;
while (((HashTab[(H(k) + i*i) mod m]) jest zajęta)
     i++;
wstaw element w miejsce HashTab[(H(k) + i*i) mod m]
```

Okazuje się, że w tym przypadku nie występuje tendencja do grupowania się obiektów wokół pierwotnych kluczy. Jej niewielką wadą jest to, że podczas próbkowania nie są przeszukiwane wszystkie pozycje tablicy *HashTab*, tj. może się zdarzyć, że przy wstawianiu nie znajdzie się wolnego miejsca, chociaż puste pozycje w tablicy *HashTab* nadal występują..

Najlepsze efekty uzyskuje się, gdy jako *m* (rozmiar tablicy *HashTab*) używamy *liczb pierwszych*.

Przykład 1
Zakładając, że każde słowo możemy zapisać przy pomocy 26 podstawowych liter alfabetu łacińskiego, oszacujemy ile jest takich słów zakładając, że interesują nas słowa nie dłuższe niże 10 znaków oraz rozróżniamy duże i małe litery.

Zamiana słów na liczby
Spróbujmy zbudować jakąś prostą funkcje haszującą działającą na dziedzinie słów (z literami bez polskich znaków diakrytycznych ą, ć itd.).
Jak wiadomo komputery używają różnych sposobów reprezentacji liter jako liczb. Jednym z takich systemów jest standard ASCII, w którym są m. in. zakodowane wszystkie litery alfabetu łacińskiego (duże i małe znaki), cyfry oraz znaki interpunkcyjne. Przykładowo litera 'a' ma kod 97, 'd' 100, 'A' to 65. Obecnie przechodzi się na kodowanie przy pomocy standardu *Unicode*, w którym znakom przypisuje się liczby dwubajtowe.
My jednak stworzymy własny kod, przyjmując: 'a' to 1, 'b' to 2, ..., 'ż' to 32. Oto odpowiednia tabela:

a	b	c	d	e	f	g	h	i	j	k	l	ł	m	n	o	p	r	s	t	u	w	y	z	ą	ę	ć	ń	ó	ś	ź	ż
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Jak możemy teraz przekształcić poszczególne słowa w jedną liczbę? Istnieje oczywiście wiele sposobów; my przeanalizujemy tylko dwa dość reprezentatywne. Przekonamy się również, że

oba mają poważne wady, co uświadomi nas, że tablice haszujące i odpowiednie funkcje haszujące (mieszające) są niezbędne.

Przykład 2 (Dodawanie kodów poszczególnych znaków)
Najprostszy sposób odwzorowania słowa na liczbę to dodanie kodów wszystkich liter. Tak więc *H('kot')=11+16+20=47*.

Przykład 3 (Mnożenie przez potęgi)
Spróbujmy odwzorować słowa na liczby inną metodą. Postarajmy się teraz, aby kodów było więcej. Niech H(x_1x_2...) = liczba, której kodem o podstawie 32 jest napis "x_1x_2 ...", gdzie wartości poszczególnych cyfr 'a', 'b', itd. są takie jak w powyższej tabeli.
Tak więc
$H('kot') = 32^2 \cdot k+32^1 \cdot o+32^0 \cdot t = 32^2 \cdot 11+32^1 \cdot 16+32^0 \cdot 20 = 11264+352+20=11636.$

Przykład 4 (Adresowanie liniowe)
Zrealizujemy algorytm *haszowania* z próbkowaniem liniowym dla zbioru danych, które są liczbami całkowitymi. Dla uproszczenia przyjmijmy, że uniwersum danych do kodowania to *S=long* (w sensie języka C). W szczególności zdefiniujemy funkcje *insert()* oraz *search()*, które odpowiednio wstawiają i wyszukują elementy.

Wskazówki
W poniższym „ogólnym" kodzie należy za `Item` przyjąć typ **`long`**. Funkcja mieszająca `H()` dokonuje pierwszego wyboru indeksu w tablicy *HashTab*. Można ją różnie definiować. Jako pierwszą najlepiej wybrać proste dzielenie modulo:

$$H(e) = e \bmod MAX\_HASHTAB$$

Deklaracje:

```
const int MAX_HASHTAB = ???; // dowolna liczba dodatnia typu int; najlepiej liczba pierwsza
const int FREE = -1; // Uwaga! Oznacza to, że nie możemy przechowywać w naszej tabeli
                     // wartości typu long równej -1
Item *HashTab;       // dalej będzie alokacja pamięci
int n;               // do sprawdzania stopnia zapełnienia tablicy
```

Funkcje:

```
int H(long e) { return (e % MAX_HASHTAB); }

void create()
{
      HashTab = new Item[MAX_HASHTAB];

      for(int i=0; i < MAX_HASHTAB; i++)
            HashTab[i] = FREE;

      n = 0;
}

void insert(Item e)
{
```

```
      int i;
      i = H(e);
      while (HashTab[i] != FREE)
            i = (i + 1) % MAX_HASHTAB;

      HashTab[i] = e;
      n++;
}

int search(Item e)
{
      int i;

      i = H(e);
      while (HashTab[i] != FREE)
            if (HashTab[i] == e)
                  return i;
            else
                  i = (i + 1) % MAX_HASHTAB;

      return -1;
}

main()
{
      long e;
      cout << ”Test działania\n”;

      create();

      for (e=7; e < 100000; e += 9)
            insert(e);
}
```

Funkcje mieszające dla kluczy napisowych

W wielu sytuacjach klucze (elementy słownika) nie są liczbami, lecz długimi ciągami znaków alfanumerycznych. Jak obliczyć funkcję mieszającą dla takiego słowa jak:

bardzudlugiklucz?

W 7-bitowym kodzie *ASCII* to słowo odpowiada liczbie:

$$b \cdot 128^{15} + a \cdot 128^{14} + ... + c \cdot 128^{1} + z \cdot 128^{0},$$

która jest zbyt wielka, aby mogła być używana przez standardowe funkcje arytmetyczne komputerów.
A oto inny sposób, który po prostu oblicza resztę z dzielenia wartości wyliczonej powyżej przez rozmiar tablicy (*m* = *MAX_HASHTAB*):

```
int H1(char* str, int m)
{
      int h = 0; a = 127;
      while (*str != 0) {
            h = (a*h + *str) % m;
            str++;
```

```
	}
	return h;
}
```

Funkcja H2(), której kod jest podany niżej ma bardziej równomierny rozkład. Oznacza to, że prawdopodobieństwo kolizji pomiędzy dwoma różnymi kluczami zbliża się do 1/m gdzie m jest rozmiarem tablicy mieszania (*m = MAX_HASHTAB*). Oto kod funkcji:

```
int H2(char* str, int m)
{
	int h = 0, a = 31415, b = 27183;

	while (*str != 0) {
		h = (a*h + *str) % m;
		a = a*b % (m-1);
		str++;
	}

	return (h < 0) ? (h + m) : h;
}
```