Notatki z AiSD. Nr ?? 16 marca 2005

HASHOWANIE

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

26 Wstęp

Hashowanie jest jedną z metod realizacji słowników. Poznaliśmy już m.in. drzewa czerwono-czarne, drzewa AVL, czy B-drzewa, struktury, które umożliwiały wykonywanie operacji słownikowych w czasie proporcjonalnym z logarytmu z wielkości słownika.

Gdy uniwersum jest małe (powiedzmy n elementowe), możemy wykorzystać n elementowe tablice bitowe (i-ty element takiej tablicy jest równy 1 wtedy i tylko wtedy gdy i-ty element uniwersum należy do zbioru; zakładamy przy tym, że umiemy efektywnie numerować elementy uniwersum). Przy takim sposobie pamiętania słownika czas wykonania operacji słownikowych jest stały.

27 Metoda funkcji hashujących

Do pamiętania elementów podzbioru wykorzystywana jest tablica $T[0..m-1]$. Zwykle m jest proporcjonalne do maksymalnej liczności słownika; wielkość uniwersum nie ma tu większego znaczenia. Metoda wykorzystuje funkcję (tzw. *funkcję haszującą*) $h : U \to \{0, ..., m-1\}$, określającą miejsce pamiętania elementów U w T.

27.1 Funkcje haszujące

Dobra funkcja haszująca powinna spełniać następujący warunek:

$$\text{(dfh)} \qquad \forall_{j=0,\ldots,m-1} \sum_{k:h(k)=j} P(k) = \frac{1}{m},$$

gdzie $P(k) =$ jest prawdopodobieństwem tego, że $k \in U$ będzie parametrem którejś z operacji słownikowych. W praktyce warunek ten jest zwykle niesprawdzalny, gdyż nie znamy P. Ponadto, jeśli metoda ma być efektywna, funkcja hashująca powinna być szybkoobliczalna.

Nie polecam wymyślania własnych funkcji hashujących (przynajmniej na początku). Lepiej skorzystać z doświadczenia innych.

Przykłady funkcji haszujących.

- $h(k) = k \bmod m$

  UWAGA. Należy wykazać się ostrożnością w wyborze m. Nie zaleca się m postaci 2^p, gdyż wówczas $h(k)$ jest równe ostatnim p bitom klucza k, a te często mają nierównomierny rozkład. Z tego samego powodu nie zaleca się brać jako m potęg liczby 10. Zwykle dobrymi wartościami m są liczby pierwsze niezbyt bliskie potęgom liczby 2.

- $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, gdzie A jest ustaloną liczbą z przedziału $(0, 1)$.

  UWAGA. Teraz wartość m nie ma takiego znaczenia jak poprzednio i zwykle bierze się m równe potędze liczby 2 (ze względu na łatwość mnożenia). Wybór A jest bardziej zależny od cech danych, jednak zwykle A równe $(\sqrt{5} - 1)/2 \approx 0.6180339887$ jest dobre.

27.2 Metody pamiętania elementów

Ponieważ wielkość tablicy T jest z reguły znacznie mniejsza od wielkości uniwersum, dość często zetkniemy się z sytuacją, gdy chcemy w T zapamiętać y, taki że $h(y) = h(x)$ dla pewnego x aktualnie pamiętanego w T. Sytuację taką nazywamy *kolizją*. Sposoby rozwiązywania kolizji zależą istotnie od tego w jaki sposób pamiętamy elementy w tablicy.

Rozważymy dwa sposoby pamiętania elementów.

27.2.1 Listy elementów

i-ty element tablicy zawiera wskaźnik na początek listy tych elementów x słownika, dla których $h(x) = i$.

Jeśli założymy, że koszt obliczenia wartości funkcji haszującej jest stały, to koszt INSERT i DELETE też jest stały, a koszt SEARCH(K) jest zależny od długości listy $T[k]$.

Fakt 20 *Przy założeniu (dfh) średni koszt operacji* SEARCH *wynosi* $\Theta(1 + \frac{n}{m})$*, gdzie* n*-liczba elementów* U *pamiętanych w* T*.*

UWAGA. $\alpha = \frac{n}{m}$ nazywane jest *współczynnikiem wypełnienia* tablicy haszującej.

Wniosek 1 *Gdy* $n = O(m)$*, to średni koszt operacji* SEARCH *(a więc także wszystkich operacji słownikowych) jest* $\Theta(1)$*.*

27.2.2 Adresowanie otwarte

Teraz elementy słownika pamiętamy bezpośrednio w elementach tablicy T. Likwidujemy w ten spoób istotny mankament poprzedniej metody: nie tracimy miejsca na pamiętanie wska"xników. Powstaje jednak nowe niekorzystne zjawisko - przepełnienie się tablicy T. Często nie potrafimy z góry określić wielkości słownika i dlatego rozpoczynamy z tablicą umiarkowanych rozmiarów. Gdy okazuje się ona za mała (jest tak nie tylko wtedy gdy w słowniku chcemy umieścić $(m+1)$-szy element, lecz już wtedy gdy duży stopień wypełnienia tablicy powoduje, że operacje słownikowe są kosztowne), powiększamy ją, zmieniamy funkcję hashującą (tak by przyjmowała wartości z nowego zakresu) i na nowo obliczamy miejsca umieszczenia wszystkich elementów słownika.

Usuwanie kolizji

Używamy funkcji haszującej

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Najpierw próbujemy umieścić element k na pozycji $h(k, 0)$. Jeśli pozycja ta jest zajęta, próbujemy $h(k, 1)$ jeśli ta jest zajęta sprawdzamy pozycję $h(k, 2)$, itd...

Funkcja h powinna spełniać następujący warunek:

(per) $\forall_{k \in U} \langle h(k, 0), \ldots, h(k, m-1) \rangle$ jest permutacją zbioru $\{0, 1, \ldots, m-1\}$.

To gwarantuje nam, źe nie znajdziemy miejsca na umieszczenie danego elementu dopiero wtedy, gdy tablica jest całkowicie zapełniona.

Przykłady:

- *Metoda liniowa*:
$$h(k, i) = (h'(k) + i) \bmod m,$$
gdzie $h' : U \to \{0, .., m-1\}$ jest pomocniczą funkcją haszującą (np. takie jak opisano powyźej).

- *Metoda kwadratowa*:
$$h(k, i) = (h'(k) + c_1 i + c_2 * i^2) \bmod m,$$
gdzie h' - jak poprzednio.
Uwaga: $c_1, c_2 \neq 0$. Stałe c_1, c_2 oraz m powinny być tak dobrane by zachodził warunek (per).

- *Podwójne haszowanie*:
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$
gdzie h_1, h_2 - pomocnicze funkcje haszujące.
Uwaga: Dla każdego $k \in U$, $h_2(k)$ powinno być względnie pierwsze z m.

W praktyce najlepsze rezultaty daje podwójne haszowanie. W metodzie liniowej (i w mniejszym stopniu w metodzie kwadratowej) występuje negatywne zjawisko tworzenia się *zlepków* (tj. zwartych obszarów tablicy T, zajętych przez elementy U), które znacznie obniźa efektywność metody.

Podczas wykonywania operacji Delete w miejscu usuwanego elementu w tablicy T należy wpisać znacznik świadczący o tym, źe to miejsce było juź kiedyś zajęte.

Analiza kosztów

Dla uproszczenia analizy stosujemy pozniźsze (nieco wyidealizowane) założenie:

(dper) ciąg $\langle h(k, 0), \ldots, h(k, m-1) \rangle$ jest z równym prawdopodobieństwem dowolną permutacją zbioru $\{0, \ldots, m-1\}$.

Twierdzenie 9 *Przy założeniu (dper) i* $\alpha = \frac{n}{m} < 1$ *oczekiwana liczba prób w poszukiwaniu zakończonym fiaskiem jest* $\leq \frac{1}{1-\alpha}$.

PRZYKŁAD. Załóżmy, że utworzyliśmy słownik i teraz wykonujemy wiele operacji SEARCH. Jeśli tablica jest zajęta w 50%, to średnia liczba prób przy poszukiwaniu zakończonym niepowodzeniem jest nie większa od 2; gdy tablica zajęta jest w 90%, to liczba ta jest nie większa od 10.

Wniosek 2 *Przy powyższych założeniach umieszczenie elementu w tablicy haszującej wymaga średnio* $\leq \frac{1}{1-\alpha}$ *prób.*

Twierdzenie 10 *Przy założeniu (dper) i* $\alpha = \frac{n}{m} < 1$ *oczekiwana liczba prób w poszukiwaniu zakończonym sukcesem jest* $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$.

PRZYKŁAD. Gdy tablica wypełniona jest w 90%, poszukiwanie zakończone sukcesem wymaga średnio nie więcej niż 3.67 prób.

UWAGA. W praktyce, pomimo niespełnienia założenia (dper), koszt operacji słownikowych jest zbliżony do kosztu wynikającego z powyższych twierdzeń.

28 Hashowanie uniwersalne

Oczywiście dla każdej funkcji hashującej istnieją dane, które powodują, że czas wykonywania operacji słownikowych jest duży (np. może się zdarzyć, że dla wszystkich elementów umieszczanych w słowniku wartość funkcji hashującej będzie ta sama). Aby uniezależnić się od takich danych wprowadzamy, podobnie jak w algorytmie *Quicksort*, randomizację: zamiast korzystać z ustalonej funkcji hashującej, losujemy ją na początku działania programu z pewnej rodziny funkcji.

Definicja 13 *Niech* H *będzie rodziną funkcji hashujących z* U *w* $\{0, \ldots, m-1\}$. *Rodzinę* H *nazywamy* uniwersalną, *jeśli* $\forall_{x,y \in U;\, x \neq y}$:

$$|\{h \in H : \ h(x) = h(y)\}| = \frac{|H|}{m}$$

Twierdzenie 11 *Niech* H *będzie uniwersalną rodziną funkcji hashujących. Dla dowolnego zbioru* $n \leq m$ *kluczy, liczba kolizji w jakich bierze udział ustalony (ale dowolny) klucz* x *jest w średnim przypadku mniejsza od* 1.

28.1 Przykład rodziny uniwersalnej

Niech m będzie liczbą pierwszą oraz $|U| < m^{r+1}$. Dla każdego $0 \leq a < m^{r+1}$ definiujemy funkcję h_a:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i,$$

gdzie $\langle a_0, a_1, \ldots, a_r \rangle$ i $\langle x_0, x_1, \ldots, x_r \rangle$ są reprezentacjami odpowiednio liczb a i x w systemie m-arnym.

Twierdzenie 12 *Rodzina* $H = \{h_a : 0 \leq a < m^{r+1}\}$ *jest rodziną uniwersalną.*

UWAGA: Dowody wszystkich twierdzeń można znale"xć w książce Cormena.

Perfect hashing.

We consider the following *perfect hashing* problem: Given a set S of n keys from a universe U, build a look-up table T of size $O(n)$ such that a membership query (given $x \in U$, is $x \in S$) can be answered in constant time.

We show that a perfect hash table can be built in linear expected time. The idea is to build a two-level table (see Fig. 1). In the first level, a hash function f partitions the set S into n subsets, denoted as *buckets*, $B_1, B_2, \ldots, B_n$. For a bucket B_i, we denote its size as $b_i = |B_i|$. In the second level, each bucket B_i has a separate memory array whose size is $\Theta(b_i^2)$, and a separate hash function g_i that maps the bucket injectively into that memory array. All the memory arrays are placed in a single table T, and for each bucket B_i we maintain the offset p_i, which gives the position in T where B_i's memory array begins.

A high level description of the algorithm is as follows:

Step 1 Find a function $f : U \to [1..n]$, that partitions S into buckets $B_1, B_2, \ldots, B_n$ such that $\sum_{i=1}^{n} b_i^2 \leq \beta n$, where β is a constant that will be determined later.

Step 2 For each bucket B_i, compute an offset $p_i = \sum_{j}^{i-1} \alpha b_j^2$, and allocate a subarray M_i of size αb_i^2 in array T between positions $p_i + 1$ and p_{i+1} in T, where α is a constant that will be determined later.

Step 3 For each bucket B_i find a function $g_i : u \to [1..\alpha b_i^2]$, such that g_i is injective on B_i. For every key $x \in B_i$, place x in $T[p_i + g_i(x)]$.

In Step 1, the function f is recorded. We use two additional arrays: $P[1..n]$ to record the offsets in Step 2, and $G[1..n]$ to record the functions g_i in Step 3. The table T is of size $\alpha \sum_{i=1}^{n} b_i^2 \leq \alpha\beta \cdot n$, and the total memory required by the data structure is therefore $O(n)$, as required. Given a key $x \in U$, a membership query for x is supported in constant time as follows:

1. Compute $i = f(x)$.
2. Read g_i from $G[i]$ and compute $j = g_i(x)$.
3. If $T[P[i] + j] = x$ then answer "$x \in S$", and otherwise answer "$x \notin S$".

More details and analysis:

In our analysis we will use four basic facts from probability theory, and a property of universal hash functions:

1. *Boole's inequality*: For any sequence of events $A_1, A_2, \ldots, A_m$, $m \geq 1$,
   $\mathbf{Pr}(A_1 \cup A_2 \cdots \cup A_m) \leq \mathbf{Pr}(A_1) + \mathbf{Pr}(A_2) + \cdots + \mathbf{Pr}(A_m)$.

2. *Markov inequality*: Let X be a nonnegative random variable, and suppose that $\mathbf{E}(X)$ is well defined. Then for all $t > 0$, $\mathbf{Pr}(X \geq t) \leq \mathbf{E}(X)/t$. Alternatively, for all $\tau > 0$, $\mathbf{Pr}(X \geq \tau \mathbf{E}(X)) \leq 1/\tau$.

3. *Linearity of expectation:* $\mathbf{E}(X+Y) = \mathbf{E}(X) + \mathbf{E}(Y)$; more generally, $\mathbf{E}(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} \mathbf{E}(X_i)$.

4. *Expectation in geometric-like distribution:* Suppose that we have a sequence of Bernoulli trials, each with a probability $\geq p$ of success and a probability $\leq 1-p$ of failure. Then the expected number of trials needed to obtain a success is at most $1/p$.

5. *Collisions in universal hash functions:* If h is chosen from a universal collection of hash functions and is used to hash N keys into a table of size B, the expected number of collisions involving a particular key x is $(N-1)/B$.

We can now provide more details on Step 1, which consists of the following sub-steps.

Step 1a Select at random a function $f : U \to [1..n]$ from a universal class of hash functions.

Step 1b Compute a hash-table T' with chaining using the hash function f, so that insertion takes constant time.

Step 1c Compute an array $B2$, so that $B2[i] = b_i^2$.

Step 1d If $\sum_{i=1}^{n} b_i^2 > \beta n$ then go to Step 1a; otherwise record the function f.

Analysis Step 1a takes constant time, Step 1b takes $O(n)$ time and $O(n)$ space, Step 1c takes $O(n)$ time using the table T', and Step 1d takes $O(n)$ time, using array $B2$. The time complexity, T_1, of Step 1 is therefore $O(tn)$, where t is the number of iterations, i.e., the number of functions f selected before the condition $\sum_{i=1}^{n} b_i^2 \leq \beta n$ is satisfied. The following claim shows that for $\beta \geq 4$ we have $\mathbf{E}(T_1) = O(n)$.

Claim: If $\beta \geq 4$ then $\mathbf{E}(t) \leq 2$.

Proof. Let C_x be the number of collisions of a key $x \in S$ under f; i.e., the number of $y \in S$, $y \neq x$, for which $f(x) = f(y)$. Due to the collision property of universal hash functions (with $N = B = n$) we have $\mathbf{E}(C_x) < 1$.

We consider the total number of collisions C_S in S. Specifically, let C_S be the number of (ordered) pairs $\langle x, y \rangle$, $x, y \in S$ and $x \neq y$, such that $f(x) = f(y)$. Clearly, $C_S = \sum_{x \in S} C_x$. Therefore, by linearity of expectation,

$$\mathbf{E}(C_S) = \sum_{x \in X} \mathbf{E}(C_x) < |S| \cdot 1 = n \ . \tag{1}$$

On the other hand, we note that collisions are defined among keys mapped into the same buckets, and can be counted as:

$$C_S = \sum_{i=1}^{n} |\{\langle x, y\rangle\} : x, y \in B_i, x \neq y\}| = \sum_{i=1}^{n} b_i \cdot (b_i - 1) = \sum_{i=1}^{n} b_i^2 - \sum_{i=1}^{n} b_i \ .$$

Therefore, since $\sum_{i=1}^{n} b_i = n$,

$$\sum_{i=1}^{n} b_i^2 = C_S + n \ ,$$

and by Eq (1)

$$\mathbf{E}\left(\sum_{i=1}^{n} b_i^2\right) = \mathbf{E}\left(C_S\right) + n < 2n \ .$$

By Markov Inequality, applied to the random variable $X = \sum_{i=1}^{n} b_i^2$,

$$\mathbf{Pr}\left(\sum_{i=1}^{n} b_i^2 \geq 4n\right) \leq 1/2 \ .$$

If $\beta \geq 4$, then for a function f selected at random the condition $\sum_{i=1}^{n} b_i^2 \leq 4n$ is satisfied with probability at least $1/2$. Therefore, the expected number, t, of functions f tried before the condition is satisfied is at most 2. ∎

To compute Step 2, note that $p_i = p_{i-1} + \alpha b_{i-1}^2$ for $i > 1$, and $p_1 = 0$. Therefore, p_i can be computed and recorded in array P by iterating for $i = 1, \ldots, n$. Step 2 takes $T_2 = O(n)$ time.

Finally, Step 3 consists of the following sub-steps, executed for all i, $i = 1, \ldots, n$:

Step 3a Initialize the subarray $T[P[i] + 1, \ldots, P[i + 1]]$ to *nil*.

Step 3b Select at random a function $g_i : U \rightarrow [1..\alpha b_i^2]$ from a universal class of hash functions.

Step 3c For each $x \in B_i$, if $T[P[i] + g_i(x)]$ is not *nil* then go to Step 3a (g_i is not injective on B_i and a new g_i is to be selected); else write x into $T[P[i] + g_i(x)]$.

Step 3d Record g_i in $G[i]$.

Analysis We analyze first Step 3 for bucket B_i. Step 3a takes time $O(b_i^2)$. Step 3b takes constant time. Step 3c can be implemented in $O(b_i)$ time, using the i'th list in the hash table T' computed in Step 1. Step 3d takes constant time. The time complexity of Step 3 for bucket B_i is therefore $t_i = O(\tau_i b_i^2)$, where τ_i is the number of iterations, i.e., the number of functions g_i selected before an injective function is found for B_i.

Comment: We could have each iteration take only $O(b_i)$ time by removing Step 3a, initializing the table T in Step 2, and modify Step 3c as follows:

Step 3c' For each $x \in B_i$, if $T[P[i] + g_i(x)]$ is not *nil* then for all $y \in B_i$ assign *nil* to $T[P[i] + g_i(y)]$ and go to Step 3a; else write x into $T[P[i] + g_i(x)]$.

The following claim shows that for $\alpha \geq 2$ we have $\mathbf{E}\left(t_i\right) = O(b_i^2)$.

Claim: If $\alpha \geq 2$ then $\mathbf{E}(\tau_i) \leq 2$.

Proof. Let C_x be the number of collisions of a key x in B_i under g_i; i.e., the number of $y \in B_i$, $y \neq x$, for which $g_i(x) = g_i(y)$. Due to the collision property of universal hash functions (with $N = b_i$ and $B = \alpha b_i^2$) we have

$$\mathbf{E}(C_x) < b_i/(\alpha b_i^2) = 1/\alpha b_i \ .$$

By Markov Inequality,

$$\mathbf{Pr}(C_x \geq 1) \leq \mathbf{E}(C_x) < 1/\alpha b_i \ . \tag{2}$$

Therefore, by Boole's inequality and Eq (2), the probability that there are any collisions in B_i is

$$\mathbf{Pr}(\exists x \in B_i \text{ such that } C_x \geq 1) \leq b_i \cdot (1/\alpha b_i) = 1/\alpha \ .$$

For $\alpha \geq 2$, the function g_i is injective with probability at least $1 - 1/\alpha \geq 1/2$, and the expected number of trials, τ_i, required before an injective function is found is at most 2. ∎

For $\alpha \geq 2$ we have

$$\mathbf{E}(t_i) = O(\mathbf{E}(\tau_i) b_i^2) = O(b_i^2) \ .$$

The total time, T_3, for Step 3 over all buckets is then

$$\mathbf{E}(T_3) = \sum_{i=1}^{n} \mathbf{E}(t_i) = O(\sum_{i=1}^{n} b_i^2) = O(\beta n)$$

The running time, T, of the entire algorithm can now be bounded as

$$\mathbf{E}(T) = \mathbf{E}(T_1 + T_2 + T_3) = \mathbf{E}(T_1) + \mathbf{E}(T_2) + \mathbf{E}(T_3) = O(n) \ .$$

Exercises

1. If h is chosen at random from an *almost-universal* collection of hash functions and is used to hash N keys into a table of size B, the collision probability of any two particular keys x and y is at most $2/B$, and the expected number of collisions involving a particular key x is at most $2(N-1)/B$.

   Modify the algorithm above so that almost-universal functions are used instead of universal functions, and such that the expected running time remains $O(n)$.

2. Modify the algorithm above and analyze it, so that the first level function f maps the input set S into $2n$ buckets, instead of n buckets.

3. (*) Generalizing (2), modify the algorithm above and analyze it, so that the first level function f maps the input set S into γn buckets, and select γ that gives favorable complexity (in terms of constants).