Słowniki dynamiczne

Wiele aplikacji wykorzystuje struktury danych jedynie do przechowywania elementów indeksowanych pewną wartością, zwaną **kluczem**. Na przykład, spis ludzi według nazwiska, lub numeru PESEL. W takich aplikacjach jedynymi operacjami wykonywanymi na strukturze danych są: INSERT, SEARCH, i DELETE. Tego typu zbiór dynamiczny jest czasami określany mianem **słownika**.

Do takich zastosowań bardzo dobrą strukturą danych są **tablice z haszowaniem** (*hash tables*). W poprawnej implementacji czas dostępu do ich elementów wynosi $\Theta(1)$ w przypadku średnim. Taki sam jest również czas operacji INSERT i DELETE.

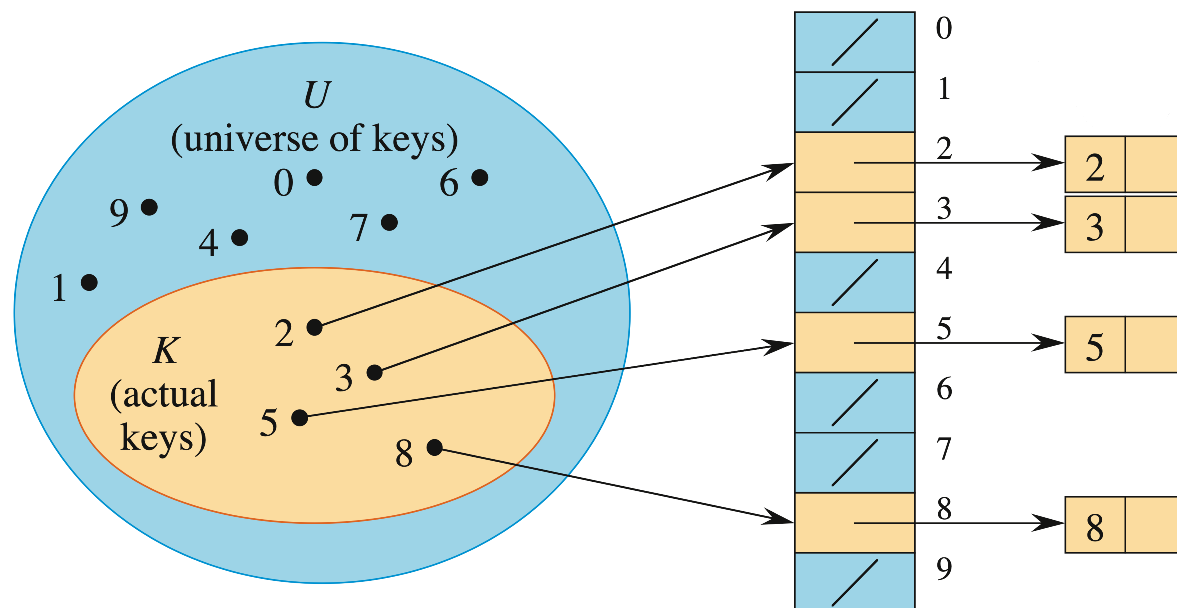
Te czasy wstawiania i usuwania elementu są więc identyczne do czasów wykonywania tych operacji na zwykłych tablicach, pod warunkiem, że znamy indeks elementu (albo pozycji do jego wpisania). Jednak w zwykłej tablicy nieuporządkowanej średni czas wyszukiwania elementu wynosi $\Theta(n)$. W tablicy uporządkowanej wynosi on $\Theta(\log n)$, jednak czas wpisywania do, i usuwania z, tablicy uporządkowanej jest znacznie dłuższy (zwykle wymaga przeorganizowania tablicy).

Zatem tablice z haszowaniem stanowią niezwykle sprawną strukturę danych. Jednak ta sprawność ma swój koszt. Tablica z haszowaniem musi być tworzona z uwzględnieniem pewnych warunków i założeń, i musi być zbudowana poprawnie. Oznacza to, że można ją również zbudować niepoprawnie. W najgorszym przypadku czasy zarówno operacji budowy tablicy INSERT/DELETE, jak i czas wyszukiwania wynoszą $\Theta(n)$.

Tablice bezpośrednio adresowane

Rozważmy na początek prostą technikę wykorzystania tablic do budowy słownika, gdzie **przestrzeń** (*universe*) możliwych kluczy jest bardzo mała. Mamy tu na myśli nie tylko małą liczbę możliwych kluczy, ale wręcz mały zakres ich wartości. Dla uproszczenia załóżmy, że wszystkie klucze należą do zakresu $U = \{0, 1, ..., m-1\}$, gdzie m jest względnie małą liczbą.

Wtedy sam klucz może służyć w roli indeksu elementu tablicy, i każdy element może być bezpośrednio wpisany na swoje miejsce, jak również odnaleziony, i/lub usunięty, w czasie stałym $\Theta(1)$.



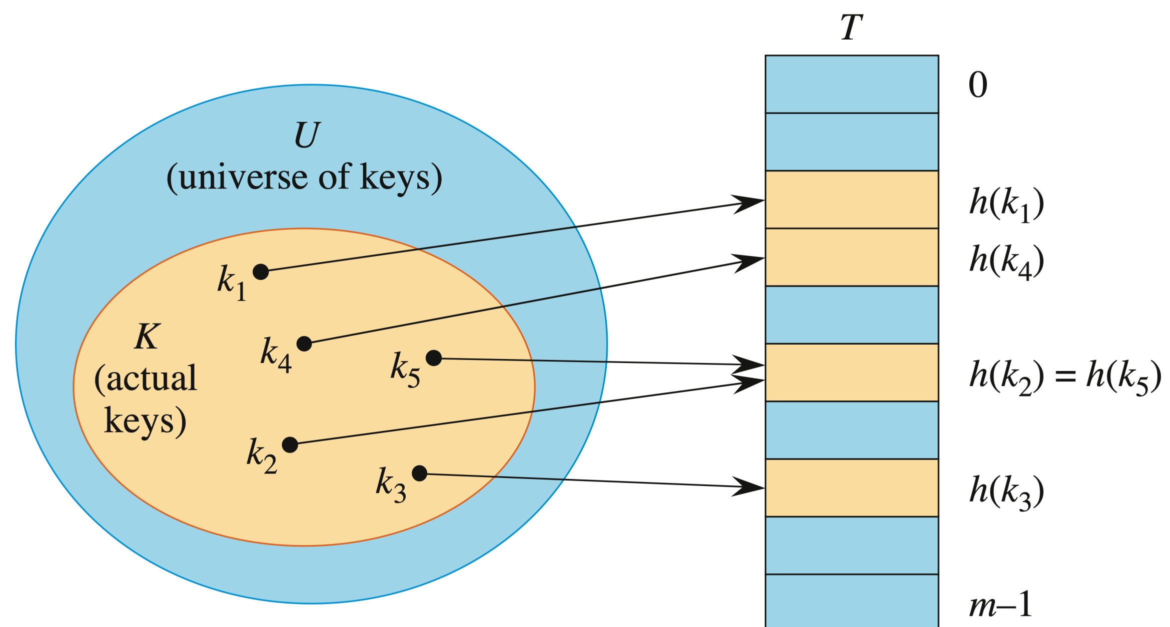
Alternatywnie do rozwiązania na rysunku, całe elementy mogą być wpisane wprost w pozycje tablicy. Wtedy trzeba tylko jakoś zaznaczyć czy element jest obecny w tablicy czy nie. Może do tego służyć specjalna wartość klucza, np. -1. A ponieważ klucze są równe indeksom tablicy, to ich samych w ogóle nie potrzeba przechowywać (jedynie wtedy trzeba w inny sposób odznaczyć brak obecności elementu w tablicy).

Tablice z haszowaniem

Tablice bezpośrednio adresowane są przykładem realizacji idei tablic z haszowaniem, ale są możliwe jedynie dla małych przestrzeni kluczy. W wielu przypadkach ten warunek nie jest spełniony. Na przykład, chcemy utworzyć zbiór do przechowywania danych osobowych grupy osób według numeru PESEL. Nawet jeśli dany zbiór ma całkiem niewielki rozmiar (na przykład zbiór studentów danego przedmiotu, maksymalnie 100 osób), to przestrzeń wszystkich wartości liczb PESEL jest ogromna (11 cyfr daje zbiór 10^{11}, czyli 100 miliardów). Mówimy o przechowywaniu danych dla 100 osób w tablicy o rozmiarze 100 miliardów pozycji.

Zamiast tego można użyć **funkcji haszującej** h odwzorowującej przestrzeń kluczy U do zbioru liczb o rozmiarze m rzeczywistej tablicy: $h : U \to \{0, 1, ..., m - 1\}$

O ile w tablicy bezpośrednio adresowanej każdy element z kluczem k jest zapisywany w pozycji k, to w tablicy z haszowaniem zostanie zapisany w pozycji $h(k)$.
Jednak mogą pojawić się **kolizje** (na rysunku klucze k_2 i k_5).



Kolizje

Przykładem prostej funkcji odwzorowującej dowolnie dużą przestrzeń liczb do dowolnie niewielkiego zbioru liczb w zakresie $[0..m-1]$ jest funkcja *modulo* $h(k) = k \bmod m$.

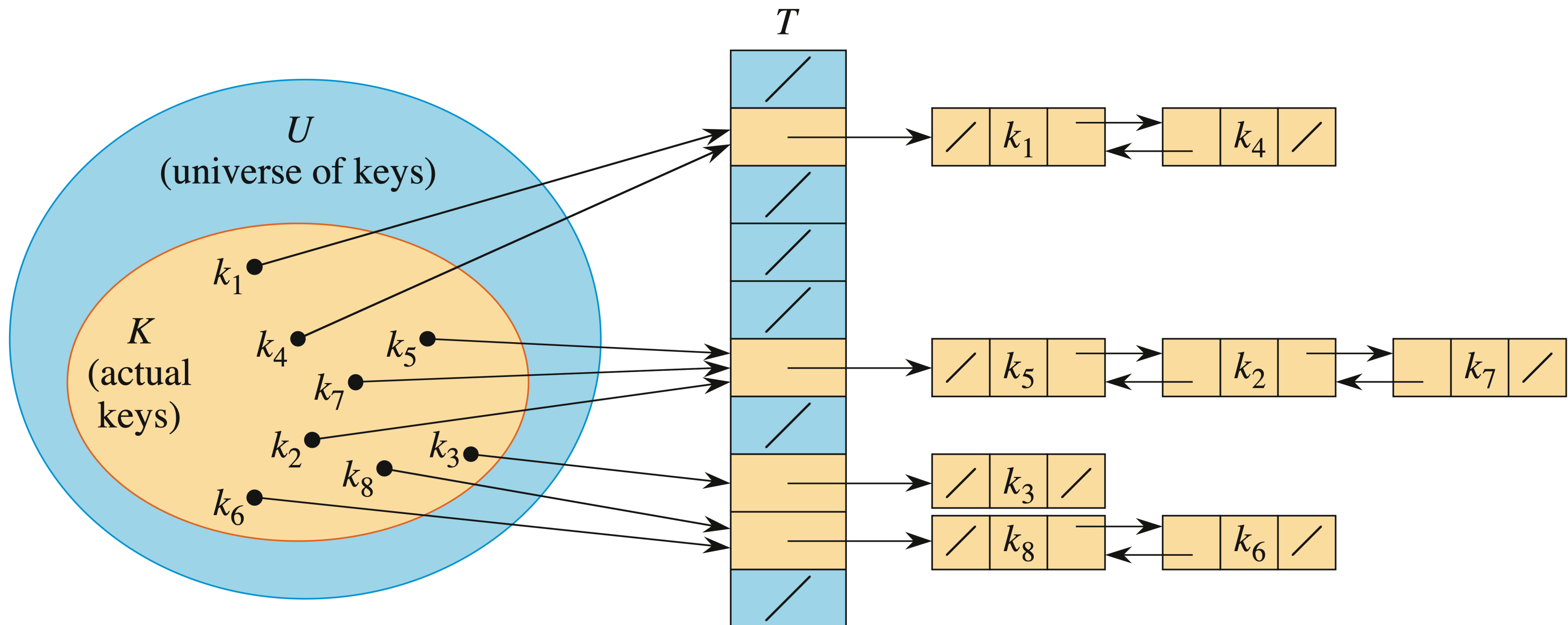
Niestety, wadą niektórych zbyt prostych funkcji haszujących są **kolizje**, które występują gdy dwa elementy o różnych wartościach klucza mają identyczną wartość funkcji haszującej. Pomimo iż wielkość zbioru $\{0, 1, ..., m-1\}$ (czyli wartość m) może być nawet z zapasem wystarczająca dla przechowania elementów danego zbioru danych, to kolizja funkcji haszującej uniemożliwia jej bezpośrednie użycie.

Podstawowym pomysłem na „zapewnienie" różnowartościowości funkcji haszującej jest zadbanie aby dawała wartości jak najbardziej przypadkowo rozrzucone. To jest zresztą oryginalne pochodzenie nazwy tych funkcji pochodzącej od angielskiego słowa *hash*, którego jednym ze znaczeń jest zbełtać. Pewnym pomysłem jest generowanie wartości losowych, jednak funkcja haszująca musi być deterministyczna, aby wielokrotnie wywoływana dawała dla każdego klucza zawsze tę samą wartość.

Pozostała część tego wykładu obejmie zarówno metody radzenia sobie z kolizjami, których w ogólności nie da się uniknąć, jak też podejścia do budowy dobrej jakości funkcji haszujących, które minimalizują prawdopodobieństwo i liczbę kolizji.

Rozwiązywanie kolizji metodą łańcuchową

Metoda **łańcuchowa** rozwiązywania kolizji polega na przechowywaniu w każdej pozycji tablicy listy elementów o danej wartości funkcji h. Zatem z góry zakładamy tu, że daną wartość funkcji może mieć więcej niż jeden element, i tworzymy z nich listę.



Wyszukiwanie elementu ma czas najgorszego przypadku $\Theta(n)$, a czas dodawania elementu $\Theta(1)$ przy założeniu, że elementu o danym kluczu nie ma w tablicy. Gdy tego założenia nie można zrobić, dodawanie elementu do tablicy można poprzedzić jego wyszukiwaniem, i wtedy czas najgorszego przypadku będzie oczywiście również $\Theta(n)$.

Metoda łańcuchowa — analiza przypadku średniego

Pesymistyczne warianty czasów działania podstawowych operacji na tablicy z haszowaniem z metodą łańcuchową są bardzo niekorzystne — praktycznie niczym się nie różnią od przechowywania elementów na zwykłej liście nieuporządkowanej. Oczywiście nie taki jest cel użycia haszowania.
Pytanie na jakie czasy działania możemy liczyć w przypadku średnim.

Czasy dostępu do tablic z haszowaniem z metodą łańcuchową w przypadku średnim zależą od wielkości tablicy. Wydaje się oczywiste, że jeśli tablica będzie miała stały, ograniczony rozmiar m, a wielkość n zbioru danych w niej zapisywanego będzie rosła, to oczekiwana długość list zapisanych w pozycjach tablicy będzie również rosła. I na odwrót, jeśli rozmiar tablicy m będzie proporcjonalny to wielkości n zbioru danych, to w pozycjach tablicy można spodziewać się niewielkich list, o ograniczonej długości.

Aby wziąć pod uwagę powyższą zależność, definiujemy dla tablicy o m pozycjach, w której znajduje się n elementów, **współczynnik zapełnienia** $\alpha = n/m$. Wyraża on średnią liczbę elementów w jednym łańcuchu. Jego wartość może być mniejsza, równa, lub większa niż 1.

Metoda łańcuchowa — analiza przypadku średniego (cd.)

Istotną odpowiedź na pytanie o czas działania operacji wyszukiwania elementu w tablicy dają twierdzenia 11.1 i 11.2 sformułowane w podręczniku CLRS:

> Tw. 11.1 i 11.2 (CLRS): W tablicy z haszowaniem wykorzystującym łańcuchową metodę rozwiązywania kolizji, przy założeniu o niezależnie jednostajnym haszowaniu, średni czas działania procedury wyszukiwania wynosi $\Theta(1+\alpha)$ zarówno w przypadku wyszukiwania zakończonego sukcesem jak i porażką.

Znaczenie współczynnika α już wyjaśniliśmy, natomiast niejasne pozostaje pojęcie niezależnie jednostajnego haszowania.

Niezależnie jednostajne haszowanie

Pojęcie **niezależnie jednostajnego haszowania** pojawiające się w powyższych twierdzeniach oznacza wyidealizowaną, hipotetyczną funkcję haszującą, która każdemu kluczowi k przyporządkowuje wartość $h(k)$ wybraną losowo z rozkładem jednostajnym z zakresu $\{0, 1, ..., m-1\}$. Ten wybór wartości losowej dokonuje się jednorazowo dla każdego klucza, natomiast każde kolejne wywołanie funkcji h dla tego samego klucza k daje już tę samą, pierwotnie wylosowaną wartość $h(k)$.

Niezależnie jednostajne haszowanie jest więc pewnym modelem wyboru wartości $h(k)$ praktycznie losowych, równomiernie rozłożonych, przez całkowicie deterministyczną funkcję. Taką funkcję bardzo trudno jest zaimplementować w praktyce (o ile jest to w ogóle możliwe), jednak wybór losowy z rozkładem równomiernym pozwala udowadniać konkretne asymptotyczne czasy działania.

Jednocześnie ten model nie jest tak zupełnie niemożliwy do zrealizowania, przynajmniej w przybliżeniu. Zobaczymy funkcje, które zachowują się podobnie do tego modelu. Jednak dla każdej takiej funkcji praktycznej — kiedy nie można przyjąć matematycznego założenia o równomiernym rozkładzie losowym — bardzo trudno jest udowadniać pożądane własności asymptotyczne.

Czas działania haszowania z metodą łańcuchową

Powyższy wynik teoretyczny moglibyśmy podsumować w ten sposób, że gdy dana funkcja haszująca jest dostatecznie bliska modelowemu haszowaniu niezależnie jednostajnemu, to czas wyszukiwania elementu w przypadku średnim będzie wynosił $\Theta(1 + \alpha)$. Jeżeli liczba elementów n w tablicy będzie nie większa niż proporcjonalna do liczby pozycji m, czyli $n = O(m)$, to $\alpha = n/m = O(m)/m = O(1)$.

Uwzględniając, że dodawanie elementu do tablicy działa w czasie $O(1)$ najgorszego przypadku, i usuwanie elementu działa również w czasie $O(1)$ najgorszego przypadku gdy listy są dwukierunkowe i argumentem usuwania jest element (a nie klucz), to można powiedzieć, że przy przyjętych założeniach dotyczącej funkcji haszującej, tablica z haszowaniem spełnia wymagania dotyczące wykonywania operacji słownikowych w czasie stałym $O(1)$ w przypadku średnim.

Tworzenie funkcji haszujących

Dobra funkcja haszująca powinna spełniać założenia haszowania niezależnie jednostajnego, to znaczy odwzorowywać klucze z jednakowym prawdopodobieństwem do każdej z pozycji tablicy, niezależnie od tego gdzie zostają odwzorowane inne klucze.

To wymaganie trudno jest spełnić gdy nie mamy informacji o rozkładzie prawdopodobieństwa z jakim będą wybierane klucze.

Pytanie czym w praktyce są klucze. W typowych zastosowaniach mogą to być:

- liczby całkowite z pewnego przedziału,
- liczby rzeczywiste z pewnego przedziału,
- napisy znakowej o zmiennej długości.

Na początek będziemy zakładać, że klucze są nieujemnymi liczbami całkowitymi o ograniczonym zakresie. Najpierw rozważymy grupę **statycznych funkcji haszujących**, dla których jedynie w przybliżeniu można zakładać, że spełnią stawiane wyżej wymagania. Następnie bardzo pobieżnie przyjrzymy się podejściu do **haszowania randomizowanego**, o znacznie lepszych własnościach. Z kolei zapoznamy się z obsługą kluczy o zmiennej długości.

Metoda dzielenia

Metoda dzielenia (zwana również **haszowaniem modularnym**) zakłada wykorzystanie reszty z dzielenia wartości klucza k przez wielkość tablicy m:

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

Jest to prosta i szybka metoda, która zarazem nie daje żadnych gwarancji dobrej oczekiwanej efektywności. Wiadomo jednak, że metoda częściej przynosi dobre efekty gdy m jest liczbą pierwszą, niezbyt bliską jakiejkolwiek potędze dwójki.

Metoda mnożenia

Metoda mnożenia wykorzystuje trochę bardziej skomplikowany wzór odwzorowujący wartość klucza w pozycję tablicy. Najpierw wartość klucza mnoży się przez pewną stałą A z przedziału $0 < A < 1$, a następnie wybiera część ułamkową wyniku. Ten ułamek pomnożony przez pojemność tablicy m daje pozycję klucza (po zaokrągleniu funkcją podłogi), co można zapisać wzorem:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

gdzie wyrażenie $(kA \bmod 1)$ oznacza wzięcie części ułamkowej z kA, czyli wartość $kA - \lfloor kA \rfloor$. Zaletą haszowania przez mnożenie jest to, że wybór wartości m przestaje być krytyczny, i może być niezależny od wyboru stałej multiplikatywnej A.

Metoda mnożenia z przesunięciem

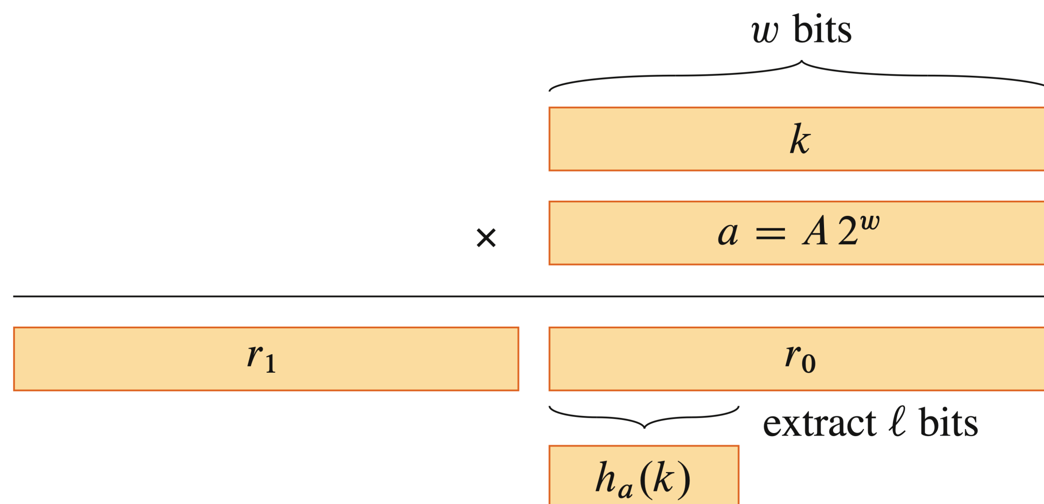
W praktyce haszowanie przez mnożenie sprawdza się najlepiej w pewnym szczególnym przypadku, gdy rozmiar tablicy $m = 2^l$ dla pewnego dodatniego $l \leq w$, gdzie w jest liczbą bitów w słowie maszynowym.

Jeśli ustalimy dodatnią w-bitową liczbę całkowitą $a = A2^w$, gdzie $0 < A < 1$ tak jak w podstawowej metodzie mnożenia, tzn. a jest z zakresu $0 < a < 2^w$, to funkcję haszującą można zdefiniować wzorem:

$$h_a(k) = (ka \bmod 2^w) \ggg (w - l)$$

Klucz k jest mnożony przez stałą a, obie wartości o w bitach, co daje $2w$-bitowy wynik zapisany w słowach r_1 i r_0.

Starsze słowo r_1 o w bitach jest odrzucane wyrażeniem $\bmod 2^w$, a z młodszego słowa r_0 wybieramy l najstarszych bitów przesunięciem $\ggg (w - l)$.



Metoda mnożenia z przesunięciem: przykład i własności

Rozważmy przykład obliczenia funkcji haszującej przez mnożenie z przesunięciem. Przyjmijmy $k = 123456, l = 14, m = 2^{14} = 16384$, i $w = 32$. Ponadto weźmy wartość $a = 2654435769$. Wtedy $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, to znaczy $r_1 = 76300$ i $r_0 = 17612864$. Biorąc 14 najstarszych bitów z r_0 otrzymujemy wartość funkcji haszującej $h_a(k) = 00000001000011_2 = 67$.

Mimo iż procedura wygląda na skomplikowaną, wartość funkcji haszującej można zaimplementować za pomocą trzech rozkazów maszynowych: mnożenia, odejmowania(???), i przesunięcia bitowego, a więc jest ona szybka (i prosta).

Metoda mnożenia z przesunięciem zasadniczo nie daje żadnych gwarancji wydajności w przypadku średnim. Jednak obecność stałej a oznacza, że jest to praktycznie rodzina funkcji haszujących. Jak wkrótce zobaczymy, jeśli dla tej wartości wybierzemy losowo nieparzystą liczbę całkowitą, to okazuje się, że metoda działa bardzo dobrze w przypadku średnim.

Haszowanie randomizowane

Określenie **haszowanie randomizowane** określa ogólne podejście, w którym funkcja haszująca jest wybierana losowo z pewnej rodziny funkcji $\mathcal{H}$. Dzięki temu losowemu wyborowi, żadne konkretne dane nie prowadzą do najgorszego przypadku zachowania funkcji haszującej, co gwarantuje dobre zachowanie tablicy z haszowaniem w przypadku średnim. Co to znaczy dobre zachowanie zależy od konkretnej rodziny funkcji $\mathcal{H}$.

Szczególny przypadek haszowania randomizowanego stanowi **haszowanie uniwersalne**. Rodzinę funkcji haszujących $\mathcal{H}$ odwzorowujących uniwersum kluczy U w przedział $\{0, 1, ...m-1\}$ nazywamy **uniwersalną**, jeśli dla każdej pary różnych kluczy $k_1, k_2 \in U$ liczba funkcji haszujących $h \in \mathcal{H}$, dla których $h(k_1) = h(k_2)$ wynosi co najwyżej $|\mathcal{H}|/m$. Inaczej mówiąc, dla losowo wybranej funkcji $h \in \mathcal{H}$ prawdopodobieństwo kolizji między dowolnymi dwoma różnymi kluczami k_1 i k_2 jest nie większe niż gdyby klucze były wybrane losowo i niezależnie z przedziału $\{0, 1, ..., m-1\}$ $(1/m)$.

Przypomnijmy, że zawsze jest możliwy przypadek pesymistyczny, w którym funkcja wylosowana z rodziny uniwersalnej $\mathcal{H}$ odwzoruje wszystkie klucze (lub ich zdecydowaną większość) w jedną wartość $h(k)$ prowadząc do czasu wyszukiwania klucza $\Omega(n)$. Jednak jest to bardzo mało prawdopodobne $\left((1/m)^n\right)$, i nie da się celowo skonstruować zbioru kluczy prowadzącego do takiego zachowania.

Uniwersalna rodzina funkcji haszujących oparta na teorii liczb

Pierwszą metodę budowy uniwersalnej rodziny funkcji haszującej przedstawiamy bez uzasadnienia ani intuicji stojących za taką konstrukcją.

Zaczynamy od wybrania liczby pierwszej p na tyle dużej, aby wszystkie klucze mieściły się w przedziale od 0 do $p-1$. Przyjmijmy $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ i $\mathbb{Z}_p^* = \{1, ..., p-1\}$.

Dla dowolnej pary liczb $a \in \mathbb{Z}_p^*$ i $b \in \mathbb{Z}_p$ definiujemy funkcję haszującą:

$$h_{ab} = ((ak + b) \bmod p) \bmod m$$

Przykład: dla $p = 17$ i $m = 6$ mamy:

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5. \end{aligned}$$

Dla danych wartości p i m rodziną funkcji jest $\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$
Każda taka rodzina zawiera $p(p-1)$ funkcji haszujących.

Uniwersalna rodzina funkcji haszujących oparta na mnożeniu z przesunięciem

Drugą rodzinę uniwersalnych funkcji haszujących stanowią funkcje mnożenia z przesunięciem opisane wcześniej:

$$\mathcal{H} = \{h_a : a \text{ jest nieparzyste}, 1 \leq a < m, h_a \text{ jest funkcją mnożenia z przesunięciem}\}$$

Dla nieparzystych a można udowodnić, że prawdopodobieństwo kolizji dwóch różnych kluczy nie przekracza wartości $2/m$. Tych funkcji jest $\lfloor m/2 \rfloor$.

Kryptograficzne funkcje skrótu

Funkcje haszujące znalazły szereg zastosowań wykraczających poza obsługę tablic z haszowaniem (np. system haszówek). Ważna grupa zastosowań obejmuje **kryptograficzne funkcje skrótu**. W kryptografii potrzebna jest możliwość generowania stałej długości kodów dla dowolnych ciągów danych, zwanych **dokumentami**. Dokumentem może być dowolny ciąg bajtów, może to być dokument tekstowy albo binarny (np. PDF), albo dowolny inny plik, np. obraz w formacie JPG, albo film w formacie MP4 lub DVD. Te kody można traktować jako **sygnatury dokumentów**, ponieważ ich podstawową własnością jest różnowartościowość, to znaczy dla różnych argumentów otrzymujemy różne wartości skrótu.

Wymagania dla kryptograficznych funkcji skrótu:

- stały, niezależny od argumentu rozmiar wyniku,
- niemożność odtworzenia dowolnej części argumentu na podstawie wartości funkcji,
- niemożność wykrycia na podstawie wartości funkcji jakiejkolwiek korelacji pomiędzy wartościami argumentu (np. 1-bitowej różnicy),
- astronomicznie małe prawdopodobieństwo kolizji.

Haszowanie z użyciem kryptograficznych funkcji skrótu

Jak widać, kryptograficzne funkcje skrótu mają szereg wymagań znacznie przekraczających wymagania dla zwykłych funkcji haszujących, np. praktyczny brak kolizji. Związane jest to, między innymi, z zakresem wartości tych funkcji, które nie są ograniczone wielkością żadnej tablicy, a wręcz przeciwnie, mogą być znacznie dłuższe.

Na przykład, uznany za przestarzały i niezbyt bezpieczny algorytm skrótu MD5 generuje skróty 128-bitowe, natomiast bardziej współczesna rodzina funkcji SHA2 generuje skróty zaczynające się od 256-bitowych.

Można jednak „zatrudnić” kryptograficzne funkcje skrótu do zastosowań w tablicach z haszowaniem, na przykład definiując (w przypadku wartości m będącej potęgą dwójki operacja modulo w reprezentacji binarnej sprowadza się do odcięcia starszych bitów):

$$h(k) = \text{ SHA-256}(k) \bmod m$$

Aby zbudować rodzinę takich funkcji można zastosować **sól**, to znaczy losowo wygenerowany string doklejony do klucza (operacją konkatenacji napisów $||$).

$$h_a(k) = \text{ SHA-256}(a||k) \bmod m$$

Dygresja — przechowywanie haseł

Przykładem zastosowania kryptograficznych funkcji skrótu jest przechowywanie haseł w systemach komputerowych. Chcąc zabezpieczyć przechowywane hasła przed możliwością wykradzenia można, zamiast przechowywać wersji jawnej każdego hasła, przechowywać tylko obliczone skróty, i każdorazowo porównywać je ze skrótem obliczonym z hasła podanego przez użytkownika. Dzięki praktycznie zerowemu prawdopodobieństwu kolizji, zgodność obliczonego skrótu z zapamiętaną wersją oznacza zgodność podanego hasła z tym pierwotnie ustalonym.

Jednak problemem w kryptografii są tzw. ataki słownikowe. Jeśli atakujący pozyska bazę haseł, nawet w postaci skrótów, to może wygenerować wiele popularnych haseł, takich jak basia123, i innych typowych kombinacji istniejących słów, imion, nazw własnych, itp. Atakujący może wręcz wstępnie wygenerować bazę skrótów ze wszystkich sensownie zbudowanych haseł. Taki proces może zająć bardzo dużo czasu procesora, ale może być przeprowadzony *off-line*.

Wtedy atak na dowolną pozyskaną/wykradzioną bazę skrótów polega już tylko na porównywaniu znalezionych skrótów z tymi z wygenerowanej bazy. Skutkiem takiego ataku jest odkrycie wszystkich słabych haseł.

Sytuację poprawia zastosowanie **soli**, która jest losowo wygenerowanym napisem dodanym do hasła użytkownika przed obliczeniem skrótu. Sól jest losowa, ale jest zapamiętana w postaci jawnej razem ze skrótem, aby po podaniu prawidłowego hasła przez użytkownika można było ponownie dodać do niego tę samą sól, i obliczyć skrót. Wtedy jednak atak słownikowy ze wstępnie wygenerowaną bazą skrótów nie jest już możliwy, i w przypadku wykradzenia bazy skrótów atakujący musi haszować na bieżąco wszystkie możliwe hasła ze znalezionymi wartościami soli.

Adresowanie otwarte

Adresowanie otwarte jest metodą rozwiązywania kolizji. W odróżnieniu od poznanej wcześniej metody łańcuchowej, adresowanie otwarte nie wykorzystuje dodatkowej pamięci na przechowywanie kluczy. Wszystkie klucze zapisywane są bezpośrednio w tablicy. Oczywiście w tym celu potrzebna jest wyróżniona nielegalna wartość klucza (NIL) jako znacznika pustych pozycji w tablicy.

Idea adresowania otwartego jest następująca. Każdy klucz jest wpisywany w pozycję tablicy zgodną z obliczoną wartością funkcji haszującej, o ile ta pozycja jest wolna.

Jeśli pozycja obliczona dla danego klucza jest już zajęta w tablicy (kolizja!) to rozpoczyna się procedura **próbkowania**, to znaczy poszukiwania innego miejsca. Jeśli to miejsce jest wolne, to klucz jest tam wpisywany. W przeciwnym wypadku, próbkowanie jest kontynuowane, aż do znalezienia wolnego miejsca.

Zatem czas wykonywania operacji zapisu elementu do tablicy nie będzie już jednostkowy, tylko może się wydłużyć, w zależności od metody próbkowania.

Analogicznie, przy wyszukiwaniu albo usuwaniu klucza, też musimy użyć próbkowania.

Jeśli poszukiwanego klucza nie ma na pozycji wyznaczonej przez funkcję haszującą, i jest tam wartość NIL, to elementu nie ma w tablicy. Ale jeśli jest wpisany inny klucz, to musimy próbkować, aby albo znaleźć właściwy klucz, albo ustalić, że go nie ma.

Otwarte adresowanie ma również inne konsekwencje. Nigdy nie uda się zapisać w tablicy więcej kluczy niż jest możliwych pozycji (współczynnik zapełnienia $\alpha \leq 1$). W metodzie łańcuchowej było to możliwe.

Inną konsekwencją jest odmienne podejście do usuwania kluczy. Gdy usuwamy klucz z tablicy, nie możemy wpisać w jego pozycję wartości NIL. Albowiem istnieje możliwość, że inny klucz miał tę samą wartość funkcji haszującej, ale został wpisany na inną pozycję, bo ta pierwotna była zajęta. Przy jego wyszukiwaniu, znaleziona wartość NIL mogłaby świadczyć, że klucza nie ma w tablicy. Musimy użyć innej wartości znacznikowej, np. DEL (symbolicznie: wartość skasowana). Ta wartość oznacza, że pozycja tablicy jest wolna, można w nią wpisywać nowe klucze, ale przy wyszukiwaniu należy ją traktować jako zajętą, i poszukując danego klucza rozpocząć, lub kontynuować, próbkowanie.

Próbkowanie liniowe

Najprostszą metodą próbkowania jest **próbkowanie liniowe**. Polega ono na przejściu do kolejnej pozycji tablicy $(h(k) + 1)$, i potem znowu do kolejnej, itd., a po osiągnięciu końca tablicy kontynuowanie od jej początku.

Ciekawostka:
Zastosowanie próbkowania liniowego pozwala zaimplementować względnie efektywną metodę usuwania elementu z odtworzeniem na zwolnione miejsce elementu(ów) wcześniej umieszczonych na nieswoim(ich) miejscu(ach) przez próbkowanie. Metoda działa jedynie z próbkowaniem liniowym, ale nie z bardziej ogólną metodą haszowania podwójnego (poniżej). Jej zastosowanie pozwala usprawnić przyszłe przeszukiwania przez ograniczenie próbkowania. Procedura nie będzie tu omawiana (patrz sekcja 11.5.1 podręcznika CLRS).

Haszowanie podwójne

Bardziej wyrafinowanym podejściem do próbkowania jest **haszowanie podwójne**. Wymaga ono zdefiniowania dwóch pomocniczych funkcji haszujących h_1 i h_2, wykorzystywanych łącznie do obliczenia właściwego hasza dla klucza:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

gdzie i jest numerem kroku próbkowania.

Jak widać, w kolejnych krokach próbkowania znaleziona pozycja klucza będzie się oddalała od pierwotnej o „kroki" wyznaczone przez funkcję h_2.

Można więc uznać, że próbkowanie liniowe jest prostą wersją haszowania podwójnego z funkcją $h_2(k) \equiv 1$.

Efektywność adresowania otwartego

Można łatwo zgadnąć, że efektywność adresowania otwartego będzie zależała od stopnia zapełnienia tablicy $\alpha = n/m$. Musi ono być ułamkiem właściwym, ale przy bardzo mocno wypełnionej tablicy, z wartością α bliską 1, można spodziewać się wielu kolizji, w miarę jak przy wstawianiu (albo poszukiwaniu) danego klucza kolejne pozycje są próbkowane, i prawie wszystkie okazują się zajęte.

Na odwrót, gdy tablica ma ogromny rozmiar, w porównaniu z liczbą kluczy, i współczynnik wypełnienia jest bliski zera, to można się spodziewać, że prawie każde wstawianie elementu napotka na pustą pozycję w tablicy, i kolizji, a w konsekwencji próbkowania, nie będzie wcale.

Rzeczywiście, można udowodnić, że przy zastosowaniu adresowania otwartego, oraz założeniu haszowania niezależnie jednostajnego, oczekiwana liczba porównań przy poszukiwaniu elementu, którego nie ma w tablicy nie przekracza $1/(1-\alpha)$.

Przy tych samych założeniach, oczekiwana liczba porównań przy poszukiwaniu elementu, który znajduje się w tablicy nie przekracza

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

(Ten wzór budzi wątpliwości dla $\alpha \approx 0$, jednak wartości okazują się bliskie 1.)

Krótkie podsumowanie — pytania sprawdzające

1. Dla tablicy z haszowaniem o 9 pozycjach z funkcją haszującą $h(k) = k \bmod 9$ zilustruj ciąg wstawień elementów o kluczach 5, 28, 19, 15, 20, 33, 12, 17, i 10, przy założeniu rozwiązywania kolizji metodą łańcuchową.

2. Profesor Marley wysunął hipotezę, że posortowanie list w metodzie łańcuchowej poprawiłoby efektywność tablicy z haszowaniem. Odpowiedz jak modyfikacja profesora wpłynie na czas wykonywania operacji wstawienia, usunięcia, oraz wyszukiwania, oddzielnie zakończonego sukcesem i porażką.

3. Rozważ tablicę z haszowaniem o rozmiarze $m = 1000$ oraz funkcję haszującą $h(k) = \lfloor m(kA \bmod 1) \rfloor$ dla $A = (\sqrt{5} - 1)/2$. Oblicz pozycje, na które trafią klucze 61, 62, 63, 64, i 65.

4. Rozważ wstawianie kluczy 10, 22, 31, 4, 15, 28, 17, 88, i 59 do tablicy z haszowaniem o długości $m = 11$ z otwartym adresowaniem. Pokaż wynik wstawiania tych kluczy przy użyciu najpierw próbkowania liniowego $h(k, i) = (k + i) \bmod m$ a następnie haszowania podwójnego z $h_1(k) = k$, oraz $h_2(k) = 1 + (k \bmod (m - 1))$.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 11.