

Hashing. Aplicații

*Alexandru Cohal
Decembrie 2012*

Cuprins

 Introducere	3
 Funcții hash	5
 Metoda Diviziunii (restul împărțirii la m)	5
 Metoda înmulțirii	5
 Metoda probabilistică (Hashing-ul universal)	5
 Interpretarea cheilor ca numere naturale	6
 Coliziunile	6
 Chaining (înlănțuire)	6
 Open-addressing (adresare deschisă)	7
 Distribuția cheilor	8
 Examinarea liniară	8
 Examinarea pătratică	8
 Hashing dublu	8
 Hashing-ul perfect	9
 Hash-uri în STL	10
 Aplicații	10
 Algoritmul Rabin - Karp	10
 Alte probleme	11
 Legături	11
 Bibliografie	11



Introducere

Multe aplicații necesită o mulțime dinamică pentru care să se aplice numai operațiile specifice pentru dicționare: *inserează*, *caută*, *șterge*.

O *tabelă de dispersie* (*tabelă hash*) este o structură eficientă de date pentru implementarea dicționarelor.

O *tabelă de dispersie* este o generalizare a noțiunii mai simple de tablou.

Adresarea directă într-un tablou folosește abilitatea noastră de a examina o poziție arbitrară în timp $O(1)$. Adresarea directă este aplicabilă numai în cazul în care ne permitem să alocăm un tablou care are câte o poziție pentru fiecare cheie posibilă.

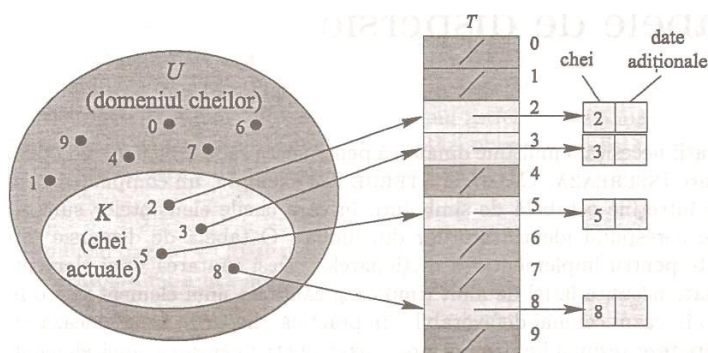


Figura 1.

Implementarea unei mulțimi dinamice printr-un tablou cu **adresare directă** T .

Fiecare cheie din universul $U = 0, 1, 2, \dots, 9$ corespunde unui indice în tablou.

Mulțimea $K = 2, 3, 5, 8$ a cheilor efective determină locațiile din tablou care conțin pointeri către elemente.

Celelalte locații, hașurate mai întunecat, conțin *NIL*.

Când numărul cheilor memorate efectiv este relativ mic față de numărul total de chei posibile, **tabelele de dispersie** devin o alternativă eficientă la adresarea directă într-un tablou, deoarece se folosește un tablou de mărime proporțională cu numărul de chei memorate efectiv.

În loc să se folosească direct cheia ca indice în tablou, indicele este calculat pe baza cheii, folosind o *funcție de dispersie* (**hash**), care face maparea spațiului de chei într-un spațiu de adrese.

Funcția generează o adresă printr-un calcul simplu, aritmetic sau logic asupra cheii sau asupra unei părți a cheii.

Să considerăm A o mulțime cu n valori care trebuie stocate în structura de date (aceste valori sunt denumite **chei**) și T un vector cu m componente (denumit **tabel hash**).

Vom considera o **funcție hash** prin care se asociază fiecărei chei un număr întreg din intervalul $[0, m - 1]$:

$$h : A \rightarrow [0, m - 1]$$

Funcția *hash* are proprietatea că valorile ei sunt uniform distribuite în intervalul specificat. Aceste valori vor fi utilizate ca indici în vectorul T . Mai exact, cheia x va fi plasată în tabela hash pe poziția $h(x)$.

Datorită faptului că spațiul de chei este de obicei mult mai mare decât spațiul de adrese, se poate întâmpla ca mai multe chei să aibă aceeași adresă. Aceasta se numește **coliziune** între înregistrări.

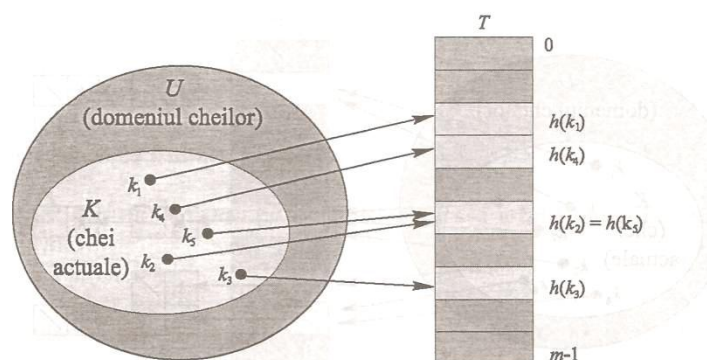


Figura 2.

Folosirea unei **funcții de dispersie** h pentru a transforma chei în poziții din tabela de dispersie. Cheile k_2 și k_5 se transformă în aceeași poziție, deci sunt în coliziune.



Funcții hash



Metoda Diviziunii (restul împărțirii la m)

$$h(x) = x \% m$$

În acest caz este indicat să evităm ca m să fie de forma 2^p (pentru că atunci $h(x)$ ar avea ca valoare ultimii p biți ai lui x). Exceptând cazul în care știm că toate secvențele binare de lungime p apar cu aceeași probabilitate ca ultimi p biți ai cheii, este de preferat să utilizăm o funcție în care se utilizează toți biții cheii.

Se alege m ca fiind un număr prim pentru ca înmulțirea a doi termeni care au suferit operația modulo să dea rezultat nul numai dacă unul dintre aceștia este nul.



Metoda înmulțirii

$$h(x) = [m * (\text{frac}(x * y))]]$$

Se înmulțește cheia x cu un număr subunitar $0 < y < 1$, considerăm partea fracționară a lui $x * y$, o înmulțim cu m și reținem doar partea întreagă a rezultatului.

Un avantaj al acestei metode este că nu există valori particulare pentru m care să trebuiască evitate ca la *Metoda Diviziunii*, prin urmare de obicei m poate fi o putere a lui 2, pentru a implementa eficient funcția hash.



Metoda probabilistică (Hashing-ul universal)

Dacă un adversar “răutăcios” va alege cheile astfel încât toate să aibă aceeași valoare pentru o anumită funcție hash, atunci obținem un timp de execuție liniar. Soluția ar fi să alegem o funcție hash independentă de cheile ce urmează a fi stocate.

Ideea de bază este să selectăm funcția hash la întâmplare dintr-o listă predefinită de funcții hash la începutul execuției programului. Se numește familie universală de funcții hash o mulțime H de funcții definite pe A cu valori în $\{0, 1, \dots, m-1\}$ astfel încât pentru orice pereche de chei x, y ($x \neq y$) numărul funcțiilor hash din familie pentru care $h(x) = h(y)$ este cel mult $\frac{|H|}{m}$. Cu alte cuvinte, dacă alegem aleator o funcție din această familie, probabilitatea de obține o coliziune pentru cheile x și y este mai mică decât $\frac{1}{m}$.

Un exemplu care se comportă foarte bine în practică este următorul: se generează aleator un număr natural impar r . Valoarea funcției pentru un întreg k se află înmulțind k cu r și păstrând cei mai semnificativi p biți ai rezultatului (înmulțirea se face pe 32 de biți, și ignorăm overflow).



Interpretarea cheilor ca numere naturale

Majoritatea funcțiilor de dispersie presupun universul cheilor din mulțimea $\mathbb{N} = 0, 1, 2, \dots$ a numerelor naturale. Astfel, dacă cheile **nu** sunt numere naturale, trebuie găsită o modalitate pentru a le interpreta ca numere naturale.

De exemplu, o cheie care este un *șir de caractere* poate fi interpretată ca un întreg într-o bază de numerație aleasă convenabil. Prin urmare, identificatorul *pt* poate fi interpretat ca o pereche de numere zecimale (112, 116), pentru că $p = 112$ și $t = 116$ în mulțimea codurilor ASCII; atunci *pt* exprimat ca un întreg în baza 128 devine $(112 \cdot 128) + 116 = 14452$.

În mod obișnuit, în orice aplicație se poate crea direct o asemenea metodă simplă de a interpreta fiecare cheie ca un număr natural (posibil mare).



Coliziunile

Așa cum spuneam și mai devreme, datorită faptului că spațiul de chei este de obicei mult mai mare decât spațiul de adrese, se poate întâmpla ca mai multe chei să aibă aceeași adresă. Aceasta se numește **coliziune** între înregistrări.

Prin urmare este necesar și un mecanism de rezolvare a *coliziunilor*. Există două metode de rezolvare a coliziunilor.



Chaining (înlănțuire)

Toate valorile sinonime (care au aceeași valoare *hash*) vor fi plasate într-o listă înlănțuită. Astfel în vectorul T pe poziția i vom reține un pointer către începutul listei înlănțuite formate din toate valorile x pentru care $h(x) = i$.

Operația de inserare se execută în $O(1)$ (se inserează valoarea x la începutul listei $T[h(x)]$).

Operația de căutare presupune căutarea elementului cu cheia x în lista înlănțuită $T[h(x)]$.

Operația de ștergere presupune căutarea elementului cu cheia x în lista înlănțuită $T[h(x)]$, apoi ștergerea din listă a acestuia.

Ultimele două operații au timpul de execuție proporțional cu lungimea listei $T[h(x)]$. Analizând complexitatea în cazul cel mai defavorabil, lungimea listei poate fi $O(n)$ (atunci când toate elementele sunt sinonime).

Analizând complexitatea în medie, observăm că lungimea listei depinde de cât de uniform distribuie *funcția hash* cheile.

Dacă presupunem că *funcția hash* va distribui uniform valorile, atunci lungimea medie a listei este $\frac{n}{m}$ (această valoare este denumită factor de încărcare a tabelii), deci timpul de execuție va fi $O(1 + \frac{n}{m})$.



Open-addressing (adresare deschisă)

În tabela *hash* nu sunt memorati pointeri, ci sunt memorate elementele mulțimii *A*. Astfel în tabelă se vor afla poziții ocupate cu elemente din *A* și poziții libere (acestea vor fi marcate cu o valoare specială pe care o vom denumi *NIL*).

În acest caz, pentru a insera un element *x* în tabela *hash* se vor executa mai multe încercări, până când determinăm o poziție liberă, în care putem plasa elementul *x*.

În acest caz, funcția *hash* va avea doi parametri: cheia *x* și numărul încercării curente (numerotarea încercărilor va începe de la 0):

$$h : A \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

Operația de inserare: Pentru a insera elementul *x* în tabela *T* se examinează în ordine pozițiile $h(x, 0)$, $h(x, 1)$, ..., $h(x, m - 1)$. Această secvență de poziții trebuie să fie o permutare a mulțimii $\{0, 1, \dots, m - 1\}$.

Operația de căutare: Căutarea unui element în tabelă presupune examinarea aceleiași secvențe de poziții ca și inserarea. Dacă în tabel nu se efectuează ștergeri, căutarea se termină fără succes la întâlnirea primei poziții libere.

Operația de ștergere: Ștergerea unui element dintr-o tabelă *hash* este dificilă. Nu putem doar marca poziția *i* pe care este plasată cheia *x* ca fiind liberă. În acest mod, căutare oricărei chei pentru care pe parcursul inserării am examinat poziția *i* și am găsit-o ocupată se va termina fără succes.

O soluție ar fi ca la ștergerea unui element pe poziția respectivă să nu plasăm valoarea *NIL*, ci o altă valoare specială (pe care o vom denumi *DELETED*). Funcția de inserare poate fi modificată astfel încât să insereze elementul *x* pe prima poziție pe care se află *NIL* sau *DELETED*.

Dacă alegem această soluție, timpul de execuție al căutării nu mai este proporțional cu factorul de încărcare a tabeli. Din acest motiv, tehnica de rezolvare a *coliziunilor* prin *înlănțuire* este mai utilizată pentru cazul în care structura de date suportă și operații de ștergere.



Distribuția cheilor

Pentru ca distribuția cheilor să fie uniformă, funcția *hash* trebuie să genereze cu egală probabilitate oricare dintre cele $m!$ permutări ale mulțimii $\{1, 2, \dots, m - 1\}$.

Pentru a determina secvența pozițiilor de examinat se utilizează 3 tehnici. Fiecare dintre aceste 3 tehnici garantează că pentru orice cheie secvența pozițiilor de examinat este o permutare a mulțimii $\{0, 1, \dots, m - 1\}$. Nici una dintre acestea nu asigură o distribuție perfect uniformă.



Examinarea liniară

Considerând o funcție *hash* “obișnuită”

$$h': U \rightarrow \{0, 1, \dots, m - 1\}$$

(denumită funcție *hash* auxiliară), tehnica examinării liniare utilizează funcția:

$$h(x, i) = (h'(x) + i) \% m.$$

Observație

➡ Deoarece prima poziție din secvență determină întreaga secvență de poziții de examinat, deducem că există doar m secvențe de examinare distincte.

Examinarea liniară este ușor de implementat, dar generează un fenomen denumit *primary clustering* (secvența de poziții ocupate devine mare, și astfel timpul mediu de execuție al căutării crește).



Examinarea pătratică

Examinarea pătratică utilizează o funcție *hash* de forma:

$$h(x, i) = h'(x) + c_1 i + c_2 i^2 \% m$$

unde h' este funcția *hash* auxiliară, iar c_1 și $c_2 \neq 0$ sunt constante auxiliare.

Observație

➡ Această metodă funcționează mai bine decât examinarea liniară, dar pentru a garanta faptul că întreaga tabelă este utilizată, m , c_1 și c_2 trebuie să îndeplinească anumite condiții.



Hashing dublu

Hashing-ul dublu utilizează o funcție de forma:

$$h(x, i) = h_1 x + i h_2(x) \% m$$

unde h_1 și h_2 sunt funcții *hash* auxiliare.

Pentru ca întreaga tabelă să fie examinată, valoarea $h_2(x)$ trebuie să fie relativ primă cu m . O metodă sigură este de a-l alege pe m o putere a lui 2, iar h_2 să producă numai valori impare. Sau m să fie un număr prim, iar h_2 să producă numere $< m$.

Observație

➡ Hashingul dublu este mai eficient decât cel liniar sau pătratic.



Hashing-ul perfect

Există aplicații în care mulțimea cheilor nu se schimbă (de exemplu, cuvintele rezervate ale unui limbaj de programare, numele fișierelor scrise pe un CD, etc). Hashing-ul poate fi utilizat și pentru probleme în care mulțimea cheilor este statică (odată ce au fost inserate cheile în tabela *hash*, aceasta nu se mai schimbă). În acest caz, timpul de execuție în cazul cel mai defavorabil este $O(1)$.

Hashing-ul se numește *hashing perfect* dacă nu există coliziuni.

Hashing-ul perfect se realizează pe două niveluri.

La primul nivel, este similar cu hashing-ul cu înlănțuire: cele n chei vor fi distribuite cu ajutorul unei funcții *hash* h în m locații.

Dar în loc de a crea o listă înlănțuită cu toate cheile distribuite în locația i , la cel de **al doilea nivel** vom utiliza câte o tabelă *hash* secundară S_i care are asociată o funcție *hash* h_i pentru fiecare locație în care există coliziuni.

Pentru a garanta faptul că nu vor exista coliziuni în tabela *hash* secundară, dimensiunea m_i a tabelii *hash* secundare S_i trebuie să fie cel puțin egală cu n_i^2 (unde n_i este numărul de chei care sunt distribuite în locația i).

Evident, descrierea grupurilor de la primul nivel trebuie să conțină și funcția *hash* aleasă pentru grupul respectiv, precum și dimensiunea spațiului de memorie alocat tabelii *hash* secundare.

O alegere a funcției *hash* h dintr-o familie universală de funcții *hash* asigură că dimensiunea totală a spațiului de memorie utilizat este $O(n)$.



Hash-uri în STL

- [hash](#)
- [hash_map](#)
- [hash_multimap](#)
- [hash_set](#)
- [hash_multiset](#)



Aplicații



Algoritmul Rabin - Karp

Fie T un șir de n caractere și P un pattern de m caractere.
Să se verifice dacă P apare sau nu ca subsecvență în șirul T .

Soluție:

Fie b numărul de caractere distincte care apar în T . Un șir de lungime m poate fi considerat un număr în baza $b + 1$ având m cifre (se folosesc cifre de la 1 la b).

Se asociază pattern-ului P o valoare p (șirul de caractere este considerat ca un număr întreg în baza $b + 1$).

Pentru fiecare secvență de m caractere din șirul T se calculează t_i , valoarea în baza $b + 1$ a subsecvenței din T care începe la poziția i ($T[i \dots i + m - 1]$). Evident, $p = t_i$ dacă și numai dacă $P = T[i \dots i + m - 1]$.

Valorile t_i pentru $i = 1, \dots, n - m + 1$ se pot calcula astfel:

$$t_{i+1} = (b + 1) * (t_i - (b + 1)^{m-1} * T[i]) + T[i + m].$$

Singura dificultate constă în faptul că p și t_i pot fi numere foarte mari. Prin urmare, operațiile cu ele nu se execută în timp constant.

Se va lucra cu p și $t_i \bmod Q$, unde Q este un număr prim convenabil ales, așadar se folosește o funcție de dispersie. Pot apărea probleme deoarece $p \bmod Q = t_i \bmod Q$ nu implică faptul că $p = t_i$ (apar coliziuni). Dar dacă $p \bmod Q \neq t_i \bmod Q$, atunci sigur $p \neq t_i$.

Așadar, algoritmul *Rabin - Karp* este unul probabilistic, eficiența în detectarea potrivirii pattern-ului depinzând de funcția de dispersie. Pentru a elimina potrivirile "false", se poate aplica o verificare normală pentru fiecare potrivire detectată sau se pot folosi mai multe funcții de dispersie.



Alte probleme

- [Hashuri](#) (Infoarena)
- [Loto](#) (Infoarena)
- [Eqs](#) (Infoarena)
- [Oite](#) (Infoarena)
- [Strmatch](#) (Infoarena)
- [Abc2](#) (Infoarena)
- [String](#) (Infoarena)
- [Circular](#) (Campion)
- [Repeat](#) (Campion)

- [Registration system](#) (Codeforces)
- [Ograzi](#) (Infoarena)
- [Rk](#) (Infoarena)
- [Patrate3](#) (Infoarena)
- [Flori2](#) (Infoarena)
- [Banana](#) (Infoarena)
- [Take5](#) (Infoarena)
- [Radio2](#) (Infoarena)
- [Petr#](#) (Codeforces)



Legături

- MIT - [Lecture 7: Hashing, Hash Functions](#)
- MIT - [Lecture 8: Universal Hashing, Perfect Hashing](#)



Bibliografie

- Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, “*Introducere în algoritmi*”, Editura Computer Libris Agora, Cluj Napoca, 2000
- Emanuela Cerchez, Marinel Șerban, “*Programarea în limbajul C/C++ pentru liceu*” volumul III, Editura Polirom, Iași, 2006
- Dana Lica, Mircea Pașoi, “*Fundamentele programării*” volumul III, Editura L&S SOFT, București, 2006.

Alexandru Cohal
alexandru.c04@gmail.com
alexandru.cohal@yahoo.com
Decembrie 2012