Hashing

prof. Emanuela Cerchez

Multe aplicații necesită o structură de date dinamică, care să permită executarea eficientă a operațiilor *Insert*, *Delete* şi *Search*. O astfel de structură de date poartă numele de *dicționar*.

Tabelele *hash* sunt o variantă de implementare a unui dicționar, foarte uşor de implementat şi frecvent utilizată în practică. În cazul cel mai defavorabil, o tabelă *hash* se poate comporta ca o listă simplu înlănțuită (deci operațiile specificate se execută în timp liniar). Dar în practică, utilizând *hashing* cele 3 operații se execută foarte eficient.

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

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

`h: A→{0, 1, ..., 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)`.

Funcții hash

O funcție *hash* bună trebuie să distribuie uniform cheile în tabela *hash*. Din păcate această condiție este greu de îndeplinit, pentru că nu cunoaştem structura cheilor.

În general, funcțiile *hash* sunt definite pe mulțimea numerelor naturale (cheile sunt considerate numere naturale). În cazul în care cheile nu sunt numere naturale, ele pot fi transformate în numere naturale.

De exemplu, pentru o cheie şir de caractere, putem considera că şirul este un număr natural scris în baza `128` (pentru codul ASCII) sau `256` (pentru codul ASCII extins).

Metoda 1: 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 ştim că ultimii toate secvențle binare de lungime `p` apar cu acceaşi probabilitate ca ultimi `p` biți ai cheii, este de preferat să utilizăm o funcție în care se utilizează toți biții cheii.

Un număr prim apropiat de o putere a lui `2` este adesea o bună alegere pentru `m`.

Observație

Putem considera orice funcție *hash* de forma `h(x)=(ax+b)%m`, cu $a \neq 0$.

Exemplu de funcție *hash* pentru chei şir de caractere:

```
int hashfunction(char *s)
{ int i;
  for (i=0; *s; s++) i = 131*i + *s;
  return( i % m ); }
```

Metoda 2: Înmulțirea

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

$h(x) = [m*(frac(x*y)]$

Un avantaj al acestei metode este că nu există valori "rele" pentru m, prin urmare de obicei m poate fi o putere a lui 2, pentru a implementa eficient funcția hash.

Metoda 3: Hashing universal

Dacă un adversar "răutăcios" va alege cheile astfel să aibă toate aceeaşi valoare *hash* pentru o anumită funcție *hash*, atunci… obținem un timp de execuție liniar.

Orice funcție *hash* fixată este vulnerabilă la o astfel de abordare. 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 numeşte familie universală de funcții *hash* o mulțime H de funcții definite pe A cu valori în $\{0, 1, \ldots, m-1\}$ 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 $|H|/m$. Cu alte cuvinte, dacă alegem aleator o funcție din acestă familie, probabilitatea de obține o coliziune pentru cheile x şi y este $<1/m$.

Exemplu practic

Alegem $m=2^p$ şi generăm aleator un număr natural impar r. Codul *hash* pentru un întreg x se află înmulțind x cu r şi păstrând cei mai semnificativi p biți ai rezultatului (înmulțirea se face pe 32 de biți, şi ignorăm *overflow*). Această variantă aproximează bine o familie universală de funcții *hash*:

```
int r = (rand() << 1) | 1;
...
inline int hash(int x) {
return((unsigned) (x * r) >> (32 - k));
}
```

Rezolvarea coliziunilor

Când funcția *hash* produce o aceeaşi valoare pentru două chei distincte se produce o *coliziune* (astfel de chei sunt denumite sinonime).

Prin urmare este necesar şi un mecanism de rezolvare a coliziunilor.

Există două metode de rezolvare a coliziunilor.

1. Chaining (înlănțuire)

Toate valorile sinonime (care au aceeaşi 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 *Insert* se execută în O(1) (se inserează valoarea `x` la începutul listei `T[h(x)]`).

Operația *Search* presupune căutarea elementului cu cheia `x` în lista înlănțuită `T[h(x)]`.

Operația *Delete* presupune căutarea elementului cu cheia `x` în lista înlănțuită `T[h(x)]`, apoi ştergerea din listă.

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`) (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 `n/m` (această valoare este denumită factor de încărcare a tabelei), deci timpul de execuție va fi O(1+`n/m`).

2. Open-addressing (adresare deschisă)

În tabela *hash* nu sunt memorați pointeri, ci sunt memorate elementele mulțimii `A`. Astfel în tabelă se vor afla poziții ocupate cu elemente din `A` şi 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`.

Pozițiile care se examinează depind de cheia `x`.

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

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

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`, …, `m-1`}.

```
Insert (T, x)
{ i=0;
  do
    {j=h(x, i)
     if (T[j]==NIL)
        {T[j]=k;  return j;}
         else i++;
    }
while (i<m);
error ("hash table overflow");
```

Operația de căutare

Căutarea unui element în tabelă presupune examinarea aceleiaşi secvențe de poziții ca şi inserarea. Dacă în tabel nu se efectuează ştergeri, căutarea se termină fără succes la întâlnirea primei poziții libere.

```
Search(T, x)
{i=0;
 do
   {j=h(x, i);
    if (T[j] == x)return j;
    i++;
   }
while (T[j]!=NILL || i<m);
return NIL;
}
```

Operația de ştergere

Ştergerea 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` şi am găsit-o ocupată se va termina fără succes.

O soluție ar fi ca la ştergerea 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 tabelei.

Din acest motiv, tehnica de rezolvare a coliziunilor prin înlănțuire este mai utilizat pentru cazul în care structura de date suportă şi operații de ştergere.

Observație

Pentru ca distribuția cheilor să fie uniformă, funcția *hash* trebuie să genereze cu egală probailitate oricare dintre cele `m!` permutări ale mulțimii `{1, 2, …, 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, …, m-1}`. Nici una dintre acestea nu asigură o distribuție perfect uniformă.

1. Examinare liniară

Considerând o funcție *hash* "obişnuită" `h': U→{0, 1, ..., m - 1}` (denumită funcție *hash* auxiliară), tehnica examinării liniare utilizează funcția:

```
h(x, i)=(h'(x)+i)% m
```

Observații

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

2. Examinarea liniară este uşor de implementat, dar generează un fenomen denumit primary clustering (secvența de poziții ocupate devine mare, şi astfel timpul mediu de execuție al căutării creşte).

2. Examinare 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 şi $c_2 \neq 0$ sunt constante auxiliare.

Observații

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

Vom considera că m este o putere a lui 2. O variantă bună de examinare pătratică este:

```
i=h'(x);
j=0;
do
  {if (T[i]==NIL) {T[i]=x; return i;}
   j=(j+1)% m ;
   i=(i+j)% m;
  }
while(1);
```

3. Hashing dublu

Hashing-ul dublu utilizează o funcție de forma

$h(x,i)=(h_1(x)+ih_2(x))\%\ m,$

unde h_1 şi h_2 sunt funcții *hash* auxiliare.

Pentru ca întreaga tabelă să fie examinată valoarea $h_2(x)$ trebuie să fie realtiv primă cu m. O metodă sigură este de a 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 $<m$.

De exemplu:

$h1(x)=x\%m;$

$h2(x)=1+(x\%m')$

unde m' este $< m$ (de exemplu $m-1$).

Observație

Hashingul dublu este mai eficient decât cel liniar sau pătratic.

Hashing perfect

Există aplicații în care mulțimea cheilor nu se schimbă (de exemplu, cuvintele rezervate ale unui limbaj de programare; numele fişierelor scrise pe un CD, etc). Hashing-ul poate fi utilizat şi 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 numeşte 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 tabelei *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 şi funcția *hash* aleasă pentru grupul respectiv, precum şi dimensiunea spațiului de memorie alocat tabelei 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 $O(n)$.

gperf

GNU `gperf` este un generator de funcții hash perfecte. Pentru listă dată de şiruri generează o funcție hash şi o tabelă hash (sub formă de cod C/C++), analizând dependențele dintre şirurile de intrare. Funcția *hash* generată este *perfectă*, în sensul că tabela *hash* nu are coliziuni, şi deci consultarea tabelei se reduce la o singură comparație de şiruri.

Aplicație – Algoritmul Rabin Karp

Fie T un şir de n caractere şi P un pattern de m caractere. Să se verifice dacă P apare sau nu ca subsecvență în şirul T.

Soluție

Michael Rabin şi Richard Karp au conceput un algoritm de pattern-matching bazat pe hashing. Ideea este că dacă două şiruri au aceeaşi valoare *hash* atunci ar putea să coincidă; în caz contrar sigur sunt diferite. Cu alte cuvinte, răspunsul NU este întotdeauna corect, iar răspunsul DA este corect cu o probabilitate mare.

Algoritmul preprocesează pattern-ul în $O(m)$. Căutarea se execută în cazul cel mai defavorabil în $O((n-m+1)m)$, dar în medie timpul de execuție este mult mai bun.

Să notăm cu d numărul de caractere distincte care apar în text. Un şir de lungime m poate fi considerat un număr în baza d având m cifre.

În continuare, vom nota cu p valoarea în baza 10 asociată patternului p, iar cu t_k valoarea în baza 10 a subsecvenței din T care începe la poziția k ($T[k..k+m-1]$). Evident, $p=t_k$ dacă şi numai dacă $P=T[k..k+m-1]$.

Valorile t_k, pentru $k=0, \ldots, n-m+1$ se pot calcula în $O(n-m+1)$, deoarece t_{k+1} se poate determina din t_k utilizând următoarea formulă:

$t_{k+1}=d*(t_k-d^{m-1}*T[k])+T[k+m]$

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

Prin urmare vom lucra cu p şi t_k modulo q, unde q este un număr prim convenabil ales (astfel încât $d*q<MAXLONGINT$), deci vom utiliza o funcție *hash*. Problema este că $p\%q==t_k\%q$ nu implică faptul că $p=t_k$ (adică apar coliziuni). Dar dacă $p\%q!=t_k\%q$, atunci sigur $P!=T[k..k+m-1]$. Deci putem utiliza testul modulo ca o euristică, urmând ulterior să testăm egalitatea dintre P şi $T[k..k+m]$.

```
RABIN-KARP (T, P, d, q)
{n=strlen(T); m=strlen(P);
 h = d^(m-1) % q;
 p=0; t0=0;
 for (i=0; i<m; i++) //preprocesare pattern
     {p=(d*p + P[i]) % q;
      t0=(d*t0 + T[i]) % q;}
 for (k=0; k<=n-m; k++)
     {if (p == t0)
         if (P== T [k..k+m-1])
            {print "YES"; return;}
      t0=(d*(t0 - T[k]*h) + T[k+ m]) % q;
     }
print "NO"; return;
}
```

Exercițiu

Modificați algoritmul Rabin-Karp astfel încât să poată fi aplicat pentru un tablou T_{nxn} și un pattern P_{mxm}.

Bibliografie

G.H. Gonnet, R Baeza-Yates – Handbook on algorithms and Data Structures

T. Cormen, L. Leiserson, R. Rivest – Introducere în algoritmi