Curs cu tema „Hashing”

Grup țintă: elevii componenți ai centrului InfO(1)

prof. Dana Lica

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

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

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

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

```
h: A→{0, 1, ..., m-1}
```

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

Funcţii hash

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

În general, funcţiile *hash* sunt definite pe mulţimea 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 împărţirii 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` biţi ai lui `x`). Exceptând cazul în care ştim că ultimii toate secvenţle binare de lungime `p` apar cu aceeaşi probabilitate ca ultimi `p` biţi ai cheii, este de preferat să utilizăm o funcţie în care se utilizează toţi biţii cheii.

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

Observaţie

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

Exemplu de funcţie *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: Înmulţirea

Se înmulţeşte cheia `x` cu un număr subunitar `0<y<1`, considerăm partea fracţionară a lui `x*y`, o înmulţim cu `m` şi reţinem 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 funcţia hash.

Implementarea celor trei operații in C++ folosind containerul <vector> din STL

```
#include <cstdio>
#include <vector>
#define Mod 666013

using namespace std;

vector <int> H[Mod];
vector <int> :: iterator it;
int i, n, op, x, ind;

vector <int> :: iterator find_v(int x){

    for(it= H[ind].begin(); it!=H[ind].end(); it++)
```

```
        if (*it==x) return it;
    return it;
}

void insert_v(int x){

    if (it == H[ind].end()) H[ind].push_back(x);
}

void erase_v(int x){

    if (it!=H[ind].end()) H[ind].erase(it);
}

void write_v (int x){

    if(it==H[ind].end()) printf("0\n"); else printf("1\n");
}

int main(){
    freopen ("hashuri.in","r", stdin);
    freopen("hashuri.out","w", stdout);

    scanf("%d\n", &n);

    for (int i=1; i<=n; i++){

        scanf("%d %d\n", &op, &x);

        ind = x % Mod;
        it = find_v(x);

        if(op==1) insert_v(x);
        if(op==2) erase_v(x);
        if(op==3) write_v(x);
    }

return 0;
}
```

Aplicația pe șiruri de caractere: Algoritmul Rabin-Karp

ENUNȚ: Se dau două șiruri *A* și *B* formate din litere mici și mari ale alfabetului englez și din cifre. Se cere găsirea tuturor aparițiilor șirului *A* ca subsecvență a șirului *B.*

```
1 function RabinKarp(string s[1..n], string sub[1..m])
2     hsub := hash(sub[1..m]);  hs := hash(s[1..m])
3     for i from 1 to n-m+1
4         if hs = hsub
5             if s[i..i+m-1] = sub // compararea O(m)
6                 return i
7         hs := hash(s[i+1..i+m])
8     return not found
```

Consider șirul S= ABDCABAC de lungime N
Și subșirul ABA de lungime M

Consider două funcții de dispersie pe care dacă le aplic asupra subșirului și obțin valorile:

H1=(A*101^2+B*101^1+C) % 666013 (unde A,B,C sunt inlocuite cu 65, 66,67 adica cu ASCII)
H2=(A*109^2+B*109^1+C) % 10003
for(i=1;i<=m) {
h1=(h1*101%666013+(int)sub[i])%666013;   h2=h2*109%10003+(int)sub[i])%10002; }
(bazele le-am considerat nr. prime mai mari decât ‚Z' respectiv 101 și 109)

Se traversează liniar șirul și se numără de câte ori subșirul curent la traversare obține aceleași două valori pe cele două funcții de dispersie. Dacă da, îl consider corect și-l număr.
Dacă sunt pe pozitia I în șirul S inițial, atunci pentru a calcula valoarea pentru subșirul care se termină pe pozitia I, calculez în O(1) astfel:

Pe șirurile noastre luate ca exemplu,

s=ABDCABAC de lungime N
și
subșir ABA de lungime M

după prima sevență verificată din șir ABD, cu funcția H1 se obține valoarea

V1 =(A*101^2+B*101^1+D) % 666013

Din valoarea curentă V1 „iese" A*101^2 și „intră" C. Calcul realizat în O(1):

Notez

K= 666013 si Q= 10003
P=101$^{(M-1)}$ si R=109$^{(M-1)}$;

Atunci algoritmul se poate scrie astfel:

```
for (i = M;  i<N; i++)
{   V1 =  ( (V1 - (S[ i - M]*P)% K + K)*101 + S[i])%K;
    V2 =  ( (V2 - (S[ i - M]*R)% Q + Q)*109 + S[i])%Q;
    if  (V1==H1 && V2==H2)  nr++;
}
```

Aplicații propuse pentru exercițiu:

http://infoarena.ro/problema/eqs
http://infoarena.ro/problema/strmatch