Aplicatii. Tabele de dispersie

– Măsurarea performanţei: funcţii pentru timp și numere aleatoare
– Tabele de dispersie
– Aplicaţie: tabele de simboluri

10 ianuarie 2005

Analiza timpului de rulare al algoritmilor

Metode teoretice

– relaţii matematice pentru cazul cel mai defavorabil (uneori şi mediu, mai rar: cel mai favorabil)
– de regulă, nu timpul fizic, ci numărul de operaţii de un anumit tip (ex. pt. sortare: numărul de comparaţii, numărul de interschimbări)

Evaluare practică

– prin rularea programelor pe diverse seturi de date (aleatoare sau cu anumite proprietăţi), şi măsurarea timpilor de execuţie

Discutăm:
– funcţii pentru generarea de numere (pseudo)aleatoare
– funcţii legate de măsurarea timpului

Funcţii pentru manipularea timpului (`time.h`)

Tipuri definite pentru reprezentarea timpului: `clock_t` şi `time_t`
(sunt de fapt tipuri aritmetice, de ex. `unsigned` sau `unsigned long`)

`clock_t clock(void);`
returnează timpul scurs de la lansarea programului, în unităţi de ceas
date de constanta `CLOCKS_PER_SEC` (în standardul POSIX, 1 milion)
e o aproximaţie dependentă de granularitatea ceasului de timp real
poate interveni depăşire (pe sistem de 32 de biţi, după cca 72 min.)

`time_t time(time_t *timer);`
returnează o valoare aritmetică reprezentând data/ora curentă
(în UNIX, numărul de secunde trecute de la 1 ian. 1970 UTC)
dacă argumentul pointer e nenul, valoarea e stocată şi la acea adresă

`double difftime(time_t time1, time_t time0);`
returnează diferenţa exprimată în secunde, ca `double`
Pentru reprezentări descompuse (zi/oră/min./etc.): tipul `struct tm`
(vezi detalii în standard)

Funcţii pentru numere pseudoaleatoare (`stdlib.h`)

Numerele generate sunt *pseudo*aleatoare (de fapt deterministe, bazate pe un algoritm, dar cu distribuţie cât mai uniformă)
(numere cu adevărat aleatoare ar trebui să fie bazate pe fenomene fizice, ex. aruncarea unei monede sau descompunerea unor particule)

`int rand(void);`
returnează un număr pseudoaleator între `0` şi `RAND_MAX` (min. 32767)
pt. un număr aleator între `1` şi `N` putem folosi `1 + rand() % N`

`void srand(unsigned int seed);`
reiniţializează generatorul de numere pseudoaleatoare cu valoarea dată
următorul număr va fi generat de `rand()` pornind de la această valoare
fără apelarea ei, două rulări generează acelaşi şir de valori cu `rand()`
se poate folosi de ex. cu `srand(unsigned)time(NULL));`

Tabele de dispersie (hashtables)

– pentru *regăsirea eficientă* a unui obiect când acesta nu are o valoare numerică de identificare utilizabilă direct ca indice într-un tablou
– ideea: găsirea unei funcţii `h` cu o *valoare numerică unică* pentru fiecare obiect considerat, într-un domeniu restrâns (utilizabil ca indice)
$\Rightarrow$ fiecare obiect `x` e memorat într-un tablou la indicele `h(x)`
– matematic: o funcţie parţială $h : D \rightarrow V$, unde D e domeniul tuturor obiectelor posibile, iar domeniul de valori V e $0, 1, \ldots, N-1$.
– ex. pt. compilator: D e mulţimea tuturor identificatorilor
– practic, $|D| >> |V|$, deci h nu poate fi injectivă pe D, dar avem nevoie de valori distincte doar pt. submulţimea $D_u \subseteq D$ a obiectelor efectiv utilizate (ex. identificatorii dintr-un anumit program C)
– se caută funcţii de dispersie (hash functions) cu proprietăţi cât mai bune (distribuţie uniformă $\Rightarrow$ probabilitate mică de valori egale)

Exemple de funcţii de dispersie

– funcţii simple, calculate rapid, folosind (aproape) toate caracterele
– adesea cu deplasări pe biţi (în loc de înmulţiri, şi pt. uniformizare)
Exemple pentru şiruri (`char *s; len=strlen(s);` parcurs secvenţial)

```
for (h=len; len--;) h = ((h<<7) ^ (h<<27)) ^ *s++;      /* Knuth */
for (h=5381; c=*s++; ) h += (h << 5) + c;             /* Bernstein */
for (h=0; c=*s++; ) h = (h<<6) + (h<<16) - h + c;        /* SDBM */
```

Pentru alte obiecte: calcule cu întregii obţinuţi grupând octeţii câte 4

În toate cazurile: valoarea finală luată modulo dimensiunea tabloului
– inserăm elementul la indicele respectiv, şi îl căutăm tot acolo

Şi funcţiile bune au *coliziuni* (valori egale pt. obiecte diferite)
⇒ trebuie rezolvate (dezambiguate) pentru a permite regăsirea corectă
– la ce indice inserăm/căutăm elementul daca la `h(x)` se găseşte altul?

Tabele de dispersie deschise și închise

Tabele de dispersie închise (closed hashing)

– dacă la indicele `idx=h(x)` se găsește alt obiect `y`, se caută succesiv după o anumită regulă: *secvențial* (`idx++`), *liniar* (`idx+=i`), cu a doua funcție (`idx+=h2(x)`), până se găsește obiectul sau o intrare vidă

– nu pot conține mai multe obiecte decât dimensiunea tabloului

⇒ la depășire, obiectele trebuie redistribuite într-un tablou mai mare

– la ștergere, intrarea în tablou trebuie marcată "șters", nu "vid", pentru a permite căutarea corectă (până la găsire sau "vid")

Tabele de dispersie deschise (open hashing)

– o intrare în tablou: *listă* de obiecte cu aceeași valoare pentru `h`

⇒ hashing + căutare liniară în listă (scurtă pentru funcții bune)

– necesită alocare dinamică pentru elementele listei (v. exemplu)

– și aici, tablou cu dimensiune cel puțin comparabilă cu nr. de obiecte