Laborator 12 – Tabele de dispersie

Marti 16-18

Tabelele de dispersie (hashtables) sunt o structura de date foarte utila daca vrem sa stocam o colectie de elemente si vrem sa verificam in timp cat mai rapid daca un element exista sau nu in multimea noastra. Este o structura de date foarte utila in aplicatii reale. O aplicatie pentru hashtables este un hashmap (un dictionar), in care pe langa faptul ca avem nevoie sa tinem obiectele din multimea noastra, fiecare obiect are si o valoare asociata, astfel incat vom retine perechi de tip (cheie, valoare), avand proprietatea ca trebuie sa avem chei unice. Putem sa ne gandim la un dictionar intre doua limbi, in care dorim sa aflam rapid care este traducerea pentru un cuvant. In cazul unui hashtable simplu, nu ne intereseaza decat sa aflam daca elementul exista sau nu (practic e ca un hashmap in care valoarea nu este relevanta). Diferenta de implementare dintre cele doua este minima, de aceea in limbaje precum C++ sau Java se ofera clase pentru hashmaps.

Un hashtable este de fapt un vector, in care fiecare element din vector stocheaza mai multe obiecte. Aceste elemente se numesc “buckets”. Practic, avem un vector de buckets, fiecare bucket avand mai multe obiecte, care pot fi stocate, de exemplu, intr-o lista.
Exista 3 operatii de baza – insert, remove si find.
Pentru a sti in ce bucket trebuie sa inseram, se foloseste o functie de hash. Aceasta preia un tip de date pe care vrem sa il stocam si intoarce un numar, folosind niste calcule ce tin cont de tipul nostru. De exemplu, pentru tipul de date string/ char*, o functie standard de hash este sa consideram cuvantul scris in baza 26 (daca avem doar litere mici ale alfabetului), si sa obtinem varianta echivalenta in baza 10. Deoarece numarul de bucketuri este limitat, si in general functiile de hash intorc numere foarte mari, se face modulo la fiecare operatie cu numarul total de bucketuri.
Ex: string = ‘abc’
Nr buckets = 50
hash(‘abc’) = (1*26^2+2*26+3)%50
Exista multe functii de hashing, conditia ca sa functioneze fiind sa distribuie uniform obiectele intr-un numar finit de bucketuri. Daca doua obiecte au aceeasi functie de hash, ele vor trebui inserate in acelasi bucket, cauzand ceea ce se numeste o “coliziune”. Cu cat mai multe coliziuni avem, cu atat operatia de cautare este mai dificila si complexitatea creste. O functie de hash perfecta reuseste sa mapeze intrarile fara nicio coliziune.
Pseudocodul pentru inserare folosind bucketuri de liste ar putea arata in felul urmator:

```
insert(Object o, HashTable h)
{
        bucket = hashfunc(o)
        nod *n = new nod
        n→next = h[bucket]
        h[bucket] = n
        return
}
```

Pentru a cauta un obiect, pur si simplu se parcurge lista corespunzatoare in bucket si se verifica daca gasim obiectul.

```
find(Object o, hashtable h)
{
        bucket = hashfunc(o)
        nod * n = h[bucket]
        while (n!=null)
```

```
            if (n→val ==o)
                return true
            n = n→next
        return false
}
```
Pentru a sterge un element din hashtable, e suficient sa stergem elementul din lista corespunzatoare bucketului lui.

Pentru implementare, puteti folosi aceste structuri pentru hashtable.

```
typedef struct hashtable{
        struct nod ** Table; // vector de nod*, initial toate sunt NULL
        int nr_buckets;
        int (*hfunc) (char*);
        int size; // cate elemente avem in hashtable
} hashtable;

typedef struct nod{
        char *val;
        struct nod * next;
} nod;
```

Teoretic, complexitatea pentru hashtable este O(1) amortizat, daca avem putine coliziuni. In general, se tine cont de un procent de umplere – daca avem mai mult de 75% din numarul de bucketuri, se dubleaza numarul de bucketuri, iar daca avem mai putin de 25% (de exemplu), se imparte numarul de bucketuri la 2. Evident, in acest proces toate elementele trebuie mutate in noul hashtable, dar pentru ca aceste operatii se fac rar, nu influenteaza complexitatea totala.

O varianta de hashtable este urmatoarea – pentru multe coliziuni, in loc sa folosim o lista pentru elemente, se foloseste un arbore binar de cautare. In acest fel, daca avem mai multe coliziuni, in loc sa cautam intr-o lista in O(n), n fiind numarul de coliziuni pentru acelasi bucket, vom cauta in O(log n).

Cerinte:
1. Implementati un hashtable folosind liste. Adaugati cateva siruri in tabel, stergeti siruri din tabel si verificati daca exista anumite siruri, pentru a verifica validitatea tabelului.

2. Modificati implementarea de mai sus, astfel incat nodul unui bucket sa nu mai fie un nod de lista, ci un nod de treap. Pentru cautare, inserare si stergere, se vor folosi functiile corespunzatoare din treap implementate in laboratorul 7.