Laborator 06 - HashTable

Responsabili:

- Claudia Cârdei
- Alex Fărcășanu

Obiective

În urma parcurgerii acestui laborator studentul va fi capabil să:

- definească tipul de date dicționar
- implementeze un dicționar folosind tabele de dispersie
- prezinte avantaje/dezavataje diverselor implementări de dicționare

Ce e un dicționar?

Un dictionar este un tip de date abstract compus dintr-o colecție de chei si o colecție de valori, în care fiecărei chei îi este asociată o valoare. Operația de găsire a unei valori asociate unei chei poartă numele de **indexare**, aceasta fiind și cea mai importantă operație (din acest motiv dicționarele se mai numesc și *arrayuri asociative* - fac asocierea între o cheie și o valoare).

Operații de bază

- **put(key, value):**
  - adaugă în dicționar o nouă valoare și o asociază unei anumite chei
  - dacă perechea există deja, valorea este înlocuită cu cea nouă
- **remove(key):**
  - elimina din dictionar cheia (si valoarea asociata acesteia) key
- **get(key):**
  - întoarce valoarea asociată cheii
  - dacă perechea nu există, întoarce corespunzător o eroare pentru a semnala acest lucru
- **has_key(key):**
  - întoarce **TRUE** dacă există cheia respectivă în dicționar
  - întoarce **FALSE** dacă nu există cheia respectivă în dicționar

Implementare: hashtable

O implementare frecvent întâlnită a unui dicționar estea cea folosind o tabelă de dispersie (hashtable). Un **hashtable** este o structură de date optimizată pentru funcția de *lookup* (în medie, timpul de căutare este constant; O(1)). Acest lucru se realizează transformând cheia într-un hash (folosind o **funcție hash**). În cel mai defavorabil caz, timpul de căutare al unui element poate fi O(n). Totuși, tabelele de dispersie sunt foarte utile în cazul în care se stochează cantități mari de date, a căror dimensiune (mărime a volumului de date) poate fi anticipat.

Funcția hash trebuie aleasă astfel încât să se **minimizeze** numărul coliziunilor (valori diferite care produc aceleași hash-uri). Coliziunile apar în mod inerent, deoarece lungimea hash-ului este fixă, iar obiectele de stocare pot avea lungimi și conținut arbitrare. În cazul apariției unei coliziuni, valorile se stochează pe *aceeasi poziție* (în același bucket). În acest caz, căutarea se va reduce la compararea valorilor efective în cadrul bucketului.

Implementarea cu liste înlănțuite

O implementare a unui hashtable care trateaza coliziunile se numește *înlănțuire directă* (direct chaining). Cea mai simplă formă folosește câte o

Search

- Reguli generale și de notare
- Catalog
- Concursuri
- Calendar

Laboratoare

- Laborator 1 - Introducere in C++
- Laborator 2 - Noțiuni de C++
- Laborator 3 - Stive
- Laborator 4 - Cozi
- Laborator 5 - Liste generice
- Laborator 6 - HashTable
- Laborator 7 - Grafuri
- Laborator 8 - Arbori Binari
- Laborator 9 - Arbori Binari de Căutare
- Laborator 10 - Heap-uri
- Laborator 11 - Treap-uri
- Laborator 12 - Mulțimi Disjuncte

Teme

- Tema 1
- Tema 2
- Tema 3
- Tema 4

Resurse

- Debugging
- Data Structure Visualization

Table of Contents

listă înlănțuită pentru fiecare bucket. Fiecare listă este asociată unei anumite chei.

- **inserarea** în hash presupune găsirea indexului corect și adăugarea elementului la lista corespunzătoare.
- **ștergerea** presupune căutarea și scoaterea elementului din lista corespunzătoare.

Avantajul tabelelor de dispersie constă în faptul că operația de ștergere este simpla, iar redimensionarea tabelei poate fi amânată mult timp, deoarece (chiar și când toate pozițile din hash sunt folosite), performanța este încă bună.

Dezavantajele acestei soluții sunt cele moștenite de la listele înlănțuite: pentru stocarea unor date mici, overhead-ul introdus poate fi semnificativ; iar parcurgerea unei liste este costisitoare.

Există și alte structuri de date cu ajutorul cărora se poate implementa un hash ca mai sus. Un exemplu ar fi un arbore binar echilibrat, pentru care timpul pe cazul cel mai defavorabil se poate reduce la O(log n) față de O(n). Totuși, această variantă se poate dovedi ineficientă dacă hash-ul este proiectat pentru puține coliziuni.

Alte implementări

- arbori binari echilibrați
- radix-tree
- prefix-tree
- array-uri judy

Acestea prezintă timpi de găsire mai buni pentru cel mai defavorabil caz și folosesc eficient spațiul de stocare în funcție de tipul de date folosit.

Exemplu

Funcție de hash pentru șiruri de caractere:

hash.h

```
#ifndef __HASH__H
#define __HASH__H

// Hash function based on djb2 from Dan Bernstein
// http://www.cse.yorku.ca/~oz/hash.html
//
// @return computed hash value

unsigned int hash_fct(char *str)
{
        unsigned int hash = 5381;
        int c;

        while ( (c = *str ++) != 0 ) {
                hash = ((hash << 5) + hash) + c;
        }

        return hash;
}

#endif // __HASH__H
```

Header-ul pentru Hashtable:

hashtable.h

```
#ifndef __HASHTABLE__H
#define __HASHTABLE__H
#include <list>

template<typename Tkey, typename Tvalue>
struct elem_info {
    Tkey key;
```

```
    Tvalue value;
};

template<typename Tkey, typename Tvalue>
class Hashtable {
private:
        std::list<struct elem_info<Tkey, Tvalue> > *H;
        int HMAX;
        unsigned int (*hash) (Tkey);
public:
        Hashtable(int hmax, unsigned int (*h)(Tkey));
        ~Hashtable();

        void put(Tkey key, Tvalue value);
        void remove(Tkey key);
        Tvalue get(Tkey key);
        bool has_key(Tkey key);
};

#endif // __HASHTABLE__H
```

Exercitii

1) [**3p**] Pornind de la header-ul de mai sus, implementați structura de date **hashtable**

- [**1p**] constructor si destructor
- [**1p**] metoda put si remove
- [**1p**] metoda has_key si get

2) [**1p**] Testați implementarea voastră folosind un cod simplist.

3) [**3p**] O aplicatie a unui hashtable este reprezentata de stocarea parolelor unor utilizatori in vederea autentificarii intr-un sistem. Consideram un fisier `text` **password.dat** ce contine pe fiecare linie o pereche de siruri de caractere reprezentand numele utilizatorului si parola sa. Dupa citirea si stocarea parolelor, programul va citi de la tastatura numele utilizatorului ce doreste sa se autentifice, precum si parola sa. Daca parola este cea aferenta utilizatorului, se va afisa un mesaj de tipul "Authentication successful", iar in caz contrar, se va afisa "Authentication failure".

password.dat

```
alex alex1234
marius k1ll3r
elena qwerty
ovidiu 13579er!
```

de intrare

```
elena
qwerty!
```

de iesire

```
Authentication failure
```

Schelet de cod: lab06_schelet_cod.zip

Pasi de rezolvare (urmariti TODO-urile din schelet):

- [**1p**] TODO 1: implementare clasa String
- [**0.5p**] TODO 2: definiti functia hash pentru clasa String
- [**1.5p**] TODO 3: rezolvati problema folosind un Hashtable cu cheie de tip String.

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a

tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

- Pentru o colecție de date cu nume, prenume și multe alte câmpuri, cum ai defini funcția hash?
- Care este complexitatea unei operatiuni de căutare într-un hashtable?
- Care este diferența dintre un hashtable și un vector?
- Descrie cum ai implementa DEX cu ajutorul unui hashtable.
- În ce condiții căutarea într-un hashtable ar putea să nu fie constantă?

Și multe altele...

Bibliografie

[1] C++ Reference

[2] Wikipedia: Hashtable

[3] Wikipedia: Hash function

[4] Open Hashing Visualization

[5] Closed Hashing Visualziation

[6] Closed Hashing (Buckets) Visualization

(cc) BY-SA CHIMERIC DE W3C CSS DOKUWIKI GET FIREFOX RSS XML FEED W3C XHTML 1.0