Laborator 5 - Dicţionar

Responsabili

- Răzvan Rădoi [mailto:mailto:razvanradoi97@gmail.com]
- Alex Selea [mailto:mailto:alexselea@gmail.com]
- Cosmin Petrișor [mailto:mailto:cosmin.ioan.petrisor@gmail.com]

Obiective

În urma parcurgerii acestui articol studentul va fi capabil să:

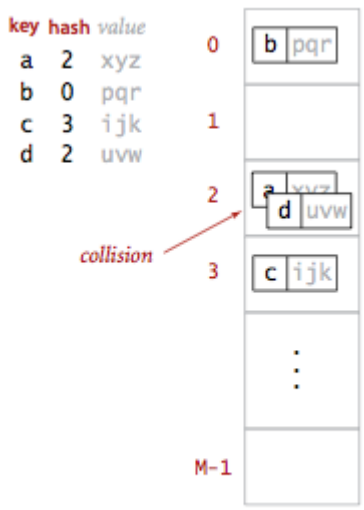
- definească tipul de date dicționar
- implementeze un dicționar folosind tabele de dispersie
- prezinte avantaje / dezavataje ale diverselor implementări de dicționare

Ce este un dicționar?

Un dicţionar este un tip de date abstract compus dintr-o colecție de chei şi 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 *array-uri asociative* - fac asocierea între o cheie și o valoare.

Operația de adăugare a unei perechi (cheie-valoare) în dicționar are **două părți**. Prima parte este transformarea cheii într-un index întreg, printr-o funcție de **hashing**. În mod ideal, chei diferite mapează indexuri diferite în dicționar, însa în realitate nu se întamplă acest lucru. De aceea, partea a doua a operației de adăugare constă în procesul de **rezolvare a coliziunilor**.



Hashing: the crux of the problem

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):**
  - elimină din dicţionar cheia **key** (şi valoarea asociată acesteia)
- **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

O implementare frecvent întâlnită a unui dicționar este cea folosind o tabelă de dispersie - **hashtable**. Un **hashtable** este o structură de date optimizată pentru funcția de *căutare* - în medie, timpul de căutare este constant: O(1). Acest lucru se realizează transformând cheia într-un hash - un număr întreg fără semn pe 16 / 32 / 64 de biţi, etc. - 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 (chei 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 aceeași poziție - în același **bucket**. În acest caz, căutarea se va reduce la compararea valorilor efective în cadrul bucket-ului.

Exemplu 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__
```

Exemplu pentru construcția de funcții hash care minimizează numărul de coliziuni
[https://stackoverflow.com/questions/1145217/why-should-hash-functions-use-a-prime-number-modulus]

Reprezentarea internă cu liste înlănțuite

O implementare a unui hashtable care trateaza coliziunile se numește înlănțuire directă - **direct chaining**. Cea mai simplă formă folosește câte o listă înlănțuită pentru fiecare bucket, practic un array de liste.

Fiecare listă este asociată unui anumit hash.

- **inserarea** în hashtable presupune găsirea indexului corect și adăugarea elementului la lista corespunzătoare.

Hash-ul poate depăşi cu mult dimensiunea array-ului de bucket-uri, ceea ce duce la necesitatea folosirii, cel mai frecvent, a operaţiei *modulo* → **index = hash % HMAX**, pentru a situa indexul bucket-ului în care va fi inserat elementul în limitele necesare.

Dacă dimensiunea array-ului este exprimată în puteri ale lui 2, se mai poate folosi şi formula următoare → **index = hash & (HMAX - 1)**.

HMAX reprezintă dimensiunea maximă a array-ului.

- **ștergerea** presupune căutarea și scoaterea elementului din lista corespunzătoare.

- **cautarea** presupune determinarea index-ului prin funcția de hashing și apoi identificarea perechii potrivite.
- **has_key** presupune determinarea existenței unei chei in dicționar

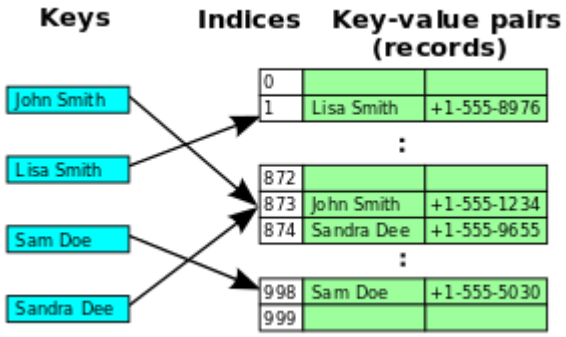
```
void put(key, value)
        index <- hash(key) % DIMENSIUNE_DICTIONAR
        pentru element it in bucketul H[index]
                // Se itereaza prin lista inlantuita de la index, pana se
                // gaseste cheia dorita; daca nu este gasita, vom insera
                // un entry nou in cadrul bucketului
                daca it->key == key
                        // Daca exista key deja in bucket
                        // doar se updateaza valoarea
                        it.value <- value
                        switchFlag <- true; // Semn ca a fost gasita cheia in bucket

        daca switchFlag == 0                // Daca nu a fost gasita cheia in bucket, inseram una noua
                creeaza un nou element pe baza key, value
                adauga elementul in hashtable

void remove(key)
        index <- hash(key) % DIMENSIUNE_DICTIONAR
        pentru element it in bucketul H[index]
                daca it->key == key
                        break

        // it va pointa ori dupa ultimul element (H.end) => nu avem ce sterge
        // ori catre un element deja existent in bucket => stergem elementul it
        daca it nu indica finalul listei
                sterge elementul de la pozitia lui it

TypeValue get(key)
        index <- hash(key) % DIMENSIUNE_DICTIONAR
        pentru element it in bucketul H[index]
                daca it->key == key
                        return it.value

        return null

bool has_key(key)
        index <- hash(key) % DIMENSIUNE_DICTIONAR
        pentru element it in bucketul H[index]
                daca it->key == key
                        return true

        return false
```

Avantajul tabelelor de dispersie constă în faptul că operația de ștergere este simplă, iar redimensionarea tabelei poate fi amânată mult timp, deoarece performanța este suficient de bună chiar și atunci când toate pozițiile din hashtable sunt folosite.

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 hashtable 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 hashtable-ul este proiectat pentru puține coliziuni.

Un alt mod de a utiliza o listă inlanțuită pentru crearea unui dicționar presupune folosirea **linear probing**. Atunci când la inserarea unei perechi (cheie-valoare) in dicționar apărea o coliziune, algoritmul caută primul “spațiu gol” si inserează acolo perechea.



Alte reprezentări interne

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

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

Exemplu de utilizare

Frequency vector

În acest exemplu vom folosi clasa std::map din STL. Pentru mai multe detalii, vă sugerăm să citiți documentația oficială - std::map [http://www.cplusplus.com/reference/map/map/?kw=map].

Atenție! Este nevoie să includem biblioteca map.

```
#include <map> // std::map
```

Clasa std::map oferă toate funcționalitățile uzuale pentru un HashTable. Menționăm că nu există o implementare unică, astfel încât apar diferențe față de implementările sugerate în articol (de exemplu cheile din map sunt unice). În continuare ne vom referi doar la următoatele funcționalități:

- metoda insert - inserează o pereche (key, value)
- metoda erase - șterge valoarea asociată cu o anumită cheie
- metoda empty - verifică dacă structura nu conține nici o pereche
- metoda size - returnează numărul de perechi din map
- operatorul de indexare [] - care primește ca paramtru o cheie și returnează referință către valoarea asociată (dacă cheia nu exista în map, aceasta va fi introdusă automat iar valoarea asociată este data de constructorul default al tipului pe care il au valorile)

Implementare cu std::map

Se dă un număr n foarte mare și n stringuri. Se cere să se afișeze pe ecran numărul de apariții al fiecărui string utilizând un spațiu de memorie cât mai mic.

Soluție: Vom simula funcționalitatea unui vector de frecvență folosinf std::map.

main.cpp

```
#include <iostream> // std::cout
#include <map>      // std::map

int main() {
    int n;                           // Numarul de elemente din lista
    std::string x;                   // Variabila temporara
    std::map <std::string, int> hash; // Map-ul (hash-ul) folosit

    // Citire elemente si adaugare in stiva
    std::cout << "n = ";
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        // Citeste un alt element
        std::cout << "x = ";
        std::cin >> x;

        // Adauga o aparitie a lui x
        ++hash[ x ];

        // Afisare statistici
        std::cout << x << " apare de " << hash[x] << " ori; hash size = " << hash.size() << "\n";
    }

    // Parcurgerea elementelor din hash
    std::cout << "Stare finala hash\n";
    std::cout << "hash size = " << hash.size() << '\n';

    for (std::map <std::string, int> :: iterator it = hash.begin(); it != hash.end(); ++it) {
        // Extrag key si value
        std::string key = it->first;
        int value = it->second;

        // Afisez de cate ori a fost intalnit key
        std::cout << key  << " apare de " << value << " ori\n";
    }

    std::cout << "Golesc hash\n";
    hash.clear();
    std::cout << (hash.empty() ? "Hash gol" : "Hash contine elemente") << "\n";

    return 0;
}
```

Testare:

```
g++ main.cpp -o main
./main
```

```
n = 10

x = SD
SD apare de 1 ori; hash size = 1

x = PL
PL apare de 1 ori; hash size = 2

x = MN
MN apare de 1 ori; hash size = 3

x = SD
SD apare de 2 ori; hash size = 3

x = SD
SD apare de 3 ori; hash size = 3

x = PL
PL apare de 2 ori; hash size = 3

x = CMOS
CMOS apare de 1 ori; hash size = 4
```

```
x = BUCURIE
BUCURIE apare de 1 ori; hash size = 5

x = cmos
cmos apare de 1 ori; hash size = 6

x = proiect
proiect apare de 1 ori; hash size = 7

Stare finala hash
hash size = 7

BUCURIE apare de 1 ori
CMOS apare de 1 ori
MN apare de 1 ori
PL apare de 2 ori
SD apare de 3 ori
cmos apare de 1 ori
proiect apare de 1 ori

Golesc hash
Hash gol
```

Schelet

Schelet

Exerciţii

Fiecare laborator va avea unul sau doua exerciții publice si un pool de subiecte ascunse, din care asistentul poate alege cum se formeaza celelalte puncte ale laboratorului.

1) [**1p**] Testaţi funcţionalitatea std::map descrisă mai sus. Nu este nevoie să copiaţi codul, acesta este disponibil în schelet.

2) [**5p**] Implementaţi structura de date dicţionar, plecând de la pseudocodul de mai sus.

- [**1p**] constructor şi destructor
- [**2p**] metodele **put** şi **has_key**
- [**1p**] metoda **get**
- [**1p**] metoda **remove**

3) [**4p**] O aplicaţie a unui hashtable este reprezentată de stocarea credenţialelor unor utilizatori în vederea autentificării într-un sistem. Pentru ficare utilizator se vor reţine următoarele date: `parolă`, `username` şi un boolean `logged in` care va reţine dacă utilizatorul este autentificat sau nu. Vom defini următoarele operaţii:

- **signup(username, password)** - adaugă un nou utilizator în sistem. Dacă utilizatorul există deja, se va afişa “User <username> already added”. Altfel, se va afişa “User <username> successfully added”.
- **login(username, password)** - autentifică utilizatorul **username** în sistem. Dacă autentificarea are loc cu succes se va afişa mesajul “User <username> logged in successfully”. Altfel, se va afişa “Username/Password incorrect”.
- **logout(username)** - deloghează utilizatorul din sistem. Dacă operaţia se finalizează cu succes, se va afişa “User <username> logged out”, în caz contrar “User does not exist”.
- **change_password(username, old_password, new_password)** - schimbă parola unui utilizator. Dacă operatia se poate realiza (vechea parola se potriveste si exista user-ul), se va afişa

"Password changed for user <username>". În caz contrar se va afişa mesajul "Could not change password".

4) [**2p**] Implementați și testați redimensionarea unui hashtable: funcția **resize** dublează dimensiunea structurii interne a tabelei de dispersie. Dublarea se va face în momentul în care raportul dintre numărul de elemente introduse în hashtable şi numărul de bucket-uri HMAX este mai mare decâti o valoare aleasă (ex: *size / HMAX > 0.75*). Comportamentul dorit pentru această funcţionalitate este următorul: se redimensionează array-ul de bucket-uri, iar apoi fiecare bucket este parcus în ordine și elementele sunt redistribuite după valoarea **noului hash**.

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?

Bibliografie

1. C++ Reference [http://www.cplusplus.com]
2. Wikipedia: Hashtable [http://en.wikipedia.org/wiki/Hashtable]
3. Wikipedia: Hash function [http://en.wikipedia.org/wiki/Hash_function]
4. Open Hashing Visualization [http://www.cs.usfca.edu/~galles/visualization/OpenHash.html]
5. Closed Hashing Visualization [http://www.cs.usfca.edu/~galles/visualization/ClosedHash.html]
6. Closed Hashing (Buckets) Visualization [http://www.cs.usfca.edu/~galles/visualization/ClosedHashBucket.html]
7. Collection of hash functions [http://www.cse.yorku.ca/~oz/hash.html]

sd-ca/2018/laboratoare/lab-05.txt · Last modified: 2019/02/01 13:24 by teodora.serbanescu