Analiza algoritmilor

Dragomir Andrei-Mihai - 322CA

andrei.dragomir1401@stud.acs.upb.ro

Abstract. Abstractizarea multimilor - reprezentarea API-ului specific unei multimi cu ajutorul structurilor de date

Keywords: Tabel de dispersie · Treap · Multime.

1 Introducere

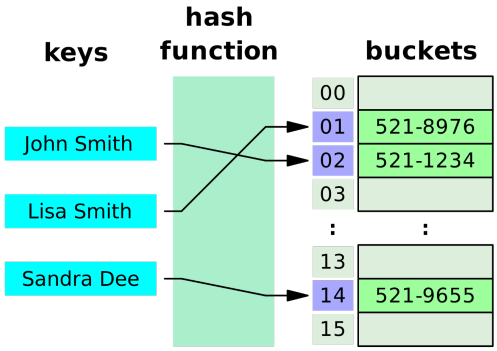
1.1 Descrierea problemei alese

Structurile de date au aparut din nevoia de a manipula si incapsula mai usor datele unei probleme pe care automatizarea isi propune sa o rezolve. Acestea facilitează **accesul utilizatorilor** și **lucrul cu datele** de care au nevoie în moduri adecvate. Cel mai important, structurile de date încadrează **organizarea informațiilor**.

O multime de date este extrem de intalnita in aproape orice aplicatie sau structura informatica. Astfel a fost nevoie de gasirea unor **solutii optime** pentru facilitarea operatiilor pe multimi prin creearea unor structuri de date precum cele ce vor fi studiate in urmatoarele pagini.

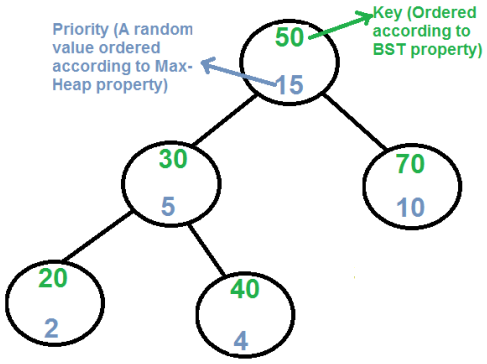
In aceste structuri de date s-a ajuns sa se poata stoca in mod abstract orice tip de informatie. Structurile de date alese trebuie sa ne faciliteze indeplinirea eficace a urmatoarelor operatii: **adaugare, stergere, cautare, inserare si afisare integrala**. Adaugarea, stergerea si inserarea sunt elemente vitale ale manipularii unei multimi cu un API specific, pe cand celelalte doua sunt o aditie utila.

Tabelele de dispersie sunt o modalitate de stocare a colecțiilor de date prin perechi **cheie-valoare**. Cheia este trecuta printr-o functie specifica denumita **hash** pentru a asigura asocierea uniforma cu valoarea specifica. Tabelul de dispersie funcționează eficient avand ca si concept corespondent in viata reala un dicționar. Cel mai valoros aspect al unui astfel de tabel față de alte structuri de date abstracte este viteza acestuia de a efectua operațiuni de **inserare, ștergere și căutare**. Tabelele de dispersie le pot face pe toate în **timp constant**



[1] medium.com - hashtable photo

Treapul este o structura de date foarte utila in lumea calculatoarelor, fiind folosita in numerosi algoritmi complexi. Utilizează amestecarea nodurilor care determină **rotații spontane** pentru a se menține **echilibrat în ceea ce privește înălțimea**, astfel încât operațiile care sunt efectuate pe ei să rămână eficiente din punct de vedere al complexității timpului.



[2] www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/

1.2 Aplicatii practice ale problemei alese

Seturile de date pot deține in cazuri practice informații precum fișe medicale sau dosare de asigurări, pentru a fi utilizate de un program care rulează pe un sistem automat. Seturile de date sunt, de asemenea, utilizate pentru a stoca informațiile necesare aplicațiilor sau **sistemului de operare** însuși, cum ar fi programele sursă, bibliotecile macro sau variabilele si parametrii de sistem.

Tabelele de dispersie sunt utilizate in general pentru implementarea de vectori(arrays) asociativi dar si a altor structuri de tipul **tabel de memorie**. Aceasta structura de date este implementata si in limbaje dinamice: Perl, Python, Javascript sau Ruby fiind concept de baza in unele dintre ele.

Alta utilizare a tabelelor de dispersie este si in algoritmi renumiti precum **Rabin-Karp**, un algortim de cautare in siruri. Utilizează hashing pentru a găsi o potrivire exactă a unui șir de caractere într-un text. Folosește un **hash** tocmai pentru a filtra rapid pozițiile textului care nu se potrivesc cu modelul, apoi verifică existenta unei potriviri la pozițiile rămase.

Treapul are numeroase utilizari in lumea calculatoarelor, dar este cel mai util pentru a rezolva probleme de **conectivitate**, probleme de **interogare a intervalului** și poate fi folosit ca alternativa buna la **segmentarea** arborilor/tabelelor.

Combinatia aceasta de **arbore binar cu heap** mai poate fi folosita si in algoritmul Huffman, dezvoltat prima data de David Huffman. Acesta este o **tehnică de comprimare a datelor** pentru a le reduce dimensiunea fără a pierde niciun detaliu, propunand astfel o solutie utila pentru a comprima fisiere în care există **caractere care apar frecvent**.

1.3 Solutiile alese

Am ales ca structuri reprezentante **Hashtable-ul** si **Treap-ul** ce pot fi folosite pentru construirea unui API specific unei multimi cu operatii precum:
- adaugarea, stergerea, gasirea si modificarea unui element
- gasirea maximului/minimului
- afisarea intregii multimi.

Tabelul de dispersie stocheaza dupa cum am spus perechi cheie valoare. Punctul lui forte este complexitatea uzuala de cautare, inserare si de stergere: **O(1)**. Este adevarat ca in cazul cel mai defavorabil se poate ajunge la **0(n)** (se parcurge practic o lista inlantuita), dar daca se utilizeaza o functie hash buna si destula memorie, acest caz nu este atins.

O functie hash buna trebuie sa fie in primul rand rapida, dar si mai important de atat este sa genereze indexuri cat mai diferite pentru intrari. Ce face de fapt aceasta functie e sa ne dea un **hash** care corespunde cu locul in care va fi stocata valoarea.

Nodurile treap-ului sunt reprezentate si ele de perechi, dar nu cheie valoare, ci **cheie-prioritate**. Treapul presupune **complexitate logaritmica** in majoritatea operatiilor ca orice alt arbore binar. Cheie indica si pastreaza invariantul de arbore binar de cautare:

$$nod.cheie \geq nodStanga.cheie$$
$$nod.cheie \leq nodDreapta.cheie$$

Invariantul de heap presupune sa avem nodul cu prioritate mai mare sa fie cat mai sus in arbore, relatie care arata astfel:

$$pentru(fiu : nod.fii)$$
$$nod.prioritate \geq fiu.prioritate$$

Pentru pastrarea **invariantului de heap** de mai sus dupa operatii care modifica structura Treap-ului se folosesc rotatii. Acestea sunt astfel gandite incat sa reechilibreze arborele si sa pastreze si **invariantul de arbore binar de cautare**.

Pentru reprezentarea API-ului unei multimi cu arbori perfect echilibrati se pot alege **arborii AVL**. Într-un **arbore AVL**, înălțimile celor doi subarbori fii ai oricărui nod diferă cu cel mult unu. Propietatea e restabilita prin reechilibrare.

1.4 Criterii de evaluare a solutiilor alese

Daca vorbim de criterii de evaluare a modalitatilor de reprezentare a API-ului specific unei multimi este clar ca principalul aspect este **complexitatea temporala**, urmata apoi de cea **spatiala**. Exista un compromis intre aceste doua cerinte, iar sistemele moderne tind sa mearga pe partea de **viteza** in defavoarea spatiului de memorie ocupat.

Generarea testelor de dimensiuni mari se va face automat cu un script sau program si se vor testa si cazurile marginale specifice comune (introducerea de **valori cu hash identic generat** sau modificarea/stergerea unei **valori inexistente**). Testele constau in succesiuni de **comenzi** de tipul:

- **0**(adaugare)
- **1**(eliminare)
- **2**(cautare)
- **3**(inlocuire)
- **4**(afisare integrala).

Ordinul de magnitudine al testelor va fi de 10^5 operatii de fiecare tip: inserari, cautari, stergeri, dar vor contine si inlocuiri de elemente si afisari integrale ale multimii. **Metodele statistice** folosite vor fi calcularea de valori medii pentru autenticitatea studiului si compararea unor metode diferite de a produce acelasi efect pentru a studia performantele.

Din punct de vedere al **tabelului de dispersie**, testele (pe langa faptul ca testeaza si corectitudinea codului) vor fi relevante si exhaustive, tratand si asa zisele cazuri marginale. Se va pune accent pe evaluare fiecare variante de tratare a coliziunilor: probare liniara si inlantuire directa. De asemenea, se vor aborda si cazurile in care este folosita o functie de hash mai slaba si vor fi inserate chei cu hash-uri identice pentru a observa cum complexitatea tinde sa nu mai fie **O(1)**.

Testele selectate vor fi relevante si pentru **Treap**, unde vor verifica, pe langa complexitatea temporala, si partea de rotatii pentru **pastrarea invariantului de heap**. Se urmareste verificarea eficientei reechilibrarii Treap-ului dupa stergeri sau inserari cu prioritate generata random. Prin comparatia celor doua structuri de date se urmareste mai ales monitorizarea **spatiului ocupat** prin amplitudinea testelor care le incarca progresiv. O incarcare mai mare **presupune un timp mai mare** la iterarea prin liste sau noduri dupa caz si de aceea este un aspect important.

2 Prezentarea solutiilor

2.1 Descrierea functionalitatii algoritmilor alesi

Dupa cum am stabilit, se vor implementa tabelul de dispersie si treapul ca structuri de date reprezentative pentru API-ul unei multimi. Limbajul ales este C, iar aceste doua structuri sunt implementate in asa fel incat sa poata sa stocheze **date generice, de tip void**.

2.1.1 Algoritmul de tabel de dispersie

Pentru implementarea tabelului de dispersie se va folosi un **array de liste inaluntuite** cu dimensiune presetata. S-a ales aceasta varianta deoarece reprezinta una dintre cele mai bune moduri de tratare a coliziunilor(hash identic pentru chei diferite). Coliziunile apar indiferent, fiind cauzate de **diferenta de dimensiune** dintre obiectele stocate care sunt diferite si lungimea hashului care este fixa.

Implementarea presupune **modificarea unui element** in momentul in care se incearca inserarea unei intrari cu cheie identica. Se verifica existenta cheii printr-o operatie de cautare si se face inlocuirea in caz de reusita.

```
double load_factor = ht->size / ht->hmax;
if (load_factor > 1)
    ht_resize(ht);

// find hash
int index = ht->hash_function(key) % ht->hmax;
void *copy_key = malloc(key_size);
memcpy(copy_key, key, key_size);
ll_node_t *aux = ht->buckets[index]->head;

while (aux) {
  if (!ht->compare_function(key, ((info *)aux->data)->key) &&
     ht->compare_function(value, ((info *)aux->data)->value))
     {
    // update entry
    free(((info *)aux->data)->value);
    ((info *)aux->data)->value = malloc(value_size);
    memcpy(((info *)aux->data)->value, value, value_size);

    free(copy_key);
    return;
  }
  aux = aux->next;
}

// Create a new entry
info *data = malloc(sizeof(info));
```

```
data->key = malloc(key_size);
data->value = malloc(value_size);

// Put memory in the new entry
memcpy(data->key, copy_key, key_size);
memcpy(data->value, value, value_size);

// Add node to the bucket in charge
ll_add_nth_node(ht->buckets[index], ht->buckets[index]->size,
     data);
```

Cautarea in hashtable urmeaza un model simplu prin cautarea in bucketul dat de indexul generat de functia de hash:

```
void *
ht_get(hashtable_t *ht, void *key)
{
    // Verifies if a ht has key and returns value
    int index = ht->hash_function(key) % ht->hmax;
    ll_node_t *aux = ht->buckets[index]->head;
    while (aux) {
        if (!ht->compare_function(key, ((info *)aux->data)->
    key))
            return ((info *)aux->data)->value;

        aux = aux->next;
    }

    return NULL;
}
```

Ca sa fie o functie de hash buna, ea trebuie sa satisfaca **doua propietati**:
1) ar trebui să fie foarte rapida in calcul;
2) ar trebui să minimizeze dublarea valorilor de ieșire (coliziuni).

Funcțiile hash se bazează pe generarea de distribuții de probabilitate favorabile pentru eficacitatea lor, reducând timpul de acces la aproape constant. Factorii mari de încărcare a tabelului, seturile de chei nefavorabile și funcțiile hash prost concepute pot duce la timpi de acces care se apropie de liniari. **Functia de hash** care a fost aleasa este una care transforma cheia intr-un numar intreg fara semn pe 64 de biti. Indexul pentru stocarea valorii corespondente cheii este calculat dupa formula:

$$INDEX = HASH \ \% \ HMAX$$

Legenda:
INDEX = indexul bucketului in care va fi adaugata valoare corespondenta cheii
HMAX = dimensiunea maxima a tabelului de dispersie
HASH = hashul obtinut de functia de hashing

```c
unsigned int
hash_function_string(void *a)
{
    /*
     * Credits: http://www.cse.yorku.ca/~oz/hash.html
     */
    unsigned char *puchar_a = (unsigned char *)a;
    uint64_t hash = 5381;
    int c;

    while ((c = *puchar_a++))
        hash = ((hash << 5u) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

Avantajul folosirii acestei functii de hash este ca ofera o distributie statistica buna a perechilor cheie-valoare, minimizand coliziunile.

2.1.2 Algoritmul de treap

Treapul este implemenat cu ajutorul unei structuri de nod ce retine cheia si piroritatea acestuia. Cheia reprezinta baza pentru operatiile de adaugare, stergere, etc. Prioritatea este folosita pentru a pastra echilibrul structurii de date, intrucat este generata aleator si limitata la MAX_PRIORITY in program.

Adaugarea in treap se face dupa model recursiv printr-o functie simpla:

```c

/* Inserare in Treap
 *
 * @param1: Nodul radacina al subarborelui din parcurgerea
    recursiva.
 * @param2: Valoare de adaugat in Treap.
 * @param3: Numarul de octeti pe care se scrie valoarea.
 * @param4: Functia de comparare pentru datele din Treap.
 */
void treap_insert(Node **node,
                  void *value,
                  int data_size,
                  int (*compar)(void *, void *))
{
    if ((*node) == NULL) {
        *node = node_create(value, data_size);
        return;
    }



```

```
    if (compar(value, (*node)->data) < 0) {
        // go to left subtree
        treap_insert(&((*node)->left), value, data_size,
    compar);

        // check if heap property is correct
        if (priority((*node)->left) > priority(*node))
            rotate_right(node);
    } else {
        // go to right subtree
        treap_insert(&((*node)->right), value, data_size,
    compar);

        // check if heap property is correct
        if (priority((*node)->right) > priority(*node))
            rotate_left(node);
    }
}
```

Stergerea in treap se face prin gasire pe baza invariantului de arbore binar de cautare si rotatii pana se respecta invariantul de heap:

```
/* Stergere din Treap
 *
 * @param1: Nodul radacina al subarborelui din parcurgerea
    recursiva.
 * @param2: Valoare de adaugat in Treap.
 * @param3: Numarul de octeti pe care se scrie valoarea.
 * @param4: Functia de comparare pentru datele din Treap.
 */
void treap_delete(Node **node,
                  void *value,
                  int data_size,
                  int (*compar)(void *, void *))
{
    if (node == NULL)
        return;

    if (compar(value, (*node)->data) < 0) {
        treap_delete(&((*node)->left), value, data_size,
    compar);

    } else if (compar(value, (*node)->data) > 0) {
        treap_delete(&((*node)->right), value, data_size,
    compar);

    } else if ((*node)->left == NULL && (*node)->right ==
    NULL) {
        node_free(node);
        *node = NULL;

```

```
26      } else if (priority((*node)->left) > priority((*node)->
    right)) {
27          rotate_right(node);
28          treap_delete(node, value, data_size, compar);
29      } else {
30          rotate_left(node);
31          treap_delete(node, value, data_size, compar);
32      }
33  }
```

Cautarea cheii este identica cu cautarea in arborele binar de cautare:

```
1   void *
2   get_key(Node *node, void *value, int data_size, int (*compar)
        (void *, void *))
3   {
4       if (compar(value, node->data) == 0)
5           return node;
6
7       if (compar(value, node->data) <= 0) {
8           if (node->left != NULL)
9               return get_key(node->left, value, data_size,
        compar);
10          else
11              return NULL;
12      } else {
13          if (node->right != NULL)
14              return get_key(node->right, value, data_size,
        compar);
15          else
16              return NULL;
17      }
18  }
```

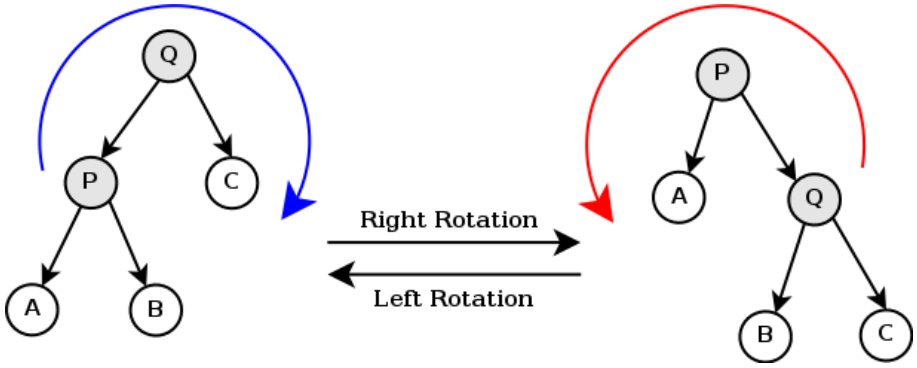
Esenta pastrarii conditiei de arbore echilibrat sunt rotatiile implementate: rotate_right si rotate_left. Generarea prioritatii la intamplare ar parea un element de instabilitate la prima vedere, insa este exact ceea ce pastreaza arborele echilibrat(precum pivotul este de exemplu generat aleator la fiecare pas in algoritmul de quick sort).

```
1   void rotate_left(Node **node)
2   {
3       Node *rson = (*node)->right;
4       Node *rlson = (*node)->right->left;
5
6       rson->left = *node;
7       (*node)->right = rlson;
8
9       // new root
10      *node = rson;
11  }
```

```c
void rotate_right(Node **node)
{
    Node *lson = (*node)->left;
    Node *lrson = (*node)->left->right;

    lson->right = *node;
    (*node)->left = lrson;

    // new root
    *node = lson;
}
```



[3] https://iq.opengenus.org/treap-randomized-cartesian-tree/

2.2 Analiza complexitatii solutiilor

2.2.1 Analiza complexitatii algoritmului de tabel de dispersie

Cand vorbim de complexitatea algoritmului ce implementeaza hashtable-ul, trebuie sa diferentiem doua cazuri de tratare a coliziunilor si sa le tratam seaparat: linear probing si direct chaining.

1. Linear probing

Aceasta metoda de **adresare deschisa** a coliziunilor se caracterizeaza prin faptul ca in momentul in care functia de hash produce un rezultat care corespunde unui spatiu deja ocupat de alta pereche cheie-valoare, se continua cautarea unui loc gol in spatiile imediat vecine.

daca $h(x)$ este ocupat atunci se verifica $h(x+1)$, $h(x+2)$, $h(x+3)$, etc
unde h este functia de hash folosita

Aceasta metoda de adresare a coliziunilor presupune urmatoarele complexitati temporale pentru operatiile de baza [6]:

A. Inserare

Complexitatea pentru cel mai bun caz: O(1)

Complexitatea pentru cel mai rău caz: O(N).
Acest lucru se întâmplă atunci când toate elementele au acelasi hash rezultat și trebuie să inserăm ultimul element verificând spațiul liber unul câte unul.

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

Presupunând că funcția hash distribuie uniform elementele, atunci complexitatea medie va fi constant O(1). În cazul în care funcția hash funcționează prost, atunci complexitatea medie va ajunge la O(N).

B. Stergere

Complexitatea pentru cel mai bun caz: O(1)

Complexitatea pentru cel mai rău caz: O(N).

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

La stergere nu este suficienta golirea celulei. Acest lucru ar afecta căutările pentru alte chei care au o valoare hash mai inainte decât celula golită, dar care sunt stocate într-o poziție de dupa celula golită. În schimb, atunci când o celulă i este golită, este necesara cautarea prin următoarele celule ale tabelului până când se gaseste fie o altă celulă goală, fie o cheie care poate fi mutată în celula i (adică o cheie a cărei valoare hash este egală cu sau mai mica decât i). Când se gaseste o celulă goală, atunci golirea celulei i este sigură și procesul de ștergere se încheie.

C. Cautare

Complexitatea pentru cel mai bun caz: O(1).

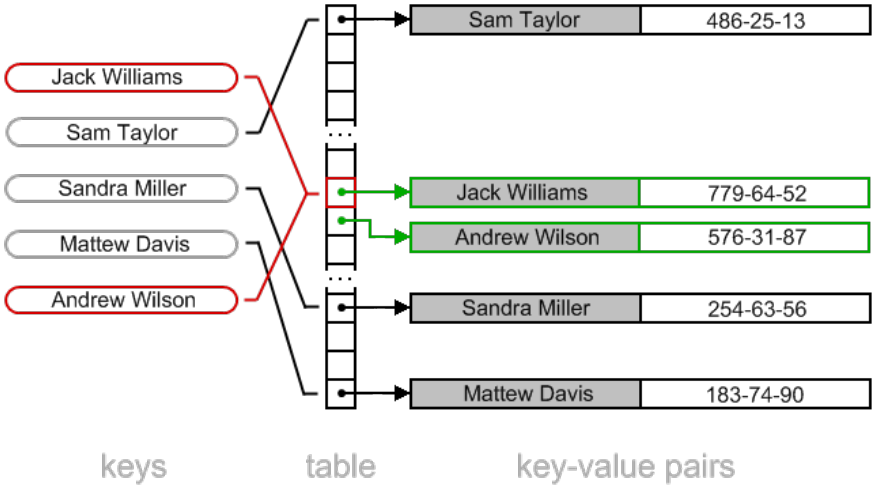
Complexitatea pentru cel mai rău caz: O(N).

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

Pentru a căuta o anumită cheie x, celulele tabeluilui de dispersie sunt examinate, începând cu celula de la indexul h(x) (unde h este funcția hash) și continuând până la celulele adiacente h(x) + 1, h(x) + 2, ..., până când găsim fie o celulă goală, fie o celulă a cărei cheie stocată este x. Dacă este găsită o celulă care conține cheia, căutarea returnează valoarea din acea celulă. În caz

contrar, dacă se găsește o celulă goală, cheia nu poate fi în tabel, deoarece ar fi fost plasată în acea celulă. În acest caz, căutarea returnează ca rezultat că cheia nu este prezentă în dicționar.

Un aspect important este ca pentru cazul de linear probing, **complexitatea spatiala este O(n)**.



[7] https://www.algolist.net/Data_structures/Hash_table/Open_addressing

2. Direct chaining

Modalitatea mai intalnita de a trata coliziunile este cea de **inalantuire directa**, ce presupune ca la fiecare index sa se afle o lista inlantuita. In momentul in care se produce un hash ce corespunde unui index deja ocupat, pur si simplu se adauga elementul in lista respectiva indexului(denumita generic **bucket**).

Aceasta metoda de **adresare inchisa** a coliziunilor presupune urmatoarele complexitati temporale pentru operatiile de baza [8]:

A. Inserare

Complexitatea pentru cel mai bun caz: O(1).

Complexitatea pentru cel mai rău caz: O(N).

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

Inserarea presupune adaugarea in bucketul corespunzator indexului, adica o operatie de inserare in lista.

B. Stergere

Complexitatea pentru cel mai bun caz: O(1).

Complexitatea pentru cel mai rău caz: O(N).

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

Stergerea presupune eliminare din bucketul corespunzator indexului, adica o operatie de stergere din lista.

C. Cautare

Complexitatea pentru cel mai bun caz: O(1).

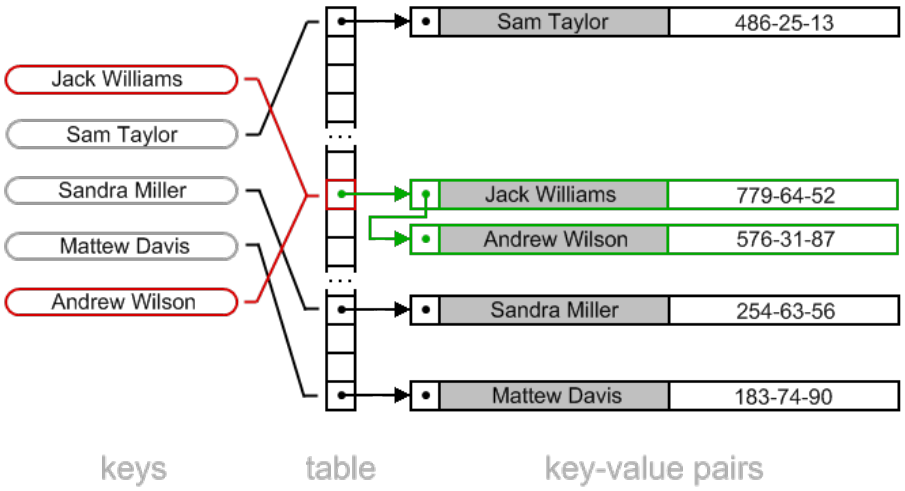
Complexitatea pentru cel mai rău caz: O(N).

Complexitatea medie: O(1) pentru o funcție hash bună;
Complexitatea medie: O(N) pentru funcția hash proastă

Cautarea unei intrari in hashtable presupune cautarea in bucketul corespunzator indexului, o operatie mai costisitoare decat inserarea si stergerea, dar suficient de rapida.

Dupa cum am mentionat, complexitatea spatiala pentru linear probing era **O(n)**. Pentru direct chaining este **O(m + n)**, asta fiind o diferenta semnificativa intre cele doua abordari.

m = dimensiunea tabelului de dispersie
n = numarul de elemente inserate

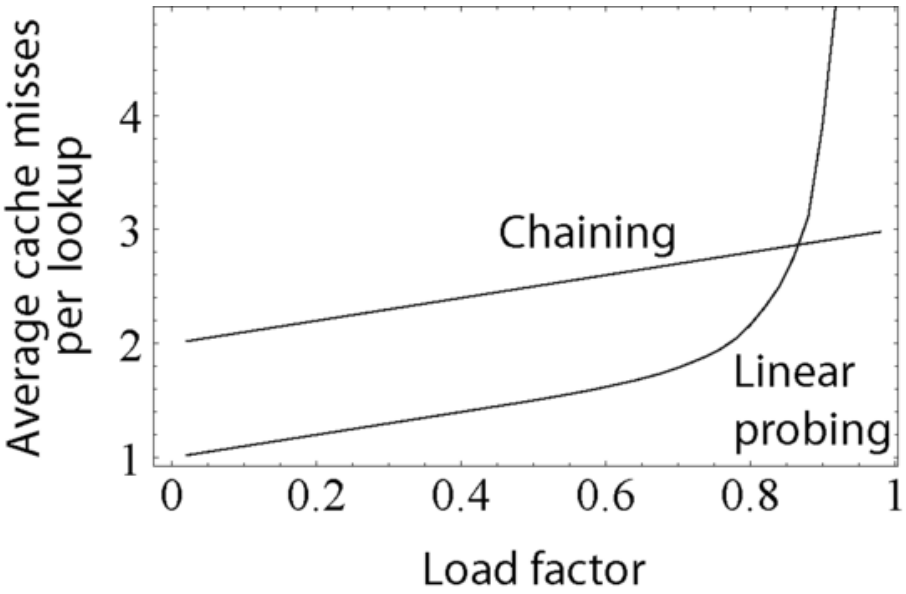


[9] https://www.algolist.net/Data_structures/Hash_table/Chaining

Load factorul este definit ca gradul de ocupare al tabelului de dispersie, fiind reprezentat de un procent calculabil dupa formula:

LOAD_FACTOR = SIZE / HMAX

Cand load factorul se apropie de 1(adica atunci cand hashtable-ul este aproape plin) varianta de **linear probing** presupune devantajul ca trebuie cautat elementul prin verificare de mai multe pozitii succesive in tabelul de dispersie. **Direct chaining** are avantajul ca pe acest caz se gaseste direct indexul cautat si se fac operatii pe bucketul respectiv fara cautari suplimentare. **Linear probing** are o performanta mai buna pe cazuri de hashtableuri cu load factor sub 0.7-0.8 deoarece nu aduce asupra sa cautarile suplimentare de care am vorbit la un load factor mai mare.



[9] https://stackoverflow.com/questions/30683511/linear-probing-vs-chaining

Dupa cum se vede si in grafic, pentru factor de incarcare mai mic este mai util **linear probing**, iar pentru factor de incarcare mai mare se recomanda **direct chaining**.

2.2.2 Analiza complexitatii algoritmului de Treap

Aspectul esential al treapului este caracterul de structura de date echilibrata. Acest lucru presupune o complexitate omogena intre cazul favorabil, mediu si defavorabil de O(log(N)) (N = numarul de intrari din treap). Echilibrarea se face prin rotiri pentru a limita dimensiunea la inaltimea permisa.

A. Inserare

Complexitatea pentru cel mai bun caz: O(log(N)).

Complexitatea pentru cel mai rău caz: O(N).
Complexitate ce se atinge daca echilibrare nu e corecta si se parcurge practic o lista inlantuita.

Complexitatea medie: O(log(N)).

Inserarea presupune coboirarea ghidata de regula de ordine de arbore binar de cautare si apoi realizarea de rotatii pentru echilibrare pe baza prioritatilor.

B. Stergere

Complexitatea pentru cel mai bun caz: O(log(N)).

Complexitatea pentru cel mai rău caz: O(N).
Complexitate ce se atinge daca echilibrare nu e corecta si se parcurge practic o lista inlantuita.

Complexitatea medie: O(log(N)).

Stergerea presupune cautarea elementului pe baza regulii de ordine, stergerea lui si apoi echilibrarea pe baza de prioritati.

C. Cautare

Complexitatea pentru cel mai bun caz: O(log(N)).

Complexitatea pentru cel mai rău caz: O(N).
Complexitate ce se atinge daca echilibrare nu e corecta si se parcurge practic o lista inlantuita.

Complexitatea medie: O(log(N)).

Cautarea unei intrari in treap presupune urmarirea regulii de ordine pana se gaseste cheia.

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

COMPARATIE STRUCTURI DE DATE ALESE PENTRU API-UL UNEI MULTIMI	
HASHTABLE	**TREAP**
Pentru cazuri cand stim **dimensiunea aproximativa** a datelor se poate obtine un **load factor favorabil**	Este mai versatil cand nu stim dimensiunea datelor deoarece este **robust** prin caracterul **echilibrat**
In cazul in care HMAX ales arbitrar la inceput nu este suficient poate fi implementat printr-un **vector redimensionabil** ce poate fi realocat la nevoie prin dublare	Nu are nevoie de redimensionari si permite numar nelimitat de inserari, ci doar de **rotatii** pentru echilibrare
Nu utilizeaza **memoria** in cel mai eficient mod intrucat pe cazul de direct chaining pot exista bucketuri care **raman goale** pe tot parcursul programului, iar pentru adresare directa pot ramane deasenenea **spatii goale** direct in array	Aloca spatiu doar pentru **intrarile reale** si utilizeaza mai eficient memoria
Poate duce cautarea in complexitate mai slaba pe cazuri marginale(lipsa redimensionarii sau numar mari de coliziuni)	Are complexitatea de cautare stabila la O(long(N))
Nu are acest beneficiu, necesitand sortare separata daca e nevoie	Fiind un arbore de cautare impune **relatia de ordine** si se pot obtine usor **elementele sortate** daca acesta este un criteriu de alegere
Inserarea si stergerea se fac in **timp constant** pe cazul mediu	Consuma timp la inserare si stergere si pe **operatiile de rotire** pentru pastrarea caracterului de arbore echilibrat

3 Evaluare

3.1 Descrierea modalitatii de construire a setului de teste folosite pentru validare

References

1. Tabel de dispersie poza introductiva - https://medium.com/@jaden.banson/hash-tables-17c9f7bf3ecf
2. Treap poza introductiva - https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/
3. Rotatii in treap - https://iq.opengenus.org/treap-randomized-cartesian-tree/
4. Resurse facultate - ocw.cs.pub.ro
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms
6. https://iq.opengenus.org/linear-probing
7. Linear probing - https://www.algolist.net/Data_structures/Hash_table/Open_addressing
8. https://iq.opengenus.org/time-complexity-of-hash-table/
9. Linear probing - https://www.algolist.net/Data_structures/Hash_table/Chaining
10. https://stackoverflow.com/questions/30683511/linear-probing-vs-chaining