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

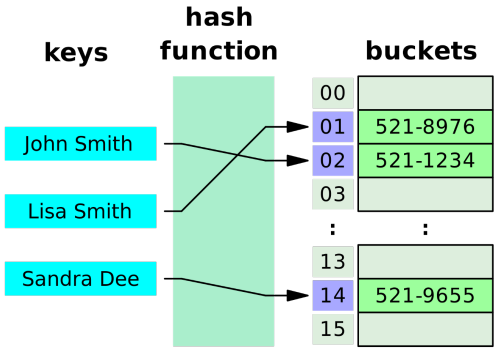
1.1 Descrierea problemei alese

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.

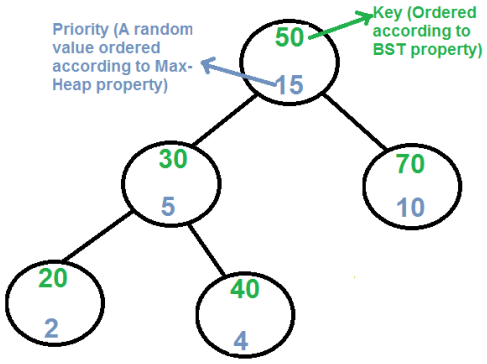
In aceste structuri de date s-a ajuns sa se poata stoca in mod abstract orice tip de informatie. Structurile de date alese trebuie sa ne faciliteze indeplinirea eficace a urmatoarelor operatii: **adaugare, stergere, cautare, inserare si afisare integrala**. Adaugarea, stergerea si inserarea sunt elemente 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.

```
1  void ht_put(hashtable_t *ht, void *key, unsigned int key_size
       , void *value, unsigned int value_size)
2  {
3      if (!ht)
4          return;
5
6      double load_factor = (double)ht->size / ht->hmax;
7      if (load_factor > 1)
8      {
9          ht_resize(ht);
10     }
11
12     unsigned int hash = ht->hash_function(key) % ht->hmax;
13
14     linked_list_t *list = ht->buckets[hash];
15     ll_node_t *node = list->head;
16
17     while (node)
18     {
19         if (!ht->compare_function(key, ((info *)node->data)->
       key))
20         {
21             ((info *)node->data)->value =
22                 realloc(((info *)node->data)->value,
       value_size);
23             memcpy(((info *)node->data)->value, value,
       value_size);
```

```
            return;
        }

        node = node->next;
    }

    info *new_info = malloc(sizeof(info));

    new_info->key = malloc(key_size);
    memcpy(new_info->key, key, key_size);

    new_info->value = malloc(value_size);
    memcpy(new_info->value, value, value_size);

    ht->size++;

    ll_add_nth_node(list, 0, new_info);
    free(new_info);
}
```

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)
{
    if (!ht)
        return NULL;

    unsigned int hash = ht->hash_function(key) % ht->hmax;

    ll_node_t *node = ht->buckets[hash]->head;

    while (node)
    {
        if (!ht->compare_function(key, ((info *)node->data)->
    key))
            return ((info *)node->data)->value;

        node = node->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 32 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

```
unsigned int hash_function(void *a)
{
    unsigned int x = *(unsigned int *)a;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = (x >> 16) ^ x;
    return x;
}
```

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:

```
void treap_insert(Node **node, int value, int (*compar)(int,
    int))
{
    if ((*node) == NULL)
    {
        *node = node_create(value);
        return;
    }

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

```
void treap_delete(Node **node, int value, int (*compar)(int,
    int))
{
    // Check if the tree is empty
    if (*node == NULL)
    {
        return;
    }

    // Compare the value to be deleted with the current node'
    s value
    int comp_result = compar(value, (*node)->data);

    // If the value to be deleted is smaller than the current
     node's value,
    // recursively call treap_delete on the left child
    if (comp_result < 0)
    {
        treap_delete(&(*node)->left, value, compar);
        // After deleting the node, check if the treap
    properties are still maintained
        // If not, perform a right rotation
        if ((*node)->left != NULL && (*node)->left->priority
    > (*node)->priority)
        {
            rotate_right(node);
        }
        return;
    }

```

```
26      // If the value to be deleted is larger than the current
        node's value,
27      // recursively call treap_delete on the right child
28      if (comp_result > 0)
29      {
30          treap_delete(&(*node)->right, value, compar);
31          // After deleting the node, check if the treap
        properties are still maintained
32          // If not, perform a left rotation
33          if ((*node)->right != NULL && (*node)->right->
        priority > (*node)->priority)
34          {
35              rotate_left(node);
36          }
37          return;
38      }
39
40      // If we reach this point, it means that the value to be
        deleted has been found
41      Node *to_delete = *node;
42
43      // If the current node has no children (i.e. it is a leaf
         node), simply delete it
44      if ((*node)->left == NULL && (*node)->right == NULL)
45      {
46          *node = NULL;
47          free(to_delete);
48      }
49      // If the current node has only one child, delete it and
        replace it with its child
50      else if ((*node)->left == NULL)
51      {
52          *node = (*node)->right;
53          free(to_delete);
54      }
55      else if ((*node)->right == NULL)
56      {
57          *node = (*node)->left;
58          free(to_delete);
59      }
60      // If the current node has two children, find the in-
        order successor (i.e. the smallest
61      // value in the right subtree) and delete it. Then,
        replace the current node with the
62      // in-order successor
63      else
64      {
65          Node *successor = (*node)->right;
66          while (successor->left != NULL)
67          {
```

```
            successor = successor->left;
        }
        (*node)->data = successor->data;
        treap_delete(&(*node)->right, successor->data, compar
    );
        // After deleting the node, check if the treap
    properties are still maintained
        // If not, perform a left rotation
        if ((*node)->right != NULL && (*node)->right->
    priority > (*node)->priority)
        {
            rotate_left(node);
        }
    }
}
```

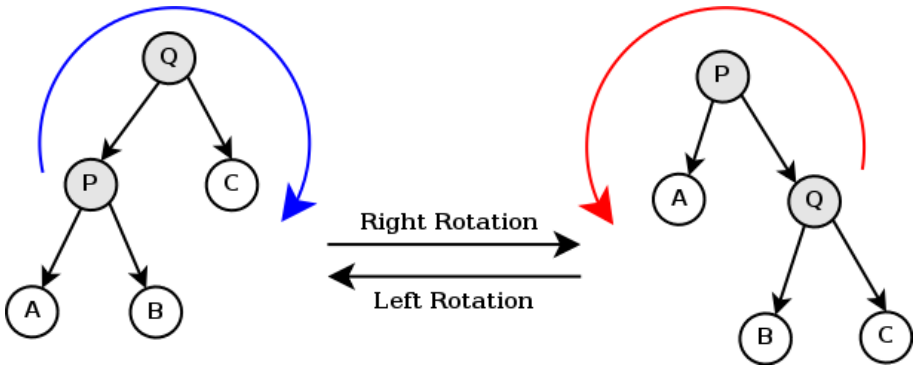
Cautarea cheii este identica cu cautarea in arborele binar de cautare:

```
void *get_key(Node *node, int value, int (*compar)(int, int))
{
    if (!node)
        return NULL;

    if (compar(value, node->data) == 0)
        return node;

    // go to left subtree
    if (compar(value, node->data) <= 0)
    {
        if (node->left != NULL)
        {
            return get_key(node->left, value, compar);
        } else {
            return NULL;
        }
        // go to right subtree
    } else {
        if (node->right != NULL)
        {
            return get_key(node->right, value, compar);
            return NULL;
        }
    }
}
```

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).

```
void rotate_left(Node **node)
{
    Node *rson = (*node)->right;
    Node *rlson = (*node)->right->left;

    rson->left = *node;
    (*node)->right = rlson;

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

```
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 separat: 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

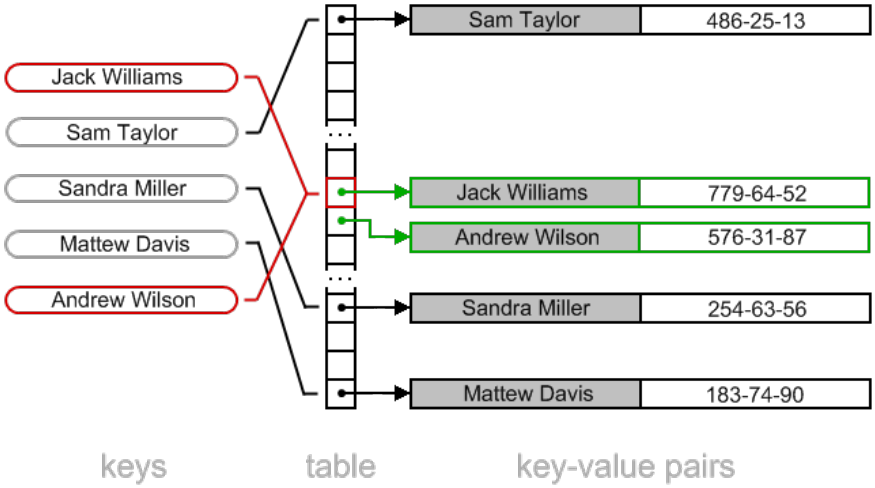
Complexitatea pentru cel mai bun caz: O(1).

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

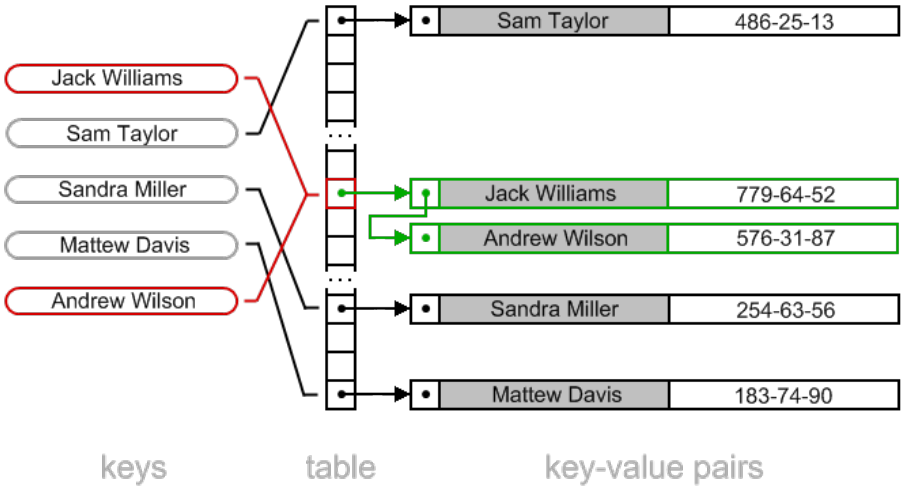
Complexitatea pentru cel mai bun caz: O(1).

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

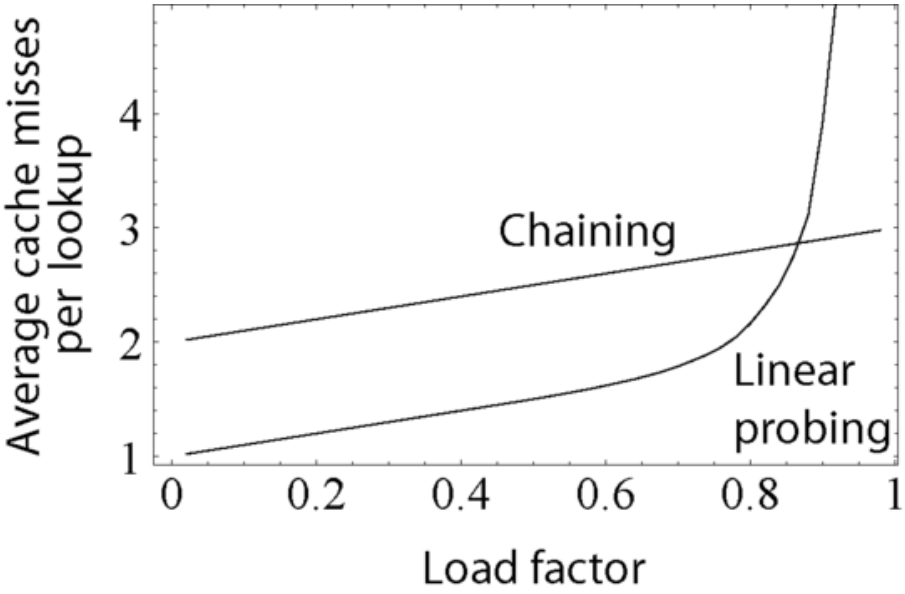


[9] https://www.algolist.net/Data_structures/Hash_table/Chaining

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

$$\text{LOAD\_FACTOR} = \text{SIZE} / \text{HMAX}$$

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(log(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

Pentru constructia testelor am folosit un generator de teste scris in C. El genereaza 30 de teste diferite cu posibilitate de customizare. Parametrii controlabili sunt:
→ tipul testului
→ magnitudinea testului
→ frecventa operatiilor de stergere
→ frecventa operatiilor de cautare
→ frecventa operatiilor de inlocuire
→ proportia de operatii de stergere a unui element valid
→ proportia de operatii de cautare a unui element ce exista/nu exista in multime
→ proportia de operatii de inlocuire a unui element existent/inexistent in multime
→ frecventa operatiilor de afisare integrala(la cate operatii de adaugare sa se faca o afisare → se alege o frecventa mica pentru ca afisarile introduc influenteaza puternic timpul de executie)

Testele sunt progresive, atat din punct de vedere al dimensiunii, cat si din punct de vedere al comenzilor folosite. Testele implementeaza dupa modelul:
→ testele 1, 3, 5 - operatii doar de adaugare si sub zece comenzi
→ testele 2, 4, 7 - operatii de adaugare si stergere, cu dimensiuni mai mari
→ testele 6 si 8 - operatii de adaugare si stergere cu frecventa fixa
→ testele 9, 10, 11 - operatii de adaugare, stergere si cautare, cu dimensiuni de ordinul sutelor si frecevente de stergere si cautare mari
→ testele 12, 13, 14, 15 - operatii de adaugare, stergere si cautare, cu dimensiuni de ordinul miilor si frecvente ridicate de cautare, stergere si inlocuire
→ testele 15, 16, 17, 18, 19, 20 - operatii de toate felurile, cu frecvente mai scazute si numere de dimensiuni medii si dimensiuni din ce in ce mai mari
→ testele 21, 22, 23, 24, 25 - teste mixte cu dimensiuni de **ordinul zecilor de mii** de comenzi si care au operatii cu numere mai mari
→ testele 26, 27, 28, 29, 30 - teste mixte de dimensiuni **foarte mari**(sute de mii, milioane de comenzi) si **numere mari**

Se folosesc aceleasi teste pentru ambele structuri de date pentru a putea compara performantele. Aceasta testeaza atat **corectitudinea implementarii**, cat si **performantele algoritmului** in sine.

Exista program separat care genereaza suita de outputuri din out/ in functie de fisierele din directorul in/. In proiect se afla si cod care **ruleaza singular** pentru ambele structuri de date un test specific din test.in si afiseaza rezultatele in test.out. Un script numit **compute** se ocupa de calcularea timpului de executie de 5 ori pentru fiecare structura de date in parte.

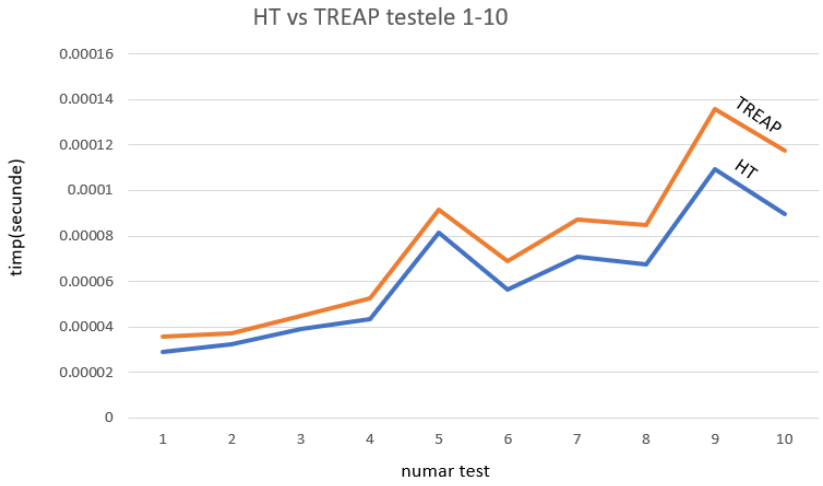
Se calculeaza timpul de executie de 5 ori si se face o **medie** pentru a face comparatia in modul cel mai corect statistic. In cazul **tabelului de dispersie**, inputurile mai mari testeaza foarte bine si capacitatea de tratare a coliziunilor prin adresare inchisa(inlantuire directa). Algoritmul are si functionalitatea de resize implementata, dar cu ea performantele nu mai sunt comparabile cu cele ale treapului deoarece introduce overhead prin faptul ca implica **restructurarea intregului tabel**.

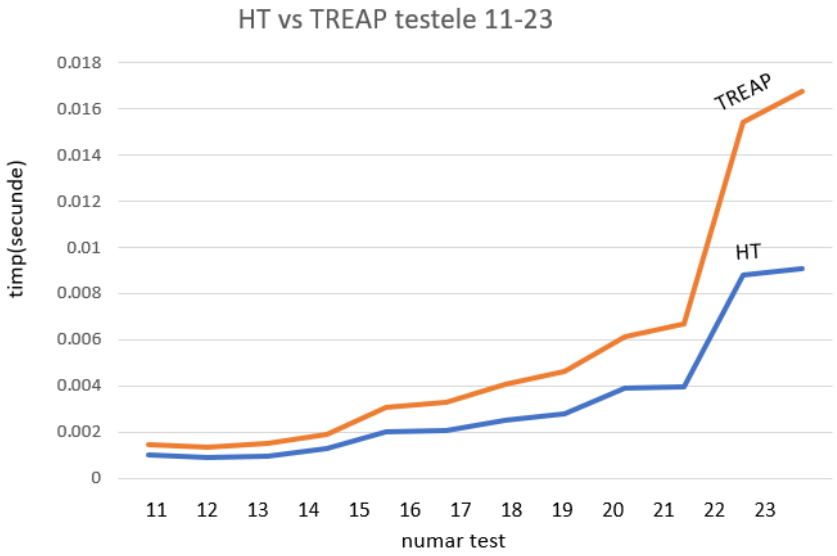
In cazul **treapului**, aceste teste foarte mari vor veni ca o validare a capacitatii sale de look-up comparativ cu hashtable-ul.

3.2 Specificatiile sistemului de calcul pe care am rulat testele

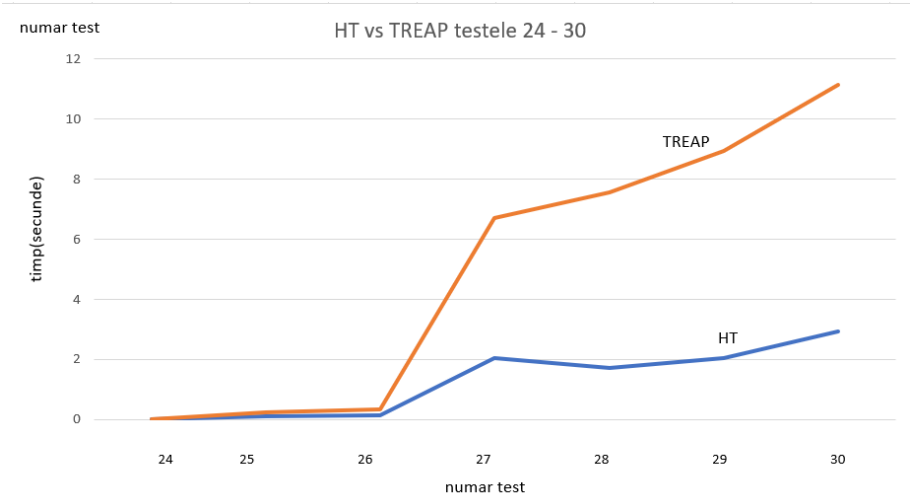
Procesor : AMD Ryzen 5 5600H with Radeon Graphics, 3301 Mhz, 6 Core(s), 12 Logical Processor(s)
Memorie : 16.0 GB RAM

3.3 Ilustrarea, folosind grafice/tabele, a rezultatelor evaluarii solutiilor pe setul de teste





Nr. Test	Nr. comenzi	Timp HT (s)	Timp Treap (s)
1	0	0.000029	0.0000066
2	20	0.0000322	0.0000048
3	23	0.0000392	0.0000056
4	50	0.0000436	0.000009
5	43	0.0000814	0.00001
6	64	0.0000566	0.0000124
7	85	0.000071	0.0000164
8	88	0.0000674	0.0000176
9	114	0.0001096	0.0000264
10	126	0.0000894	0.000028
11	328	0.0001442	0.0000598
12	1 687	0.0010264	0.0004098
13	2 060	0.0008956	0.0004716
14	2533	0.0009528	0.0005536
15	2 838	0.001303	0.0006184
16	3 099	0.0020314	0.0010566
17	3 549	0.0020892	0.0011958
18	4 444	0.002509	0.0015426
19	5 166	0.0028122	0.0018404
20	6 309	0.003933	0.0021944
21	6 921	0.0039652	0.0027248
22	15 475	0.008825	0.0066088
23	17 995	0.0090994	0.0076694
24	19 814	0.0103938	0.0086634
25	198 894	0.1138336	0.1346512
26	287 646	0.1285108	0.2112322
27	2 736 849	2.0336982	4.6624452
28	3 223 215	1.713182	5.850319
29	3 851 980	2.063241	6.8893932
30	4 298 484	2.9422468	8.18147



3.4 Prezentarea valorilor obtinute pe teste

Dupa cum se observa si din tabelulu si graficele prezentate:

→ Pentru **primele 10 teste** care sunt de dimensiune mica si testeaza **comenzile de baza** performantele sunt similare, tabelul de dispersie avand totusi un **mic avantaj**.

→ Pentru **testele intermediare** se poate observa cum **complexitatea constanta** a tabelului de dispersie la cautari, inserari si stergeri il desprinde de performanta treapului pe parcurs ce dimensiunea testelor creste.

→ Pentru ultimele teste unde dimensiunile ajung la cateva **milioane de comenzi** se poate observa o **desprindere clara** a hashtableului de treap. Performanta hashtableului este de pana la trei ori mai buna decat a treapului. Acestea testeaza foarte bine si cazul de cautare a unui element ce nu exista in multime si pe parcurs ce **demandul computational** creste, complexitatea mai buna a hashtableului face diferenta fata de cea **logaritmica** a treapului.

Aceste complexitati sunt masurate fara a lua in calcul **operatiile de afisare**, care adauga overhead urias datorita multitudinii de apeluri de sistem pe care le face si durata efectiva de scriere. Se observa cum diferentele pe ultimele teste **intensiv computationale** diferenta e de la aproximativ 3 secunde la peste 50 pentru hashmap si de la 8 secunde la aproape 50 pentru treap. Cu tot cu **operatiile de afisare**, care pe ultimele teste au o freceventa de aparitie de una la 10-15 mii de comenzi, complexitatile se modifica astfel:

Nr. Test	Nr. comenzi	Timp HT cu print (s)	Timp Treap cu print (s)
1	0	0.000021	0.0000428
2	20	0.0000278	0.000026
3	23	0.0000388	0.0000078
4	50	0.0000452	0.0000514
5	43	0.0000562	0.0000322
6	64	0.000049	0.000016
7	85	0.0000608	0.000016
8	88	0.000071	0.000041
9	114	0.0000884	0.0000736
10	126	0.0000814	0.000054
11	328	0.0001696	0.0000886
12	1687	0.0006798	0.0005662
13	2060	0.0007424	0.000519
14	2533	0.0008476	0.000623
15	2838	0.0009174	0.0007814
16	3099	0.2229722	0.004113
17	3549	0.2194254	0.0051312
18	4444	0.2645248	0.0077128
19	5166	0.2838194	0.0099044
20	6309	0.3739844	0.0142156
21	6921	0.0032642	0.0033498
22	15475	0.019312	0.0089136
23	17995	0.0209114	0.0101452
24	19814	0.0083754	0.0096094
25	198894	0.2938486	0.2544168
26	287646	0.3640652	0.3644836
27	2736849	33.671288	29.044404
28	3223215	40.755004	35.520608
29	3851980	48.538706	42.51607
30	4298484	55.474568	49.866816

4 Concluzii

API-ul unei multimi poate fi foarte bine reprezentat utilizand structurile de date prezentate, cu performante demonstrate diferite. De multe ori, o anumita structura de date reprezinta fundamentul unui algoritm sau sistem, iar o performanta buna a acesteia (complexitate spatiala si temporala cat mai mica) influenteaza performanta sistemului per total.

Dupa analiza realizata pe cele doua structuri de date, daca ar fi sa aleg una dintre ele strict pentru **operatiile de adaugare, stergere, cautare si inlocuire** as alege **tabelul de dispersie**. Rationamentul este ca prezita **performante temporale** superioare Treapului, iar daca obiectiv principal al proiectului este acesta, hashmapul este alegerea corecta. Alta situatie cand as alege hashtableul este cand stiu dimensionea aproximativa a datelor ce urmeaza sa fie stocate pentru a alege un **HMAX potrivit** pentru a nu contine spatii goale si pentru a avea o distributie a cheilor uniforma.

Cu toate acestea, Treapul prezinta **beneficii esentiale** care nu exista la tabelul de dispersie:

→ putem obtine **cheile in ordine sortata** doar prin a **parcurge in inordine** treapul, pe langa hashmap unde e nevoie de efort suplimentar considerabil

→ daca avem operatii de **gasirea celui mai apropiat** element mai mare/mic decat unul dat sau orice alta comanda ce necesita **intervale** sunt mult mai usor de facut pe treap, mai ales cautarea pe un interval dat

→ treapul are structura **robusta** si garanteaza **complexitate logaritmica**, pe cand hashmapul poate deveni ineficient mai ales daca e nevoie de redimensionarea lui; analiza temporala realizata pe tabelul de dispersie s-a facut **fara redimensionare**, deoarce adaugarea acestui mecanism facea performantele de **aproape 5 ori mai slabe** pentru testele foarte mari

→ treapul este **mai eficient din punct de vedere spatial** decat hashmapul, iar daca acesta este un punct important al programului, treapul este alegerea recomandata; tabelul de dispersie poate **contine bucketuri** goale pe tot parcursul rularii si de aceea poate presupune ineficienta a utilizarii memoriei

→ un alt aspect important si practic este ca treapul este **mai usor de implementat**, in afara de partea de rotatii pentru echilibrare; la fel treapul este mai customizabil in limbaje de programare mai high end decat C-ul, implementarea de hashtable bazandu-se pe implementari eficiente deja existente in limbaj; tabelul de dispersie, daca este implementat custom, are nevoie si de o **functie de hashing foarte buna** pentru a prezenta performante foarte bune

Asadar pentru un program care are nevoie de **consistenta a complexitatii temporale**, de operatii pe intervale, de afisari ordonate si eventual sa fie mai **eficient din punct de vedere al spatiului ocupat**, Treapul este alegerea superioara fata de tabelul de dispersie.

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