Laborator 6: Tabele de dispersie

1. Obiective

In acest laborator vom lucra cu tabele de dispersie. In contextul tabelelor de dispersie se vor studia urmatoarele operatii: creare, inserare, gasire si retragerea unor elemente.

2. Recapitulare scurta

Tipul Tabel

Un tabel reprezinta o colectie de elemente de acelasi tip. Aceste elemente sunt identificate prin intermediul unor chei. Tabelele sunt organizate in doua moduri:

- Tabele Fixe – In acest caz numarul de elemente este cunoscut de la inceput programului.
- Tabele Dinamice – Aceste tabele au un numar variabil de elemente.

Un exemplu de tabel fix poate fi considerat un tabel de maxim 7 elemente continand toate zilele saptamanii. Tabelele dinamice pot fi organizate ca si liste liniare simplu inlantuite, arbori de cautare sau tabele de dispersie. Daca tabelele dinamice sunt organizate ca si liste, cautarile sunt effectuate linear, lucru care incetineste operatia de gasire a unui element. Un arbore de cautare reduce timpul de gasire al unui element, precum am vazut in laboratorul precedent gasirea unui element era realizata in timp logaritmic. In acest laborator vom merge mai departe si vom incerca sa construim o structura de date in care gasirea unui element se va efectua in timp constant O(1).

Un **tabel de dispersie** reprezinta o colectie de itemi care sunt stocati in asa maniera incat gasirea lor sa fie foarte usoara. Fiecare pozitie a tabelului de dispersie se numeste **slot** si poate retine un element. Indexarea tabelelor de dispersie incepe cu indicele 0 si se termina cu un indice n, care reprezinta numarul maxim de elemente care pot fi stocate in tabelul de dispersie. Initial tabelul de dispersie nu contine nici un element, adica fiecare slot al lui e gol. In figura 1 se prezinta un tabel de dispersie de dimensine 11 in care fiecare element este nul. Cu alte cuvinte avem allocate m =11 sloturi in tabelul de dispersie numerotate de la 0 la 10.

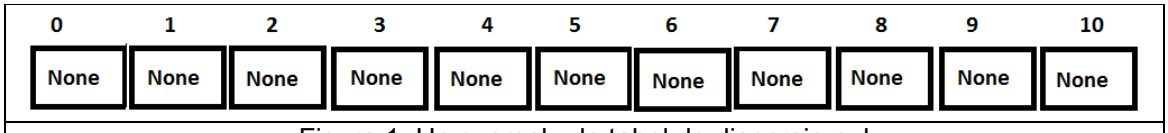

Figura 1. Un exemplu de tabel de dispersie nul

Maparea dintre un item si un slot, adica locul in care trebuie plasat itemul respective in tabelul de dispersie, este numita functie de dispersie sau functie de hashing. Functia de dispersie va lua orice item dintr-o colectie si il va transforma intr-o valoare numerica in intervalul numarului de sloturi (de la 0 la m - 1 in exemplul de mai sus).

3. Functii de dispersie

Fiind data o colectie de itemi, o functie de dispersie care mapeaza fiecare item intr-un slot unic este considerata o functie perfecta de hashing. Daca am stii ca itemii din colectie nu se vor schimba niciodata, atunci am putea construii o functie perfecta de dispersie. Cu toate acestea avand in vedere natura aleatorie a cheilor, nu exista o modalitate sistematica de a construi functia de hashing perfecta. O metoda de a avea o functie de dispersie perfecta ar fi de a marii dimensiunea tabelului in asa fel incat orice valoare sa isi poata gasi un loc unic. Aceasta abordare este practica pentru un numar mic de elemente dar devine nepractica in momentul in care dorim sa mapam numere mai mari (spre exemplu daca pentru o clasa de 30 de elevi am vrea sa mapam fiecare elev folosind CNP-ul lui – in acest caz am irosi multa memorie). Din fericire nu avem nevoie ca functia de dispersie sa fie perfecta pentru a ne bucura performanta.

$$f : K \rightarrow H$$

In maparea de mai sus K reprezinta un set de chei iar H reprezinta o multime de numere naturale. Functia f reprezinta o mapare many to one. Presupunand ca avem doua key, $\mathrm{k1\ si\ k2\ cu\ k1} != \mathrm{k2}$ si rezultatul functiei de hashing e acelasi i.e. $f(k1) = = f(k2)$, atunci se spune ca intre cele doua chei apare o coliziune iar datele corespunzatoare sunt numite inregistrari sinonime. Doua restrictii sunt impuse asupra functiei f:

1. Pentru orice k, valoarea corespunzatoare trebuie obtinuta cat de repede posibil
2. Functia trebuie sa minimizeze numarul de coliziuni.

Un exemplu de functie de hashing este ilustrat mai jos:

$$f(k) = \gamma(k)\ modulus\ B\ ,$$

unde γ repezinta o functie care mapeaza o cheie la un numar natural, iar B reprezinta un numar natural (de regula prim). Modul in care se construieste functia γ depinde de cheile folosite. Daca cheile sunt valori numerice atunci $\gamma(k) = k$ poate fi o posibila functie de dispersie. Pentru chei care reprezinta siruri de caractere cea ma simpla functie este suma codurilor ASCII ale fiecarui character al cheii. O functie simpla pe siruri de caractere este ilustrata mai jos:

```
#define B ? /* se ofera o valoare corespunzatoare lui B */
int f( char *key )
{
    int i, sum;
    sum = 0;
    for ( i = 0; i < strlen( key ); i++ )
```

```
            sum += key[ i ];
    return ( sum % B );
}
```

Alte exemple de functii de dispersie pot fi: suma cifrelor, metoda gruparii (folding method) etc.

4. Rezolutia de coliziune

In momentul in care doua elemente sunt dispersate in acelasi slot trebuie sa gasim o metoda sistematica de a plasa al doilea item in tabelul de dispersie. Acest process se numeste rezolutia coliziunii. Precum am mentionat mai sus, daca functia de dispersie ar fi perfecta, niciodata nu ar aparea coliziuni. Cu toate acestea,functiile perfecte de dispersie nu exista asadar rezolutia devine un element important al tratarii coliziunilor.

O metoda de rezolvare a coliziunilor este de a cauta in tabelul de dispersie un slot disponibil si neocupat care sa poata tine elementul care a cauzat coliziunea. O metoda simpla prin care acest lucru poate fi realizat este sa se porneasca o cautare de la pozitia in care ar fi trebuit elementul sa fie situat, si sa se verifice secvential sloturile pana cand se gaseste primul slot disponibil. De notat este faptul ca este posibil sa nu se gaseasca o pozitie libera pana la finalul parcurgerii tabelului, fiind necesar in acest caz sa reincepem cautarea de la pozitia 0, intr-o maniera circulara. Aceasta metoda de rezolvare a coliziunilor se numeste **open addressing** incrucat incearca sa gaseasca urmatorul slot liber sau urmatoarea adresa libera in tabelul de dispersie. Prin vizitarea sistematica a fiecarui slot la un moment dat efectuam o tehnica de open addressing numita **linear probing.**

Presupunand ca avem urmatorul sir de numere intregi : 54, 26, 93, 17, 77, 31, 44, 55, 20 si functia de hashing f.

$$f(x) = x \,\%\, 11$$

In figura 2 este ilustrata plasarea primelor 6 valori in tabelul de dispersie.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Figura 2. Primele 6 valori din sirul de numere intregi

In momentul in care dorim sa plasam elementul 44 pe pozitia 0 observam ca in acea pozitie avem deja un element, asadar avem de a face cu o coliziune. In cazul rezolutiei cu linear probing, ne uitam in tabel slot cu slot pana gasim o pozitie libera. In acest caz gasim pozitia 1. Pentru elementul 55 iar va trebuii sa cautam un slot liber intrucat pozitia 0 este ocupata. Procedand ca si mai sus gasim pozitia 2 in care

plasam elementul 55. Ultima valoare , 20, ar trebuii plasata pe pozitia slotului 9. Observam ca acea pozitie este ocupata si incepem sa cautam urmatoarea pozitie libera. Pozitiile in care se cauta sunt urmatoarele: 10, 0 , 1 si 2. In final se gaseste pozitia 3 ca fiind libera si se plaseaza elementul 20. In figura 3 este ilustrat tabelul de dispersie final.

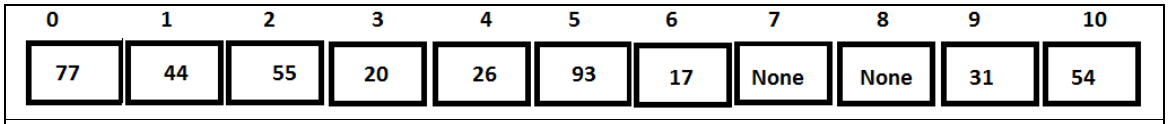

Figura 3. Tabelul de dispersie dupa plasarea numerelor in sloturile corespunzatoare

Este important de mentionat ca modul de gasire al unui elemnent in tabelul de dispersie va trebuii sa fie la fel ca si modul in care elementul a fost inserat – in cazul de mai sus utilizand tehnica de open addressing numita linear probing.

Presupunem ca dorim sa vedem daca elementul 93 a fost adaugat in tabel. Functia de dispersie apelata pentru elementul 93 ne va intoarce pozitia 5. In momentul in care ne uitam pe slotul 5 observam elementul 93. Daca dorim sa gasim elementul 20 functia de hashing ne va intoarce pozitia 9. Cu toate aceastea elementul de pe pozitia 9 este 31. Pentru a verifica daca elementul 20 se afla intradevar in tabelul de dispersie va trebuii sa efectuam o cautare secventiala pornind de la pozitia 10 pana cand fie intalnim elementul 20 sau intalnim un slot gol. Un dezavantaj al metodei folosite este acela ca in cazul in care avem multe coliziuni pentru o pozitie, elementele vor fi tinde sa fie accumulate in jurul pozitiei respective. O metoda de a elimina clusteringul este de a cauta urmatoarea pozitie libera intr-un mod nelinear, spre exemplu cu un pas de 3 pozitii. Asta inseamna ca in momentul in care apare o coliziune vom cauta urmatoarea pozitie valida din 3 in 3. Acest process de a cauta o noua pozitie(slot) dupa ce a avut loc o coliziune se numeste **rehashing**.

$$newHashValue = rehashing(oldHashValue)$$

$$rehashing(pos) = (pos + 3)\%marime\_tabel$$

O variatie a metodei linear probing este quadratic probing. In aceasta metoda urmatorul slot se cauta la o distanta patratica (h + 1, h + 4, h + 9 etc; unde h reprezinta pozitia initiala).

O alternativa de a trata coliziunile este de a lasa fiecare slot sa tina minte o referinta la o colectie de elemente. Spre exemplu fiecare slot poate tine o referinta catre o lista lineara simplu inlantuita. In momentul in care o coliziune se intampla noul element este adaugat in coada listei. Aceasta metoda se numeste **chaining** sau inlantuire. In figura 4 este ilustrata o imagine intuitiva la ce inseamna rezolvarea coliziunilor folosind chaining.

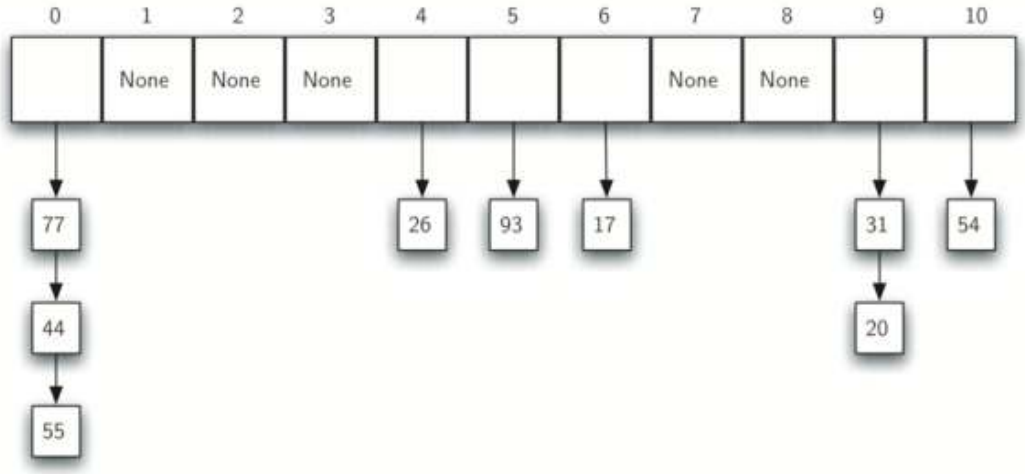

Figure 4. Rezolutia Coliziunilor prin Chaining

Ne putem da seama si intuitiv ca problema pe care aceasta metoda o are este ca mai multe elemente pot fi accumulate in acelasi slot. In cel mai rau caz toate elementele pot fi accumulate in aceeasi pozitie, cautarea facandu-se in O(N).

In cele ce urmeaza o metoda de rezolutie a coliziunilor folosind chaining este implementata si prezentata.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100 //numarul maxim de elmente din tabelul de dispersie
#define NR 11 //numarul la care facem restul impartirii
using namespace std;

typedef struct cell
{
    int val;
    struct cell *next;
}NodeT;

//tabelul nostru de dispersie
NodeT *hTabel[MAX];

int hFunction(int value)
{
    //calculam codul de hashing folosind tehnica modulo
    return value % NR;
}
int FindElement(int Key)
{
    //obtinem codul de hashing
    int h = hFunction(Key);
    NodeT *p;
    //luam elementul corespunzator din tabel
    p = hTabel[h];
```

```
    //verificam daca mai avem o cheie similara in tabel
    while(p != NULL){
        //intoarcem 1 daca am gasit o cheie
        if(p->val == Key) return 1;
        p = p->next;
    }
    // 0 daca nu am gasit nici o cheie la fel
    return 0;
}

void insertElement(int key)
{
    int h;
    NodeT *p;
    p = (NodeT *)malloc(sizeof(NodeT));
    p->val = key;
    h = hFunction(key);
    //daca nu mai avem elemente
    if(hTabel[h] == NULL)
    {
        hTabel[h] = p;
        p->next = NULL;
    }
    else
    {
        int nr = FindElement(key);
        if(nr == 1)
        {
            //daca gasim un element cu aceeasi cheie afisam mesajul de
duplicat si nu mai inseram elementul
            printf(" The element %d is allready present in the tabel \n",key);
        }
        else
        {
            //altfel noul element devine headul
            p->next = hTabel[h];
            hTabel[h] = p;
        }
    }
}
//afisarea tuturor elmentelor din tebela de dispersie
void showAll()
{
    for(int i = 0; i < MAX; i++)
    {
        //verificam daca la slotul i am adaugat ceva
        if(hTabel[i] != NULL)
        {
            printf(" %d :",i);
            NodeT *p;
            p = hTabel[i];
            while (p != NULL)
            {
                printf(" %d ",p->val);
                p = p->next;
```

```
            }
            printf("\n");
        }
        else
        {
            printf(" %d :\n", i);
        }
    }
}
int main(int argc, CHAR* argv[])
{
    int n,x;
    //citim un numar n de elemente
    scanf("%d",&n);
    //initializam tabelul nostru de dispersie
    for(int i = 0; i < MAX; i++)
    {
        hTabel[i] = NULL;
    }
    //afisam toate elementele tabelului
    showAll();
    for(int i = 0; i < n; i++)
    {
        scanf("%d",&x);
        //inseram un nou element in tabel
        insertElement(x);
    }
    //afisam toate elementele din tabel din nou
    showAll();

    return 0;
}
```

Mersul Lucrarii

1. Probleme Obligatorii

1. Sa se implementeze si sa se testeze exemplul de cod din laborator .

2. Sa se implementeze metoda de rezolutie a coliziunilor prin open addressing folosind linear probing.

3. Fiind dat un text al unei limbi si o multime de cuvinte din limba respectiva, care sunt toate de aceeasi dimensiune, determinati numarul de pozitii candidat la care se pot gasii cuvintele textului. Pe prima linie a fisierului Date.in se gaseste alfabetul in care vom cauta, iar pe a doua linie se gasesc cuvintele. In fisierul Date.out sa se scrie numarul de pozitii candidat la care apar cuvintele.

Date.in	Date.out
bbcabbabcba abba bbca	3

abcb bbca aaaa	

Explicatie: bbca se gaseste pe pozitia 0, abba se gaseste pe pozitia 3, abcb se gaseste pe pozitia 6. Celelalte cuvinte nu se gasesc in text. In total am gasit 3 pozitii posibile.

2. Probleme optionale

1. Stergeti elementele duplicate dintr-un array nesortat de dimensiune n.Pe prima linie a fisierului Date.in se gaseste numarul n iar pe a doua linie se gaseste secventa de numere. In fisierul Date.out sa se scrie secventa fara duplicate.

Date.in	Date.out
9 1 2 10 6 2 5 3 1 15 6	1 2 10 6 5 3 15

2. Fiind date un numar de N siruri de intregi sa se determine punctul sau punctele lor de intersectie.