17. TABELE DE DISPERSIE

17.1 Structura de date de tip tabela de dispersie

Cel mai eficient algoritm de căutare este acela în care fiecărei valori din colecţia de elemente i se asociază o poziţie unică în cadrul mulţimii. Astfel, pe baza valorii căutate se determină poziţia acesteia în cadrul colecţiei.

În cadrul vectorilor, această abordare se implementează cu uşurinţă deoarece asocierea dintre valoarea unei chei numerice întreagă şi poziţia acesteia în vector se realizează prin:

- definirea unui vector cu număr de elemente egală cu valoarea maximă posibilă a cheii de căutare;
- se stabileşte o valoare neutilizată în cadrul problemei de rezolvat pentru a indica dacă elementul cu cheia căutată există – deoarece vectorul reprezintă o zonă de memorie continuă, se implementează principiul conform căruia elementele există sau sunt şterse logic; pentru o cheie de căutare ce ia valori în intervalul [0…255], valoarea -1 este fi utilizată pentru a indică inexistenţa elementului căutat în colecţie.

Principalul avantaj al căutării bazate pe acces direct este dat de rapiditate cu care se găseşte elementul căutat sau este indicată inexistenţa acestuia.

Pentru structura *element* funcţia de regăsire este descrisă în secvenţa de cod:

```
struct element
{
int cheie
int valoare
}
int ValoareVector(element* vector, int n, int cheie)
{
      if(cheie >= n) return -1;
            else if(vector [cheie].cheie == -1) return -1;
                  else return vector [cheie].valoare;
}
```

Cu toate acestea, în practică este evitată această soluţie directă care, deşi minimizează timpul de regăsirea prezintă numeroase dezavantaje generate de:

- dimensiunea memoriei ocupate – spaţiul de memorie, MEMORIE, necesar implementării acestei structuri este dat de relaţia

*MEMORIE = maxim(valoare cheie căutare) * dimensiune(element)* (17.1)

  iar în cazul în care valoarea maximă este foarte mare atunci trebuie să fie rezervat un spaţiu considerabil;
- gradului de utilizare a spaţiului – deoarece dimensiunea structurii este dependentă de valoarea maximă a cheii de căutare, nu se ţine cont de numărul real de elemente utilizate; cea mai nefavorabilă situaţie este dată de un raport foarte mic dintre

numărul de elemente şi valoarea maximă a cheii; de exemplu, se consideră o aplicaţie informatică ce gestionează studenţii din cadrul unei facultăţi reprezentaţi de structura:

```
struct Student
{
      char nume[20];
      int varsta;
      char facultate[20];
      int nrMatricol;
}
```

pentru a minimiza timpul de regăsire, se implementează o structură cu acces direct în care numărul matricol al studentului este egal cu poziţia în cadrul colecţiei a respectivului element; în cazul în care valoarea maximă a câmpului *nrMatricol* este egală 55630, iar numărul real de studenţi este egal cu 1450 rezultă un spaţiu ocupat egal cu

$$MEMORIE=max(nrMatricol)*dimensiune(Student)=55630*48=2,54\ MB \quad (17.2)$$

iar gradul de utilizare a acestui spaţiu este egal cu:

$$GUM = \frac{MEM_{elemente}}{MEM} * 100 = \frac{1450 * 48}{55630 * 48} * 100 = 2,6\% \quad (17.3)$$

- tipului cheii de căutare – acesta trebuie să aibă un tip numeric întreg deoarece trebuie să corespundă indexului cu care se accesează elementele unui vector.

Pentru a remedia dezavantajele utilizării structurilor cu acces direct sunt definite tabelele de dispersie, *hash tables*, ce reprezintă colecţii de date în care pe baza unei funcţii *hash* cheia de căutare este pusă în corespondenţă cu poziţia elementului în cadrul colecţiei. Avantajul acestei tip de structură de stocare şi căutare este dat de:

- utilizarea mai eficientă a spaţiului fără a se stoca memorie pentru elemente care nu sunt utilizate – în cazul aplicaţiei de gestiune a studenţilor se observă că din totalul de 55630 de elemente, doar 1450 sunt utilizate; faptul se datorează concentrării valorilor cheii de căutare în intervalul [54130, 55630]; reducerea spaţiului ocupat de tabela de dispersie se realizează în această situaţie prin implementarea unei funcţii *hash*: [54130, 55630] → [0, 1500] care să reducă valoarea maximă a cheii de căutarea prin conversia valorii din intervalul [54130, 55630] într-o valoare din intervalul [0, 1500]; o astfel de funcţie *hash* cu un nivel de complexitate scăzut este

$$hash(X) = X \text{ modulo } 1500, \text{ cu } X \in [54130,55630] \quad (17.4)$$

- implementarea de chei alfanumerice – prin utilizarea unei funcţii care să prelucreze cheia de căutare este depăşită bariera care limita pentru structurile cu acces direct tipul cheii de căutare la o

valoarea numerică întreagă; în această situaţie rolul funcţiei *hash* este de a translata valoarea alfanumerică într-o valoare întreagă pozitivă; de exemplu, în limbajul Java este implementată funcţia *hash* pentru un string

$$hash(S) = s[0]*31^{n-1}+s[1]*31^{n-2}+\ldots+s[n-2]*31+s[n-1] \qquad (17.5)$$

unde:

- s[i] – codul ASCII al caracterului *i* din şir;
- n – dimensiunea şirului de caractere.

Pentru cheia alfanumerică cu valoarea *salut* aplicarea funcţiei *hash* conduce la obţinerea valorii

$$hash(\text{“salut”})=115*31^4+97*31^3+108*31^2+117*31+116=109202173 \quad (17.6)$$

valoarea obţinută este introdusă din nou într-o funcţie *hash* care să identifice poziţia corespondentă din mulţime pentru această valoare numerică foarte mare.

Dezavantajul tabelelor de dispersie în cadrul proceselor de căutare este descris de:

- efortul suplimentar de prelucrare dat de funcţia *hash* care poate avea în unele situaţii un nivel de complexitate ridicat;
- apariţia în cadrul tabelei a coliziunilor – acestea sunt generate de situaţia în care două valori Xh şi YH ce aparţin domeniului de definiţie a funcţie *hash* conduc la obţinerea de valori identice, hash(Xh) = hash(Yh); evitarea coliziunilor implică realizarea de operaţii suplimentare prin care să se identifice poziţia elementului în cadrul mulţimii; printre cele mai utilizate tehnici de evitare a coliziunilor se numără *chaining, re-hashing, linear probing, quadratic probing* şi *overflow area*, [Will96], [Seng94].

Din aceste puncte de vedere, tabela de dispersie utilizată în procesul de regăsire a informaţiei se descrie ca o structură în care datele sunt stocate în tabele cu adresare directă, iar poziţia acestora este determinată prin prelucrarea cheii de căutare cu o funcţie *hash*, figura 17.1.

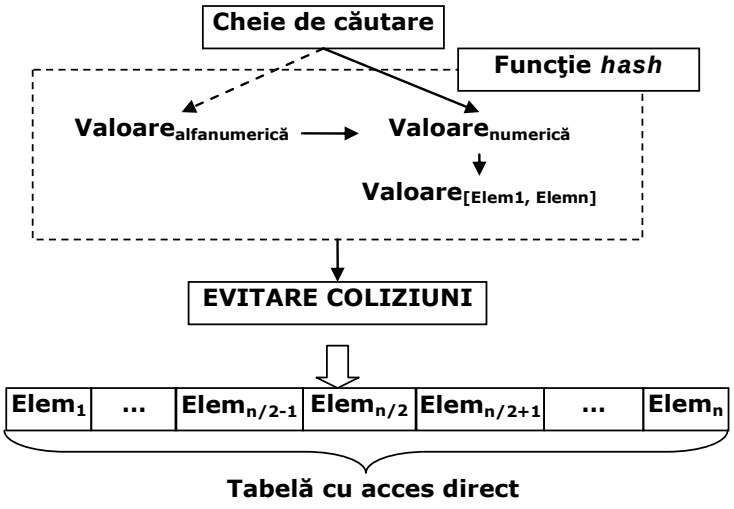

Figura 17.1 *Tabela de dispersie*

17.2 Chei si funcţii *hash*

Pentru a identifica în mod unic fiecare înregistrare sunt utilizate valori dintr-un domeniu bine definit. Asocierea dintre aceste valori unice şi înregistrările pe care le reprezintă este de unu la unu. În funcţie de tipul valorii cu rol de cheie se identifică:

- chei numerice reprezentate prin intermediul tipurilor fundamentale definite de limbajul de programare utilizat;
- chei alfanumerice reprezentate prin şiruri de caractere;
- chei compuse definite pe baza a mai multor atribute;

Rolul funcţiei *hash* este de a prelucra cheia asociată fiecărei înregistrări şi de a determina poziţia în cadrul tabelei de dispersie a elementului respectiv. Deoarece nu există o funcţie *hash* generală, iar alegerea acesteia se face în funcţie de caracteristicile mulţimii de valori chei, sunt utilizate modele matematice bazate pe:

- împărţire în modul; această metodă are un grad ridicat de utilizare în diferitele modele de implementare a tabelelor de dispersie prin prisma complexităţii scăzute a modelului şi a uşurinţei în implementare; indiferent de forma cheii de căutarea, aceasta este transformată într-o valoare numerică şi apoi transpusă în mulţimea [0…nht-1], unde *nht* reprezintă dimensiunea tabelei de dispersie; acest tip de funcţie are asociat modelul:

$$pozitie\_tabela = val\_cheie \ \% \ val\_baza \qquad (17.7)$$

unde:

- *pozitie_tabela* – valoarea *hash* obţinută;
- *val_cheie* – valoarea numerică ce identifică unic fiecare înregistrare;
- *val_baza* – valoarea numerică stabilită la definirea modelului;

Valoarea utilizată ca bază în modelul funcţiei *hash* este dată de dimensiunea tabelei de dispersie; obiectivul definirii funcţiei *hash* este de a obţine un model care să minimizeze mulţimea de chei diferite ce conduc la aceeaşi valoare *pozitie_tabela*; în acest sens se utilizează numere prime apropiate de numărul total de înregistrări; astfel, dacă cheile înregistrărilor definesc o mulţime continuă de valori unice, valorile *hash* determinate vor acoperi toată mulţimea [0..*val_baza-1*]; pentru a stabili dimensiunea iniţială a tabelei de dispersii sunt alese numere prime particulare; de exemplu limbajul Java defineşte o clasă *HashTable* ce descrie o tabelă de dispersie cu dimensiunea iniţială egală cu 101; valoarea nu este aleasă la întâmplare deoarece permite redimensionarea tabelei la o nouă valoare primă; pentru cazuri generale, este indicat să se folosească un număr prim determinat prin relaţia:

$$dimensiune\_hash = (4*i+3) \text{ cu } i=0, 1, 2, 3,\ldots \qquad (17.8)$$

- înmulţirea cu un număr real aleatoriu şi prelucrarea ulterioară a părţii zecimale – valoarea cheii de căutare este înmulţită cu un număr din intervalul [0;1) ce este ales aleatoriu; în urma

înmulţirii se obţine o valoarea temporara, *temp*, a cărei parte zecimală are valori cuprinse în intervalul [0 ;1); înmulţind *temp* cu dimensiunea tabelei de dispersie, *dimensiune_hash*, se obţine o valoare în intervalul [0, *dimensiune_hash-1*] ce reprezintă poziţia elementului căutat în tabelă; relaţia 17.9 descrie procesul de determinare a valorii *hash*:

$$val\_hash=((val\_cheie*random_{[0;1)})-[(val\_cheie*random_{[0;1)})])*nth \quad (17.9)$$

unde:

- *val_hash* – valoarea *hash* calculată;
- *val_cheie* – valoarea cheii de căutare;
- $random_{[0;1)}$ – număr aleatoriu din mulţimea [0;1);
- *nth* – dimensiunea tabelei de dispersie.

- utilizarea unui bloc de biţi obţinut prin prelucrarea cheii de căutare – de exemplu se defineşte funcţia care multiplică valoarea codului şi care determină valoarea *hash* pe baza celor 10 biţi din interiorul acesteia; dacă se consideră cheile reprezentate pe 32 de biţi atunci relaţia este:

$$val\_hash = ((val\_chei * val\_cheie)>>11) \% nhash \quad (17.10)$$

- prelucrarea codurilor ASCII ale caracterelor alfanumerice – în cazul în care cheia de căutare este dată de o secvenţă de caractere atunci este necesar ca aceasta să fie prelucrată pentru a se genera un număr întreg care să indice poziţia elementului în tabelă; prelucrarea se face pe baza codului ASCII asociat caracterului; pe baza primului caracter din cheie se defineşte relaţia:

$$val\_hashs_1 = string\_cheie[0] \% 255 \quad (17.11)$$

unde:

- val_hashs_1 – valoarea *hash* calculată;
- *string_cheie* – valoarea cheii de căutare;

Relaţia anterioară defineşte un model cu un nivel de complexitate scăzut ce este destinat gestiunii unei colectivităţi mici de elemente; în cazul în care valoarea cheii reprezintă numele unei persoane, modelul este ineficient deoarece generează multe coliziuni pentru nume diferite dar care încep cu acelaşi caracter; rafinarea acestui model se poate face prin preluarea mai multor caractere din şirul respectiv; de exemplu se defineşte o nouă relaţie

$$val\_hashs_2=(string\_cheie[0]+string\_cheie[lungime_{string\_cheie}])\%dim\_has \quad (17.12)$$

unde:

- val_hashs_2 – valoarea *hash* calculată;
- *string_cheie* – valoarea cheii de căutare;
- $lungime_{string\_cheie}$ – dimensiunea şirului de caractere;

- *dim_hash* – dimensiunea tabelei de dispersie;

ce ia în calcul primul şi ultimul caracter; de asemenea, pentru a nu reduce dimensiunea tabelei la maxim 255 elemente, se utilizează un număr prim, *dim_hash* suficient de mare; alte funcţii *hash* de prelucrare a cheilor alfanumerice analizează toate caracterele din şir; de exemplu relaţia

$$val\_hashs_3 = \sum_{i=1}^{lungime\_cheie} ASCII(string\_cheie[i]) \ \%\ dim\_hash \qquad (17.13)$$

unde:

- *val_hashs₃* – valoarea *hash* calculată;
- *string_cheie* – valoarea cheii de căutare;
- lungime_cheie – dimensiunea şirului de caractere;
- *dim_hash* – dimensiunea tabelei de dispersie;

determină valoarea *hash* pe baza sumei codurilor ASCII asociate tuturor caracterelor din cheie;

- înmulţirea cu un număr real definit; această valoare este determinată prin evaluarea expresiei $m = \frac{\left(\sqrt{5}-1\right)}{2}$ sau a expresiei $m = 1 - \frac{\left(\sqrt{5}-1\right)}{2}$; funcţia *hash* definită pe baza acestui model este

$$f(val\_cheie) = [val\_cheie * m] \qquad (17.14)$$

acest model este implementat în cadrul funcţiile de *rehashing* ce sunt utilizate în cadrul procedurilor de rezolvare a coliziunilor.

Dificultatea în definirea funcţiilor *hash* constă în identificare acelui model care să minimizeze numărul coliziunilor şi care să genereze o distribuţie uniformă a valorilor pe întreg intervalul.

17.3 Evitarea coliziunilor

Funcţiile de evitare a coliziunilor definesc metode de regăsire a elementelor ce sunt descrise de chei cu valori diferite dar care conduc la valori *hash* identice ceea ce implică poziţii identice în cadrul structurii:

- *chaining* implementrează lucrul cu liste – fiecare poziţiei din cadrul tabelei de dispersie conţine capătul unei liste de elemente cu valori hash egale; regăsirea unui element presupune determinarea poziţiei în cadrul tabelei prin calcularea valorii *hash* şi parcurgerea secvenţială a listei ataşate poziţiei respective după valoarea cheii de căutare, figura 17.2;

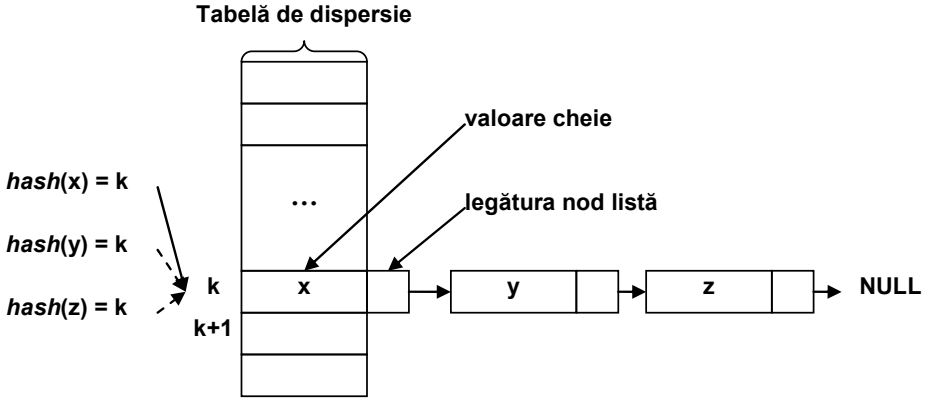

Figura 17.2 *Metoda chaining de evitare a coliziunilor în tabele de dispersie.*

- *re-hashing* presupune aplicarea în cascadă a aceleiaşi funcţii *hash* sau a altui model dintr-o mulţime de funcţii până când valoarea obţinută reprezintă o poziţie liberă din cadrul tabelei de dispersie; la fiecare pas al procesului de regăsire, valoarea cheii de căutare este introdusă într-o listă de funcţii *hash* până când se identifică elementul cu valoarea căutată sau nu mai există alte posibilităţi de a recalcula valoarea *hash*; tipul şi numărul de funcţii *hash* aplicate valorilor intermediare descriu o procedură bine definită de căutare, respectiv iniţializare element nou, pentru a conduce de fiecare dată la aceleaşi rezultate, figura 17.3;

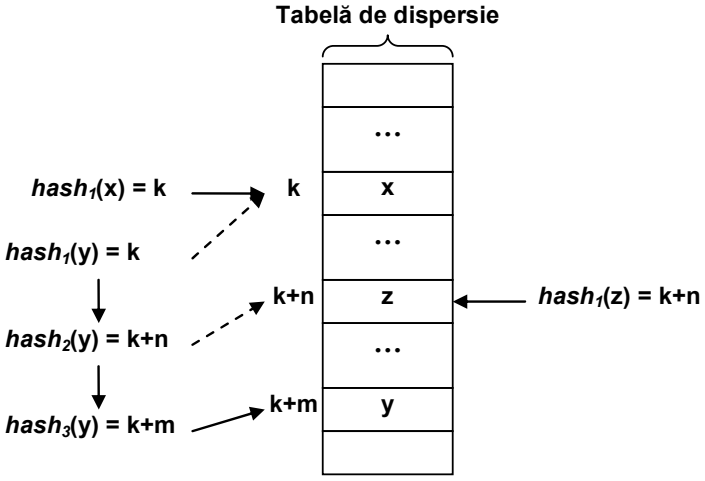

Figura 17.3 *Metoda rehashing de evitare a coliziunilor în tabele de dispersie*

- *linear probing* se bazează pe căutarea secvenţială a primei poziţii libere în care să fie inserat elementul nou, poziţie aflată la stânga sau la dreapta coliziunii; în cazul căutării, procesul presupune verificarea elementelor adiacente poziţiei indicate de valoarea *hash*; cu toate că are un grad scăzut de complexitate, această metodă de evitare a coliziunilor are ca efecte secundare gruparea coliziunilor de acelaşi tip în aceeaşi zonă, *cluster*, fapt care conduce la creşterea probabilităţii de apariţie a coliziunilor pentru valorile *hash* adiacente, figura 17.4;

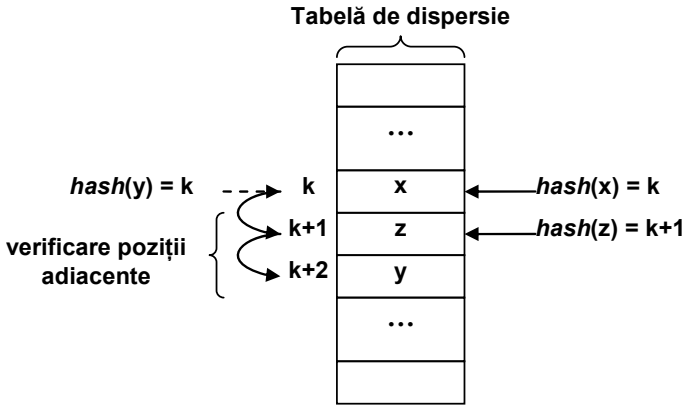

Figura 17.4 *Metoda* linear probing *de evitare a coliziunilor în tabele de dispersie*

- *quadratic probing* este o metodă de tipul *linear probing* care evită crearea grupurilor de coliziuni prin utilizarea unui pas de regăsire a următoarei poziţii libere diferit de 1; astfel, în caz de coliziuni au loc salturi în tabela de dispersie din două în două poziţii sau din patru în patru; în general, pentru a determina următoare poziţie în care să se insereze un element nou sau în care să se caute un element existent este dat de funcţia:

$$poziţie = hash(X)+c*i^2 \qquad (17.15)$$

unde:

- *poziţie* – noua poziţie din tabela de dispersie în care se inserează sau se caută un element;
- *X* – valoarea cheii asociate elementului;
- *hash(X)* – poziţia indicată de valoarea *hash* a elementului care se adaugă sau se căută;
- *c* – valoare constantă definită în mulţimea {1, 2, 4};
- *i* – numărul operaţiei de *rehash* sau numărul de poziţii verificate.

Prin utilizarea de repoziţionări pe poziţii neadiacente se evită gruparea coliziunilor în aceeaşi zonă din tabela de dispersie, figura 17.5;

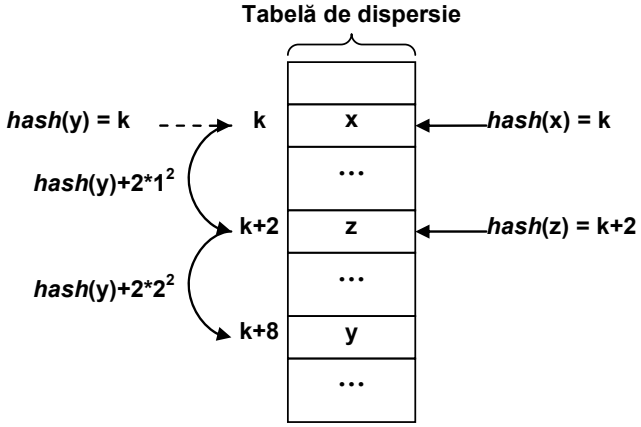

Figura 17.5 *Metoda* quadratic probing *de evitare a coliziunilor în tabele de dispersie*

- *overflow area* presupune o abordare ce împarte tabela de dispersie în două zone, primară pentru reţinerea elementelor iniţiale şi secundară alocată elementelor ce generează coliziuni; la apariţia unei coliziuni, fie la operaţia de regăsire sau la adăugarea unui element nou, se utilizează un element al zonei secundare pentru a reţine noua valoare sau pentru a continua căutarea; accesul la zona secundară se realizează doar prin intermediul unui pointer din zona primară, funcţia hash negenerând poziţii în acest interval de valori; această metodă, descrisă în figura 17.6, permite o regăsire mai rapidă a informaţiilor decât metoda *chaining* care presupune parcurgerea de liste;

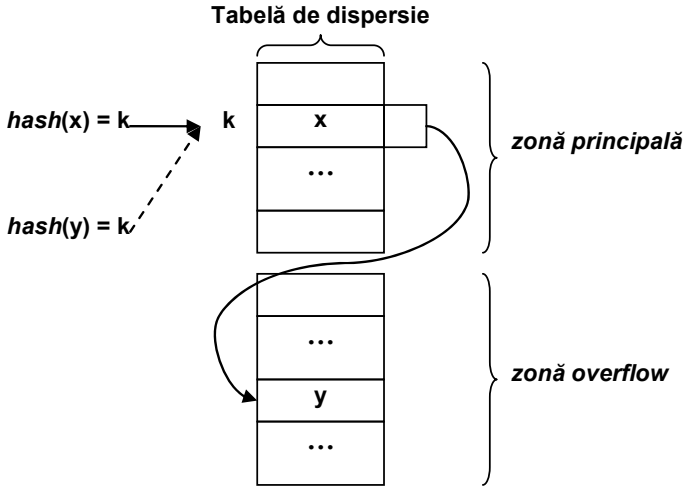

Figura 17.6 *Metoda* overflow *de evitare a coliziunilor în tabele de dispersie*

Se observă în cazul tabelelor de dispersie că probabilitatea de apariţie a coliziunilor la inserare sau la căutare creşte proporţional cu gradul de utilizare a tabelei. În cele mai multe cazuri, funcţiile *hash* cu un grad redus de complexitate nu conduc la rezultate unice pentru valori de intrare distincte. Din acest motiv, cu cât tabela pune la dispoziţie un număr din ce în ce mai mic de poziţii disponibile, creşte riscul de a avea numeroase elemente cu chei de căutare diferite care ar trebui să se regăsească pe poziţii identice. Metodele de evitare a coliziunilor rezolvă parţial problema deoarece ele plasează elementele noi pe alte poziţii definind o nouă cauză pentru coliziuni viitoare.

Pentru a rezolva această situaţie şi pentru a menţine eficienţa operaţiei de căutare la un nivel acceptabil, astfel încât să nu se parcurgă toată mulţimea de elemente la fiecare operaţie de regăsire de informaţie, se analizează gradul de utilizare a tabelei *hash*, GU_{hash}.

$$GU_{hash} = \frac{NPO}{NTP} * 100 \qquad (17.16)$$

unde:
- *NPO* – numărul de poziţii ocupate;
- *NTP* – numărul total de poziţii.

care este menţinut la o valoarea mai mică de 50%. În ciuda faptului că are loc o gestiune ineficientă a spaţiului, eficienţa abordării se bazează pe o viteză de regăsire mare. Pentru a menţine valoarea indicatorului GU_{hash} sub limită impusă, tabela se redimensionează prin dublarea numărului de poziţii. În cazul în care modelul funcţiei hash este bazat pe împărţirea în modul la dimensiunea acesteia, redimensionarea trebuie să conducă la o nouă dimensiune care să reprezinte tot un număr prim. Având în vedere importanţa acestor tipuri de numere, lucru evidenţiat la descrierea tipurilor de funcţii hash, redimensionarea se face prin intermediul relaţiei

*dimensiune_noua = dimensiune_curenta*2 + 1 (17.17)*

În cazul limbajului Java, unde dimensiunea iniţială a tabelei de dispersie este 101, redimensionarea ulterioară a acesteia utilizând relaţia descrisă conduce la o serie de valori care reprezintă tot valori prime. De exemplu 203 şi 407. În cazul în care relaţia utilizată pentru a determina numărul prim este

*dimensiune_hash = (4*i+3) cu i=0, 1, 2, 3, … (17.18)*

redimensionarea se realizează prin înlocuirea lui *i* cu *2*i*. Astfel, noua dimensiune va fi reprezentată tot de un număr prim.

17.4 Implementare

În continuare se exemplifică printr-un program scris în limbajul de programare C++ implementarea unei tabele de dispersie pentru înregistrări de tip *Student*, căutările realizându-se după numărul matricolei. Ca funcţie de dispersie a fost ales un exemplu simplu cu ajutorul operatorului modulo (%). Dimensiunea implicită a tabelei a fost aleasă 101, o valoare cuprinzătoare pentru exemplul ales.

Tabela de dispersie este implementată sub forma unui masiv unidimensional alocat dinamic, ale cărui elemente sunt liste care conţin datele propriu zise. Acest mod de implementare permite tratarea coliziunilor, în cazul în care pentru indexul generat există o înregistrarea în tabelă, noua înregistrare va fi adăugată ca element la lista simplu înlănţuită de la nodul respectiv.

```
#include "stdafx.h"
#include <iostream>

using namespace std;

//structura de date stocata in tabela de dispersie
//pentru simplificarea codului s-a utilizat o structura
struct Student
{
	int matricola;
	char nume[20];
	char adresa[30];
	char localitate[20];
	char e_mail[20];
	char telefon[15];
```

```
	void Afisare()
	{
		cout<<"Matricola: "<< matricola <<endl;
		cout<<"Matricola: "<< nume <<endl;
		cout<<"Matricola: "<< adresa<<endl;
		cout<<"Matricola: "<< localitate<<endl;
		cout<<"Matricola: "<< e_mail <<endl;
		cout<<"Matricola: "<< telefon <<endl;
	}
};

//elementele tabelei de dispersie
//pentru simplificarea codului s-a utilizat o structura
struct NodDate
{
	NodDate *urm;
	Student data;
};

//Cheia este matricola studentului
//
class HashStudent
{
	int size;
	NodDate ** elem;

	//functie simpla de dispersie care utilizeaza operatorul modulo
	//pentru a genera pozitia in tabela
	int HashFunction(int cheie)
	{
		return cheie%size;
	};

	//aloca memorie pentru structura si initializeaza cu null
valorile
	void AlocaMemorie()
	{
		elem = new NodDate*[size];
		for (int i=0;i<size;i++)
			elem[i] = NULL;
	}
	void ElibereazaMemorieLista(NodDate *&);
public:
	HashStudent():size(101)
	{
		AlocaMemorie();
	}
	~HashStudent();
	HashStudent(int iSize): size (iSize)
	{
		AlocaMemorie();
	}
	int Insereaza(Student data);
	Student Cauta(int matricola);
	int Sterge(int matricola);
	void HashStudent::Afiseaza();
};


//insereaza un element pe o pozitie generata de functia de dispersi
```

```
//pe baza valorii matricolei din structura
int HashStudent::Insereaza(Student s)
{
	int index = -1;

	if (s.matricola<0)
		return index;

	if (elem!=NULL)
	{
		index = HashFunction(s.matricola);
		//prima inregistrare
		NodDate *nod_nou = new NodDate;;
		nod_nou->urm = NULL;
		nod_nou->data = s;

		if (elem[index]== NULL)
		{
			elem[index] = nod_nou;
		}
		else //coliziune
		{
			NodDate *t = elem[index];
			while(t->urm!=NULL)
				t = t->urm;
			t->urm = nod_nou;

		}
	}
	return index;
}

//cauta un element dupa chiea matricola
//in cazul in care nu este gasita nici o inregistrare este
//returnata o structura Student cu matricola -1
Student HashStudent::Cauta(int matricola)
{
	//valoare de retur pentru cazurile in care nu este gasit
elementul
	Student s_negasit;
	s_negasit.matricola = -1;

	if (matricola<0)
		return s_negasit;

	if (elem!=NULL)
	{
		int index = HashFunction(matricola);
		if (elem[index]==NULL)
		{
			return s_negasit;
		}
		else
		{
			if (elem[index]->data.matricola ==matricola)
				return elem[index]->data;
			else//coliziune
			{
			NodDate *t = elem[index];
			while (t!= NULL && t->data.matricola!=matricola)
					t = t->urm;
```

```
                        if (t==NULL)
                              return s_negasit;
                        else
                              return t->data;
                  }
            }
      }
      //neinitializat sau nu exista inregistrarea
      return s_negasit;
}

//functie recursiva care eliberaza memoria ocupata de nodurile listei
void HashStudent::ElibereazaMemorieLista(NodDate *&inceput)
{
      if (inceput == NULL)
            return;
      else
      {
            ElibereazaMemorieLista(inceput->urm);
            cout<<"Eliminat nodul cu codul: " <<inceput-
>data.matricola<<endl;
            delete inceput;
      }

}
//elibereaza memoria ocupa de tabela de dispesie
HashStudent::~HashStudent()
{
      if (elem != NULL)
      {
            for (int i=0;i<size;i++)
                  ElibereazaMemorieLista(elem[i]);
            delete [] elem;
      }
}

//Sterge elementul a carui matricola coincide cu matricola trimisa ca
parametru
//returneaza pozitia din tabela de a fost eliminat elementul
//sau -1 in cazul in care nu exista elementul cu cheia data
int HashStudent::Sterge(int matricola)
{
      if (matricola<0)
            return matricola;

      if (elem!=NULL)
      {
            int index = HashFunction(matricola);
            if (elem[index]==NULL)
            {
                  return -1;
            }
            else
            {
                  //este primul
                  if (elem[index]->data.matricola ==matricola)
                  {
                        if(elem[index]->urm == NULL)
                        {
                              delete elem[index];
                              elem[index]=NULL;
```

```
                        }
                        else//mai sint si alte elemente in lista
                        {
                              NodDate *t = elem[index];
                              elem[index] = t->urm;
                              delete t;
                        }
                  }
                  else//coliziune, nu este primul, il cautam
                  {
                        NodDate *t = elem[index];
                        while (t->urm!= NULL && t->urm->data.matricola
!=matricola)
                              t = t->urm;
                        if (t->urm==NULL)
                              return -1;
                        else//t->urm.data.matricola == matricola
                        {
                              NodDate *p = t->urm;
                              if (p->urm ==NULL)
                              {
                                    t->urm = NULL;
                                    delete p;
                              }
                              else
                              {
                                    t->urm = p->urm;
                                    delete p;
                              }
                        }

                  }
                  return index;
            }

            return -1;
      }
}

//Afiseaza continutul tabelei de dispersie
void HashStudent::Afiseaza()
{
      if (elem!=NULL)
      {
            for (int i=0;i<size;i++)
            {
                  if (elem[i]!= NULL)
                  {

                        NodDate *t = elem[i];
                        while (t!=NULL)
                        {
                              cout<<"S-a inserat in pozitia: " << i <<
" cheia: "<<t->data.matricola<<endl;
                              t = t->urm;
                        }

                  }
            }
      }
```

```
}

int _tmain(int argc, _TCHAR* argv[])
{
      Student grupa[] ={
            {1000,"Ion Vlad", "Str. Norilor
20","Cluj","ion@server.ro", "0101010101"},
            {204,"Mihai Vlad", "Str. Norilor
20","Cluj","mihai@server.ro", "0101010101"},
            {406,"Dan Vlad", "Str. Norilor 20","Cluj","dan@server.ro",
"0101010101"},
            {305,"Lili Vlad", "Str. Norilor
20","Cluj","lili@server.ro", "0101010101"},
            {1022,"Silviu Vlad", "Str. Norilor
20","Cluj","silviu@server.ro", "0101010101"},
            {1021,"Alina Vlad", "Str. Norilor
20","Cluj","alina@server.ro", "0101010101"},
            {1030,"Sorin Vlad", "Str. Norilor
20","Cluj","sorin@server.ro", "0101010101"},
            {1032,"Titi Vlad", "Str. Norilor
20","Cluj","titi@server.ro", "0101010101"},
            {1200,"Gigel Vlad", "Str. Norilor
20","Cluj","gigel@server.ro", "0101010101"},
            {2021,"Anca Vlad", "Str. Norilor
20","Cluj","ioanca@server.ro", "0101010101"},
            {1230,"Maria Vlad", "Str. Norilor
20","Cluj","maria@server.ro", "0101010101"},
            {1008,"Elena Vlad", "Str. Norilor
20","Cluj","elena@server.ro", "0101010101"}
      };
      HashStudent hs;

      for (int i=0;i<sizeof(grupa)/sizeof(Student);i++)
      {
            hs.Insereaza(grupa[i]);
      }
      hs.Afiseaza();
      int m;
      cout<<"Matricola: ";
      cin>>m;

      while (m!=-1)
      {

            Student rez = hs.Cauta(m);
            if (rez.matricola != -1)
                  rez.Afisare();
            cout<<"Matricola: ";
            cin>>m;

      }
      hs.Sterge(1008);
      hs.Sterge(406);
      hs.Sterge(305);
      hs.Afiseaza();
      cin.get();
      return 0;
```

După rulare, programul produce următoarele rezultate la dispozitivul standard de ieşire:

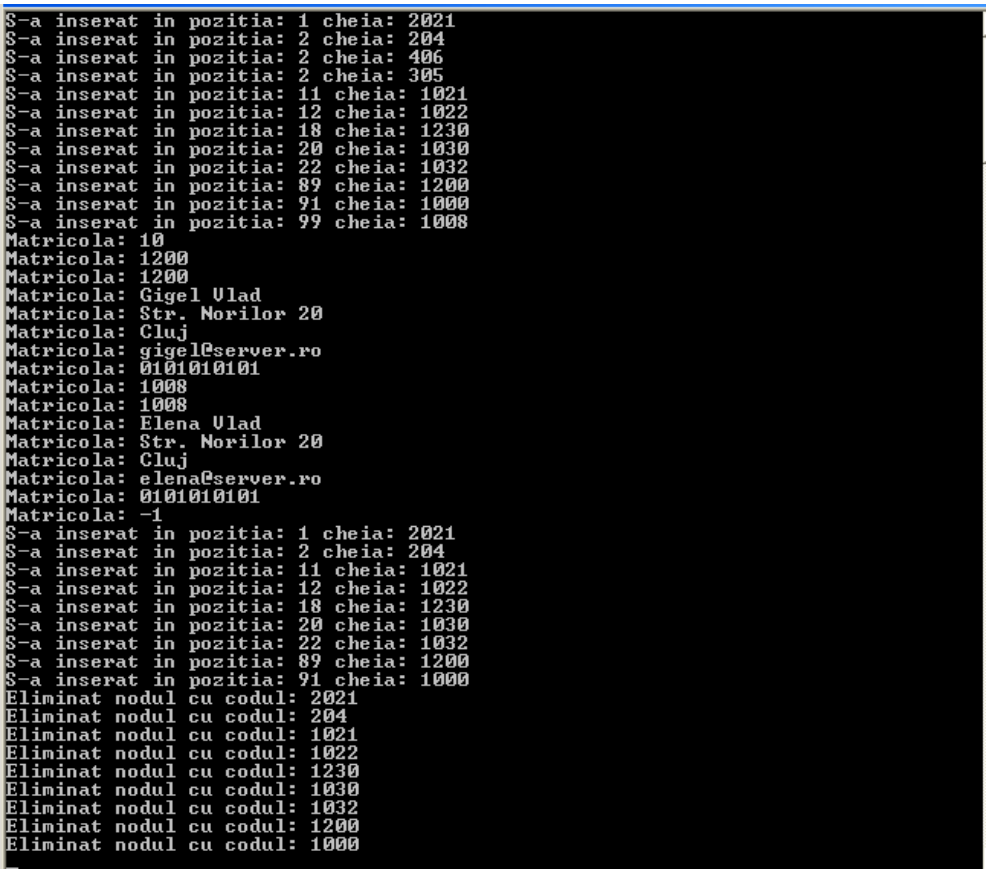

Figura 17.7 *Rezultatele programului HashStudent*