7. Tabele

7.1. TDA Tabelă

- Noțiunea de tabelă este foarte cunoscută, fiecare dintre noi consultând tabele cu diverse ocazii.
- Ceea ce interesează însă în acest context se referă la evidențierea acelor caracteristici ale tabelelor care definesc **tipul de date abstract tabelă**.
- În figura 7.1.a apare un exemplu de tabelă în două variante.

Nume Prenume	An	Medie
Antonescu Ion	3	7,89
Bărbulescu Petre	2	9,20
Card Gheorghe	5	9,80
Mare Vasile	2	8,33
Suciu Horia	1	9,60

Nume Prenume	An	Medie
Suciu Horia	1	9,60
Bărbulescu Petre	2	9,20
Mare Vasile	2	8,33
Antonescu Ion	3	7,89
Card Gheorghe	5	9.80

Fig.7.1.a. Exemple de tabele

- După cum se observă, tabela din figură, ca şi marea majoritate a tabelelor, este alcătuită din **articole**.
  - Acest lucru **nu** este însă obligatoriu, deoarece spre exemplu prima coloană a acestei tabele poate fi considerată la rândul ei o **tabelă** (tabela studenților înscrişi).
  - În prima variantă studenții sunt aranjați în **ordine alfabetică**, în a doua în **ordinea crescătoare** a **anilor** de studiu.
- Se spune că tabela este **ordonată** după o "cheie" element care facilitează regăsirea informațiilor pe care le conține.
  - Astfel o "**cheie**" este un câmp sau o parte componentă a unui câmp care **identifică** în mod unic o **intrare** în tabelă.
  - **Nu** este necesar ca tabela să fie **sortată** după o cheie însă o astfel de sortare simplifică căutarea în tabelă.
- Dintre operațiile frecvente care se execută asupra tabelelor se menționează:
  - În primul rând **căutarea** în tabelă în vederea localizării informațiilor care satisfac una sau mai multe condiții.
  - În anumite situații se impune prelucrarea **integrală** a tabelei într-un proces de "**traversare**" a acesteia.
  - Prelucrarea unei anumite intrări a tabelei în cadrul procesului de traversare se numește "**vizitare**" ("**visiting**").
  - Vizitarea poate include toate genurile de prelucrări începând de la **afişare** şi terminând cu **modificarea** conținutului intrării.
- În definitiv TDA Tabelă poate fi descris în mod sintetic după cum urmează [7.1.a].

TDA Tabelă

Modelul Matematic: O secvență finită de elemente. Fiecare

element are cel puţin o cheie care identifică în mod unic intrarea în tabelă şi care este un câmp sau un subcâmp al elementului.

Notaţii:

```
TipElement    - tipul elementelor tabelei.
TipCheie          - tipul cheii.
t: TipTabela;
e: TipElement;
k: TipCheie;
b: boolean.                                    [7.1.a]
```

Operatori:

1. ***CreazăTabelaVidă****(t: TipTabela);* - operator care face tabela *t* vidă.
2. ***TabelăVidă****(t: TipTabela): boolean;* - operator boolean care returnează **true** dacă tabela *t* este goală.
3. ***TabelăPlină****(t: TipTabela): boolean;* - operator boolean care returnează **true** dacă tabela e plină.
4. ***CheieElemTabelă****(e: TipElement): TipCheie;* -operator care returnează cheia elementului *e*.
5. ***CautăCheie****(t: TipTabela,k: TipCheie): boolean;* - operator care returnează **true** dacă cheia *k* se găseşte în tabela *t*.
6. ***InserElemTabelă****(t: TipTabela,e: TipElement);* - operator care înserează elementul *e* în tabela *t*. Se presupune că *e* nu există în tabelă.
7. ***SuprimElemTabelă****(t: TipTabela,k: TipCheie);* - operator care suprimă din *t* elementul cu cheia *k*. Se presupune că există un astfel de element în *t*.
8. ***FurnizeazăElemTabelă****(t: TipTabela,k: TipCheie): TipElement;* - operator care returnează elementul cu cheia *k*. Se presupune că elementul aparţine lui *t*.
9. ***TraverseazăTabelă****(t: TipTabela, Vizitare (Listă Argumente))* - operator care execută procedura *Vizitare* pentru fiecare element al tabelei *t*.

- În anumite implementări este convenabil să se cunoască **numărul** curent de elemente conţinute în tabelă.
  - În acest scop se poate asocia tabelei un **contor** de elemente şi un operator care-l furnizează *(**DimensTabelă**(t:TipTabela):TipContor).*
  - Contorul se iniţializează la crearea tabelei şi se actualizează ori de câte ori se realizează inserţii sau suprimări în tabelă.
- În unele aplicaţii este convenabil de asemenea să se definească operaţia ***Actualizare****(t: TipTabela, e: TipElement)*
  - Operaţia realizează **actualizarea** acelei intrări din tabelă care conţine elementul *e*.
  - În termenii operatorilor anterior definiţi, actualizarea poate fi implementată astfel [7.1.b]:

```
{TDA Tabelă - operatorul Actualizare}

    SuprimElemTabelă(t,CheieElemTabelă(e));
    InserElemTabelă(t,e)                       [7.1.b]
```

7.2. Tehnici de implementare a TDA Tabelă. Analiză comparată

- Există în principiu mai multe posibilități de **implementare** a tipului de date abstract tabelă.
- Este vorba despre tablouri, liste înlănțuite, tabele de dispersie, arbori etc.
- Dintre aceste posibilități în cadrul paragrafului de față vor fi analizate cele bazate pe structurile **tablou** şi listă **înlănțuită**.
- Implementarea bazată pe tehnica dispersiei face obiectul secțiunii următoare iar cea bazată pe arbori binari va fi tratată ulterior.
- În general **listele înlănțuite** sau **tablourile**, ambele în variantă **ordonată** sau **neordonată** se constituie în 4 metode posibile de implementare a tabelelor.
    - În continuare se va realiza o **analiză comparată** a acestor metode, detaliile de implementare fiind lăsate ca exercițiu.
    - Aceasta analiză are un caracter de **generalitate** care se răsfrânge şi asupra altor tipuri de date în a căror implementare sunt implicate structurile tablou respectiv listă înlănțuită.
    - În esență, în fiecare dintre cele 4 situații vor fi evidențiate **avantajele**, **dezavantajele**, **concluziile** şi **recomandările** referitoare la utilizare.
- Observația **fundamentală** de la care porneşte orice implementare a unui **TDA** este aceea că **particularitatea aplicației** în care este el utilizat, **determină** cea mai potrivită **metodă de implementare**
- **Particularitatea** unei aplicații derivă direct din **natura** şi din **dinamica operațiilor** efectuate asupra structurii de date.

7.2.1. Implementarea TDA Tabelă cu ajutorul tabourilor ordonate

a) Avantaje

- Posibilitatea utilizării tehnicii **căutării binare** *(*O*(log*$_2$ *n))*. Acest avantaj se răsfrânge asupra **fazei de căutare** a cheii în operațiile de inserție, suprimare, furnizare element şi căutare cheie.
- **Furnizarea** informației şi **căutarea cheii** este foarte rapidă O*(log*$_2$ *n)*. De fapt furnizarea constă din doi timpi: **faza de căutare** *(*O*(log*$_2$ *n))* şi din **faza de returnare** O*(1)*.
- **Traversarea ordonată** a cheilor este liniară (O*(n)*).

b) Dezavantaje

- Faza de instalare la **inserție** este lentă (O*(n)*) deoarece se mută **toate** intrările începând cu punctul de inserție pentru a face loc în tabelă.
- **Inserția** este lentă(O*(n)*). Ea constă din **faza de căutare** care este rapidă (O*(log*$_2$ *n)*) şi din **faza de instalare** care realizează inserția propriu-zisă (O*(n)*).
- Faza de extragere la **suprimare** este lentă (O*(n)*) deoarece se mută toate intrările începând cu locul de suprimare.
- **Suprimarea** este lentă (O*(n)*). Ea constă din faza de căutare (O*(log*$_2$ *n)*) şi din faza de extragere (O*(n)*).
- **Crearea** tabelei lentă (O*(n log*$_2$ *n)*). Pentru fiecare din cele n elemente se realizează o inserție O*(log*$_2$ *n)*.
- Trebuie cunoscut apriori **numărul total** de intrări în tabelă. Operatorul `TabelăPlină` trebuie executat înaintea fiecărei inserții.

c) Concluzii

- Utilizarea acestei implementări a tabelelor este **avantajoasă** atunci când:
  - Se realizează multe consultări, verificări sau rapoarte asupra tabelei,
  - Numărul de inserţii şi suprimări este redus
  - Tabela este de mari dimensiuni.
- Trebuie cunoscută cu bună precizie apriori **dimensiunea** maximă a tabelei.

7.2.2. Implementarea tabelelor cu ajutorul tablourilor neordonate

a) Avantaje

- **Inserţie** rapidă care se realizează la sfârşitul tabloului (O*(1)*).
- Faza de extragere la **suprimare** este rapidă (O*(1)*) dacă se mută ultima intrare a tabelei peste cea suprimată.
- **Construcţia** tabelei este rapidă (O*(n)*).

b) Dezavantaje

- **Căutarea** se realizează **secvenţial** O*(n)*. Acest dezavantaj se reflectă asupra suprimării, furnizării şi căutării cheilor.
- **Suprimarea** este lentă (O*(n)*). Ea constă din faza de căutare (O*(n)*) urmată de faza de extragere (O*(1)*).
- **Furnizarea** este lentă (O*(n)*).
- **Căutarea** unei chei este lentă (O*(n)*). Ea constă din căutarea propriu-zisă (O*(n)*) urmată de faza de returnare (O*(1)*).
- **Traversarea ordonată** a cheilor este lentă, ea necesitând un efort de calcul cuprins între O$(n \log_2 n)$ şi O(n^2) în dependenţă de metoda de sortare utilizată.
- Trebuie cunoscut cu precizie **numărul total** de intrări în tabelă.

c) Concluzii

- Acest mod de implementare este recomandat a fi utilizat atunci când se execută multe traversări neordonate ale tabelei, sau când se fac comparaţii sau calcule care afectează fiecare intrare.
- Construcţia şi inserţia sunt simple.
- Trebuie cunoscută apriori dimensiunea maximă a tabelei.
- Implementarea **nu** este recomandabilă atunci când se efectuează frecvent suprimări şi căutări sau traversări ordonate.

7.2.3. Implementarea tabelelor cu ajutorul listelor înlănţuite ordonate

a) Avantaje

- Faza de instalare la **inserţie** este rapidă (O*(1)*).
- Faza de extragere la **suprimare** este rapidă (O*(1)*).
- **Traversarea** în manieră ordonată a tabelei este rapidă (O*(n)*).
- **Nu** este necesară precizarea **dimensiunii maxime** a tabelei.
- Operatorul ***`TabelăPlină`*** este întotdeauna fals.

b) Dezavantaje

- **Căutarea** se realizează în manieră **secvenţială** (O*(n)*). Acest dezavantaj se răsfrânge asupra fazei de căutare la inserţie, suprimare, furnizare element şi căutare cheie.

- **Inserția** este lentă (O*(n)*), ea constând din căutare (O*(n)*) şi instalare (O*(1)*).
- **Suprimarea** este lentă (O*(n)*) ea constând din căutare (O*(n)*) şi extragere (O*(1)*).
- **Furnizarea** este lentă (O*(n)*) deoarece presupune o căutare (O*(n)*) urmată de un retur (O*(1)*).
- **Căutarea** cheii este lentă (O*(n)*). Ea constă dintr-o căutare (O*(n)*) urmată de un retur (O*(1)*). Totuşi în situația în care cheia **nu** există în tabelă, căutarea se poate opri chiar după poziția unde ar trebui să se găsească cheia.
- **Construcția** tabelei este lentă (O$(n \cdot log_2 n)$ – O(n^2)) funcție de tehnica utilizată pentru sortare.
- Efortul de calcul necesitat de **gestionarea înlănțuirilor**.

c) Concluzii

- Structura este eficientă pentru situațiile în care:
  - Se realizează **traversări** frecvente în manieră **ordonată** ale tabelei
  - Nu se cunoaşte **numărul maxim** de elemente.
- Structura **nu** este propice investigațiilor.
- Inserția şi suprimarea consumă timp în tabele de dimensiuni mari.

7.2.4. Implementarea tabelelor cu ajutorul listelor înlănțuite neordonate

a) Avantaje

- **Inserția** este rapidă ea realizându-se la începutul sau la sfârşitul listei (O*(1)*).
- Faza de extragere la **suprimare** este rapidă (O*(1)*).
- **Construcția** tabelei este rapidă (O*(n)*) deoarece constă dintr-o succesiune de inserții.
- **Nu** este necesară precizarea dimensiunii maxime a tabelei.
- Operatorul ***TabelăPlină*** este întotdeauna fals.

b) Dezavantaje

- **Căutarea** este **secvențială** (O*(n)*). Dezavantajează inserția, suprimarea, furnizarea şi căutarea propriu-zisă.
- **Suprimarea** este lentă (O*(n)*) deoarece ea constă din faza de căutare (O*(n)*) urmată de faza de extragere (O*(1)*).
- **Furnizarea** este lentă (O*(n)*). Constă din căutare (O*(n)*) şi retur (O*(1)*).
- **Căutarea** cheii este lentă (O*(n)*). Similar ca şi la furnizare.
- **Traversarea** în ordinea cheilor este lentă (O$(n \cdot log_2 n)$ – O(n^2)) funcție de metoda de sortare utilizată.
- Regia **legăturilor**.

c) Concluzii

- Această modalitate de implementare este indicată atunci când:
  - Se cere construcția rapidă a tabelei,
  - Se realizează multe inserții adiționale
  - Nu se cunoaşte dimensiunea maximă a tabelei.

- Investigările, suprimările sau traversările ordonate sunt consumatoare de timp.

7.3. Implementarea tabelelor prin tehnica dispersiei

- Una dintre metodele avansate de implementare a tabelelor o reprezintă **tehnica dispersiei**.
- Această tehnică cunoscută şi sub denumirea de "**hashing**", reprezintă un exemplu de rezolvare elegantă şi deosebit de eficientă a problemei căutării într-un context bine precizat.

7.3.1. Consideraţii generale

- Formularea problemei:
  - Se dă o **mulţime** `S` de noduri identificate fiecare prin valoarea unei **chei**, organizate într-o **structură tabelă**.
  - Pe mulţimea cheilor se consideră definită o **relaţie de ordonare**.
  - Se cere ca tabela `S` să fie organizată de o asemenea manieră încât **regăsirea** unui nod cu o cheie precizată `k`, să necesite un **efort** cât mai redus.
- În ultimă instanţă, **accesul** la un anumit nod presupune **determinarea** acelei intrări din tabelă, la care el este **memorat**.
- Astfel, problema formulată se **reduce** la găsirea unei **asocieri** specifice `(H)` a **mulţimii cheilor** `(K)` cu **mulţimea intrărilor** `(A)` [7.3.1.a].

```
-------------------------------------------------------------
      H: K → A                                      [7.3.1.a]
-------------------------------------------------------------
```

- Problema **nu** este nouă, ea a mai fost abordată pe parcursul acestei lucrări.
  - Până la momentul de faţă astfel de asocieri au fost implementate cu ajutorul unor algoritmi de căutare în tablouri şi liste în diferite strategii de abordare şi în diverse moduri de organizare.
- În secţiunea de faţă se propune o altă **metodă**, simplă şi foarte eficientă în multe situaţii.
- Metoda se bazează pe **tehnica dispersiei** şi se aplică structurilor de date **tablou**.
- Deşi este o metodă care utilizează o **structură statică** de memorie, prin performanţele sale este un competitor serios al metodelor bazate pe structuri de date avansate.
- **Ideea** pe care se bazează tehnica dispersării este următoarea:
  - Se rezervă **static** un volum constant de memorie pentru "**tabela dispersată**", care conţine **nodurile** cu care se lucrează.
  - Tabela se implementează în forma unei structuri de date **tablou** `T` având drept elemente nodurile în cauză.
  - Notând cu `p` numărul elementelor tabloului, indicele `a` care **precizează** un nod oarecare, ia valori între `0` şi `p-1`.
  - Se notează cu `K` mulţimea tuturor **cheilor** şi cu `k` o cheie oarecare.
  - Numărul cheilor este de obicei mult mai mare decât `p`. Un exemplu realist în acest sens este acela în care cheile reprezintă identificatori cu cel mult zece caractere, caz în care există aproximativ 10^{15} chei diferite în timp ce valoarea practică a lui `p` este 10^3.

- Se consideră în aceste condiții funcția `H`, care definește o **aplicație** a lui `K` pe mulțimea tuturor indicilor, mulțime care se notează cu `L (L = {0,1,2,...,p-1})` [7.3.1.b].

`a = H(k)` *unde* `k ∈ K şi a ∈ L` [7.3.1.b]

- Funcția de **asociere** `H` este de fapt o funcție de **dispersie** şi ea permite ca pornind de la o cheie `k` să se determine indicele asociat `a`.
  - Este evident că `H` **nu** este o funcție **bijectivă** deoarece numărul cheilor este mult mai mare decât numărul indicilor.
  - Practic există o mulțime de chei (clasă) cărora le corespunde acelaşi indice `a`.
  - Din acest motiv, o altă denumire uzuală sub care este cunoscută tehnica dispersării este aceea de "**transformare a cheilor**" ("**key transformation**"), întrucât cheile se transformă practic în indici de tablou.
- **Principiul metodei** este următorul.
  - Pentru **înregistrarea** unui nod cu cheia `k`:
    - (1) Se determină mai întâi indicele asociat `a=H(k)`,
    - (2) Se depune nodul la `T[a]`.
  - Dacă în continuare apare o altă cheie `k'` care are acelaşi indice asociat `a = H(k') = H(k)`, atunci s-a ajuns la aşa numita "**situație de coliziune**",
  - În acest caz care trebuie adoptată o anumită **strategie** de rezolvare a acestei probleme.
  - La **căutare** se procedează similar.
    - (1) Dată fiind o cheie `k`, se determină `a = H(k)`,
    - (2) Se verifică dacă `T[a]` este nodul căutat, adică dacă `T[a].cheie=k`.
    - (3) Dacă da, atunci s-a găsit nodul, în caz contrar s-a ajuns la coliziune.
- În concluzie, aplicarea tehnicii dispersiei presupune soluționarea a două **probleme**:
  - (1) Definirea **funcției de dispersie** `H`;
  - (2) Rezolvarea situațiilor de **coliziune**.
- Rezolvarea favorabilă a celor două probleme conduce la rezultate remarcabil de eficiente, cu toate că metoda prezintă şi anumite dezavantaje, asupra cărora se va reveni.

7.3.2. Alegerea funcției de dispersie

- **Funcția de dispersie** trebuie:
  - Pe de o parte să repartizeze cât mai "**uniform**" cheile pe mulțimea indicilor deoarece în felul acesta se reduce probabilitatea coliziunilor,
  - Pe de altă parte trebuie să fie uşor şi rapid **calculabilă**.
- Proprietățile acestei funcții sunt aproximate în literatura de specialitate de termenul "**hashing**" (amestec) motiv pentru care funcția se notează generic cu `H` fiind denumită funcție de **amestec** ("**hash function**").
- Din acelaşi motiv tehnica dispersiei se mai numeşte şi **tehnică "hashing".**

- În literatură se definesc diverse funcții de dispersie fiecare cu avantaje şi dezavantaje specifice, activitate care conturează un domeniu de cercetare extrem de activ [Kn76].
- În cele ce urmează se va prezenta o variantă a unei astfel de funcții.
  - Fie `ORD(k)` o valoare întregă unică ataşată cheii `k`, valoare care precizează **numărul** de ordine al cheii `k` în **mulțimea ordonată** a tuturor cheilor (§1.3.2.).
  - Funcția `H` se defineşte în aceste condiții astfel [7.3.2.a]:

```
-------------------------------------------------------
     H(k)= ORD(k) MOD p                        [7.3.2.a]
-------------------------------------------------------
```

- Această funcție care asigură distribuția cheilor pe mulțimea indicilor `(0,p-1)` stă la baza mai multor metode de repartizare.
- S-a demonstrat **experimental** că în vederea repartizării cât mai uniforme a cheilor pe mulțimea indicilor este **recomandabil** ca `p` să fie un număr **prim**.
  - Cazul în care `p` este o putere a lui 2 este extrem de favorabil din punctul de vedere al eficienței calculului funcției, dar foarte nefavorabil din punctul de vedere al coliziunilor, mai ales în cazul în care cheile sunt secvențe de caractere [Wi76].

7.3.3. Tratarea situației de coliziune

- Prezența unei situații de coliziune presupune **generarea** unui nou indice în tabelă (**indice secundar**), pornind de la cel anterior.
- Există două modalități principiale de generare a indicilor secundari:
  - (1) O modalitate care presupune un **spațiu suplimentar** asociat tabelei şi care prefigurează tehnica **dispersiei deschise**
  - (2) O a doua modalitate care exploatează **doar** spațiul de **memorie** alocat tabelei şi care prefigurează tehnica **dispersiei închise**.

7.3.3.1. Tehnica dispersiei deschise

- Tehnica **dispersiei deschise** presupune înlănțuirea într-o **listă înlănțuită** a tuturor nodurilor ale căror indici primari sunt identici, metodă care se mai numeşte şi **înlănțuire directă.**
- Elementele acestei liste pot sau **nu** aparține tabloului inițial: în ultimul caz memoria necesară alocându-se din aşa numita "**zonă de depăşire**" a tabelei.
- Structurile de date aferente acestei tehnici sunt prezentate în [7.3.3.1.a].

```
-------------------------------------------------------
/* Tehnica dispersiei deschise */


enum { dimensiunetabela = 29};   /*numar prim*/

typedef unsigned char tipindex;

typedef int tipinfo;
typedef int tipcheie;
typedef struct tipelement {
  tipcheie cheie;
  tipinfo info;
} tipelement;
```

```
typedef struct tipnod* tipreferinta;    /*[7.3.3.1.a]*/
typedef struct tipnod {
  tipelement element;
  tipreferinta urm;
} tipnod;
typedef tipreferinta tiptabela[dimensiunetabela+1];

tiptabela t;
-------------------------------------------------------------
```

- Această metodă are **avantajul** că permite **suprimarea** elementelor din tabelă, operație care **nu** este posibilă în toate implementările.
- Dintre **dezavantaje** se evidențiază două:
  - Necesitatea menținerii unei **liste secundare**
  - Prelungirea fiecărui nod cu un spațiu de memorie pentru pointerul (indexul) necesar înlănțuirii.

7.3.3.2. Tehnica dispersiei închise

- Tehnica **dispersiei închise** utilizează, după cum s-a precizat anterior, **strict** spațiul de memorie alocat tabelei.
- În cazul unei coliziuni se realizează parcurgerea după o anumită **regulă** a tabloului dispersat `T` până la găsirea primului loc liber.
- Structurile de date aferente acestei tehnici apar în [7.3.3.2.a].

```
-------------------------------------------------------------
/* Tehnica dispersiei închise*/

enum { dimensiunetabela = 29};    /*numar prim*/

typedef unsigned tipindex;

typedef int tipinfo;
typedef int tipcheie;
typedef struct tipelement {
  tipcheie cheie;
  tipinfo info;                          /*[7.3.3.2.a]*/
} tipelement;
typedef tipelement tiptabela[dimensiunetabela+1];
-------------------------------------------------------------
```

- Schița de principiu a algoritmului care realizează **căutarea** în tabelă este prezentat în secvența [7.3.3.2.b].

```
-------------------------------------------------------------
/* Tehnica dispersiei închise - căutarea unui element în
tabelă */

tiptabela t;
tipindex l;

a= h(k); i= 0;
do {                                     /*[7.3.3.2.b]*/
  if (t[a].cheie==k) ;
      /*element gasit*/
    else
      if (t[a].cheie==liber) ;
          /*elementul nu este în tabela*/
        else
```

```
          {    /*coliziune*/
              i= i+1;
              a= a+g(i);
          }
} while (!((gasit) | (nu_este_in_tabela) |
(tabela_plina)));
-------------------------------------------------------------
```

- Funcția `g(i)` este aceea care precizează **regula** după care se parcurge tabela în caz de coliziune.
- Există numeroase astfel de funcții `g` cărora le corespund modalități diverse în conformitate cu care se realizează parcurgerea tabelei.
- Dintre acestea cele mai cunoscute sunt "**adresarea deschisă liniară**" şi "**adresarea deschisă patratică**".

Adresarea deschisă liniară

- Adresarea deschisă liniară prefigurează cea mai simplă metodă de parcurgere
- Conform **adresării deschise liniare**, intrările tabelei se parcurg **secvențial**, tabela considerându-se **circulară**. Cu alte cuvinte `g(i)=i` [7.3.3.2.c].

```
-------------------------------------------------------------
   a0 = H(k)
   ai =(a0+i) MOD p     i=1...p-1              [7.3.3.2.c]
-------------------------------------------------------------
```

- În fragmentul de program din secvența [7.3.3.2.d], se prezintă procesul de **căutare** în tabelă a intrării cu cheia `k`.
    - Dacă intrarea se găseşte, se asignează variabila de ieşire `v` cu conținutul său.

```
-------------------------------------------------------------
/* Adresarea deschisă liniară - căutarea unui element în
tabelă */

tiptabela t;
 tipindex a,i;
 unsigned gasit,absent;
 tipelement v;

#define true (1)
#define false (0)

a= h(k); i= l; gasit= false; absent= false;
do {
  if ((t[a].cheie==liber.cheie)
       && (t[a].info==liber.info))
      absent= true;
    else
      if (t[a].cheie==k){
            v= t[a];
            gasit= true;
          }                                  /*[7.3.3.2.d]*/
        else{    /*coliziune*/
            a= a+1;
            if (a==p)  a= 0;
            if (a==i)  absent= true;
          }
} while (!(gasit || absent));
```

- Acest algoritm are ca **dezavantaj** tendința de a **îngrămădi** nodurile în continuarea celor deja înregistrate.
- Dezavantajul **nu** este inerent metodei de adresare deschisă liniară, el datorându-se **regulii** de parcurgere a elementelor tabelei.
- Ideal ar fi ca în caz de coliziune să se aplice o regulă care distribuie încercările, cu **probabilități** egale, pe **locurile libere** rămase în tabelă. Realizarea practică a acestui deziderat este însă deosebit de complexă.

Adresarea deschisă pătratică

- **Adresarea deschisă pătratică** reprezintă o soluție de compromis relativ simplă şi eficientă.
- **Funcția de repartizare** are expresia $g(i)=i^2$ [7.3.3.2.e].

$a_0 = H(k)$
$a_i = (a_0+i^2)$ **MOD** p *i=1,2,3,...* [7.3.3.2.e]

- O manieră simplă de **implementare** a acestei funcții de parcurgere este următoarea:
  - Dacă un indice `a` conduce la **coliziune**, atunci proxima încercare se face la indicele `a + r` unde `r` este o mărime care se inițializează pe `1` şi care după fiecare încercare nereuşită se incrementează cu `2`.
  - Astfel valorile lui `r` formează şirul numerelor fără soț, şir a cărui sumă este un **pătrat perfect.**
- În caz de coliziuni repetate, numărul încercărilor se va limita prin impunerea condiției formale `r < p`.
- Algoritmul de **inserție** a unui nou element `v` în tabelă este prezentat în secvența [7.3.3.2.f].

```
/* Adresarea deschisă patratică - inserția unui element în
tabelă */

     tiptabela t;
     tipindex a,r;
     boolean depus;
     tipelement v;

a= h(k); r= 1; depus= false;
do {
  if ((t[a].cheie==liber.cheie)
       && (t[a].info==liber.info)){
        t[a]= v; depus= true;
      }                                          /*[7.3.3.2.f]*/
    else{
        a= a+r;
        if (a>=p)  a= a-p;
        r= r+2;
      }
} while (!(depus || (r==p)));
```

- Un **dezavantaj** minor al acestei metode este că se poate întâmpla să **nu** se găsească nici un element liber, ajungându-se la depășirea tabelei, cu toate că în realitate în tabelă mai există locuri libere.
    - Se demonstrează însă că **probabilitatea** acestui eveniment este **foarte mică** și că în general metoda este foarte performantă.
- **Traversarea** ordonată a tabelei este greu de realizat,
- **Dezavantajul major** al acestei implementări se referă la faptul că ea **nu** permite suprimarea elementelor din tabelă.
    - Suprimarea unui element oarecare al tabelei poate **întrerupe** secvența liniară sau pătratică de parcurgere a tabelei fapt care poate conduce la **pierderea** iremediabilă a unor intrări.
- Pentru a remedia acest dezavantaj major se poate recurge la următoarea **stratagemă**.
    - Se asociază fiecărei intrări a tabelei un câmp de tip enumerare numit `stare` care poate lua valorile `liber, ocupat` sau `șters`
    - În consecință `TipElement` din secvența [7.3.3.2.a] se modifică după cum urmează [7.3.3.2.g].

```
-------------------------------------------------------------
enum { dimensiunetabela = 29};     /*numar prim*/

typedef unsigned char tipindex;

typedef int tipinfo;
typedef int tipcheie;               /*[7.3.3.2.g]*/
typedef struct tipelement {
                    tipcheie cheie;
                    tipinfo info;
                    enum {liber,ocupat,sters} stare;
                    }tipelement;
-------------------------------------------------------------
```

- Inițial la crearea tabelei se trec **toate** locațiile sale în starea `liber`.
- Pe măsură ce se inserează elemente, locațiile corespunzătoare se trec în starea `ocupat`.
- În aceste condiții, **suprimarea** se poate realiza simplu fără îndepărtarea efectivă a elementului prin simpla trecere a stării câmpului asociat în `șters`.
- În acest mod, în procesul de căutare a unei chei se parcurg **și** câmpurile aflate în starea `șters`, secvența de adresare creată rămânând nemodificată.
- Câmpurile marcate cu `șters` pot fi însă **reutilizate** la introducerea unor noi elemente în tabelă.

7.3.4. Analiza performanței tehnicii dispersiei închise

- Pentru analiza performanței tehnicii dispersiei închise se presupune pentru început că:
    - (1) Toate cheile au **probabilități** de apariție **egal posibile**
    - (2) Că funcția de distribuție `H` le repartizează **uniform** pe mulțimea intrărilor tabelei.
- Se presupune de asemenea, că o cheie oarecare se inserează într-o tabelă de dimensiune `n`, care conține deja `m` elemente, adică `m` intrări sunt **deja ocupate**.

- Se noteaă cu α=m/n a raportul dintre locaţiile ocupate şi cele disponibile în cadrul tabelei dispersate.
  - α se mai numeşte şi **factor de umplere**
  - α = 0 precizează tabela vidă
  - α = 1 precizează tabela plină.
- Valoarea **numărului mediu probabil de încercări** E , necesar realizării **accesului** la o cheie oarecare din tabelul dispersat având factorul de umplere α apare în [7.3.4.f]. El se obţine în urma uni raţionament matematic sofisticat [Cr00].

$$E = -\frac{1}{\alpha} \cdot ln(1-\alpha) \qquad [7.3.4.f]$$

- Numărul **probabil** de încercări E necesar găsirii sau inserţiei unei chei k funcţie de factorul de umplere α , apare tabelat în figura 7.3.4.a.

Nr. crt.	α	E(α)
1	0.10	1.05
2	0.25	1.15
3	0.50	1.39
4	0.75	1.85
5	0.90	2.56
6	0.95	3.15
7	0.99	4.66

Fig.7.3.4.a. Tabelarea numărului probabil de încercări funcţie de factorul de umplere al tabelei (cazul ideal)

Nr. crt.	α	E(α)
1	0.1	1.06
2	0.25	1.17
3	0.5	1.50
4	0.75	2.50
5	0.9	9.90
6	0.95	10.50

Fig.7.3.4.b. Tabelarea relaţiei experimentale E(α) funcţie de factorul de umplere α

- Rezultatele numerice sunt surprinzătoare, ele **nu** depind de dimensiunea tabelei şi în acelaşi timp explică comportarea excepţională a acestei metode.
- Chiar dacă tabela este plină în proporţie de 90 %, în medie sunt necesare numai 2.56 încercări pentru a localiza o cheie sau a găsi un loc liber în ea.
- Rezultatele prezentate presupun însă cazul ideal, respectiv o metodă de tratare a coliziunilor care repartizează în mod perfect uniform cheile pe mulţimea locaţiilor rămase libere.
- În realitate, acest lucru este practic irealizabil.
- Analiza **experimentală** a comportamentului real a dispersiei închise bazate pe adresarea deschisă liniară a condus la relaţia experimentală [7.3.4.g]

$$E = \frac{1-\alpha/2}{1-\alpha} \qquad [7.3.4.g]$$

- Reprezentarea grafică ale valorilor sale numerice funcţie de factorul de umplere α apar reprezentate în figura 7.3.4.b [Wi74].
- Rezultatele obţinute chiar pentru metoda cea mai puţin elaborată de tratare a coliziunilor (adresarea deschisă liniară) sunt atât de bune încât ar putea exista tentaţia de a considera metoda dispersiei drept soluţie pentru orice situaţie.

- Performanțele metodei, cel puțin din punctul de vedere al numărului de comparații necesare pentru găsire sau inserție, sunt superioare celor mai sofisticate metode bazate pe structuri de arbori.
- Desigur, metoda are și o serie de **dezavantaje**.
- (1) Primul și cel mai important, este acela că, dimensiunea tabelei fiind fixă aceasta **nu** poate fi ajustată conform cererii curente.
    - O estimare a priori a dimensiunii sale maxime este de regulă greu de realizat și conduce fie la o slabă utilizare a memoriei fie la performanțe scăzute prin depășirea tabelei dispersate.
    - Chiar dacă numărul de elemente este cunoscut exact, în vederea obținerii unei bune performanțe, dimensiunea tabelei trebuie aleasă cam cu 10 % mai mare.
- (2) O a doua deficiență majoră se referă la suprimarea cheilor.
    - Aceasta se realizează într-o manieră laborioasă și complexă mai puțin în cazul metodei înlănțuirii care utilizează un spațiu de memorie suplimentar pentru înlănțuirea directă, elemente care au fost deja abordate în cadrul paragrafului de față.
- În **concluzie**, utilizarea tehnicii dispersiei **nu** este recomandabilă dacă:
    - Volumul de date este cunoscut cu mică probabilitate
    - Volumul de date este puternic variabil
    - Volumul de date crește în timp.

7.4. Rezumat

- **Tabela** este o structură de date alcătuită din **articole**. Principalii operatori definiți pe **TDA tabelă** se referă la **inserția, suprimarea, căutarea** și **traversarea** structurii.
- Există mai multe **modalități de implementare a tabelelor** dintre care se precizează **tablourile ordonate** și **neordonate** respectiv **listele înlănțuite ordonate** și **neordonate** fiecare cu **avantaje** și **dezavantaje** specifice.
- O manieră de implementare performantă a tabelelor o reprezintă **tehnica dispersiei** sau **hashing**-ul.

7.5. Exerciții

1. Definiți structura *tabelă*. Dați cîteva exemple de tabele.
2. Care sunt *principalii operatori* definți de *TDA tabelă*?
3. Realizați o *analiză comparată* a implementării structuri tabelă cu *ajutorul tablourilor ordonate* respectiv *neordonate*. Precizați câteva *concluzii* utile pentru aplicații.
4. Realizați o *analiză comparată* a implementării structuri tabelă cu *ajutorul listelor înlănțuite ordonate* respectiv *neordonate*. Precizați câteva *recomandări* utile pentru aplicații.
5. Redactați un program C care implementează operatorii de inserție, suprimare și căutare într-o tabelă utilizînd *tehnica dispersiei deschise*.
6. Redactați un program C care implementează operatorii de inserție și căutare într-o tabelă utilizînd *tehnica dispersiei închise*. Ce măsuri trebuiesc luate pentru a realiza și implementarea operatorului de suprimare. Implementați și o astfel de variantă.
7. Care este *performanța tehnicii dispersiei*? Comentați. Ce *dezavantaje* majore presupune tehnica dispersiei? Cum se poate realiza parcurgerea în manieră ordonată a unei tabele de dispersie?