29. STRUCTURI FOLOSITE ÎN PROCESE DE CĂUTARE

29.1 Tipologii de aplicaţii

Aplicaţiile informatice diferă funcţie de obiectivele pentru care au fost elaborate.

Tipologiile aplicaţiilor sunt considerate în raport cu criterii de clasificare elaborate atât de elaboratori, cât şi de dealeri, dar mai ales de către clienţi.

După criteriul structurii, aplicaţiile informatice sunt:

- *seriale*, în care componentele sunt referite una câte una; calitatea execuţiei componentei precedente determină referirea componentei următoare în secvenţă şi deci, execuţia întregului lanţ;
- *arborescentă*, caz în care componentele se dispun pe niveluri şi referirea depinde de evaluarea unei expresii relaţionale; sunt situaţii în care pe o ramură a arborescenţei prelucrările conduc la rezultate corecte, în schimb pe alte ramuri rezultatele sunt incerte;
- *reţea*, care presupune atât execuţii în secvenţă cât şi evaluări de expresii relaţionale; referirile sunt uneori comune pentru mai multe componente, atât pentru componente următoare, cât şi pentru componente precedente ca poziţie, care devin următoare în execuţie.

Abordările moderne impun clasificări spre constituirea de clase, de module, de interacţiuni, ceea ce conduce la construcţii simple, medii sau complexe.

De asemenea, în raport cu evoluţiile ulterioare se caută accentuarea laturilor de mentenanţă sau portabilitate. Aplicaţiile informatice sunt sau nu sunt înzestrate cu anumite proprietăţi, caz în care sunt incluse într-una din categoriile aparţinând criteriilor care se definesc ad-hoc. Din punct de vedere al utilizatorilor, aplicaţiile informatice sunt:

- pentru un singur client, de regulă specializat în realizarea de prelucrări complex şi este vorba de aplicaţii particulare, cu volum mare de decizii care presupun nivele de agregare ridicate; aplicaţia presupune numeroase convenţii referitoare la tipurile de operaţii, procedurile care se activează şi ramurile care se traversează în reţeaua asociată; există numeroase puncte de intervenţie pentru a actualiza parametrii şi chiar proceduri de calcul şi de selecţie a datelor din bazele de date pentru a urmări dinamica sistemului economic din care aplicaţia este parte;
- cu număr restrâns de utilizatori, destinate lucrului cu baze de date al căror volum de actualizare este ridicat; aplicaţiile sunt specializate iar operatorii sunt profesionişti în domeniu; se definesc tipologii de operaţii cărora le corespund coduri de selecţie şi date de intrare specifice; se urmăreşte rezolvarea de probleme într-o interval dat de timp, cu controlul riguros asupra volumului de date; se elaborează chei de control pentru garantarea calităţii datelor de intrare care garantează la rândul lor calitatea rezultatelor; aceste aplicaţii determină plăţi, acceptarea unor stări de fapt, selecţia de candidaţi şi alte forme de tranzacţii

sau de construire a argumentaţiilor; operatorii sunt instruiţi atât în utilizarea produselor software, cât mai ales în semnalarea de erori în efectuarea de corecţii acolo unde este posibil;

- număr de utilizatori foarte mare, de masă, cu grad de neomogenitate ridicat; interfeţele sunt astfel construite încât toţi utilizatorii să obţină maximum de satisfacţie; se impune ca aceste aplicaţii să fie atent elaborate, să ţină seama de cerinţele reale ale utilizatorilor şi să nu necesite procese de instruire speciale; în cazul în care numărul de paşi care se parcurg este indicat, trecerea de la un pas la altul este marcată prin comenzi de acelaşi tip; problema rescrierii la pasul iniţial (de start) se va realiza de la oricare din etapele interacţiunii; neomogenitatea clienţilor impune utilizarea de interfeţe grafice; clientul trebuie să obţină maximum de beneficii cu minimum de date introduse de la tastatură; se efectuează o cercetare sistematică pentru a evidenţia care sunt situaţiile destinate care trebuie trataţi pentru cazurile cu frecvenţa cea mai mare se definesc fluxuri de opţiuni care prin tastări simple să conducă la traversarea de paşi şi obţinerea de efecte în concordanţă cu aşteptările utilizatorilor; clienţii efectuează un număr limitat de selecţii, iar modul de reprezentare a datelor reduce repetitivitatea de operaţii generată de erori de interpretare; simplitatea dialogului om-maşină, lipsa de rigiditate şi luarea în considerare a tipurilor de clienţi cu comportamentul specific are rolul de a genera aplicaţii informatice destinate de succes, viabile.

29.2 Chei de regăsire

Aplicaţiile informatice cele mai frecvente sunt caracterizate prin utilizarea în procesul de regăsire a unei singure chei.

De regulă, în procesul de analiză a problemei de rezolvat se parcurg următorii paşi:

- se identifică mulţimile cu care se operează precum: mulţimea persoanelor, mulţimea materialelor, mulţimea documentelor, mulţimea operaţiilor etc;
- se stabilesc numărul maxim de componente care alcătuiesc mulţimile;
- se alege algoritmul de căutare a cheilor aşa fel încât să se asigure unicitatea în concordanţă cu apartenenţa elementelor colectivităţii la eventuale submulţimi;
- se generează mecanisme de stare şi de căutare care să reducă duratele de prelucrare.

În cazul în care dinamica înregistrată de colectivitate permite să construim chei numerice.

Pentru salariaţii unei întreprinderi, marca salariatului – unică – este un număr. Salariatul X are marca 7250, ceea ce înseamnă că a fost a 7250 a persoană care s-a angajat. Cei dinaintea sa mai lucrează, s-au transferat sau au plecat la pensie.

Aplicaţiile bazate pe marca salariatului impun manipularea unei legitimaţii în mod obligatoriu.

În cazul în care se doreşte diferenţierea muncitorilor pe secţii se face o codificarea secţiilor şi o codificarea a muncitorilor şi se concatenează cele două coduri, rezultând marca muncitorului. De exemplu, pentru o întreprindere cu 25 de secţii numerotarea secţiilor se face cu 01, 02, …, 25, iar pentru codul muncitorului se constituie secvenţe formate din patru cifre.

Marca salariatului 140014 arată că este vorba de salariatul de la secţia 14, având codul 0014 în codul secţiei respective. Aplicaţiile bazate pe o singură cheie sunt proiectate astfel încât să se poată gestiona cu uşurinţă toate datele. Simplitatea acestor aplicaţii impune un nivel ridicat de rigiditate. Generarea codurilor în mod serial corespunzătoare mulţimilor omogene face dificil procesul de regăsire atunci când lipseşte suportul pe care este indicat codul elementului ce trebuie căutat. Elaborarea de structuri care se asociază cheilor şi a mecanismelor de continuare a elementelor elimină parţial acest impediment.

Este important ca părţile ce formează structura să fie alcătuit din mulţimi deja existente, cunoscute, iar numărul de elemente generate să fie cât mai restrâns.

Dacă o firmă comercializează produse electrocasnice, autoturisme şi materiale de construcţii către persoane fizice codul contractului este o construcţie rezultată din concatenarea secvenţelor de numere următoare:

- secvenţa corespunzătoare tipului de produs achiziţionat;
- secvenţa de identificare a cumpărătorului.

Se stabileşte un algoritm de construcţie a codului contractului. Fiecare literă din alfabet are o poziţie: litera a are poziţia 01, litera b are poziţia 02, litera i are poziţia 09, litera r are poziţia 17 etc.

Lunile anului se numerotează 01 (ianuarie) sau 07 (iulie) şi 12 (decembrie). Zilele lunii se numerotează de la 01 la 31. Anul de naştere se numerotează cu patru cifre pentru a elimina ambiguităţile de tipul celei generate de trecerea la anul 2000.

Dacă sunt luate în considerare şi părţi din numele persoanei fizice care achiziţionează produse, cu certitudine este asigurată unicitatea codului contractului.

Dacă structura codului de control este descrisă prin:

- poziţia 1: tipul produsului E – electrocasnice, A – autoturism, C – material de construcţii;
- poziţiile 2, 3, 4, 5: anul naşterii cumpărătorului;
- poziţia 6,7: luna naşterii cumpărătorului;
- poziţia 8,9: ziua de naştere a cumpărătorului;
- poziţiile 10, 11, 12: primele 3 litere din numele cumpărătorului la completare se asigură unicitatea codului contractului ca cheie de regăsire.

Întrucât un rol esenţial în evidenţe îl are codul numeric personal, în dezvoltarea aplicaţiilor moderne va fi inclus în structura cheilor în vederea regăsirii.

În operaţiile cu clienţii posesivi de posturi telefonice numărul postului şi codul personal deja formează un cod redundant.

La proiectarea cheilor de regăsire este preferabil să fie utilizate secvenţe care descriu elemente deja existente precum:

- două litere ce reprezintă prescurtările numelui de judeţ;
- literă M, F prin care se indică sexul persoanei;
- data naşterii prezentată prin an, lună, zi;
- alte componente din structura codului personal;

- cuvinte prin care se desemnează în clar produse (telefon, frigider, cuier, tablă, cablu, geam);
- denumiri de localităţi, străzi, instituţii.

Schimbarea concepţiei privind definirea cheilor rezidă din trecerea de la aplicaţiile bazate pe informaţii simple utilizate în programul de regăsire, oferite complet, după reguli specificate, la aplicaţii cu clienţi diferiţi, neomogeni ca pregătire în raport cu o aplicaţie informatică, unde regăsirea trebuie să se facă dacă informaţiile pe care clientul le are.

Cheile de regăsire sunt numeroase şi la proiectarea aplicaţiilor se impune comutarea de arborescenţă ale cor frunze sunt de fapt codurile unice.

În cazul informaţiilor incomplete sunt extrase submulţimi dintre care este identificat elementul căutat.

De exemplu, se ia în considerare codul numeric personal, căruia i se asociază arborescenţa dată în figura 29.1.

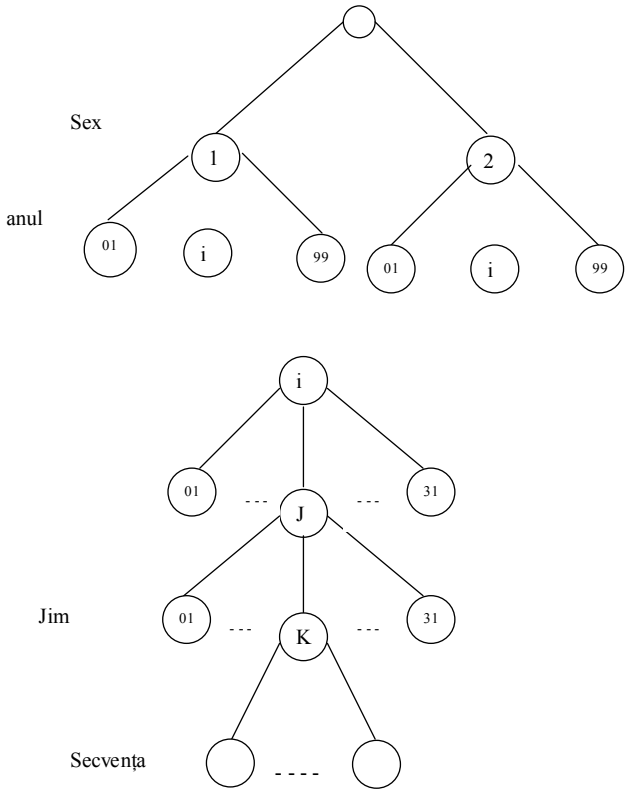


Figura 29.1 *Structura arborescentă asociată CNP*

Este important ca la un volum cât mai restrâns de date de intrare să se obţină o fineţe de selecţie cât mai bună

Pentru efectuarea de studii statistice se ajunge la forma de structură arborescentă care să genereze un număr cât mai restrâns de elemente.

În cazul persoanelor încorporabile anul naşterii conduce la submulţimi formate din multe elemente.

În schimb ziua de naştere urmată de luna de naştere conduce la o mulţime de elemente din care se extrage cu uşurinţă un element pentru a efectua prelucrări solicitate.

Problema cheilor de regăsire trebuie rezolvată adecvat pentru a genera echilibrul aplicaţiilor informatice şi pentru a reflecta particularităţile

evoluţiei fiecărei aplicaţii în parte, în raport cu clienţii cărora este destinată. Se au în vedere aplicaţiile distribuite cu accesul direct al clienţilor.

Este important să se utilizeze pentru regăsire *chei grafice* date sub forma unor simboluri, liste de cuvinte care corespund numelor de localităţi, denumiri de produse, mărci de produse, categorii de produse, valori numerice, denumiri de evenimente etc.

Este important ca fiecare client să tasteze cât mai puţine informaţii de intrare, iar produsul software să aibă o serie de algoritmi de normalizare care să conducă la realizarea acelui nivel de flexibilitate cerut aplicaţiilor distribuite care se adresează publicului larg.

Astfel, dacă se doreşte un număr de telefon al unei persoane vor fi solicitate următoarele date:

Numele: x x x…x
Prenumele: x x x …x
Oraşul: lista de oraşe
Localitatea în ordine alfabetică sau posibilitatea de a tasta altă localitate
Strada: x……x
Bloc: xxx
Apartament: xxx
Număr: xxxx

Atunci când aplicaţia este destinată utilizatorilor specializaţi sau care au efectuat un stagiu de instruire, introducerea datelor este clară, întrucât meniul este suficient de riguros.

În cazul utilizatorilor cu grad de instruire, cu cunoştinţe incomplete asupra modului de formulare a cererii apar următoarele situaţii:

- numele este ortografiat astfel decât forma stocată în baze de date; de exemplu Brâncuşi din baza de date poate fi tastat BRINCUŞ sau Brâncuş sau Brancusi sau Brancus;
- numelui îi lipseşte o vocală sau o consoană dublă; de exemplu, cuvântul BELLER este ortografiat Beler sau Belăr, Johnson este ortografiat Jonson
- numele este interschimbat cu prenumele; Bălcescu Nicolae este ortografiat Nicolae Bălcescu
- prenumele este înlocuit cu o literă sau grup de litere; Nicolae e scris N Gheorghe e scris Gh. sau este înlocuit cu un formular sau diminutiv. Ion este înlocuit cu Nelu, Dan este înlocuit cu Daniel, Gheorghe este înlocuit cu Gică sau Gicu sau Gigel. Aceleaşi probleme sunt puse şi în cazul ortografiei cu litere mari. În cazul în care numele şi prenumele este introdus într-o singură rubrică cea a numelui se impune o operaţie opusă concatenării.

Se consideră colectivitatea $A = \{a_1, a_2, \ldots, a_n\}$ pentru care un element a_i este descris prin şablonul *Sc* definit de caracteristicile sc_1, sc_2, …, sc_m în mulţimea $Sc = \{sc_1, sc_2, \ldots, sc_m\}$. Pentru elementul a_i se generează şirul de caractere c_{ij} pentru a specifica şirul caracteristicii Sc_j din şablon.

Descrierea este completă atunci când pentru toate elementele colectivităţii *A* au fost măsurate sau generate şiruri de caractere pentru toate caracteristicile ce definesc şablonul. O astfel de definire este dată în tabelul 29.1.

Tabelulul nr. 29.1 *Măsurarea elementelor colectivităţii materialelor*

Denumire	Cod	U.M.	Stoc iniţial	Intrări	Ieşiri	Preţ unitar
Var	100	kg	100	10	50	5
Ciment	122	kg	500	30	200	100
Cărămidă	400	buc.	10000	1000	5000	1
Cuie	310	kg.	100	50	25	25
Rigips	133	buc.	200	20	50	60

Sunt situaţii în care nu sunt efectuate definiri complete sau măsurători datorită neluării în considerare a caracteristicilor sau datorită costurilor foarte ridicate de a măsura, aşa cum se prezintă în tabelul 12.2.

Tabelul nr. 29.2 *Înregistrări incomplete pentru colectivitatea elevilor*

Nume	Sex	Data naşterii	Înălţime (cm)	Şcoală
Popescu Ion	M	10.08.1985	173	174
Alexandrescu	M		180	174
Gheorghe Ion	M	11.05.1984		3
Pana Marius		14.03.1984	165	
Popescu Alina	F	10.07.1985	167	3
Jderu Ioana	F	22.05.1984	165	

Pentru fiecare dintre elementele mulţimii *A* se stabilesc şiruri de caractere care se constituie în vocabularul asociat colectivităţii *A*.
Se iau în considerare frecvenţele de apariţie ale cuvintelor.
Pentru o colectivitate *B* definită în tabelul se măsoară frecvenţele de apariţie a cuvintelor în vocabular.

Tabelul nr. 29.3 *Nivelurile generate pentru caracteristicile C_1, C_2, C_3 şi C_4 care definesc colectivitatea B*

C_1	C_2	C_3	C_4
aa	x	u	wwww
bbb	yy	u	zzz
cc	x	vvv	zzz
dddd	x	vvv	zzz
e	yy	u	zzz
ffff	x	u	zzz
g	yy	u	wwww
h	yy	vvv	zzz
iii	yy	vvv	zzz
jj	yy	vvv	zzz

Vocabularul asociat colectivităţii A, V_A este definit de şirurile de caractere V_A = *{aa bbb cc dddd e ffff g h iii jj x yy u vvv wwww zzz}*.

Frecvenţele de apariţie a cuvintelor sunt date pentru caracteristici în tabelul 29.4

Tabelul nr. 29.4 *Frecvenţele pentru caracteristicile C_1, C_2, C_3 şi C_4 care definesc colectivitatea B*

Cuvânt	fC_1	fC_2	fC_3	fC_4
aa	0	0	0	0
bbb	0	0	0	0
cc	0	0	0	0
dddd	0	0	0	0
e	0	0	0	0
ffff	0	0	0	0
g	0	0	0	0
h	0	0	0	0
iii	0	0	0	0
jj	0	0	0	0
x	0	0	0	0
yy	0	0	0	0
u	0	0	0	0
vvv	0	0	0	0
wwww	0	0	0	0
zzz	0	0	0	0
TOTAL:	10	10	10	10

Analizând frecvenţele rezultă că subvocabularul VC_1 este format din cuvinte ce constituie chei unice în timp ce vocabularele VC_2, VC_3 şi VC_4 au un şir de cuvinte mult mai restrâns şi un şir se atribuie mai multor elemente din colectivitatea B.

Compararea şirurilor de caractere este o operaţie uzuală care evidenţiază măsura în care şirurile diferă sau nu unele de celelalte.

Dacă se consideră mulţimea şirurilor de caractere S = {C_1, C_2, …, C_{nsir})

Se calculează indicatorul de asemănare *AS* după relaţia:

$$AS = \frac{KS}{C_{nsir}^2} \qquad (29.1)$$

unde:

- *KS* – numărul de şiruri identice;
- C_{nsir}^2 – numărul combinaţiilor din mulţimea S de şiruri luate câte două.

În mulţimea S_0 = {aa, bb, cc, dd} perechile care se construiesc sunt în număr de $C_4^2 = 6$. Acestea sunt:

(aa, bb), (aa, cc), (aa, dd), (bb, cc), (bb, dd) şi (cc, dd) *(29.2)*

Aceste perechi de şiruri sunt date diferite, rezultând KS = 0 şi indicatorul $AS_{(S0)} = 0$.

În cazul mulţimii de şiruri:

$$S_1 = \{aa, bb, cc, aa\} \qquad (29.3)$$

rezultă perechile:

(aa, bb), (aa, cc), (aa, aa), (bb, cc), (bb, aa) şi (cc, aa) *(29.4)*

Cum operaţia de comparare este comutativă rezultă că operaţiile de comparare aplicate perechilor de şiruri (aa, bb) şi (bb,aa) conduc la acelaşi rezultat. O situaţie identică se regăseşte şi în cazul perechilor (aa, cc) şi (cc, aa).

Indicatorul $AS_{(S1)}$ este:

$$AS_{(S1)} = \frac{3}{6} = \frac{1}{2} \qquad (29.5)$$

Se defineşte indicatorul:

$$AS_{(Si)} = \frac{C_m^2}{C_n^2} \qquad (29.6)$$

unde:

- *m* – numărul de cuvinte diferite din mulţime;
- *n* – numărul total de cuvinte;

$$AS_{(S1)} = \frac{C_3^2}{C_4^2} = \frac{1}{2} \qquad (29.7)$$

Cu cât valoarea indicatorului AS tinde către 1, cu atât rezultă că şirurile datorită asemănării dintre nu se constituie într-o mulţime de chei de regăsire a informaţiei. Concatenarea şirurilor este operaţia prin care două şiruri C_i şi C_j se alipesc rezultând un nou şir $C_{ij} = C_i \,||\, C_j$.

Dacă se consideră mulţimile de şiruri S_1 şi S_2 cu acelaşi număr de componente

$$S_1 = \{C_1, C_2, \ldots, C_{nsir}\} \qquad (29.8)$$

$$S_2 = \{d_1, d_2, \ldots, d_{nsir}\} \qquad (29.9)$$

prin concatenarea elementelor $l_i = c_i \,||\, d_i$ rezultă mulţimea:

$$S_3 = \{l_1, l_2, \ldots, l_{nsir}\} \qquad (29.10)$$

Indicele de asemănare pentru mulţimile S_1, S_2 şi S_3 trebuie să difere astfel încât:

$$AS(S_1) \geq AS(S_3) \text{ şi } AS(S_2) \geq AS(S_3) \qquad (29.11)$$

numai în cazul în care mulţimile S_1 şi S_2 sunt identice aceste condiţii nu sunt îndeplinite.

Pentru $S_1 = \{a, b, c, d\}$ şi $S_2 = \{x, y, z, w\}$ rezultă valorile:

$$AS(S_1) = 0; \qquad (29.12)$$

$$AS(S_2) = 0; \qquad (29.13)$$

iar mulţimea S_3 generată, are valorile

$$S_3 = \{ax, by, cz, dw\} şi\ AS(S_3) = 0 \quad (29.14)$$

Se observă că în ipoteza $AS(S_1) = 0$, $AS(S_2) = 0$ rezultă $AS(S_3) = 0$.
Pentru $S_1 = \{a, b, c, a\}$ şi $S_2 = \{x, y, z, x\}$ rezultă:

$$AS(S_1) = \frac{1}{2};\ AS(S_2) = \frac{1}{2};\ AS(S_3) = \frac{1}{2};\ S_3 = \{ax, by, cz, ax\} \quad (29.15)$$

Pentru $S_1 = \{a, b, c, a\}$ şi $S_2 = \{x, y, y, w\}$ rezultă:

$$AS(S_1) = \frac{1}{2};\ AS(S_2) = \frac{1}{2};\ AS(S_3) = 0;\ S_3 = \{ax, by, cy, aw\} \quad (29.16)$$

Concatenarea şirurilor corespondente din mulţimi presupune ca:
- mulţimile să aibă acelaşi număr de componente;
- operaţia să se efectueze pe elemente de pe aceeaşi poziţie din fiecare mulţime.

Mulţimile de şiruri se dispun sub forma unor coloane într-un tabel cu *MS* coloane şi *NSIR* linii dacă numărul de mulţimi de şiruri este *MS* şi numărul de elemente într-un şir este *NSIR*, tabelul 29.5.

Tabelul nr.29.5 *Agregarea şirurilor*

Poziţie	**S_1**	**S_2**	**...**	**S_j**	**...**	**S_{MS}**
E_1	x_{11}	x_{12}	**...**	x_{1j}	**...**	x_{1MS}
E_2	x_{21}	x_{22}	**...**	x_{2j}	**...**	x_{2MS}
...	**...**	**...**	**...**	**...**	**...**	**...**
E_i	x_{i1}	x_{i2}	**...**	**x_{ij}**	**...**	x_{iMS}
...	**...**	**...**	**...**	**...**	**...**	**...**
E_{NSIR}	x_{NSIR1}	x_{NSIR2}	**...**	x_{NSIRj}	**...**	x_{NSIRMS}

în care:
- E_i – poziţia elementului în şir;
- S_j – mulţimea *j* de şiruri;
- X_{ij} – şirul de pe poziţia *i* din mulţime S_j.

Se calculează indicatorii $AS(S_i)$, i = 1,2, …, MS. Dacă există mulţimile Si_1, Si_2, …, Si_k pentru care indicatorii AS() = 0 rezultă că oricare din aceste chei se folosesc drept cheie unică de regăsire a informaţiei.

Se construiesc cele C_{MS}^2 mulţimi $S_j = S_i \,||\, S_j$ ale căror elemente rezultă prin concatenarea de elemente corespondente.

Se calculează indicatorii:

$$AS(S_{11}),\ AS(S_{12}),\ \ldots,\ AS(S_{ij})_{i>j},\ \ldots,\ AS(S_{MS-1\ MS}) \quad (29.17)$$

Dacă există indicatori cu valorile AS = 0 , acele mulţimi se utilizează ca chei unice.

Procedeul de concatenare se extinde la trei, patru mulţimi până rezultă mulţimea cu şirurile cele mai lungi obţinute din concatenarea celor MS mulţimi.

$$S_{1, 2, \ldots, MS} = \prod_{i=1}^{MS} S_i \quad (29.18)$$

De exemplu, se consideră mulţimile:

$$S_1 = \{a, b, c, d\} \quad (29.19)$$

$$S_2 = \{x, y, z, x\} \quad (29.20)$$

$$S_3 = \{1, 2, 3, 1\} \quad (29.21)$$

$$S_{123} = S_1 || S_2 || S_3 = \{ ax1, by2, cz3, ax1\} \quad (29.22)$$

pentru care se determină valorile:

$$AS(S_1) = \frac{1}{2};\ AS(S_2) = \frac{1}{2};\ AS(S_{123}) = \frac{1}{2} \qquad (29.23)$$

Rezultă că operaţia de concatenare nu a condus la obţinerea unor chei unice.

Bordarea este operaţia prin care unui tablou i se adaugă coloane sau linii. Bordarea unei coloane cu valori pentru care indicatorul AS are valoarea 0 înseamnă adăugarea la tabloul dat a unei coloane de chei unice.

Se obţine în acest fel un nou tablou în care elementele se identifică fără dificultate prin chei unice.

Atunci când cheia unică este adresa fizică a articolului sau poziţia pe care articolul o ocupă în fişier, baza de date sau în depozitul de date, procesul de regăsire a informaţiei este accelerat.

Dacă cheia unică este invariabilă atunci când rezultă din măsurătorile caracteristicilor, îşi pierde acest caracter când se produce trecerea pe un alt suport dacă cheia este rezultatul bordării cu o coloană de adrese.

Când cheia unică indică poziţia elementului prin inserare se produce schimbarea de poziţie ceea ce impune existenţa unui algoritm recalculare a adresei relative.

29.3 Algoritmi de căutare

Orice tip de aplicaţie informatică are componente de prelucrare a datelor indiferent de tipul sau sursa de provenienţă a acestora. Obiectivul este de a obţine rezultate, de a atinge obiectivul final pentru care a fost dezvoltată aplicaţia, rezolvarea unei probleme bine definite.

Datele prelucrate de aplicaţie sunt stocate în colecţii conform şi sunt aranjate conform unor metode considerate de programator importante şi eficiente. Prelucrarea acestor date, presupune iteraţii în care seturi de una sau mai multe valori sunt preluate din mulţime şi sunt introduse ca operanzi

în secvenţe de instrucţiuni. Pentru aceasta programatorul utilizează rutine de căutare pentru a avea acces la valorile utilizate.

Algoritmii de regăsire au menirea de a identifica în mulţimea de valori, articolul a cărui cheie este specificată.

În funcţie de complexitatea operaţiei de căutare, există următoarele tipuri de algoritmi de regăsire:

- algoritmi bazaţi pe operaţii de comparare a cheii specificate cu chei din mulţimea de articole; procesul se încheie atunci când cheia specificată este localizat sau când a fost epuizată mulţimea articolelor; performanţa algoritmilor este dată de numărul de comparări;
- algoritmi în care se calculează pe baza cheii adresa relativă şi adresa fizică a articolului; performanţa este dată de numărul de operaţii şi de densitatea poziţionării articolelor.

În funcţie de locaţia datelor, operaţia de căutare are loc:

- în memoria internă a calculatorului; dimensiunea setului de date este redusă însă procesul este caracterizat de viteza mare de execuţie datorită efortului redus al procesorului de acces la date; pentru a putea rula rutinele de căutare, la fiecare execuţie a aplicaţiei trebuie construit setul de date deoarece acesta nu există în afara memoriei rezervate aplicaţiei;
- în memoria externă, datele fiind stocate în fişiere pe diverse suporturi de stocare; dimensiunea setului de date este foarte mare, iar procesul este caracterizat de viteza redusă de prelucrare datorită efortului procesor ridicat dat de citirea datelor şi aducerea acestora în memoria internă; datele sunt independente de aplicaţie din punctul de vedere al existenţei acestora, nefiind nevoie să fie recreate de fiecare dată când aplicaţia este lansată în execuţie;
- în memoria internă şi externă; se combină tehnicile de căutare în memoria internă, aplicate unor structuri auxiliare, cu metodele de acces la date stocate pe disc.

Legătura de interdependenţă dintre procesul de căutare şi structurile de date are la baza necesitatea de a stoca informaţia fie în memoria virtuală sau în cea externă într-o formă care să asigure integritatea datelor şi care să permită ulterior citirea acestora.

Căutarea secvenţială este procesul prin care datele sunt regăsite prin parcurgerea lineară a tuturor elementelor din mulţimea de date pornind de la începutul acesteia. Procesul se finalizează cu regăsirea elementului căutat sau cu atingerea sfârşitului colecţiei de date, lucru care indică terminarea operaţiei de căutare. Figura 29.2 descrie modul în care este aplicată căutarea secvenţială în cadrul unei colecţii de date formată din elementele $Elem_1$, $Elem_2$, …, $Elem_n$ pentru a găsi valoarea XVal.

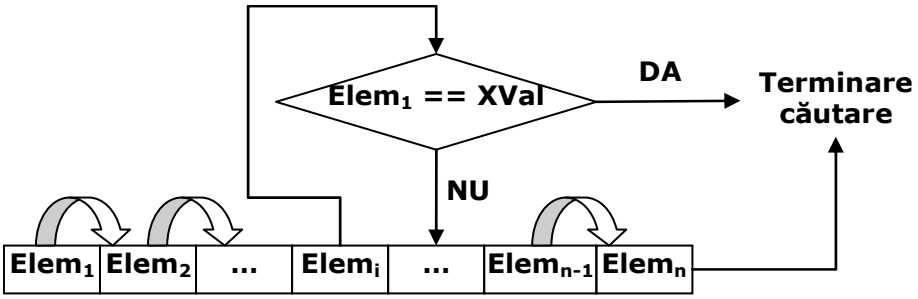


Figura 29.2 *Căutarea secvenţială a unei valori într-o colecţie*

Pentru a implementa într-un limbaj de programare această metodă se ţine seama că aceeaşi rutină de verificare a valorii căutate se aplică pentru fiecare element în parte.

```
int SecventialSearch(int * array, int arrayLength, int searchvalue){
      int i;
      for( i = 0;  i < arrayLength; i++ ){
         if(array[i] == searchvalue)
               return i;
      }
      return -1;
}
```

În cazul în care structura de date utilizată pentru a stoca valorile mulţimii este de tip listă simplu înlănţuită atunci rutina utilizată pentru căutarea secvenţială a unei valori este *ListSecventialSearch*.

```
struct SLList
{
      int value;
      SLList *next;
};
SLList* ListSecventialSearch(SLList * head, int searchvalue){
      SLList *temp;
      for( temp = head;  temp!=NULL; temp = temp->next ){
         if(temp->value == searchvalue)
               return temp;
      }
      return NULL;
}
```

Această metodă de căutare este îmbunătăţită prin reducerea numărului de comparaţii realizate în cadrul ciclului repetitiv, [Knut73], tehnică specifică optimizării la nivel de text sursă, [Ivan07a].

```
int QuickSecventialSearch(int * array, int arrayLength,
int searchvalue){
      int i;
      array[arrayLength] = searchvalue;
      for( i = 0;; i++ ){
         if(array[i] == searchvalue)
               if(i==arrayLength)
                     return -1;
               else
                     return i; }}
```

Se observă că această metodă diferă prin:

- utilizarea unui element suplimentar pe poziţia *arrayLength* în cadrul vectorului pentru a fi utilizat la verificarea condiţiei de terminare a căutării;
- reducerea la unu a numărului de condiţii ce sunt evaluate la fiecare iteraţie a ciclului repetitiv;
- inserarea unei noi condiţii de verificare a terminării căutării; procesul se încheie de fiecare dată cu găsirea elementului căutat, însă dacă acesta se găseşte pe poziţia *arrayLength* rezultă că elementul nu se regăseşte în mulţime.

Această simplă modificare conduce la reducerea numărului de cicluri maşină necesare realizării căutării. În ciuda faptului că a două metodă necesită un element suplimentar în cadrul vectorului, volumul redus de cicluri maşină reprezintă un efect mult mai important al procesului de optimizare.

Pe baza metodelor de optimizare a textului sursă, [Ivan07a] se maximizează viteza de căutare prin desfăşurarea ciclului repetitiv şi multiplicarea numărului de elemente verificate. De exemplu secvenţa

```
int FastSecventialSearch(int * array, int arrayLength,
int searchvalue){
      int i;
      array[arrayLength] = searchvalue;
      for( i = 0;; i+=3 ){
         if(array[i] == searchvalue)
               if(i==arrayLength)
                     return -1;
               else
                     return i;
         if(array[i+1] == searchvalue)
               if(i==arrayLength)
                     return -1;
               else
                     return i+1;
         if(array[i+1] == searchvalue)
               if(i==arrayLength)
                     return -1;
               else
                     return i;
         if(array[i+2] == searchvalue)
               if(i==arrayLength)
                     return -1;
               else
                     return i+2;
      }
}
```

creşte la trei numărul de elemente verificate în cadrul fiecărei iteraţii. Deoarece ciclul repetitiv se termină cu regăsirea elementului căutat pe o poziţie $0 \leq i \leq arrayLength$ sau pe poziţia *arrayLength* nu are loc depăşirea acestuia şi nu se generează o eroare de citire în afara vectorului.

Pe baza metodei parcurgerii secvenţiale a unei colecţii se defineşte căutarea cu elemente sortate după valoarea cheii.

```
void SortSecventialSearch(int * array, int arrayLength,
int searchvalue){
      int i;
      for( i = 0;  i < arrayLength; i++ ){
         if(array[i] >= searchvalue)
               break;
      }
      if(array[i]==searchvalue)
            return i;
      else
            return -1;
}
```

În cazul în care valoarea căutată se regăseşte în mulţimea de valori, această metodă nu îmbunătăţeşte nivelul de eficienţă al căutării secvenţiale, funcţia *SecventialSearch*. Rezultatele, minimizarea efortului prin reducerea numărului de valori verificate, sunt evidente în cazul în care valoarea căutată nu se află în mulţime. În această situaţie, ciclul repetitiv este întrerupt dacă se întâlneşte un element cu valoarea mai mare decât cel căutat şi nu se mai parcurge tot vectorul.

O altă metodă de căutare secvenţială se bazează pe frecvenţa de referire a elementelor. Cu cât frecvenţa unui element este mai ridicată, cu atât acestea sunt mai la început şi astfel numărul de comparări regăsirii lor scade.

De exemplu, se consideră şirul cheilor de căutare pentru care se înregistrează frecvenţele

a	*b*	*c*	*d*	
1	*1*	*3*	*7*	*(29.24)*

Datele sunt aranjate astfel încât, cuvintele să fie sortate în ordinea descrescătoare a frecvenţelor

7d 3c 1a 1b (29.25)

Utilizarea unei astfel de structuri presupune implementarea unui câmp în structura de date utilizată pentru a înregistra numărul de căutări. De exemplu, se consideră problema stocării studenţilor dintr-o facultate pentru care se implementează structura

```
struct Student
{
      char nume[20];
      int varsta;
      int cod
      int frecventa;
};
```

Metoda utilizată la căutarea unui student după codul acestuia este identică cu rutina *SecventialSearch* din punctul de vedere al algoritmului implementat, însă ia în considerare actualizarea câmpului *frecventa* pentru elementele căutate şi resortarea mulţimii.

Eficienţa acestei metode în raport cu o căutare secvenţială pe o mulţime neordonată sau sortată se bazează pe faptul că în practică, fiecărui

element din mulţime i se asociază o probabilitate de utilizare în funcţie problema ce trebuie rezolvată.

Există numeroase studii în acest domeniu, [Knut73], însă două reguli stabilesc importanţa unui set restrâns de elemente, cu frecvenţă ridicată de utilizare, în cadrul operaţiilor de căutare:

- conform legii lui Zipf, [Wiki07a], probabilitate de utilizare a elementelor dintr-o mulţime respectă o distribuţie dată de relaţia

$$p_1 = \frac{c}{1},\ p_2 = \frac{c}{2},\ \ldots,\ p_N = \frac{c}{N},\ cu\ c = \frac{1}{\sum_{i=1}^{N}\frac{1}{i}} \qquad (29.26)$$

  acesta a observat în că frecvenţa celui mai utilizat *k* cuvânt este invers proporţională cu poziţia sa în tabela de frecvenţe; astfel cel mai utilizat cuvânt va apărea în texte de aproximativ de două ori mai mult decât al doilea cel mai utilizat cuvânt; acesta, la rândul său are de două ori mai multe apariţii decât al patrulea cuvânt din lista de frecvenţe; spre deosebire de alte modele de distribuţie artificiale, această teorie a fost stabilită pe cale empirică şi de multe ori există situaţii reale în care se respectă;
- legea *80-20,* sau principiul Pareto, [Wiki07a], este un model empiric definit de economistul Vilfredo Pareto pe baza analizei veniturilor Italiei; acesta a observat că 80% din veniturile Italiei provin de la 20% din populaţie; principiul a fost extins pe baza de cercetări şi în domeniu optimizării aplicaţiilor informatice prin faptul că 80% din resurse sunt utilizate de către 20% din componentele aplicaţiei; la nivel de viteză de execuţie, se observă că 90% din prelucrările efectuate de program au loc la nivelul a doar 10% din codul sursă;

Pentru a evita definirea de câmpuri suplimentare în interiorul structurii de date cu scopul de a gestiona frecvenţa de utilizare, este definită o metodă care se bazează pe principiul repoziţionării la începutul colecţiei de date a fiecărui element căutat. Astfel, se obţine acelaşi rezultat, prin poziţionarea elementelor căutate des la începutul mulţimii.

Repoziţionarea unui element în cadrul unui vector implică recopierea elementelor, anterioare elementului mutat, pe poziţii următoare. Acest lucru implică un efort suplimentar de prelucrare a datelor. Pentru a fi evitat, colecţia de date este reprezentată printr-o listă simplă înlănţuită. În această situaţie, repoziţionarea este echivalentă cu iniţializarea unui set restrâns de pointeri.

Căutarea secvenţială cu elemente sortate după valoarea cheii este îmbunătăţită prin algoritmul de căutare binară, ce presupune începerea procesului dintr-un punct mai apropiat de valoarea căutată decât de la început. Primul reper în procesul de căutare este definit de poziţia din mijloc a şirului de valori sortate. În funcţie de mărimea relativă a valorii căutate faţă de valoarea mediană, căutarea continuă în şirul elementelor aflate la stânga sau la dreapta acesteia prin repetarea procesului.

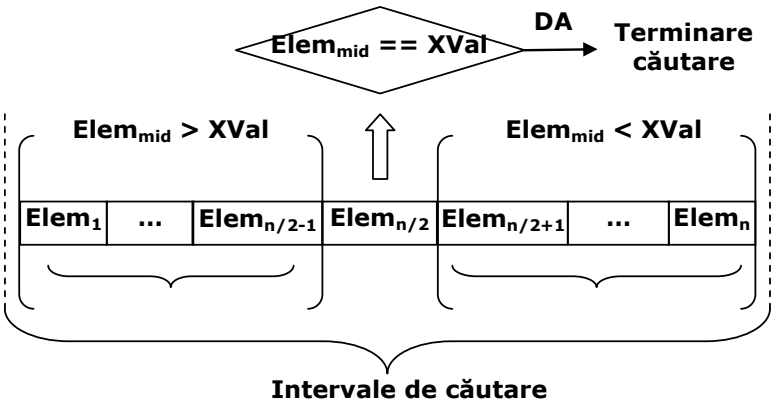


Figura 29.3 *Căutarea binară a unei valori într-o colecţie sortată crescător*

Presupunând că procesul începe cu un vector de lungime LV = n atunci poziţia mediană este determinată prin relaţia

$$mediana = (poz_{start} + poz_{final})/2 \qquad (29.27)$$

unde:

- *mediana* – poziţia mediană;
- poz_{start} – poziţia de start a intervalului de căutare; iniţial aceasta este egală cu 0;
- poz_{final} – poziţia finală a intervalului de căutare; iniţial aceasta este egală cu *n*.

Prin compararea valorii căutate cu valoarea de pe poziţia mediană se determină:

- momentul găsirii, dacă cele două valori sunt egale;
- continuarea căutării în intervalul inferior, dacă valoarea căutată este mai mică;
- continuarea căutării în intervalul superior, dacă valoarea căutată este mai mică;
- încetarea procesului dacă valorile nu sunt egale şi nu există alte intervale de valori.

```
void BinaraSearch(int * array, int arrayLength, int searchvalue){
            int middlevalue;
            int start = 0;
            int end = arrayLength-1;
            int middle;
            while(start <= end){
                  middle = (start + end)/2;
                  middlevalue  = array[middle];
                  if(searchvalue == middlevalue){
                        printf("\n  SEARCH VALUE FOUND !");
                        return;
                  }
                  else
                        if(searchvalue < middlevalue)
                              end = middle -1;
                        else
                              start = middle +1;
            } }
```

Faţă de căutarea secvenţială, căutarea binară reduce cu mult numărul de comparări, deoarece la fiecare iteraţie se elimină din aria de căutarea jumătate din valorile neverificate.

Între tipurile de algoritmi de regăsire şi tipurile de structuri de date utilizate în programare există o legătură strânsă de interdependenţă. Semnificaţia legăturii este evidenţiată şi de faptul că au fost definite tipuri particulare de structuri de date pentru a permite minimizarea efortului de căutare şi regăsire a informaţiilor. Astfel de exemple, sunt definite de structurile arborescente de căutare:

- arborii binari în care fiecare nod numit părinte are maxim două noduri numite fiu între care există relaţia

$$Val_{fiu\_stânga} < Val_{parinte} < Val_{fiu\_dreapta} \qquad (29.28)$$

  regăsirea informaţiei în arborii binari de căutare se face prin parcurgerea structurii şi prin verificarea în fiecare nod a egalităţii dintre valoarea căutată şi cheia nodului curent; în cazul în care nu există egalitate, se continuă procesul pe fiul din stânga sau pe cel din dreapta în funcţie de sensul relaţiei dintre valorile comparate

```
struct BST
{
    int info;
    BST * right;
    BST * left;
}

void BinarySearchTreeSearch(BST * root, int info)
{
    if(! root)
          return;
    else
    {
          if(info == root->info)
          {
                      printf("\n  SEARCH VALUE FOUND !");
                      return;
          }
          else
                if(info > root->info)
                      BinarySearchTreeSearch(root->right, info);
                else
                      BinarySearchTreeSearch(root->left, info);
    }
}
```

- pentru a creşte eficienţa operaţiei de căutare pe acest tip de structură, se iau în considerare tipuri particulare de arbori binari, ce sunt echilibraţi, AVL sau Roşu & Negru; structurile echilibrate contribuie la creşterea eficienţei algoritmului de căutare prin rearanjarea valorilor astfel încât să fie minimizat efectul situaţiei cel mai puţin favorabile; în cazul arborilor binari de căutare, situaţia cel mai puţin favorabilă este dată de obţinerea unei structuri arborescente cu înălţimea egală cu numărul de

elemente, figura 29.4 (a); reechilibrarea acestei structuri, figura 3.4 (b), minimizează timpul de căutare deoarece înălţimea structurii este minimizată şi astfel scade numărul maxim de comparări ce trebuie efectuate pentru a găsi valoarea căutată;

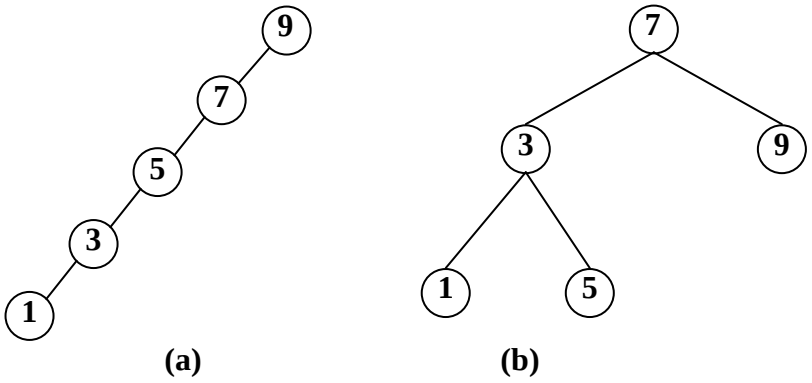


Figura 29.4 Tipuri de structuri arborescente binare de căutare.

echilibrarea arborilor de căutare nu influenţează rutina de căutare, deoarece relaţia dintre valoarea nodului părinte şi valorile nodurilor fiu se menţine; diferenţa se observă la nivelul secvenţelor de cod de inserare, respectiv de ştergere a unei valori, deoarece structura trebuie reechilibrată după fiecare operaţie de acest tip;

- arborii B sunt structuri arborescente de ordin N echilibrate, în care un nod poate conţine maxim 2*N chei şi minimum N chei, cu excepţia nodului rădăcină care poate avea mai puţin de N chei; această situaţie este întâlnită în momentul construirii arborelui B, atunci când au fost inserate mai puţin de N chei; pentru fiecare pereche de chei $KValue_i$ şi $KValue_{i+1}$ cu *i = 1..n* există un nod fiu NF_{i+1} pentru care:

$$KValue_i < Valori\_chei\{NF_{i+1}\} < KValue_{i+1} \qquad (29.29)$$

Figura 29.5 descrie modul în care este definit un nod al arborelui B de ordin N.

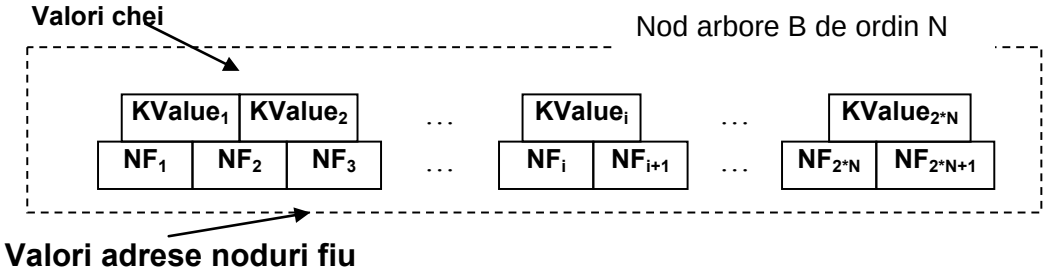


Figura 29.5 *Structura unui nod al arborelui B*

Dacă se consideră implementarea structurii arborelui B prin intermediul listelor de chei şi de noduri fiu atunci secvenţa de cod de căutarea a unei valori este

```
typedef struct NodArboreB_liste;

struct ListaChei
{
      int valCheie;
      ListaChei * next;
};

struct ListaFii
{
      NodArboreB_liste * Fiu;
      ListaFii * next;
};

struct NodArboreB_liste
{
      int NrChei;
      ListaChei *capListaChei;
      NodArboreB_liste *NodParinte;
      ListaFii *capListaFii;
};

struct NodArboreB
{
      int chei[MAX_CHEI];
      int NrChei;
      NodArboreB * NodParinte;
      NodArboreB * NodFiu[MAX_CHEI+1];
};


void  cautaValoare(NodArboreB  *NodStart,int  ValCheie,  NodArboreB  *&
NodGasit, int &pozCheie)
{
if(NodStart){
      int nrCheiNod = NodStart->NrChei;
      if((ValCheie<NodStart->chei[0])&&(ValCheie>NodStart-
>chei[nrCheiNod-1]))
      if((NodStart->NodFiu[0]      ==      NULL)      &&      (NodStart-
>NodFiu[nrCheiNod])){
            Printf("\n Cheie inexistenta !");
            return;
            }
      for(int i=0;i<nrCheiNod;i++)
            if(ValCheie==NodStart->chei[i]){
                  NodGasit=NodStart;
                  pozCheie=i;
            }
            else
            if(ValCheie>NodStart->chei[i])
                break;
if((ValCheie<NodStart->chei[i]) && (NodStart->NodFiu[i]!=NULL))
      cautaValoare(NodStart->NodFiu[i], ValCheie, NodGasit, pozCheie);
if((ValCheie>NodStart->chei[i-1]) && (NodStart-> NodFiu[i]!=NULL))
      cautaValoare(NodStart->NodFiu[i], ValCheie, NodGasit, pozCheie);
      }
};
```

Pentru a gestiona informaţii stocate pe suport extern, limbajele de programare pun la dispoziţia programatorilor funcţii de lucru cu fişiere. În funcţie de tehnicile de stocare şi regăsire a datelor pe suport extern, sunt definite:

- fişiere cu acces secvenţial în care adăugarea de noi înregistrări se face întotdeauna la sfârşit, neexistând o procedură specială de aranjare a datelor; fişierele cu acces secvenţial sunt asemănătoare ca modalitate de lucru cu masivele unidimensionale şi din acest motiv căutarea se realizează prin parcurgerea fiecărei înregistrări până când se găseşte informaţia căutată sau până când se ajunge la sfârşitul fişierului;

```
struct Student{
int ID;
char Name[10];
int Age;
}
void FILESearch(char * FileName, int SearchValue){
      Student value;
      FILE *pFile = fopen(FileName,"rb");
      while(!feof(pFile)){
            fread(&value,sizeof(Student),1, pFile);
            if(SearchValue == value){
                  printf("\n  FILE SEARCH VALUE FOUND !");
                  return;
            }
      }
}
```

- fişierele cu acces direct sunt structuri de date externe, ce se regăsesc pe disc, în care există o corespondenţă directă între poziţia în fişier a unui element şi valoarea cheii de căutare; deoarece această legătură se bazează pe unicitatea cheii şi pe ipoteza că numărul de articole din fişier este egal cu valoarea maximă a cheii de căutare, fiecare înregistrare din fişier se găseşte în una din stările, validă sau ştearsă logică;
- fişierele de tip invers permit căutarea de informaţii pe bază de chei compuse; sunt utilizate în situaţiile în care se caută submulţimi de articole ce corespund unei filtrări multicriteriale; pentru a minimiza spaţiul ocupat de fişierul invers se vor implementa structuri pe biţi ce corespund criteriilor de căutare, [Smeu04]; se consideră problema gestiunii elevilor dintr-un judeţ; pentru fiecare se înregistrează date descrise de structura *elev;*

```
struct elev
{
      char sex;
      char nationalitate;
      char cetatenie;
      char mediu;
      char nume[20];
      char note[10];
      int nrmatricol;
}
```

Deoarece o parte din câmpuri au valori fixe:

- câmpul *sex* ia valori în mulţimea {0 – masculin, 1 – feminin};
- câmpul *nationalitate* ia valori în mulţimea {0 – română, 1 – romă, 2 – maghiară, 3 – alta};
- câmpul *cetatenie* are valorile {0 – română, 1 – alta};
- câmpul *mediu* are valorile {0 – rural, 1 – urban};

Pentru că aplicaţia trebuie să permită căutări după aceste criterii se defineşte o structură pe biţi:

```
struct elevI
{
    unsigned sex:1;
    unsigned nationalitate:2;
    unsigned cetatenie:1;
    unsigned mediu:1;
    unsigned :3;
};
```

ce defineşte elementele fişierului de tip invers utilizat în procesul de căutare; motivul implementării unei structuri pe biţi este dat de minimizarea spaţiului; setul de date al elevilor este gestionat prin intermediul a două fişiere, unul ce conţine toate informaţiile şi unul ce conţine valorile elementelor de tip *elevI*; legătura dintre cele două structuri de date externe este definită de corespondenţa ce există între elementele de pe poziţii identice; rolul fişierului invers, ce conţine doar valorile criteriilor utilizate la căutare, este de a permite implementarea unui proces de căutare mult mai eficient; operaţia de căutare multicriterială se realizează prin intermediul variabilelor de tip *mască* şi *fltru*; masca este utilizată pentru a testa dacă elementul analizat îndeplineşte criteriul de selecţie prin intermediul operaţiei de sau exclusiv pe biţi; filtrul este utilizat pentru a aduce în prim plan doar acele câmpuri care compun condiţia multicriterială de selecţie; figura 29.6 descrie procesul de selecţie a elevilor care au naţionalitatea română şi provin din mediul urban;

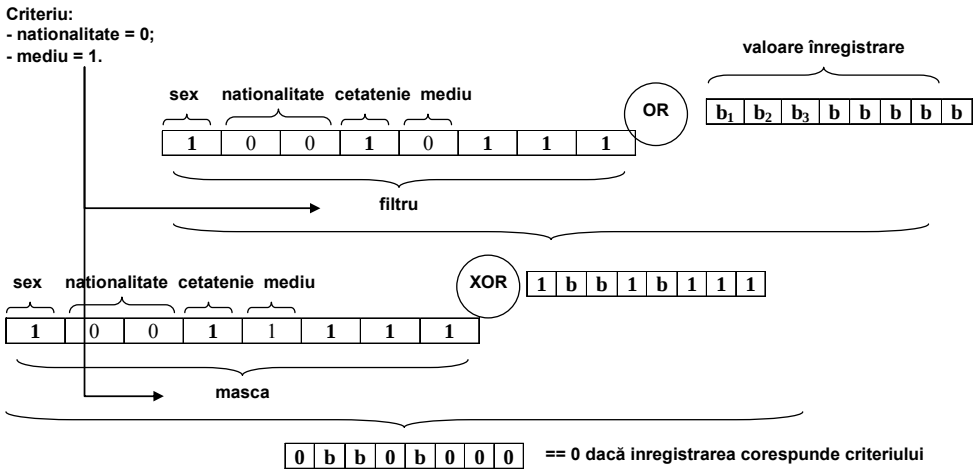


Figura 29.6 *Selecţia multicriterială prin intermediul fişierelor de tip invers*

- fişierele indexate sunt fişiere secvenţiale pentru care se construiesc o structură auxiliară de tip arbore care să eficientizeze

procesul de căutare; spre deosebire de fişiere secvenţiale, căutarea înregistrării după o chei unică se face în totalitate în structura arborescentă asociată fişierului; deoarece această structură are forma:

```
struct IndexFisier
{
      int ValoareCheie;
      int PozitieFisier;
};
```

odată ce este găsită valoarea căutată, este determinată poziţia în fişier a înregistrării respective; evitând o serie de citiri secvenţiale din fişier prin secvenţa de instrucţiuni

```
int PozInregistrare = PozitieFisier – 1;
fseek(Fisier, PozInregistrare * sizeof(Inregistrare), SEEK_SET);
```

este citită înregistrarea căutată;

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 poate 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 DirectAccess(element* array, int arrayLength, int searchvalue)
{
      if(searchvalue >= arrayLength) return -1;
            else if(array[searchvalue].cheie == -1) return -1;
                  else return array[searchvalue].valoare;
}
```

Cu toate acestea, în practică este evitată această soluţie directă care, deşi minimizează timpul de regăsirea are efecte negative asupra:

- memoriei ocupate; dimensiunea spaţiului, MEM, necesar implementării acestei structuri este dat de relaţia

*MEM = max(valoare cheie căutare) * dimensiune(element)* (29.30)

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

*MEM=max(nrMatricol)*dimensiune(Student)=2,54 MB* (29.31)

iar gradul de utilizare a acestui spaţiu este egal cu:

$$GUM = \frac{MEM_{elemente}}{MEM} * 100 = \frac{1450 * 48}{55630 * 48} * 100 = 2{,}6\% \quad (29.32)$$

- 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;
- 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.

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.

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 29.7.

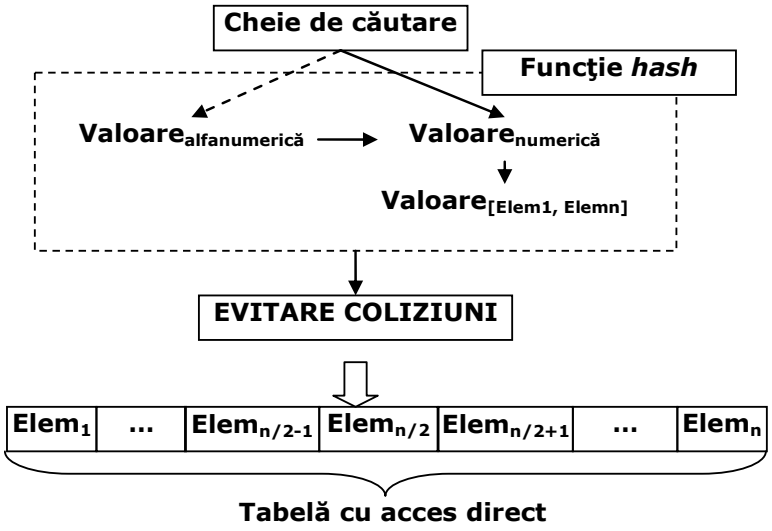


Figura 29.7 *Căutarea în tabela de dispersie*

Funcţiile de evitare a coliziunilor, descrise în cadrul capitolului dedicat tabelelor de dispersie, 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.

Analiza comparată a metodelor de căutare în vederea alegerii metodei care să dea cele mai bune rezultate se bazează pe:

- măsurarea timpului de execuţie; metoda optimă este caracterizată de cea mai mică valoare înregistrată;
- măsurarea efortului procesor prin prisma numărului de cicluri maşină realizate pentru a prelucra secvenţa de cod asociată metodei; codul care va genera cel mai mic nivel de cicluri maşină evidenţiază algoritmul care se va executa cel mai repede pentru datele de test şi care este cel mai eficient din punctul de vedere al sistemului;

- măsurarea zonei de memorie utilizată; algoritmul care va necesita cel mai mic nivel de memorie internă pentru a prelucra datele este considerat cel mai eficient din punct de vedere a spaţiului.
- analiza complexităţii algoritmului din punctul de vedere a calculelor efectuate şi alegerea modelului care este cel mai eficient.

Primele trei tehnici presupun realizarea unui program care să implementeze metodele de căutare analizate şi generarea unei proceduri de testare cu înregistrarea directă a valorilor indicatorilor. Ultima abordare ia în considerare o analiză a modelului asociat algoritmului de căutare înainte ca acesta să fie implementat.

Această tehnică se aplică în faza de analiză a problemei de rezolvat şi din acest motiv face abstracţie de mediul de implementare neluând în calcul:

- platforma hardware şi software pe care va rula programul;
- memoria disponibilă;
- limbajul de programare utilizat;
- tipul şi relaţiile dintre datele de test.

Indicatorii utilizaţi pentru a determina complexitatea algoritmului au caracter abstract şi descriu metoda prin prisma structurii interne, a numărul de instrucţiuni de comparare şi a numărului de iteraţii.

Din punct de vedere matematic, fiecare algoritm de căutare are asociat un model bazat pe o funcţie *f(n)* care descrie modul în care acesta este aplicat. De exemplu, în cazul căutării secvenţiale într-un vector *vector*,cu *n* elemente, modelul asociat are forma:

$$\text{fcautaresecv}(n)=\begin{cases}\textbf{dacă vector[0] = XVal, element gasit}\\ \textbf{dacă vector[1] = XVal, element gasit}\\ \textbf{.....}\\ \textbf{dacă vector[n-1] = XVal, element gasit}\\ \textbf{altfel, element inexistent}\end{cases}$$

Figura 29.8 *Model asociat căutării secvenţiale*

Dacă se consideră că principala operaţie pe care e bazează metoda de căutare secvenţială este compararea elementului curent din vector cu valoarea căutată, atunci complexitatea algoritmului este egală cu *n*, fiind notată *O(n)*.

În seturi reale de date de test, elementul poate fi găsit după o singură operaţie de căutare, după n/2 verificări sau acesta se poate găsi pe ultima poziţie din vector. Din acest motiv şi pentru a normaliza valorile înregistrate pentru complexitatea algoritmilor analizaţi se iau în calcul cazuri generale de execuţie:

- cel mai bun caz descrie situaţia în care este necesar minimul de efort procesor pentru a regăsi valoarea; în practică are probabilitatea cea mai mică de apariţie; în cazul căutării secvenţiale a unei valori într-o listă simplu înlănţuită, cel mai bun caz este dat de găsirea elementului căutat pe prima poziţie;

- cazul mediu descrie o analiză a unui set de operaţii de căutare pe baza căruia se determină un efort mediu necesar pentru a regăsi un element; în cazul algoritmilor complecşi este dificil de determinat acest nivel mediu;
- cel mai ineficient caz reprezintă situaţia în care este necesar cel mai ridicat nivel de efort pentru a căuta o valoare; pentru o analiză obiectivă a algoritmilor de căutare se utilizează acest criteriu în vederea alegerii rutinei mai eficiente; în cazul căutării secvenţiale a unei valori într-o listă simplu înlănţuită, cel mai ineficient caz este dat de găsirea elementului căutat pe ultima poziţie.

Pe baza modelului asociat algoritmului de prelucrare se determină nivelul de complexitate al acestuia. Chiar daca funcţia matematică pe baza căreia este definit modelul conţine numeroşi factori, pentru a determina gradul de complexitate se ia în considere dominantul relaţiei.

Din punct de vedere matematic, [Rile87], o funcţie, f(x), domină o altă funcţie, g(x), dacă există o valoare constantă *const* astfel încât pentru orice valoare x este adevărată relaţia

$$const*f(x) > g(x) \qquad (29.33)$$

În domeniul informatic, pentru a determina nivelul de complexitate al unui algoritm se utilizează o altă versiune a relaţiei anterioare, numită dominare asimptotica. Conform acestei teoreme, o funcţie f(x) domină asimptotic o altă funcţie g(x) dacă f(x) domină pe g(x) pentru toate valorile mari ale lui x.

Dacă se consideră funcţiile

$$f(x) = 2*x^3 + 1 \qquad (29.34)$$

$$g(x) = 5*x^2 + 3*x + 10 \qquad (29.35)$$

se observă din graficul 29.9 că pe intervalul [1;100] ambele funcţii cresc constant, însă de la valoarea 31 funcţia f(x) prezintă o creştere mult mai rapidă decât g(x).

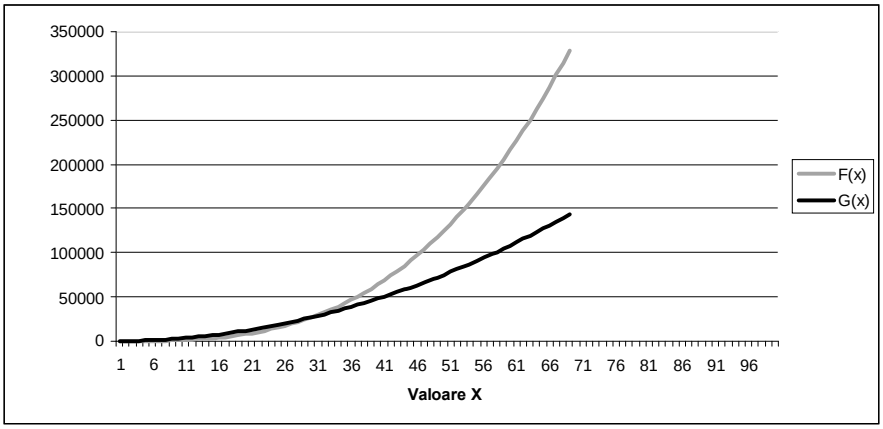


Figura 29.9 *Exemplu funcţii dominante*

În [Wiki07b] se prezintă principalele caracteristici şi reguli de determinare a nivelului complexităţii, indicator denumit *big-O.*

Principalele niveluri de complexitate regăsite la structurile de prelucrare a datelor, căutare sau sortare, sunt bazate pe funcţii:

- constante, de tipul f(n) = 1;
- liniare, de tipul f(n) = n;
- logaritmice, de tipul f(n) = $\log_2 n$;
- pătratice, de tipul f(n) = n^2;
- cubice, de tipul f(n) = n^3
- polinomial, de tipul f(n) = n^c, cu c >1;
- exponenţiale, de tipul f(n) = 2^n sau f(n) = a^n, cu a > 1.
- factoriale, de tipul f(n) = n!

Tabelul 29.6 descrie comparativ influenţa pe care o au diferitele grade de complexitate asupra efortului de prelucrare

Tabelul nr. 29.6 *Analiză comparată a nivelurilor de complexitate*

Valoare n	**f(n) = 1**	**f(n) = n**	**f(n)= $\log_2 n$**	**f(n) = n^2**	**f(n) = 2^n**
10	1	10	3.32	100	1024
100	1	100	6.64	10000	$1{,}26 * 10^{30}$
1000	1	1000	9.97	1000000	-
10000	1	10000	13.29	100000000	-

Pentru a analiza complexitatea structurilor de căutare descrise se ia în considerare cazul cel mai puţin favorabil.

Tabelul 29.7 descrie complexitatea algoritmilor de căutare analizaţi.

Tabelul nr. 29.7 *Complexitatea metodelor de căutare, cazul cel mai nefavorabil*

	Complexitate
metode de căutare în memoria internă	
Căutare cu acces direct	*O(1)*
Căutare secvenţială	*O(n)*
Căutare binară	*O($\log_2 n$)*
Căutarea în tabele de dispersie	*O(GU_{hash})*
Căutare în arbori binari de căutare echilibraţi	*O($\log_2 n$)*
Căutare în arbori B	$1+\log_N((n+1)/2)$, unde N este ordinul arborelui
metode de căutare în memoria externă	
Căutare secvenţială în fişier	*O(n)*
Căutare cu acces direct în fişier	*O(1)*
Căutare în fişier indexate	*O($\log_2 n$)* pentru index implementat pe arbori binari de căutare echilibraţi
Căutare în fişier inverse	*O(n)*

unde GU_{hash} descrie gradul de utilizare a tabelei de dispersie.

Se observă că din punctul de vedere al complexităţii utilizarea metodelor de căutare în vector şi în fişier secvenţial conduce la acelaşi nivel

de eficienţă. În cazul implementării celor două metode, se observă că citirea unui element dintr-un vector este mult mai rapidă decât citirea unui element din fişier. De exemplu secvenţa

```
        for( i = 0;  i < arrayLength; i++ ){
           if(array[i] == searchvalue)
               printf("\n  SEARCH VALUE FOUND !");
   }
```

necesită pentru un vector de 30000 de elemente distincte NCMV = 835 cicluri maşină, fiind de două ori mai eficientă decât secvenţa

```
        int value;
        FILE *pFile = fopen("SearchTest.dat","rb");
        while(!feof(pFile))
        {
            fread(&value,sizeof(int),1, pFile);
            if(searchvalue == value)
            {
                        printf("\n  FILE SEARCH VALUE FOUND !");
                        return;
            }

        }
```

care necesită pentru acelaşi set de date NCMF = 17317 cicluri maşină.

Programatorul trebuie să acorde atenţie acestui aspect şi trebui să-şi fundamenteze decizia de a implementa o structură de căutare în defavoarea alteia pe imposibilitatea de a stoca volumul mare de date în memoria internă sau pe obiectivul de a maximiza viteza de prelucrare a unui set redus de date.

Dacă este considerată a colectivitate C formată din elementele c_1, c_2, …, c_n şi identificate prin cuvinte cu valoare de cheie de regăsire având grade de utilizare diferite, algoritmii de regăsire performanţi revin la a:

- determina durata operaţiei de regăsire stabilă; această caracteristică a procesului de regăsire este măsurată în funcţie de valoarea dispersiei statistice; aceasta este calculată pentru şirul valorilor înregistrate la testarea performanţei algoritmului; gradul de stabilitate a operaţiei tinde către un nivel optim pe măsură ce valoarea dispersiei se apropie de nivelul zero; o altă abordare este dată de considerarea a două loturi de procese de regăsire; cele două serii de valori sunt analizate prin prisma mediilor m_1 şi m_2, în condiţiile în care acestea nu diferă semnificativ; (se iau 2 utilizatori şi se înregistrează tranzacţiile) procesul se extinde la *k* utilizatori ai algoritmului de regăsire;
- numărul de situaţii ambigue este controlat; numărul situaţiilor în care se produceau ambiguităţi e redus;
- există o situaţie în care se determină un volum maxim prelucrări şi o situaţie în care se determină un volum minim de prelucrări.

În cazul algoritmilor secvenţiali pentru cheia primului articol este nevoie de o comparare, pentru a doua cheie sunt necesare două comparări, iar pentru ultimul articol *n* comparări. Numărul total de comparări necesar regăsirii tuturor elementelor este

$$NTC = \frac{n*(n+1)}{2} \qquad (29.36)$$

Pentru cele *n* articole, numărul mediu de comparări este:

$$NMC = \frac{NTC}{n} = \frac{n+1}{2} \qquad (29.37)$$

Dacă este dată o structură arborescentă de căutare:

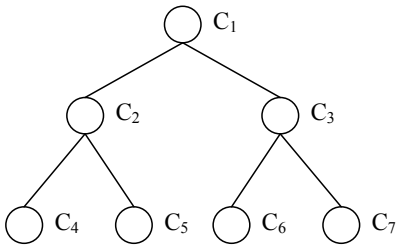


Figura 29.10 *Structura arborescentă de căutare*

Pentru regăsirea elementelor sunt necesare numărul de căutări:

C_1 – 1 comparare
C_2 – 2 comparări
C_3 – 2 comparări
C_4 – 3 comparări
C_5 – 3 comparări
C_6 – 3 comparări
C_7 – 3 comparări

În această situaţie, numărul total de comparări necesar identificării fiecărui element din arbore este

$$NTC = 1*1 + 2*2 + 3*4 = \sum_{k=1}^{m} k*2^{k-1} \qquad (29.38)$$

unde *k* reprezintă numărul de niveluri din arbore. Numărul mediu de comparări efectuate pentru oricare element este

$$NMC = \frac{\sum_{k=1}^{nv} k*2^{k-1}}{2^{nv+1}-1} \qquad (29.39)$$

unde *nv* reprezintă numărul de niveluri din arbore şi se consideră că $2^{nv+1}-1$ reprezintă numărul de noduri dintr-o structură arborescentă binară completă.

În ipoteza că arborele este complet echilibrat, atunci acesta conţine pe fiecare dintre cele i = 1,2, …, nv niveluri câte 2^{nv} noduri.

A optimiza o astfel de structură, înseamnă a găsi momentul de reechilibrare a arborelui. Fie V_i volumul de prelucrări necesar echilibrării *i* a

arborelui în care au fost adăugate noduri, prin adăugarea de frunze cu chei mai mari.

De exemplu, se consideră şirul valorilor 10, 15, 21, 43 şi 67 ce formează arborele binar de căutare echilibrat din figura 29.11.

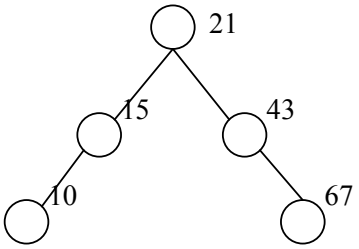


Figura 29.11 *Arbore binar de căutare*

Dacă se adaugă cheile 75, 101, 119, 400 se obţine arborele de căutare:

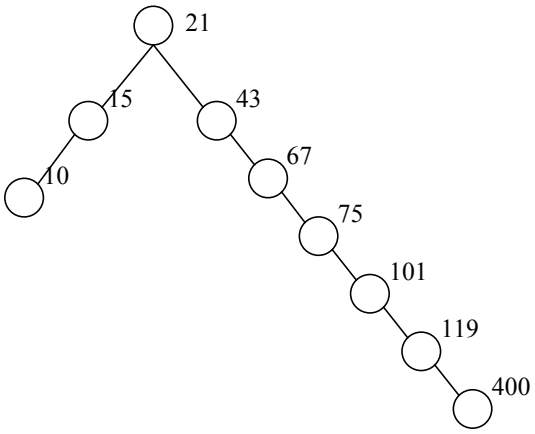


Figura 29.12 *Arbore binar de căutare dezechilibrat*

Volumul total de comparări pe arborele neechilibrat, din figura 29.12 este $VCA_1 = 33$. De la reorganizarea *i* a arborelui se trece la reorganizarea *i+1* dacă şi numai dacă

$$VCA_{k1} + VCA_{k2} + \ldots + VCA_{Kl} > V_{kl} \qquad (29.40)$$

Dacă se consideră operaţia de reechilibrare a arborelui bazată pe o parcurgere în inordine şi o reconstrucţie a arborelui prin metoda *divide et impera* pentru a insera la fiecare iteraţie elementul din mijlocul intervalului, atunci se obţine arborele echilibrat din figura 29.13.

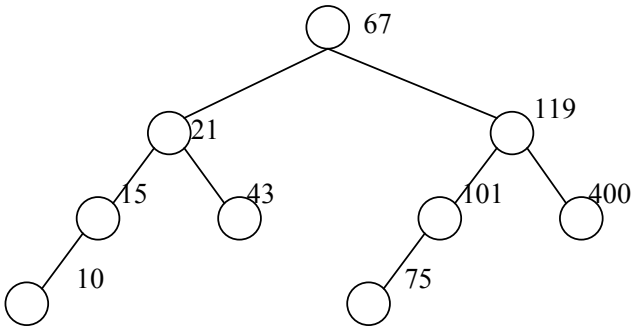


Figura 29.13 *Arbore binar de căutare reechilibrat*

Luând în considerare că operaţia de inserare a unui nod nou este echivalentă cu căutarea elementului respectiv, volumul de prelucrări al reechilibrării arborelui ca număr de comparaţii efectuate este VRA = 25. De asemenea, această valoare este egală cu volumul total de comparări pe arborele echilibrat obţinut, $VCA_2 = 25$. Se observă, minimizarea volumul total obţinut prin rearanjarea structurii, $VCA_2 < VCA_1$.

Atunci când este îndeplinită restricţia, K_l reprezintă momentul care precede echilibrarea arborescenţei, optimizând procesul de regăsire a informaţiei. Această abordare a optimizării regăsirii în structuri arborescente urmăreşte nu numai reducerea efortului de căutare, ci şi operaţiile de gestiune a arborelui, inserarea, respectiv, ştergere de valori. Obiectivul este de a reechilibra atunci când eficienţa căutării este afectată şi nu după fiecare operaţie de adăugare sau ştergere noduri.

Căutarea informaţiei sortate reprezintă un mod de optimizare. Se consideră şirul valorilor sortate crescător

10, 20, 25, 40, …., 200 (29.41)

Dacă informaţia nu e sortată este dificil de stabilit un mod de oprire a căutării dacă s-a depăşit o valoare.

În exemplul dat, dacă se caută elementul cu cheia 43 şi s-a trecut de cheia 50 din fişier, se conchide că articolul cu cheia 43 nu există.

Căutarea din mai multe direcţii oferă o soluţie alternativă pentru optimizarea timpului de regăsire. În ciuda faptului că soluţia implică o creşterea a nivelului de memorie ocupat, obiectivul minimizării timpului de prelucrare a datelor are prioritate din acest punct de vedere. Soluţii practice care implementează această abordare definesc mai multe repere în cadrul colectivităţii ţintă.

Există numeroase aplicaţii software ce utilizează articole cu cheie alfabetică. În funcţie de poziţionarea relativă a valorii cheii faţă de începutul setului sau de sfârşitul acestuia, căutarea are loc în direcţii diferite.

Dacă sunt chei care încep cu caracterul ’a’ , atunci căutarea are loc de la stânga la dreapta. Dacă sunt articole căutate după cheia *j* ce are valori apropiate caracterului ’z’ atunci căutarea se realizează de la sfârşitul seriei de valori.

Este necesar ca toate produsele software să fie înzestrate cu componente care înregistrează informaţii referitoare la modul în care s-au efectuat prelucrările.

De exemplu, se consideră programul P care lucrează cu un fişier în care se accesează articole după valori cheie, ce sunt sortate crescător.

Accesul la datele aplicaţiei se realizează de la stânga spre dreapta, având ca punct de start începutul fişierului.

Programul contorizează numărul de citiri din fişier şi numărul de comparări necesar identificării articolului căutat. Acesta va gestiona informaţiile prin realizarea unei structuri asemenea celei descrise în tabelul 29.8.

Tabelul nr. 29.8 *Analiza procesului de căutare şi citire din fişier*

Valoare cheie	**Număr comparări**	**Număr citiri**
k_1	a_1	cit_1
k_2	a_2	cit_2
…	…	…
k_i	a_i	cit_i
…	…	…
k_n	a_n	cit_n

În urma analizei datelor din tabel, rezultă $\overline{\alpha}$ numărul mediu de comparări şi o dispersie σ^2 ce descrie omogenitatea înregistrărilor. Dacă există *T* valori vor exista $\overline{\alpha}_1$, $\overline{\alpha}_2$, …, $\overline{\alpha}_T$ şi σ_1^2, σ_2^2, …, σ_T^2 indicatori.

Prin teste statistice se verifică egalitatea mediilor şi egalitatea dispersiei. În ipoteza de egalitate, procesul este stabil şi face obiectul procesului de optimizare. Se împarte fişierul în două zone de căutare, de la articolul art_1 la articolul $art_{\left[\frac{n}{2}\right]}$ şi de la acest punct până la art_n. În această situaţie, dacă valoarea căutată are o valoare mai mică decât A = $art_{\left[\frac{n}{2}\right]}$, căutarea are loc în prima jumătate. Dacă valoarea cheii este mai mare ca A = $art_{\left[\frac{n}{2}\right]}$, atunci căutarea este aplicată în a doua jumătate.

Pentru aceleaşi set de chei de căutare, k_1, k_2, …, k_n se repetă procesul, rezultând şirul numărului de căutări efectuate β_1, β_2, …, β_n. Pentru alte R loturi se obţin indicatorii $\overline{\beta}_1$, $\overline{\beta}_2$, …, $\overline{\beta}_R$ şi σ_1^2, σ_2^2, …, σ_R^2.

Dacă mediile şi dispersiile sunt egale revine că procesul este stabil.

Se compară mediile şi dispersiile rezultate în urma celor două teste pentru a identifica care dintre cele două procedee este mai bun.

Odată selectată abordarea care conduce la cele mai bune rezultate, se împarte fişierul în trei şi procedeul continuă.

Se va găsi nivelul optim de împărţire a fişierului pentru ca numărul de comparări să fie, statistic, cel mai mic.

Căutarea este un proces ce implică mai mult decât operaţia de comparare. Când aceasta este precedată de citire, este important ca numărul de citiri sa fie cât mai redus.

Regăsirea informaţiei presupune asocierea unor chei textului. Se consideră o mulţime de fişiere F_1, F_2, …, F_n şi se pune problema la căutare:

- găsirea fişierului F_i;
- găsirea în fişierul F_i a anumitor informaţii.

Se ia fiecare fişier F_1, F_2, …, F_n şi se construieşte o structură informaţională privind conţinutul acestuia, în care sunt reţinute:

C_1 – termeni generali;
C_2 – termeni specifici;
C_3 – grad de cuprindere mare;
C_4 – grad de cuprindere redus, particular conţinutului şi domeniului de apartenenţă.

Pentru fiecare fişier se construieşte vocabularul şi se iau termenii de bază după frecvenţe. Se asociază coeficienţii de complexitate (C_1 F_1), (C_2 F_2), …, (C_i F_i). Fişierele sunt împărţite în patru categorii de complexitate C_1, C_2, C_3 şi C_4. Se studiază elementele de vocabular şi se identifică tipul de complexitate pentru fiecare. Se ierarhizează cuvintele şi vor rezultat la motorul de căutare că pentru cuvântul cv_i sunt informaţii în fişiere de complexitate cuprinse între [α_i, β_i). Din mulţimea de fişiere se extrag numai acelea care au această complexitate după care se merge pe vocabular şi se iau fişierele unde cv_i au frecvenţa cea mai mare de apariţie.

Toate acestea presupun analiza comportamentului statistic pentru entităţi text, [Ivap05].

29.4 Mecanisme flexibile de regăsire

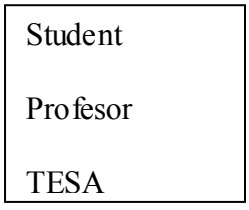
Spre deosebire de aplicaţiile clasice, aplicaţiile distribuite care se adresează unui număr foarte mare de clienţi trebuie să fie caracterizate prin nivel de flexibilitate foarte ridicat.

În primul rând flexibilitatea se manifestă prin diversitatea modurilor de introducere a datelor. Datele se introduc de la tastatură, prin coduri bară, prin scanare sau de pe un suport extern. Sunt aplicaţii care preiau comenzi vocale.

În al doilea rând flexibilitatea se manifestă prin structura datelor de intrare. Sunt acceptate atât date complete, cât mai ales date incomplete, care permit totuşi identificarea corectă a elementului căutat.

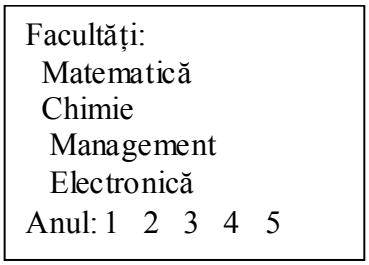
Expresiile care combină atribute se construiesc astfel încât să fie accesat din aproape în aproape acel conglomerat informaţional care corespunde evenimentului, tranzacţiei produsului sau serviciului de care este interesat clientul.

Pentru a afla nota obţinută la examen, studentul foloseşte adresa de Internet a universităţii la care acesta studiază. Primul meniu va fi:



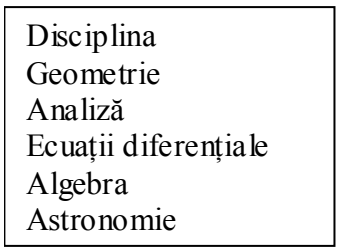
Studentul va alege opţiunea STUDENT.

Meniul care se activează este referitor la facultate:



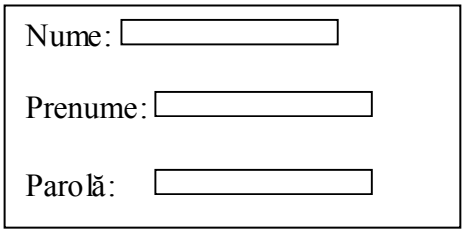
Studentul selectează MATEMATICĂ şi cifra 4.

Meniul următor se referă la disciplinele din anul şi facultatea selectată:

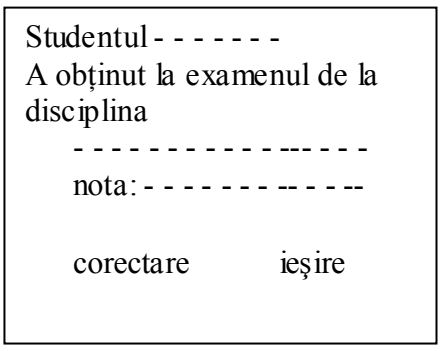


Este selectată opţiunea ALGEBRĂ

Meniul următor vizează introducerea numelui, prenumelui parolei



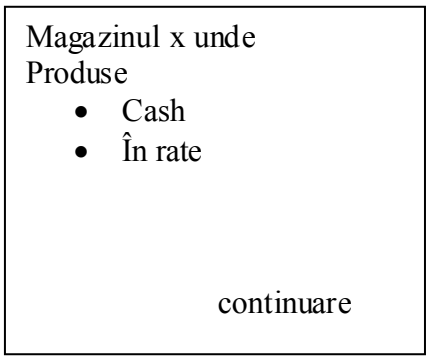
În cazul în care datele introduse sunt corecte, se afişează meniul:



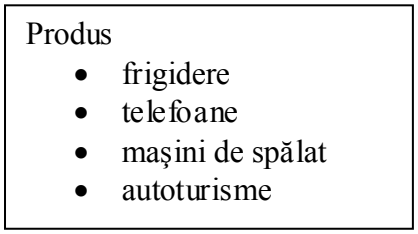
Selectarea opţiunii continuare permite apariţia meniului disciplinelor şi pot fi selectate 1, un grup sau toate şi vor fi afişate note.

Se observă că accesarea presupune selecţie. Datele de intrare sunt numele prenumele şi parola care trebuie ortografiate strict după reguli convenite la înscrierea în facultate privind literele mari, literele mici şi diacriticele.

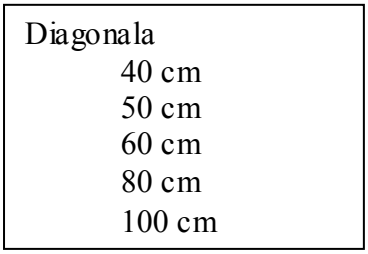
În cazul achiziţionării unui produs, clientul accesează un magazin virtual. Meniul de legătură este:



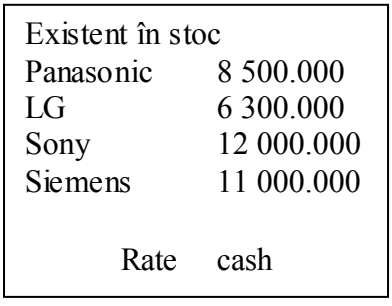
Clientul alege opţiunea convenabilă. Meniul următor se referă la tipuri de produse:



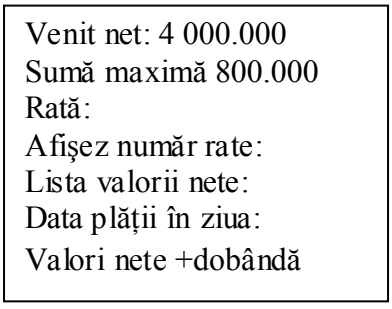
Selectează televizoare:



Selectează 60 cm:



Se tastează rate:



Problema căutării este definită printr-o mulţime de elemente $a_1, a_2, \ldots, a_n$ înzestrate cu diferite proprietăţi, a căror dispunere este de asemenea diferită. Trebuie localizat un element din mulţime în vederea utilizării. Localizarea este efectuată prin procese de căutare şi identificare elementul găsit este cel căutat, are loc prelucrarea.

Problema căutării are numeroase aplicaţii şi mai ales dezvoltări teoretice în domeniul valutar întrucât apar multe necunoscute în privinţa numărului elementelor $a_1, a_2, \ldots, a_n$, poziţiei şi proprietăţilor acestora.

Asemenea căutării în aplicaţiile valutare, implicaţiile Internet reprezintă numeroase necunoscute pentru clienţi. Fiecare aplicaţie are specificul ei.

Modul în care este definită interfaţa are rolul de a determina sistemul selecţiilor de căutare a revoluţionat întregul sistem al căutării.

Este important ca întregul sistem al căutării să se bucure de caracteristica de continuitate.

În primul rând se impune studiul interfeţei aplicaţiilor cele mai frecvent referite.

În al doilea rând, se utilizează caracteristicile de căutare deja cunoscute.

În al treilea rând se procedează la înzestrarea interfeţei cu caracteristice de reorientare, care va da din aproape în aproape o caracteristică apropiată de cerinţele materiale. Se înregistrează ferestrele de referire a opţiunilor după care se reorganizează structura mesajelor.

În al patrulea rând, se evaluează opţiunile în aşa fel încât situaţia în care se tastează date se reduc cât mai mult posibil.

În al cincilea rând se are în vedere găsirea tuturor posibilităţilor de introducere date şi de realizare de operaţii în aşa fel încât să se asigure completitudinea operaţiilor (mod de plată de prelucrare informaţii).

Căutarea clienţilor necesită algoritmi flexibili şi un nou mod de abordare.

Se subordonează criteriile de definire chei de selecţie şi mai ales opţiuni care ţin seama de cerinţele clienţilor şi nu de restricţiile echipelor care dezvoltă aplicaţiile.