

CUPRINS

1. STRUCTURI DE DATE SI TIPURI DE DATE ABSTRACTE	
1.1 Structuri de date fundamentale	3
1.2 Clasificări ale structurilor de date	3
1.3 Tipuri abstracte de date	4
1.4 Eficienta structurilor de date	6
2. STRUCTURI DE DATE ÎN LIMBAJUL C	
2.1 Implementarea operatiilor cu structuri de date	9
2.2 Utilizarea de tipuri generice	12
2.3 Utilizarea de pointeri generici	13
2.4 Structuri si functii recursive	16
3. VECTORI	
3.1 Vectori	24
3.2 Vectori ordonati	25
3.3 Vectori alocati dinamic	27
3.4 Aplicatie: Componente conexe	29
3.5 Vectori multidimensionali	31
3.6 Vectori de biti	32
4. LISTE CU LEGĂTURI	
4.1 Liste înlănțuite	35
4.2 Colectii de liste	39
4.3 Liste înlănțuite ordonate	41
4.4 Variante de liste înlănțuite	44
4.5 Liste dublu-înlănțuite	46
4.6 Comparatie între vectori si liste	48
4.7 Combinatii de liste si vectori	50
4.8 Tipul abstract listă (secvență)	54
4.9 Liste Skip	56
4.10 Liste neliniare	59
5. MULTIMI SI DICTIONARE	
5.1 Tipul abstract “Multime”	62
5.2 Aplicatie: Acoperire optimă cu multimi	63
5.3 Tipul “Colectie de multimi disjuncte”	64
5.4 Tipul abstract “Dictionar”	66
5.5 Implementare dictionar prin tabel de dispersie	68
5.6 Aplicatie: Compresia LZW	71
6. STIVE SI COZI	
6.1 Liste stivă	75
6.2 Aplicatie: Evaluare expresii	77
6.3 Eliminarea recursivității folosind o stivă	82
6.4 Liste coadă	84
6.5 Tipul “Coadă cu priorități”	90
6.6 Vectori heap	91
7. ARBORI	
7.1 Structuri arborescente	96
7.2 Arbori binari neordonati	97

7.3 Arbori binari pentru expresii	103
7.4 Arbori Huffman	104
7.5 Arbori multicăi	108
7.6 Arbori “rope”	114
7.7 Arbori Trie	115
7.8 Arbori kD si QuadTree	116
 8. ARBORI DE CAUTARE	
8.1 Arbori binari de căutare	120
8.2 Arbori binari echilibrati	122
8.3 Arbori Splay si Treap	126
8.4 Arbori AVL	130
8.5 Arbori RB	133
8.6 Arbori de căutare multicăi	135
 9. STRUCTURI DE GRAF	
9.1 Grafuri ca structuri de date	139
9.2 Reprezentarea grafurilor prin alte structuri	140
9.3 Metode de explorare a grafurilor	144
9.4 Sortare topologică	147
9.5 Aplicatii ale explorării în adâncime	149
9.6 Drumuri minime în grafuri	154
9.7 Arbori de acoperire de cost minim.....	158
9.8 Grafuri virtuale	162
 10. STRUCTURI DE DATE EXTERNE	
10.1 Specificul datelor pe suport extern	168
10.2 Sortare externă	169
10.3 Indexarea datelor	170
10.4 Arbori B	171
 11. STRUCTURI DE DATE ÎN LIMBAJUL C++	
11.1 Avantajele utilizării limbajului C++	177
11.2 Clase si obiecte în C++	178
11.3 Clase sablon (“template”) în C++	184
11.4 Clase container din biblioteca STL	187
11.5 Utilizarea claselor STL în aplicatii	191
11.6 Definirea de noi clase container	193
 PROBLEME PROPUSE	196

Capitolul 1

STRUCTURI DE DATE SI TIPURI DE DATE ABSTRACTE

1.1 STRUCTURI DE DATE FUNDAMENTALE

Gruparea unor date sub un singur nume a fost necesară încă de la începuturile programării calculatoarelor. Prima structură de date folosită a fost structura de vector (tabel), utilizată în operațiile de sortare (de ordonare) a colecțiilor și prezentă în primele limbaje de programare pentru aplicații numerice (Fortran și Basic).

Un vector este o colecție de date de același tip, în care elementele colecției sunt identificate prin indici ce reprezintă poziția relativă a fiecărui element în vector.

La început se puteau declara și utiliza numai vectori cu dimensiuni fixe, stabilite la scrierea programului și care nu mai puteau fi modificate la execuție.

Introducerea tipurilor pointer și alocării dinamice de memorie în limbajele Pascal și C a permis utilizarea de vectori cu dimensiuni stabilite și/sau modificate în cursul execuției programelor.

Gruparea mai multor date, de tipuri diferite, într-o singură entitate, numită “articol” (“record”) în Pascal sau “structură” în C a permis definirea unor noi tipuri de date de către programatori și utilizarea unor date dispersate în memorie, dar legate prin pointeri : liste înlănțuite, arbori și altele. Astfel de colecții se pot extinde dinamic pe măsura necesităților și permit un timp mai scurt pentru anumite operații, cum ar fi operația de căutare într-o colecție.

Limbajul C asigură structurile de date fundamentale (vectori, pointeri, structuri) și posibilitatea combinării acestora în noi tipuri de date, care pot primi și nume sugestive prin declarația *typedef*.

Dintr-o perspectivă independentă de limbajele de programare se pot considera ca structuri de date fundamentale vectorii, listele înlănțuite și arborii, fiecare cu diferite variante de implementare. Alte structuri de date se pot reprezenta prin combinații de vectori, liste înlănțuite și arbori. De exemplu, un tabel de dispersie (“Hash table”) este realizat de obicei ca un vector de pointeri la liste înlănțuite (liste de elemente sinonime). Un graf se reprezintă deseori printr-un vector de pointeri la liste înlănțuite (liste de adiacente), sau printr-o matrice (un vector de vectori în C).

1.2 CLASIFICĂRI ALE STRUCTURILOR DE DATE

O structură de date este caracterizată prin relațiile dintre elementele colecției și prin operațiile posibile cu această colecție.

Literatura de specialitate actuală identifică mai multe feluri de colecții (structuri de date), care pot fi clasificate după câteva criterii.

Un criteriu de clasificare folosește relațiile dintre elementele colecției:

- Colecții liniare (secvențe, liste), în care fiecare element are un singur succesor și un singur predecesor;
- Colecții arborescente (ierarhice), în care un element poate avea mai mulți succesori (fii), dar un singur predecesor (părinte);
- Colecții neliniare generale, în care relațiile dintre elemente au forma unui graf general (un element poate avea mai mulți succesori și mai mulți predecesori).

Un alt criteriu poate fi modul de reprezentare a relațiilor dintre elementele colecției:

- Implicit, prin dispunerea lor în memorie (vectori de valori, vectori de biți, heap);
- Explicit, prin adrese de legătură (pointeri).

Un alt criteriu grupează diferitele colecții după rolul pe care îl au în aplicații și după operațiile asociate colecției, indiferent de reprezentarea în memorie, folosind noțiunea de tip abstract de date. Astfel putem deosebi :

- Structuri de căutare (multimi și dicționare abstracte);
- Structuri de păstrare temporară a datelor (containere, liste, stive, cozi s.a.)

După numărul de aplicații în care se folosesc putem distinge între:

- Structuri de date de uz general ;
- Structuri de date specializate pentru anumite aplicații (geometrice, cu imagini).

Organizarea datelor pe suport extern (a fișierelor și bazelor de date) prezintă asemănări dar și diferite față de organizarea datelor în memoria internă, datorită particularităților de acces la discuri față de accesul la memoria internă.

Un fișier secvențial corespunde oarecum unui vector, un fișier de proprietăți este în fond un dicționar și astfel de paralele pot continua. Pe suport extern nu se folosesc pointeri, dar se pot folosi adrese relative în fișiere (ca număr de octeți față de începutul fișierului), ca în cazul fișierelor index.

Ideea unor date dispersate dar legate prin pointeri, folosită la liste și arbori, se folosește mai rar pentru fișiere disc pentru că ar necesita acces la articole neadiacente (dispersate în fișier), operații care consumă timp pe suport extern. Totuși, anumite structuri arborescente se folosesc și pe disc, dar ele țin seama de specificul suportului: arborii B sunt arbori echilibrați cu un număr mic de noduri și cu număr mare de date în fiecare nod, astfel ca să se facă cât mai puține citiri de pe disc.

Salvarea unor structuri de date interne cu pointeri într-un fișier disc se numește serializare, pentru că în fișier se scriu numai date (într-o ordine prestabilită), nu și pointeri (care au valabilitate numai pe durata executiei unui program). La încărcarea în memorie a datelor din fișier se poate reconstrui o structură cu pointeri (în general alți pointeri la o altă execuție a unui program ce folosește aceste date).

Tot pe suport extern se practică și memorarea unor colecții de date fără o structură internă (date nestructurate), cum ar fi unele fișiere multimedia, mesaje transmise prin e-mail, documente, rapoarte s.a. Astfel de fișiere se citesc integral și secvențial, fără a necesita operații de căutare în fișier.

1.3 TIPURI ABSTRACTE DE DATE

Un tip abstract de date este definit numai prin operațiile asociate (prin modul de utilizare), fără referire la modul concret de implementare (cu elemente consecutive sau cu pointeri sau alte detalii de memorare).

Pentru programele nebanale este utilă o abordare în (cel puțin) două etape:

- o etapă de concepție (de proiectare), care include alegerea tipurilor abstracte de date și algoritmilor necesari;
- o etapă de implementare (de codificare), care include alegerea structurilor concrete de date, scrierea de cod și folosirea unor funcții de bibliotecă.

În faza de proiectare nu trebuie stabilite structuri fizice de date și trebuie gândită aplicația în termeni de tipuri abstracte de date. Putem decide că avem nevoie de un dicționar și nu de un tabel de dispersie, putem alege o coadă cu priorități abstractă și nu un vector heap sau un arbore ordonat, s.a.m.d.

În faza de implementare putem decide ce implementări alegem pentru tipurile abstracte decise în faza de proiectare. Ideea este de a separa interfața (modul de utilizare) de implementarea unui anumit tip de colecție. În felul acesta se reduc dependentele dintre diferite părți ale unui program și se facilitează modificările care devin necesare după intrarea aplicației în exploatare.

Conceptul de tip abstract de date are un corespondent direct în limbajele orientate pe obiecte, și anume o clasă abstractă sau o interfață. În limbajul C putem folosi același nume pentru tipul abstract și aceleași nume de funcții; înlocuirea unei implementări cu alta poate însemna un alt fișier antet (cu definirea tipului) și o altă bibliotecă de funcții, dar fără modificarea aplicației care folosește tipul abstract.

Un tip de date abstract poate fi implementat prin mai multe structuri fizice de date.

Trebuie spus că nu există un set de operații unanim acceptate pentru fiecare tip abstract de date, iar aceste diferențe sunt uneori mari, ca în cazul tipului abstract "listă" (asa cum se pot vedea comparând bibliotecile de clase din C++ și din Java).

Ca exemplu de abordare a unei probleme în termeni de tipuri abstracte de date vom considera verificarea formală a unui fișier XML în sensul utilizării corecte a marcajelor ("tags"). Exemplele care urmează ilustrează o utilizare corectă și apoi o utilizare incorectă a marcajelor:

```
<stud>
  <nume>POPA</nume>
  <prenume>ION</prenume>
  <medie> 8.50</medie>
</stud>
```

```
<stud><nume>POPA<prenume> ION </nume> </prenume> <medie>8.50</medie>
```

Pentru simplificare am eliminat marcasele singulare, de forma <tag/>.

Algoritmul de verificare a unui fisier XML dacă este corect format folosește tipul abstract “stivă” (“stack”) astfel: pune într-o stivă fiecare marcaj de început (<stud>, <nume>,...), iar la citirea unui marcaj de sfârșit (</stud>, </nume>,...) verifică dacă în vârful stivei este marcajul de început pereche și îl scoate din stivă :

```
initializare stiva
repetă până la sfârșit de fisier
  extrage următorul marcaj
  dacă marcaj de început
    pune marcaj în stivă
  dacă marcaj de sfârșit
    dacă în vârful stivei este perechea lui
      scoate marcajul din vârful stivei
    altfel
      eroare de utilizare marcate
  dacă stiva nu e goală
    eroare de utilizare marcate
```

În această fază nu ne interesează dacă stiva este realizată ca vector sau ca listă înlănțuită, dacă ea conține pointeri generici sau de un tip particular.

Un alt exemplu este tipul abstract “multime”, definit ca o colecție de valori distincte și având ca operație specifică verificarea apartenenței unei valori la o multime (deci o căutare în multime după valoare). În plus, există operații generale cu orice colecție: inițializare, adăugare element la o colecție, eliminare element din colecție, afisare sau parcurgere colecție, s.a. Multimele se pot implementa prin vectori de valori, vectori de biti, liste înlănțuite, arbori binari și tabele de dispersie (“hash”).

În prezent sunt recunoscute câteva tipuri abstracte de date, definite prin operațiile specifice și modul de utilizare: multimi, colecții de multimi disjuncte, liste generale, liste particulare (stive, cozi), cozi ordonate (cu priorități), dicționare. Diferitele variante de arbori și de grafuri sunt uneori și ele considerate ca tipuri abstracte.

Aceste tipuri abstracte pot fi implementate prin câteva structuri fizice de date sau combinații ale lor: vectori extensibili dinamic, liste înlănțuite, matrice, arbori binari, arbori oarecare, vectori “heap”, fiecare cu variante.

Conceperea unui program cu tipuri abstracte de date permite modificarea implementării colecției abstracte (din motive de performanță, de obicei), fără modificarea restului aplicației.

Ca exemplu de utilizare a tipului abstract dicționar vom considera problema determinării frecvenței de apariție a cuvintelor într-un text. Un dicționar este o colecție de perechi cheie-valoare, în care cheile sunt unice (distincte). În exemplul nostru cheile sunt siruri (cuvinte), iar valorile asociate sunt numere întregi ce arată de câte ori apare fiecare cuvânt în fisier.

Aplicația poate fi descrisă astfel:

```
initializare dicționar
repetă până la sfârșit de fisier
  extrage următorul cuvânt
  dacă cuvântul există în dicționar
    aduna 1 la numărul de apariții
  altfel
```

pune in dictionar cuvant cu număr de aparitii 1
afisare dictionar

Implementarea dictionarului de cuvinte se poate face printr-un tabel hash dacă fisierele sunt foarte mari si sunt necesare multe căutări, sau printr-un arbore binar de căutare echilibrat dacă se cere afisarea sa numai în ordinea cheilor, sau printr-un vector (sau doi vectori) dacă se cere afisarea sa ordonată si după valori (după numărul de aparitii al fiecărui cuvânt).

Existenta unor biblioteci de clase predefinite pentru colectii de date reduce problema implementării structurilor de date la alegerea claselor celor mai adecvate pentru aplicatia respectivă si conduce la programe compacte si fiabile. "Adecvare" se referă aici la performantele cerute si la particularitățile aplicatiei: dacă se cere mentinerea colectiei în ordine, dacă se fac multe căutari, dacă este o colectie statică sau volatilă, etc.

1.4 EFICIENȚA STRUCTURILOR DE DATE

Unul din argumentele pentru studiul structurilor de date este acela că alegerea unei structuri nepotrivite de date poate influența negativ eficiența unor algoritmi, sau că alegerea unei structuri adecvate poate reduce memoria ocupată si timpul de executie a unor aplicatii care folosesc intens colectii de date.

Exemplul cel mai bun este al structurilor de date folosite atunci când sunt necesare căutări frecvente într-o colectie de date după continut (după chei de căutare); căutarea într-un vector sau într-o listă înlănțuită este inefficientă pentru un volum mare de date si astfel au apărut tabele de dispersie ("hash table"), arbori de căutare echilibrati, arbori B si alte structuri de date optimizate pentru operatii de căutare.

Alt exemplu este cel al algoritmilor folositi pentru determinarea unui arbore de acoperire de cost minim al unui graf cu costuri (Prim, Kruskal), care au o complexitate ce depinde de structurile de date folosite.

Influența alegerii structurii de date asupra timpului de executie a unui program stă si la baza introducerii tipurilor abstracte de date: un program care folosește tipuri abstracte poate fi mai usor modificat prin alegerea unei alte implementări a tipului abstract folosit, pentru îmbunătățirea performanțelor.

Problema alegerii unei structuri de date eficiente pentru un tip abstract nu are o solutie unică, desi există anumite recomandări generale în acest sens. Sunt mai multi factori care pot influența această alegere si care depind de aplicatia concretă.

Astfel, o structură de căutare poate sau nu să păstreze si o anumită ordine între elementele colectiei, ordine cronologică sau ordine determinată de valorile memorate. Dacă nu contează ordinea atunci un tabel de dispersie ("hash") este alegerea potrivită, dacă ordinea valorică este importantă atunci un arbore binar cu autoechilibrare este o alegere mai bună, iar dacă trebuie păstrată ordinea de introducere în colectie, atunci un tabel hash completat cu o listă coadă este mai bun.

În general un timp mai bun se poate obtine cu pretul unui consum suplimentar de memorie; un pointer în plus la fiecare element dintr-o listă sau dintr-un arbore poate reduce durata anumitor operatii si/sau poate simplifica programarea lor.

Frecvența fiecărui tip de operatie poate influența de asemenea alegerea structurii de date; dacă operatiile de stergere a unor elemente din colectie sunt rare sau lipsesc, atunci un vector este preferabil unei liste înlănțuite, de exemplu. Pentru grafuri, alegerea între o matrice de adiacente si o colectie de liste de adiacente tine seama de frecvența anumitor operatii cu graful respectiv; de exemplu, obtinerea grafului transpus sau a grafului dual se face mai repede cu o matrice de adiacente.

În fine, dimensiunea colectiei poate influența alegerea structurii adecvate: o structură cu pointeri (liste de adiacente pentru grafuri, de exemplu) este bună pentru o colectie cu număr relativ mic de elemente si care se modifică frecvent, iar o structură cu adrese succesive (o matrice de adiacente, de exemplu) poate fi preferabilă pentru un număr mare de elemente.

Eficiența unei anumite structuri este determinată de doi factori: memoria ocupată si timpul necesar pentru operatiile frecvente. Mai des se folosește termenul de "complexitate", cu variantele "complexitate temporală" si "complexitate spațială".

Operatiile asociate unei structuri de date sunt algoritmi, mai simpli sau mai complicati, iar complexitatea lor temporală este estimată prin notatia $O(f(n))$ care exprimă rata de crestere a timpului de executie în raport cu dimensiunea n a colectiei pentru cazul cel mai nefavorabil.

Complexitatea temporală a unui algoritm se estimează de obicei prin timpul maxim necesar în cazul cel mai nefavorabil, dar se poate tine seama si de timpul mediu si/sau de timpul minim necesar. Pentru un algoritm de sortare în ordine crescătoare, de exemplu, cazul cel mai defavorabil este ca datele să fie ordonate descrescător (sau crescător pentru metoda "quicksort"). Cazul mediu este al unui vector de numere generate aleator, iar cazul minim al unui vector deja ordonat.

În general, un algoritm care se comportă mai bine în cazul cel mai nefavorabil se comportă mai bine si în cazul mediu, dar există si exceptii de la această regulă cum este algoritmul de sortare rapidă QuickSort, care este cel mai bun pentru cazul mediu (ordine oarecare în lista inițială), dar se poate comporta slab pentru cazul cel mai nefavorabil (functie si de modul de alegere a elementului pivot).

Pentru a simplifica compararea eficienței algoritmilor se apreciază volumul datelor de intrare printr-un singur număr întreg N , desi nu orice problemă poate fi complet caracterizată de un singur număr. De exemplu, în problemele cu grafuri contează atât numărul de noduri din graf cât si numărul de arce din graf, dar uneori se consideră doar numărul arcelor ca dimensiune a grafului (pentru cele mai multe aplicatii reale numărul de arce este mai mare ca numărul nodurilor).

O altă simplificare folosită în estimarea complexității unui algoritm consideră că toate operatiile de prelucrare au aceeasi durată si că putem număra operatii necesare pentru obtinerea rezultatului fără să ne intereseze natura acelor operatii. Parte din această simplificare este si aceea că toate datele prelucrate se află în memoria internă si că necesită acelasi timp de acces.

Fiecare algoritm poate fi caracterizat printr-o functie ce exprimă timpul de rulare în raport cu dimensiunea n a problemei; aceste functii sunt mai greu de exprimat printr-o formulă si de aceea se lucrează cu limite superioare si inferioare pentru ele.

Se spune că un algoritm are complexitatea de ordinul lui $f(n)$ si se notează $O(f(n))$ dacă timpul de executie pentru n date de intrare $T(n)$ este mărginit superior de functia $f(n)$ astfel:

$T(n) = O(f(n))$ dacă $T(n) \leq k * f(n)$ pentru orice $n > n_0$
unde k este o constantă a cărei importanță scade pe măsură ce n crește.

Relatia anterioară spune că rata de crestere a timpului de executie a unui algoritm $T(n)$ în raport cu dimensiunea n a problemei este inferioară ratei de crestere a functiei $f(n)$. De exemplu, un algoritm de complexitate $O(n)$ este un algoritm al cărui timp de executie crește liniar (direct proportional) cu valoarea lui n .

Majoritatea algoritmilor utilizati au complexitate polinomială, deci $f(n) = n^k$. Un algoritm liniar are complexitate $O(n)$, un algoritm pătratic are complexitate $O(n^2)$, un algoritm cubic are ordinul $O(n^3)$ s.a.m.d.

Diferenta în timpul de executie dintre algoritmii de diferite complexități este cu atât mai mare cu cât n este mai mare. Tabelul următor arată cum crește timpul de executie în raport cu dimensiunea problemei pentru câteva tipuri de algoritmi.

n	$O(\log(n))$	$O(n)$	$O(n * \log(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	2.3	10	23	100	1000	10e3
20	3.0	20	60	400	8000	10e6
30	3.4	30	102	900	27000	10e9
40	3.7	40	147	1600	64000	10e12
50	3.9	50	195	2500	125000	10e15

Complexitatea unui algoritm este deci echivalentă cu rata de crestere a timpului de executie în raport cu dimensiunea problemei.

Algoritmii $O(n)$ si $O(n \log(n))$ sunt aplicabili si pentru n de ordinul 10^9 . Algoritmii $O(n^2)$ devin nepracticabili pentru $n > 10^5$, algoritmii $O(n!)$ nu pot fi folositi pentru $n > 20$, iar algoritmii $O(2^n)$ sunt inaplicabili pentru $n > 40$.

Cei mai buni algoritmi sunt cei logaritmici, indiferent de baza logaritmului.

Dacă durata unei operatii nu depinde de dimensiunea colectiei, atunci se spune că acea operatie are complexitatea $O(1)$; exemple sunt operatiile de introducere sau de scoatere din stivă, care operează la

vârful stivei si nu depind de adâncimea stivei. Un timp constant are si operatia de apartenență a unui element la o multime realizată ca un vector de biti, deoarece se face un calcul pentru determinarea pozitiei elementului căutat si o citire a unui bit (nu este o căutare prin comparatii repetate).

Operatiile de căutare secvențială într-un vector neordonat sau într-o listă înlântuită au o durată proporțională cu lungimea listei, deci complexitate $O(n)$ sau liniară.

Căutarea binară într-un vector ordonat si căutarea într-un arbore binar ordonat au o complexitate logaritmă de ordinul $O(\log_2 n)$, deoarece la fiecare pas reduce numărul de elemente cercetate la jumătate. Operatiile cu vectori heap si cu liste skip au si ele complexitate logaritmă (logaritm de 2). Cu cât dimensiunea colectiei n este mai mare, cu atât este mai mare câstigul obtinut prin căutare logaritmă în raport cu căutarea liniară.

Căutarea într-un arbore ordonat are o durată proporțională cu înălțimea arborelui, iar înălțimea este minimă în cazul unui arbore echilibrat si are valoarea $\log_2 n$, unde 'n' este numărul de noduri din arbore. Deci complexitatea operatiei de căutare într-un arbore binar ordonat si echilibrat este logaritmă în raport cu numărul de noduri (cu dimensiunea colectiei).

Anumite structuri de date au ca specific existenta unor operatii de durată mare dar care se execută relativ rar: extinderea unui vector, restructurarea unui arbore, s.a. Dacă am lua durata acestor operatii drept cazul defavorabil si am însuma pe toate operatiile am obtine rezultate gresite pentru complexitatea algoritmilor de adăugare elemente la colectie. Pentru astfel de cazuri devine importantă analiza amortizată a complexității unor secvente de operatii, care nu este neapărat egală cu suma complexităților operatiilor din secvență. Un exemplu simplu de analiză amortizată este costul adăugării unui nou element la sfârșitul unui vector care se extinde dinamic.

Fie C capacitatea momentană a unui vector dinamic. Dacă numărul de elemente din vector N este mai mic ca C atunci operatia de adăugare nu depinde de N si are complexitatea $O(1)$. Dacă N este egal cu C atunci devine necesară extinderea vectorului prin copierea celor C elemente la noua adresă obtinută. In caz că se face o extindere cu un singur element, la fiecare adăugare este necesară copierea elementelor existente în vector, deci costul unei operatii de adăugare este $O(N)$.

Dacă extinderea vectorului se va face prin dublarea capacității sale atunci copierea celor C elemente se va face numai după încă $C/2$ adăugări la vectorul de capacitate $C/2$. Deci durata medie a $C/2$ operatii de adăugare este de ordinul $3C/2$, adică $O(C)$. In acest caz, când timpul total a $O(N)$ operatii este de ordinul $O(N)$ vom spune că timpul amortizat al unei singure operatii este $O(1)$. Altfel spus, durata totală a unei secvente de N operatii este proporțională cu N si deci fiecare operatie este $O(1)$.

Această metodă de analiză amortizată se numeste metoda "agregat" pentru că se calculează un cost "agregat" pe o secvență de operatii si se raportează la numărul de operatii din secvență.

Prin extensie se vorbeste chiar de structuri de date amortizate, pentru care costul mare al unor operatii cu frecvență mică se "amortizează" pe durata celorlalte operatii. Este vorba de structuri care se reorganizează din când în când, cum ar fi tabele de dispersie (reorganizate atunci când listele de coliziuni devin prea lungi), anumite variante de heap (Fibonacci, binomial), arbori scapegoat (reorganizati când devin prea dezechilibrati), arbori Splay (reorganizati numai când elementul accesat nu este deja în rădăcină), arbori 2-3 si arbori B (reorganizati când un nod este plin dar mai trebuie adăugată o valoare la acel nod), s.a.

Diferentele dintre costul mediu si costul amortizat al unor operatii pe o structură de date provin din următoarele observatii:

- Costul mediu se calculează ca medie pe diferite intrări (date) si presupune că durata unei operatii (de adăugare de exemplu) nu depinde de operatiile anterioare;
- Costul amortizat se calculează ca medie pe o secvență de operatii succesive cu aceleasi date, iar durata unei operatii depinde de operatiile anterioare.

Capitolul 2

PROGRAMAREA STRUCTURILOR DE DATE IN C

2.1 IMPLEMENTAREA OPERATIILOR CU STRUCTURI DE DATE

Operatiile cu anumite structuri de date sunt usor de programat si de aceea pot fi rescrise în aplicatiile care le folosesc, pentru a tine seama de tipul datelor sau de alte particularități ale aplicatiei. Din această categorie fac parte vectori, matrice, stive, cozi, liste înlănțuite simple si chiar arbori binari fără reechilibrare.

Pentru alte structuri operatiile asociate pot fi destul de complexe, astfel că este preferabil să găsim o bibliotecă sau surse care pot fi adaptate rapid la specificul aplicatiei. Din această categorie fac parte arborii binari cu autoechilibrare, tabele de dispersie, liste cu acces direct ("skip list"), arbori B s.a.

Biblioteci generale de functii pentru operatii cu principalele structuri de date există numai pentru limbajele orientate pe obiecte (C++, C#, Java). Pot fi găsite însă si biblioteci C specializate cum este LEDA pentru operatii cu grafuri.

Limbajul de programare folosit în descrierea si/sau în implementarea operatiilor cu colectii de date poate influenta mult claritatea descrierii si lungimea programelor. Diferenta cea mai importantă este între limbajele procedurale (Pascal si C) si limbajele orientate pe obiecte (C++ si Java).

Multe exemple din acest text folosesc din limbajul C++ parametri de tip referință în functii (declarati cu "tip &").

Utilizarea tipului referință permite simplificarea definirii si utilizării functiilor care modifică continutul unei structuri de date, definite printr-un tip structură. In C, o functie nu poate modifica valoarea unui argument de tip structură decât dacă primește adresa variabilei ce se modifică, printr-un argument de un tip pointer. Exemplul următor foloseste o structură care reunește un vector si dimensiunea sa, iar functiile utilizează parametri de tip pointer.

```
#define M 100          // dimensiune maxima vectori
typedef struct {       // definire tip Vector
    int vec[M];
    int dim;           // dimensiune efectiva vector
} Vector;
// operatii cu vectori
void initV (Vector * pv) {      // initializare vector
    pv->dim=0;
}
void addV ( Vector * pv, int x) { // adaugare la un vector
    pv->vec[pv->dim]=x;
    pv->dim ++;
}

void printV ( Vector v) {       // afisare vector
    for (int i=0; i< v.dim;i++)
        printf ("%d ", v.vec[i]);
    printf("\n");
}

int main() {                // utilizare operatii cu vectori
    int x;  Vector v;
    initV ( &v);             // initializare vector
    while (scanf("%d",&x) != EOF)
        addV ( &v,x);        // adaugari repetate
    printV (v);               // afisare vector
}
```

Pentru o utilizare uniformă a funcțiilor și pentru eficiență am putea folosi argumente pointer și pentru funcțiile care nu modifică vectorul (de ex. "printV").

În C++ și în unele variante de C se pot folosi parametri de tip referință, care simplifică mult definirea și utilizarea de funcții cu parametri modificabili. Un parametru formal referință se declară folosind caracterul '&' între tipul și numele parametrului. În interiorul funcției parametrul referință se folosește la fel ca un parametru de același tip (transmis prin valoare). Parametrul efectiv care va înlocui un parametru formal referință poate fi orice nume de variabilă (de un tip identic sau compatibil). Exemple de funcții din programul anterior cu parametri referință:

```
void initV (Vector & v) {
    v.dim=0;
}
void addV ( Vector & v, int x) {
    v.vec[v.dim]=x; v.dim ++;
}
void main() {                // utilizare funcții cu parametri referință
    int x; Vector v;
    initV ( v);
    while (scanf("%d",&x) != EOF)
        addV ( v,x);
    printV (v);
}
```

În continuare vom folosi parametri de tip referință pentru funcțiile care trebuie să modifice valorile acestor parametri. În felul acesta utilizarea funcțiilor este uniformă, indiferent dacă ele modifică sau nu variabila colecție primită ca argument.

În cazul vectorilor sunt posibile și alte soluții care să evite funcții cu argumente modificabile (de ex. memorarea lungimii la începutul unui vector de numere), dar vom prefera soluțiile general aplicabile oricărei colecții de date.

O altă alegere trebuie făcută pentru funcțiile care au ca rezultat un element dintr-o colecție: funcția poate avea ca rezultat valoarea elementului sau poate fi de tip *void* iar valoarea să fie transmisă în afară printr-un argument de tip referință sau pointer. Pentru o funcție care furnizează elementul (de un tip T) dintr-o poziție dată a unui vector, avem de ales între următoarele variante:

```
T get ( Vector & v, int k);           // rezultat obiectul din poziția k
void get (Vector& v, int k, T & x);    // extrage din poziția k a lui v în x
int get (Vector& v, int k, T & x);    // rezultat cod de eroare
```

unde T este un tip specific aplicației, definit cu "typedef".

Alegerea între prima și ultima variantă este oarecum subiectivă și influențată de limbajul utilizat (de ex. în Java nu este posibilă decât prima variantă).

O alternativă la funcțiile cu parametri modificabili este utilizarea de variabile externe (globale) pentru colecțiile de date și scoaterea acestor colecții din lista de argumente a subprogramelor care operează cu colecția. Soluția este posibilă deseori deoarece multe aplicații folosesc o singură colecție de un anumit tip (o singură stivă, un singur graf) și ea se întâlnește în textele mai simple despre structuri de date. Astfel de funcții nu pot fi reutilizate în aplicații diferite și nu pot fi introduse în biblioteci de subprograme, dar variabilele externe simplifică programarea și fac mai eficiente funcțiile recursive (cu mai puțini parametri de pus pe stivă la fiecare apel).

Exemplu de utilizare a unui vector ca variabilă externă:

```
Vector a;                // variabila externa
void initV() {
    a.dim=0;
}
void addV (int x) {        // adaugare la vectorul a
    a.vec[a.dim++]=x;
```

```

}
// utilizare operatii cu un vector
void main() {
    int x; initV ();           // initializare vector a
    while (scanf("%d",&x) != EOF)
        addV (x);             // adauga la vectorul a
    printV ();                 // afisare vector a
}

```

Funcțiile de mai sus pot fi folosite numai într-un program care lucrează cu un singur vector, declarat ca variabilă externă cu numele "a". Dacă programul folosește mai mulți vectori, funcțiile anterioare nu mai pot fi folosite. În general se recomandă ca toate datele necesare unui subprogram și toate rezultatele să fie transmise prin argumente sau prin numele funcției.

Majoritatea subprogramelor care realizează operații cu o structură de date se pot termina anormal, fie din cauza unor argumente cu valori incorecte, fie din cauza stării colecției; de exemplu, încercarea de adăugare a unui nou element la un vector plin. În absența unui mecanism de tratare a excepțiilor program (cum sunt cele din Java și C++), soluțiile de raportare a acestei condiții de către un subprogram sunt :

- Terminarea întregului program după afisarea unui mesaj, cu sau fără utilizarea lui "assert" (pentru erori grave dar puțin probabile) . Exemplu:

```

// extragere element dintr-un vector
T get ( Vector & v, int k) {
    assert ( k >=0 && k <v.dim );      // daca eroare la indicele k
    return v.vec[k];
}

```

- Scrierea tuturor subprogramelor ca funcții de tip boolean (întreg în C), cu rezultat 1 (sau altă valoare pozitivă) pentru terminare normală și rezultat 0 sau negativ pentru terminare anormală. Exemplu:

```

// extragere element dintr-un vector
int get ( Vector & v, int k, T & x) {
    if ( k < 0 || k >=v.dim )          // daca eroare la indicele k
        return -1;
    x=v.vec[k];
    return k;
}
// utilizare
...
if ( get(v,k,x) < 0) {
    printf("indice gresit în fct. get \n"); exit(1);
}

```

2.2 UTILIZAREA DE TIPURI GENERICE

O colecție poate conține valori numerice de diferite tipuri și lungimi sau siruri de caractere sau alte tipuri agregat (structuri), sau pointeri (adrese). Se dorește ca operațiile cu un anumit tip de colecție să poată fi scrise ca funcții generale, adaptabile pentru fiecare tip de date ce poate face parte din colecție.

Limbajele orientate pe obiecte au rezolvat această problemă, fie prin utilizarea de tipuri generice, neprecizate (clase "template"), fie prin utilizarea unui tip obiect foarte general pentru elementele unei colecții, tip din care pot fi derivate orice alte tipuri de date memorate în colecție (tipul "Object" în Java).

Realizarea unei colecții generice în limbajul C se poate face în două moduri:

- 1) Prin utilizarea de tipuri generice (neprecizate) pentru elementele colecției în subprogramele ce realizează operații cu colecția. Pentru a folosi astfel de funcții ele trebuie adaptate la tipul de date

cerut de o aplicatie. Adaptarea se face partial de către compilator (prin macro-substitutie) si partial de către programator (care trebuie să dispună de forma sursă pentru aceste subprograme).

2) Prin utilizarea unor colectii de pointeri la un tip neprecizat ("void *") si a unor argumente de acest tip în subprograme, urmând ca înlocuirea cu un alt tip de pointer (la date specifice aplicatiei) să se facă la executie. Utilizarea unor astfel de functii este mai dificilă, dar utilizatorul nu trebuie să intervină în textul sursă al subprogramelor si are mai multă flexibilitate în adaptarea colectiilor la diverse tipuri de date.

Primul exemplu arată cum se definește un vector cu componente de un tip T neprecizat în functii, dar precizat în programul care folosește multimea :

```
// multimi de elemente de tipul T
#define M 1000          // dimensiune maxima vector
typedef int T ;         // tip componente multime
typedef struct {
    T v[M];             // vector cu date de tipul T
    int dim;             // dimensiune vector
} Vector;
// operatii cu un vector de obiecte
void initV (Vector & a ) {      // initializare vector
    a.dim=0;
}
void addV ( Vector & a, T x) {   // adauga pe x la vectorul a
    assert (a.n < M);           // verifica daca mai este loc in vector
    a.v [a.n++] = x;
}
int findV ( Vector a, T x) {     // cauta pe x in vectorul a
    int j;
    for ( j=0; j < a.dim;j++)
        if( x == a.v[j] )
            return j;           // gasit in pozitia j
    return -1;                  // negasit
}
```

Functiile anterioare sunt corecte numai dacă tipul T este un tip numeric pentru că operatiile de comparare la egalitate si de atribuire depind în general de tipul datelor. Operatiile de citire-scriere a datelor depind de asemenea de tipul T , dar ele fac parte în general din programul de aplicatie.

Pentru operatiile de atribuire si comparare avem două posibilități:

a) Definirea unor operatori generalizati, modificati prin macro-substitutie :

```
#define EQ(a,b) ( a==b)        // equals
#define LT(a,b) (a < b)        // less than
```

Exemplu de functie care foloseste acesti operatori:

```
int findV ( Vector a, T x) {     // cauta pe x in vectorul a
    int j;
    for ( j=0; j < a.dim;j++)
        if( EQ (x, a.v[j]) )     // comparatie la egalitate
            return j;           // gasit in pozitia j
    return -1;                  // negasit
}
```

Pentru o multime de siruri de caractere trebuie operate următoarele modificări în secvențele anterioare :

```
#define EQ(a,b) ( strcmp(a,b)==0) // equals
#define LT(a,b) (strcmp(a,b) < 0)  // less than
```

```
typedef char * T ;
```

b) Transmiterea funcțiilor de comparare, atribuire, s.a ca argumente la funcțiile care le folosesc (fără a impune anumite nume acestor funcții). Exemplu:

```
typedef char * T;
typedef int (*Fcmp) ( T a, T b) ;

int findV ( Vector a, T x, Fcmp cmp) {    // cauta pe x in vectorul a
    int j;
    for ( j=0; j < a.dim;j++)
        if ( cmp ( x, a.v[j] ) ==0 )      // comparatie la egalitate
            return j;                     // gasit in pozitia j
    return -1;                             // negasit
}
```

În cazul structurilor de date cu elemente legate prin pointeri (liste și arbori) mai există o soluție de scriere a funcțiilor care realizează operații cu acele structuri astfel ca ele să nu depindă de tipul datelor memorate: crearea nodurilor de listă sau de arbore se face în afara funcțiilor generale (în programul de aplicatie), iar funcțiile de inserție și de ștergere primesc un pointer la nodul de adăugat sau de șters și nu valoarea ce trebuie adăugată sau eliminată. Această soluție nu este adecvată structurilor folosite pentru căutarea după valoare (multimi, dicționare).

Uneori tipul datelor folosite de o aplicatie este un tip agregat (o structură C): o dată calendaristică ce grupează numere pentru zi, lună, an , descrierea unui arc dintr-un graf pentru care se memorează numerele nodurilor și costul arcului, s.a. Problema care se pune este dacă tipul T este chiar tipul structură sau este un tip pointer la acea structură. Ca și în cazul sirurilor de caractere este preferabil să se manipuleze în programe pointeri (adrese de structuri) și nu structuri. În plus, atribuirea între pointeri se face la fel ca și atribuirea între numere (cu operatorul '=').

În concluzie, tipul neprecizat T al elementelor unei colecții este de obicei fie un tip numeric, fie un tip pointer (inclusiv de tip "void *"). Avantajul principal al acestei soluții este simplitatea programelor, dar ea nu se poate aplica pentru colecții de colecții (un vector de liste, de exemplu) și nici pentru colecții neomogene.

2.3 UTILIZAREA DE POINTERI GENERICI

O a doua soluție pentru o colecție generică este o colecție de pointeri la orice tip (void *), care vor fi înlocuiți cu pointeri la datele folosite în fiecare aplicatie. Și în acest caz funcția de comparare trebuie transmisă ca argument funcțiilor de inserție sau de căutare în colecție. Exemplu de vector generic cu pointeri:

```
#define M 100                // dimensiune maxima vector
typedef void * Ptr;          // pointer la un tip neprecizat
typedef int (* fcmp) (Ptr,Ptr); // tip functie de comparare
typedef void (* fprnt) (Ptr); // tip functie de afisare

typedef struct {              // tipul vector
    Ptr v[M];                 // un vector de pointeri
    int dim;                  // nr elem in vector
} Vector;

void initV (Vector & a) {    // initializare vector
    a.dim = 0;
}

//afisare date de la adresele continute in vector
void printV ( Vector a, fprnt print ) {
    int i;
```

```

    for(i=0;i<a.dim;i++)
        print (a.v[i]);
    printf ("\n");
}
// adaugare la sfirsitul unui vector de pointeri
void addV ( Vector & a, Ptr p) {
    assert (a.dim < M);
    a.v [a.dim ++] = p;
}
// cautare in vector de pointeri
int findV ( Vector v, Ptr p, fcmp cmp) {
    int i;
    for (i=0;i<v.dim;i++)
        if ( cmp (p,v.v[i]) == 0)
            return i;                // gasit in pozitia i
    return -1;                        // negasit
}

```

Secventa următoare arată cum se poate folosi un vector de pointeri pentru a memora arce dintr-un graf cu costuri:

```

typedef struct {
    int x,y,cost;                // extremitati si cost arc
} arc;
void prntarc ( Ptr p) {         // afisare arc
    arc * ap = (arc*)p;
    printf ("%d %d %d \n",ap->x,ap->y, ap->cost);
}
int readarc (Ptr p) {           // citire arc
    arc * a =(arc*)p;
    return scanf ("%d%d%d",&a->x,&a->y,&a->cost);
}
int cmparc (Ptr p1, Ptr p2) {  // compara costuri arce
    arc * a1= (arc *)p1;
    arc * a2= (arc*)p2;
    return a1->cost - a2->cost;
}
int main () {                  // utilizare functii
    arc * ap, a; Vector v;
    initV (v); printf ("lista de arce: \n");
    while ( readarc(&a) != EOF) {
        ap = (arc*)malloc(sizeof(arc));    // aloca memorie ptr fiecare arc
        *ap=a;                             // copiaza date
        if ( findV ( v,ap, cmparc)) < 0 )   // daca nu exista deja
            addV (v,ap);                   // se adauga arc la lista de arce
    }
    printV (v, prntarc);            // afisare vector
}

```

Avantajele asupra colectiei cu date de un tip neprecizat sunt:

- Functiile pentru operatii cu colectii pot fi compilate si puse într-o bibliotecă si nu este necesar codul sursă.
- Se pot crea colectii cu elemente de tipuri diferite, pentru că în colectie se memorează adresele elementelor, iar adresele pot fi reduse la tipul comun "void*".
- Se pot crea colectii de colectii: vector de vectori, lista de liste, vector de liste etc.

Dezavantajul principal al colectiilor de pointeri (în C) este complexitatea unor aplicatii, cu erorile asociate unor operatii cu pointeri. Pentru o colectie de numere trebuie alocată memorie dinamic pentru

fiecare număr, ca să obținem câte o adresă distinctă pentru a fi memorată în colecție. Exemplu de creare a unui graf ca vector de vectori de pointeri la întregi (liste de noduri vecine pentru fiecare nod din graf):

```
void initV (Vector* & pv) {          // initializare vector
    pv=(Vector*)malloc(sizeof(Vector));
    pv->n = 0;
}
// adaugare la sfirsitul unui vector de pointeri
void addV ( Vector* & pv, Ptr p) {
    assert (pv->n < MAX);
    pv->v[pv->n ++] = p;
}
// afisare date reunite în vector de orice pointeri
void printV ( Vector* pv, Fprnt print) {
    int i;
    for(i=0;i<pv->n;i++)
        print (pv->v[i]);
    printf ("\n");
}
void main () {                      // creare si afisare graf
    Vector * graf, *vecini;
    int n,i,j; int * p;              // n=nr de noduri in graf
    initV (graf);                    // vectorul principal
    printf("n= "); scanf ("%d",&n);
    for (i=0;i<n;i++) {
        initV (vecini);              // un vector de vecini la fiecare nod
        printf("vecinii nodului %d: \n",i);
        do {
            scanf ("%d",&j);          // j este un vecin al lui i
            if (j<0) break;           // lista de vecini se termina cu un nr negativ
            p=(int*) malloc(sizeof(int)); *p=j; // ptr a obtine o adresă distincta
            addV(vecini,p);            // adauga la vector de vecini
        } while (j>=0);
        addV(graf,vecini);            // adauga vector de vecini la graf
    }
}
```

Pentru colecții ordonate (liste ordonate, arbori parțial ordonați, arbori de căutare) trebuie comparate datele memorate în colecție (nu numai la egalitate) iar comparația depinde de tipul acestor date. Soluția este de a transmite adresa funcției de comparație la funcțiile de căutare, adăugare, eliminare ș.a. Deoarece comparația este necesară în mai multe funcții este preferabil ca adresa funcției de comparație să fie transmisă la initializarea colecției și să fie memorată alături de alte variabile ce definesc colecția de date. Exemplu de operații cu un vector ordonat:

```
typedef int (* fcmp) (Ptr,Ptr); // tip funcție de comparație
typedef struct {
    Ptr *v;                      // adresa vector de pointeri alocat dinamic
    int dim;                     // dimensiune vector
    fcmp comp;                   // adresa funcției de comparație date
} Vector;
// operații cu tipul Vector
void initV (Vector & a, fcmp cmp) { // initializare
    a.n = 0;
    a.comp=cmp;                  // reține adresa funcției de comparație
}
int findV ( Vector a, Ptr p) {    // căutare în vector
```

```

int i;
for (i=0;i<a.n;i++)
    if ( a.comp(a.v[i],p)==0)
        return 1;
return 0;
}

```

Generalitatea programelor C cu structuri de date vine în conflict cu simplitatea și ușurința de înțelegere; de aceea exemplele care urmează sacrifică generalitatea în favoarea simplității, pentru că scopul lor principal este acela de a ilustra algoritmi. Din același motiv multe manuale folosesc un pseudo-cod pentru descrierea algoritmilor și nu un limbaj de programare.

Limbajul C permite o mare diversitate în exprimarea operațiilor asociate unor structuri de date, vizibilă și în programele publicate.

2.4 STRUCTURI DE DATE SI FUNCTII RECURSIVE

Un subprogram recursiv este un subprogram care se apelează pe el însuși, o dată sau de mai multe ori. Orice subprogram recursiv poate fi rescris și nerecursiv, iterativ, prin repetarea explicită a operațiilor executate la fiecare apel recursiv. O funcție recursivă realizează repetarea unor operații fără a folosi instrucțiuni de ciclare.

În anumite situații exprimarea recursivă este mai naturală și mai compactă decât forma nerecursivă. Este cazul operațiilor cu arbori binari și al altor algoritmi de tip “divide et impera” (de divizare în subprobleme).

În alte cazuri, exprimarea iterativă este mai naturală și mai eficientă ca timp și ca memorie folosită, fiind aproape exclusiv folosită: calcule de sume sau de produse, operații de căutare, operații cu liste înlănțuite, etc. În plus, funcțiile recursive cu mai mulți parametri pot fi inutilizabile pentru un număr mare de apeluri recursive, acolo unde mărimea stivei implicite (folosită de compilator) este limitată.

Cele mai simple funcții recursive corespund unor relații de recurență de forma $f(n) = r(f(n-1))$ unde n este un parametru al funcției recursive. La fiecare nou apel valoarea parametrului n se diminuează, până când n ajunge 0 (sau 1), iar valoarea $f(0)$ se calculează direct și simplu.

Un alt mod de a interpreta relația de recurență anterioară este acela că se reduce (succesiv) rezolvarea unei probleme de dimensiune n la rezolvarea unei probleme de dimensiune $n-1$, până când reducerea dimensiunii problemei nu mai este posibilă.

Funcțiile recursive au cel puțin un argument, a cărui valoare se modifică de la un apel la altul și care este verificat pentru oprirea procesului recursiv.

Orice subprogram recursiv trebuie să conțină o instrucțiune “if” (chiar la început), care să verifice condiția de oprire a procesului recursiv. În caz contrar se ajunge la un proces recursiv ce tinde la infinit și se oprește numai prin umplerea stivei.

Structurile de date liniare și arborescente se pot defini recursiv astfel:

- O listă de N elemente este formată dintr-un element și o sublistă de $N-1$ elemente;
- Un arbore binar este format dintr-un nod rădăcină și cel mult doi subarbori binari;
- Un arbore multicăi este format dintr-un nod rădăcină și mai mulți subarbori multicăi.

Aceste definiții recursive conduc la funcții recursive care reduc o anumită operație cu o listă sau cu un arbore la una sau mai multe operații cu sublista sau subarborii din componenta sa, ca în exemplele următoare pentru numărarea elementelor dintr-o listă sau dintr-un arbore:

```

// numara elementele unei liste
int count ( struct nod * list) {
    if (list==NULL)                // daca lista vida
        return 0;
    else                            // daca lista nu e vida
        return 1+ count(list->next); // un apel recursiv
}
// numara nodurile unui arbore binar

```



```

int count ( struct tnod * r) {
    if (r==NULL)                // daca arbore vid
        return 0;
    else                        // daca arbore nevid
        return 1+ count(r->left) + count(r->right);    // doua apeluri recursive
}
    
```

În cazul structurilor liniare funcțiile recursive nu aduc nici un avantaj față de variantele iterative ale acelorasi funcții, dar pentru arbori funcțiile recursive sunt mai compacte și chiar mai ușor de înțeles decât variantele iterative (mai ales atunci când este necesară o stivă pentru eliminarea recursivității). Totuși, ideea folosită în cazul structurilor liniare se aplică și în alte cazuri de funcții recursive (calcul de sume și produse, de exemplu): se reduce rezolvarea unei probleme de dimensiune N la rezolvarea unei probleme similare de dimensiune $N-1$, în mod repetat, până când se ajunge la o problemă de dimensiune 0 sau 1, care are o soluție evidentă.

Exemplele următoare arată că este important locul unde se face apelul recursiv:

```

// afisare vector in ordine inversa -recursiv
void print1 (int a[],int n) {
    if (n > 0) {
        printf ("%d ",a[n-1]);
        print1 (a,n-1);
    }
}
// afisare vector in ordine directa - recursiv
void print2 (int a[],int n) {
    if (n > 0) {
        print2 (a,n-1);
        printf ("%d ",a[n-1]);
    }
}
    
```

Ideea reducerii la două subprobleme de același tip, de la funcțiile recursive cu arbori, poate fi folosită și pentru anumite operații cu vectori sau cu liste liniare. În exemplele următoare se determină valoarea maximă dintr-un vector de întregi, cu unul și respectiv cu două apeluri recursive:

```

// maxim dintre doua numere (functie auxiliară)
int max2 (int a, int b) {
    return a>b? a:b;
}
// maxim din vector - recursiv bazat pe recurență
int maxim (int a[], int n) {
    if (n==1)
        return a[0];
    else
        return max2 (maxim (a,n-1),a[n-1]);
}
// maxim din vector - recursiv prin înjumătățire
int max (int a[], int i, int j) {
    int m;
    if ( i==j )
        return a[i];
    m= (i+j)/2;
    return max2 (max(a,i,m), max(a,m+1,j));
}
    
```

Exemple de căutare secvențială a unei valori într-un vector neordonat:

```

// cautare in vector - recursiv (ultima aparitie)
int last (int b, int a[], int n) {
    
```

```

if (n<0)
    return -1; // negasit
if (b==a[n-1])
    return n-1; // gasit
return last (b,a,n-1);
}
// cautare in vector - recursiv (prima aparitie)
int first1 (int b, int a[], int i, int j) {
    if (i>j)
        return -1;
    if (b==a[i])
        return i;
    return first1(b,a,i+1,j);
}

```

Metoda împărțirii în două subprobleme de dimensiuni apropiate (numită “divide et impera”) aplicată unor operații cu vectori necesită două argumente (indici initial și final care definesc fiecare din subvectori) și nu doar dimensiunea vectorului. Situația funcției “max” din exemplul anterior se mai întâlnește la căutarea binară într-un vector ordonat și la ordonarea unui vector prin metodele “quicksort” și “mergesort”. Diferența dintre funcția recursivă care folosește metoda “divide et impera” și funcția nerecursivă poate fi eliminată printr-o funcție auxiliară:

```

// determina maximul dintr-un vector a de n numere
int maxim1 (int a[], int n) {
    return max(a,0,n-1);
}

// cauta prima aparitie a lui b intr-un vector a de n numere
int first (int b, int a[], int n) {
    return first1(b,a,0,n-1);
}

```

Căutarea binară într-un vector ordonat împarte succesiv vectorul în două părți egale, compară valoarea căutată cu valoarea mediană, stabilește care din cei doi subvectori poate conține valoarea căutată și deci va fi împărțit în continuare. Timpul unei căutări binare într-un vector ordonat de n elemente este de ordinul $\log_2(n)$ față de $O(n)$ pentru căutarea secvențială (singura posibilă într-un vector neordonat). Exemplu de funcție recursivă pentru căutare binară:

```

// căutare binară, recursivă a lui b între a[i] și a[j]
int caut(int b, int a[], int i, int j) {
    int m;
    if (i > j)
        return -1; // b negăsit în a
    m=(i+j)/2; // m= indice median între i și j
    if (a[m]==b)
        return m; // b gasit in pozitia m
    else // daca b != a[m]
        if (b < a[m]) // daca b in prima jumatate
            return caut (b,a,i,m-1); // cauta între i și m-1
        else // daca b in a doua jumatate
            return caut (b,a,m+1,j); // cauta între m+1 și j
}

```

Varianța iterativă a căutării binare folosește un ciclu de înjumătățire repetată:

```

int caut (int b, int a[], int i, int j) {
    int este,m;

```

```

este=0;
while (i < j && este==0) {    // repeta cat timp negasit si i<j
    m=(i+j)/2;
    if (a[m]==b)
        este=1;
    else
        if (a[m] < b)
            i=m+1;
        else
            j=m-1;
}
return este ? m: -1;    // -1 daca este=0
}

```

Sortarea rapidă (“quicksort”) împarte repetat vectorul în două partitii, una cu valori mai mici si alta cu valori mai mari ca un element pivot, până când fiecare partitie se reduce la un singur element. Indicii i si j delimitează subvectorul care trebuie ordonat la un apel al functiei qsort:

```

void qsort (int a[], int i, int j) {
    int m;
    if (i < j) {
        m=pivot(a,i,j);    // determina limita m dintre partitii
        qsort(a,i,m);    // ordoneaza prima partitie
        qsort (a,m+1,j);    // ordoneaza a doua partitie
    }
}

```

Indicele m este pozitia elementului pivot, astfel ca $a[i] < a[m]$ pentru orice $i < m$ si $a[i] > a[m]$ pentru orice $i > m$. De observat că nu se compară elemente vecine din vector (ca în alte metode), ci se compară un element $a[p]$ din prima partitie cu un element $a[q]$ din a doua partitie si deci se aduc mai repede valorile mici la începutul vectorului si valorile mari la sfârșitul vectorului.

```

int pivot (int a[], int p, int q) {
    int x,t;
    x=a[(p+q)/2];    // x = element pivot
    while ( p < q) {
        while (a[q]> x) q--;
        while (a[p] < x) p++;
        if (p<q) {
            t=a[p]; a[p]=a[q]; a[q]=t;
        }
    }
    return p;    // sau return q;
}

```

Eliminarea recursivității din algoritmul quicksort nu mai este la fel de simplă ca eliminarea recursivității din algoritmul de căutare binară, deoarece sunt două apeluri recursive succesive.

În general, metoda de eliminare a recursivității depinde de numărul si de pozitia apelurilor recursive astfel:

- O functie recursivă cu un singur apel ca ultimă instructiune se poate rescrie simplu iterativ prin înlocuirea instructiunii “if” cu o instructiune “while” (de observat că metoda de căutare binară are un singur apel recursiv desi sunt scrise două instructiuni; la executie se alege doar una din ele);
- O functie recursivă cu un apel care nu este ultima instructiune sau cu două apeluri se poate rescrie nerecursiv folosind o stivă pentru memorarea argumentelor si variabilelor locale.
- Anumite functii recursive cu două apeluri, care generează apeluri repetate cu aceiasi parametri, se pot rescrie nerecursiv folosind o matrice (sau un vector) cu rezultate ale apelurilor anterioare, prin metoda programării dinamice.

Orice compilator de funcții recursive folosește o stivă pentru argumente formale, variabile locale și adrese de revenire din fiecare apel. În cazul unui mare număr de argumente și de apeluri se poate ajunge ca stiva folosită implicit să depășească capacitatea rezervată (funcție de memoria RAM disponibilă) și deci ca probleme de dimensiune mare să nu poată fi rezolvate recursiv. Acesta este unul din motivele eliminării recursivității, iar al doilea motiv este timpul mare de execuție necesar unor probleme (cum ar fi cele care pot fi rezolvate prin programare dinamică).

Eliminarea recursivității din funcția “qsort” se poate face în două etape:

- Se reduc cele două apeluri la un singur apel recursiv;
- Se folosește o stivă pentru a elimina apelul recursiv neterminat.

```
// qsort cu un apel recursiv
void qsort (int a[], int i, int j) {
    int m;
    while (i < j) {        // se ordonează alternativ fiecare partiție
        m=pivot(a, i, j); // indice element pivot
        qsort(a, i, m);   // ordonare partiție
        i=m+1; m=j;       // modifica parametri de apel
    }
}
```

Cea mai simplă structură de stivă este un vector cu adăugare și extragere numai de la sfârșit (vârful stivei este ultimul element din vector). În stivă se vor pune argumentele funcției care se modifică de la un apel la altul:

```
void qsort (int a[], int i, int j) {
    int m; int st[500], sp;
    sp=0;
    st[sp++]=i; st[sp++]=j; // pune i și j pe stiva
    while (sp>=0) {
        if (i < j) {
            m=pivot(a, i, j);
            st[sp++]=i; st[sp++]=m; // pune argumente pe stiva
            i=m+1; m=j;             // modifica argumente de apel
        }
        else {                      // refacere argumente pentru revenire
            j=st[--sp]; i=st[--sp];
        }
    }
}
```

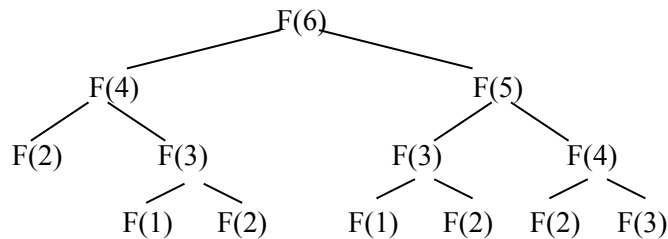
O funcție cu două apeluri recursive generează un arbore binar de apeluri. Un exemplu este calculul numărului n din sirul Fibonacci $F(n)$ pe baza relației de recurență:

$$F(n) = F(n-2) + F(n-1) \quad \text{și primele 2 numere din sir } F(0)=F(1)=1;$$

Relația de recurență poate fi transpusă imediat într-o funcție recursivă:

```
int F(int n) {
    if (n < 2) return 1;
    return F(n-2)+F(n-1);
}
```

Utilizarea acestei funcții este foarte ineficientă, iar timpul de calcul crește exponențial în raport cu n . Explicația este aceea că se repetă rezolvarea unor subprobleme, iar numărul de apeluri al funcției crește rapid pe măsură ce n crește. Arborele de apeluri pentru $F(6)$ va fi:



Deși n este mic și nu am mai figurat unele apeluri (cu argumente 1 și 0), se observă cum se repetă apeluri ale funcției recursive pentru a recalcula aceleași valori în mod repetat.

Ideea programării dinamice este de a înlocui funcția recursivă cu o funcție care să completeze vectorul cu rezultate ale subproblemelor mai mici (în acest caz, numere din sirul Fibonacci):

```

int F(int n) {
    int k, f[100]={0}; // un vector ( n < 100) initializat cu zerouri
    f[0]=f[1]=1; // initializari in vector
    for ( k=2;k<=n;k++)
        f[k]= f[k-2]+f[k-1];
    return f[n];
}
  
```

Un alt exemplu de trecere de la o funcție recursivă la completarea unui tabel este problema calculului combinațiilor de n numere luate câte k :

$$C(n,k) = C(n-1,k) + C(n-1,k-1) \quad \text{pentru } 0 < k < n$$

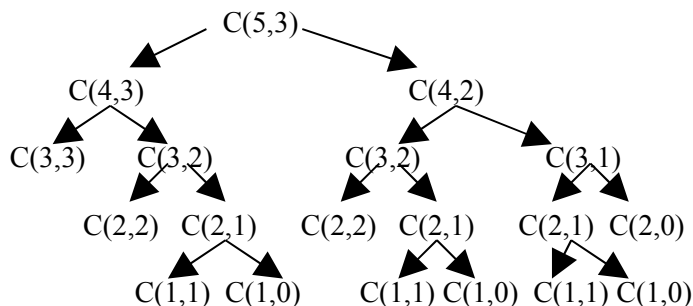
$$C(n,k) = 1 \quad \text{pentru } k=0 \text{ sau } k=n$$

Această relație de recurență exprimă descompunerea problemei $C(n,k)$ în două subprobleme mai mici (cu valori mai mici pentru n și k). Traducem direct relația într-o funcție recursivă:

```

long comb (int n, int k) {
    if (k==0 || k==n)
        return 1L;
    else
        return comb (n-1,k) + comb(n-1,k-1);
}
  
```

Dezavantajul acestei abordări rezultă din numărul mare de apeluri recursive, dintre care o parte nu fac decât să recalculeze aceleași valori (funcția "comb" se apelează de mai multe ori cu aceleași valori pentru parametri n și k). Arborele acestor apeluri pentru $n=5$ și $k=3$ este :



Dintre cele 19 apeluri numai 11 sunt diferite.

Metoda programării dinamice construiește o matrice $c[i][j]$ a cărei completare începe cu prima coloană $c[i][0]=1$, continuă cu coloana a doua $c[i][1]$, coloana a treia s.a.m.d. Elementele matricei se

calculează unele din altele folosind tot relația de recurență anterioară. Exemplu de funcție care completează această matrice :

```
long c[30][30];           // matricea este variabila externa
void pdcomb (int n) {
    int i,j;
    for (i=0;i<=n;i++)
        c[i][0]=1;        // coloana 0
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j)
                c[i][j]=1;    // diagonala principala
            else
                if (i < j)
                    c[i][j]=0;    // deasupra diagonalei
                else
                    c[i][j]=c[i-1][j]+c[i-1][j-1]; // sub diagonala
}
```

Urmează două probleme clasice, ale căror soluții sunt cunoscute aproape exclusiv sub forma recursivă, deoarece variantele nerecursive sunt mai lungi și mai greu de înțeles: afisarea numelor fișierelor dintr-un director dat și din subdirectoarele sale și problema turnurilor din Hanoi.

Structura de directoare și subdirectoare este o structură arborescentă și astfel se explică natura recursivă a operațiilor cu astfel de structuri. Pentru obținerea numelor fișierelor dintr-un director se folosesc funcțiile de bibliotecă “findfirst” și “findnext” (parte a unui obiect iterator). La fiecare nume de fișier se verifică dacă este la rândul lui un subdirector, pentru examinarea conținutului său. Procesul se repetă până când nu mai există fișiere director, ci numai fișiere normale.

Funcția care urmează mai include unele detalii, cum ar fi:

- afisare cu indentare diferită la fiecare nivel de adâncime (argumentul “sp”);
- evitarea unei recursivități infinite pentru numele de directoare “.” și “..”;
- construirea sirului de forma “cale/*.*” cerut de funcția “_findfirst”

```
void fileList ( char * path, int sp) { // listare fișiere identificate prin “path”
    struct _finddata_t fb;           // structura predefinită folosită de findfirst (atribute fișier)
    int done=0; int i;               // done devine 1 când nu mai sunt fișiere
    char tmp[256];                   // ptr creare cale la subdirectoarele unui director
    long first;                      // transmis de la findfirst la findnext
    first = _findfirst(path,&fb);     // caută primul dintre fișiere și pune atribute în fb
    while (done==0) {                // repetă cât mai sunt fișiere pe calea “path”
        if (fb.name[0] != '.')       // dacă numele de director nu începe cu ‘.’
            printf ("%*c %-12s \n",sp,' ', fb.name); // afisare nume fișier
            // dacă subdirector cu nume diferit de “.” și “..”
        if ( fb.attrib == _A_SUBDIR && fb.name[0] != '.') {
            i= strchr(path,'/')-path; // extrage nume director
            strncpy(tmp,path,i+1);    // copiaza calea în tmp
            tmp[i+1]=0;               // ca sir terminat cu zero
            strcat(tmp,fb.name);      // adaugă la cale nume subdirector și /*.*
            fileList (tmp,sp+3);       // listează conținut subdirector, decalat cu 3
        }
        done=_findnext (first,&fb);  // pune numele următorului fișier în “fb”
    }
}
```

Problema turnurilor din Hanoi este un joc cu 3 tije verticale și mai multe discuri de diametre diferite. Inițial discurile sunt stivuite pe prima tijă astfel că fiecare disc stă pe un disc mai mare. Problema cere ca toate discurile să ajungă pe ultima tijă, ca în configurația inițială, folosind pentru mutări și tija din mijloc. Mutările de discuri trebuie să satisfacă următoarele condiții:

- Se mută numai un singur disc de pe o tijă pe alta
- Se poate muta numai discul de deasupra si numai peste discurile existente in tija destinatie
- Un disc nu poate fi asezat peste un disc cu diametru mai mic

Se cere secventa de mutări care respectă aceste conditii, afisată în mod text sau cu efect asupra imaginii celor 3 tije (în mod grafic).

Functia următoare afisează discul mutat, originea si destinatia unei mutări:

```
// muta discul numarul k de pe tija s pe tija d
void mutadisc (int k, int s ,int d ) {
    printf (" muta discul %d de pe %d pe %d \n",k,s,d);
}
```

Functia recursivă care urmează rezolvă problema pentru n discuri si 3 tije:

```
// muta n discuri de pe a pe b folosind si t
void muta ( int n, int a, int b, int t) {
    if (n==1)                // daca a ramas un singur disc
        mutadisc (1,a,b);    // se muta direct de pe a pe b
    else {                   // daca sunt mai multe discuri pe a
        muta (n-1,a,t,b);    // muta n-1 discuri de pe a pe t
        mutadisc (n,a,b);    // muta discul n (de diametru maxim) de pe a pe b
        muta (n-1,t,b,a);    // muta n-1 discuri de pe t pe b
    }
}
```

Solutiile nerecursive pornesc de la analiza problemei si observarea unor proprietăti ale stărilor prin care se trece; de exemplu, s-a arătat că se poate repeta de (2^n-1) ori functia “mutadisc” recalculând la fiecare pas numărul tije sursă si al tije destinatie (numărul discului nu contează deoarece nu se poate muta decât discul din vârful stivei de pe tija sursă).

Capitolul 3

VECTORI

3.1 VECTORI

Structura de vector ("array") este foarte folosită datorită avantajelor sale:

- Nu trebuie memorate decât datele necesare aplicației (nu și adrese de legătură);
- Este posibil accesul direct (aleator) la orice element dintr-un vector prin indici;
- Programarea operațiilor cu vectori este foarte simplă.
- Căutarea într-un vector ordonat este foarte eficientă, prin căutare binară.

Dezavantajul unui vector cu dimensiune constantă rezultă din necesitatea unei estimări a dimensiunii sale la scrierea programului. Pentru un vector alocat și realocat dinamic poate apărea o fragmentare a memoriei dinamice rezultate din realocări repetate pentru extinderea vectorului. De asemenea, eliminarea de elemente dintr-un vector compact poate necesita deplasarea elementelor din vector.

Prin vectori se reprezintă și anumite cazuri particulare de liste înlănțuite sau de arbori pentru reducerea memoriei ocupate și timpului de prelucrare.

Ca tipuri de vectori putem menționa:

- Vectori cu dimensiune fixă (constantă);
- Vectori extensibili (realocabili dinamic);
- Vectori de biti (la care un element ocupă un bit);
- Vectori "heap" (care reprezintă compact un arbore binar particular);
- Vectori ca tabele de dispersie.

De obicei un vector este completat în ordinea crescătoare a indicilor, fie prin adăugare la sfârșit a noilor elemente, fie prin inserție între alte elemente existente, pentru a menține ordinea în vector.

Există și excepții de la cazul uzual al vectorilor cu elemente consecutive : vectori cu interval ("buffer gap") și tabele de dispersie ("hash tables").

Un "buffer gap" este folosit în procesoarele de texte; textul din memorie este împărțit în două siruri păstrate într-un vector ("buffer" cu text) dar separate între ele printr-un interval plasat în poziția curentă de editare a textului. În felul acesta se evită mutarea unor siruri lungi de caractere în memorie la modificarea textului; inserția de noi caractere în poziția curentă mărește secvența de la începutul vectorului și reduce intervalul, iar stergerea de caractere din poziția curentă mărește intervalul dintre caracterele aflate înainte și respectiv după poziția curentă.

Mutarea cursorului necesită mutarea unor caractere dintr-un sir în celălalt, dar numai ca urmare a unei operații de modificare în noua poziție a cursorului.

Caracterele sterse sau inserate sunt de fapt memorate într-un alt vector, pentru a se putea reconstitui un text modificat din greșeală (operația "undo" de anulare a unor operații și de revenire la o stare anterioară).

Vectorii cu dimensiune constantă, fixată la scrierea programului, se folosesc în unele situații particulare când limita colecției este cunoscută și relativ mică sau când se dorește simplificarea programelor, pentru a facilita înțelegerea lor. Alte situații pot fi cea a unui vector de constante sau de cuvinte cheie, cu număr cunoscut de valori.

Vectori cu dimensiune fixă se folosesc și ca zone tampon la citirea sau scrierea în fișiere text sau în alte fluxuri de date.

Vectorul folosit într-un tabel de dispersie are o dimensiune constantă (preferabil, un număr prim) din cauza modului în care este folosit (se va vedea ulterior).

Un fișier binar cu articole de lungime fixă poate fi privit ca un vector, deoarece are aceleași avantaje și dezavantaje, iar operațiile sunt similare: adăugare la sfârșit de fișier, căutare secvențială în fișier, acces direct la un articol prin indice (poziție relativă în fișier), sortare fișier atunci când este nevoie, s.a. La fel ca într-un vector, operațiile de inserție și de stergere de articole consumă timp și trebuie evitate sau amânate pe cât posibil.

3.2 VECTORI ORDONATI

Un vector ordonat reduce timpul anumitor operatii, cum ar fi: căutarea unei valori date, verificarea unicității elementelor, găsirea perechii celei mai apropiate, calculul frecvenței de apariție a fiecărei valori distincte s.a.

Un vector ordonat poate fi folosit și drept coadă cu priorități, dacă nu se mai fac adăugări de elemente la coadă, pentru că valoarea minimă (sau maximă) se află la una din extremitățile vectorului, de unde se poate scoate fără alte operații auxiliare.

Mentinerea unui vector în ordine după fiecare operație de adăugare sau de ștergere nu este eficientă și nici nu este necesară de multe ori; atunci când avem nevoie de o colecție dinamică permanent ordonată vom folosi un arbore binar sau o listă înlăntuită ordonată. Ordonarea vectorilor se face atunci când este necesar, de exemplu pentru afisarea elementelor sortate după o anumită cheie.

Pe de altă parte, operația de sortare este eficientă numai pe vectori; nu se sortează liste înlăntuite sau arbori neordonati sau tabele de dispersie.

Sunt cunoscuți mai mulți algoritmi de sortare, care diferă atât prin modul de lucru cât și prin performanțele lor. Cei mai simpli și ineficienți algoritmi de sortare au o complexitate de ordinul $O(n^2)$, iar cei mai buni algoritmi de sortare necesită pentru cazul mediu un timp de ordinul $O(n \log_2 n)$, unde “n” este dimensiunea vectorului.

Uneori ne interesează un algoritm de sortare “stabilă”, care păstrează ordinea inițială a valorilor egale din vectorul sortat. Mai mulți algoritmi nu sunt “stabili”.

De obicei ne interesează algoritmi de sortare “pe loc”, care nu necesită memorie suplimentară, deși există câțiva algoritmi foarte buni care nu sunt de acest tip: sortare prin interclasare și sortare prin distribuție pe compartimente.

Algoritmi de sortare “pe loc” a unui vector se bazează pe compararea de elemente din vector, urmată eventual de schimbarea între ele a elementelor comparate pentru a respecta condiția ca orice element să fie mai mare ca cele precedente și mai mic ca cele care-i urmează.

Vom nota cu T tipul elementelor din vector, tip care suportă comparația prin operatori ai limbajului (deci un tip numeric). În cazul altor tipuri (structuri, siruri) se vor înlocui operatorii de comparație (și de atribuire) cu funcții pentru aceste operații.

Vom defini mai întâi o funcție care schimbă între ele elementele din două poziții date ale unui vector:

```
void swap (T a[ ], int i, int j) {           // interschimb a[i] cu a[j]
    T b=a[i]; a[i]=a[j]; a[j]=b;
}
```

Vom prezenta aici câțiva algoritmi ușor de programat, chiar dacă nu au cele mai bune performanțe.

Sortarea prin metoda bulelor (“Bubble Sort”) compară mereu elemente vecine; după ce se compară toate perechile vecine (de la prima către ultima) se coboară valoarea maximă la sfârșitul vectorului. La următoarele parcurgeri se reduce treptat dimensiunea vectorului, prin eliminarea valorilor finale (deja sortate). Dacă se compară perechile de elemente vecine de la ultima către prima, atunci se aduce în prima poziție valoarea minimă, și apoi se modifică indicele de început. Una din variantele posibile de implementare a acestei metode este funcția următoare:

```
void bubbleSort(T a[ ], int n) { // sortare prin metoda bulelor
    int i, k;
    for (i = 0; i < n; i++) { // i este indicele primului element comparat
        for (k = n-1; k > i; k--) // comparatie incepe cu ultima pereche (n-1,n-2)
            if (a[k-1] > a[k]) // daca nu se respecta ordinea crescatoare
                swap(a,k,k-1); // schimba intre ele elemente vecine
    }
}
```

Timpul de sortare prin metoda bulelor este proportional cu pătratul dimensiunii vectorului (complexitatea algoritmului este de ordinul n^2).

Sortarea prin selectie determină în mod repetat elementul minim dintre toate care urmează unui element $a[i]$ și îl aduce în poziția i , după care crește pe i .

```
void selSort( T a[ ], int n) {    // sortare prin selectie
    int i, j, m;                // m = indice element minim dintre i,i+1,...n
    for (i = 0; i < n-1; i++) {  // in poz. i se aduce min (a[i+1],...[a[n]])
        m = i;                  // considera ca minim este a[i]
        for (j = i+1; j < n; j++) // compara minim partial cu a[j] (j > i)
            if ( a[j] < a[m] )    // a[m] este elementul minim
                m = j;
        swap(a,i,m);            // se aduce minim din poz. m in pozitia i
    }
}
```

Sortarea prin selectie are și ea complexitatea $O(n^2)$, dar în medie este mai rapidă decât sortarea prin metoda bulelor (constanta care înmulțește pe n^2 este mai mică).

Sortarea prin insertie consideră vectorul format dintr-o partiție sortată (la început de exemplu) și o partiție nesortată; la fiecare pas se alege un element din partiția nesortată și se inserează în locul corespunzător din partiția sortată, după deplasarea în jos a unor elemente pentru a crea loc de insertie.

```
void insSort (T a[ ], int n) {
    int i,j; T x;
    for (i=1;i<n;i++) {        // partiția nesortată este între pozițiile i și n
        x=a[i];                // x este un element
        j=i-1;                 // caută poziția j unde trebuie inserat x
        while (x<a[j] && j >=0) {
            a[j+1]=a[j];       // deplasare în jos din poziția j
            j--;
        }
        a[j+1]=x;              // muta pe x în poziția j+1
    }
}
```

Nici sortarea prin insertie nu este mai bună de $O(n^2)$ pentru cazul mediu și cel mai nefavorabil, dar poate fi îmbunătățită prin modificarea distanței dintre elementele comparate. Metoda cu increment variabil (ShellSort) se bazează pe ideea (folosită și în sortarea rapidă QuickSort) că sunt preferabile schimbări între elemente aflate la distanță mai mare în loc de schimbări între elemente vecine; în felul acesta valori mari aflate inițial la începutul vectorului ajung mai repede în pozițiile finale, de la sfârșitul vectorului.

Algoritmul lui Shell are în cazul mediu complexitatea de ordinul $n^{1.25}$ și în cazul cel mai rău $O(n^{1.5})$, față de $O(n^2)$ pentru sortare prin insertie cu pas 1.

În funcția următoare se folosesc rezultatele unor studii pentru determinarea valorii initiale a pasului h , care scade apoi prin împărțire succesivă la 3. De exemplu, pentru $n > 100$ pașii folosiți vor fi 13,4 și 1.

```
void shellSort(T a[ ], int n) {
    int h, i, j; T t;
    // calcul increment maxim
    h = 1;
    if (n < 14) h = 1;
    else if ( n > 29524) h = 3280;
    else {
        while (h < n) h = 3*h + 1;
        h /= 3; h /= 3;
    }
}
```

```
// sortare prin insertie cu increment h variabil
while (h > 0) {
    for (i = h; i < n; i++) {
        t = a[i];
        for (j = i-h; j >= 0 && a[j] > t; j -= h)
            a[j+h] = a[j];
        a[j+h] = t;
    }
    h /= 3;    // urmatorul increment
}
}
```

3.3 VECTORI ALOCATI DINAMIC

Putem distinge două situații de alocare dinamică pentru vectori:

- Dimensiunea vectorului este cunoscută de program înainte a valorilor ce trebuie memorate în vector și nu se mai modifică pe durata execuției programului; în acest caz este suficientă o alocare inițială de memorie pentru vector (“malloc”).
- Dimensiunea vectorului nu este cunoscută de la început sau numărul de elemente poate crește pe măsură ce programul evoluează; în acest caz este necesară extinderea dinamică a tabloului (se apelează repetat “realloc”).

În limbajul C utilizarea unui vector alocat dinamic este similară utilizării unui vector cu dimensiune constantă, cu diferența că ultimul nu poate fi realocat dinamic. Funcția “realloc” simplifică extinderea (realocarea) unui vector dinamic cu păstrarea datelor memorate. Exemplu de ordonare a unui vector de numere folosind un vector alocat dinamic.

```
// comparatie de întregi - pentru qsort
int intcmp (const void * p1, const void * p2) {
    return *(int*)p1 - *(int*)p2;
}

// citire - sortare - afisare
void main () {
    int * vec, n, i;                // vec = adresa vector
    // citire vector
    printf ("dimens. vector= "); scanf ("%d", &n);
    vec = (int*) malloc (n*sizeof(int));
    for (i=0; i<n; i++)
        scanf ("%d", &vec[i]);
    qsort (vec, n, sizeof(int), intcmp); // ordonare vector
    for (i=0; i<n; i++)               // afisare vector
        printf ("%4d", vec[i]);
    free (vec);                       // poate lipsi
}
```

În aplicațiile care prelucrează cuvintele distincte dintr-un text, numărul acestor cuvinte nu este cunoscut și nu poate fi estimat, dar putem folosi un vector realocat dinamic care se extinde atunci când este necesar. Exemplu:

```
// cauta cuvant in vector
int find ( char ** tab, int n, char * p) {
    int i;
    for (i=0; i<n; i++)
        if ( strcmp (p, tab[i]) ==0)
            return i;
    return -1;                       // negasit
}
```

```

#define INC 100
void main () {
    char cuv[80], * pc;
    char ** tab;           // tabel de pointeri la cuvinte
    int i, n, nmax=INC;     // nc= numar de cuvinte in lista
    n=0;
    tab = (char**)malloc(nmax*sizeof(char*)); // alocare initiala ptr vector
    while (scanf ("%s",cuv) > 0) {           // citeste un cuvânt
        pc =strdup(cuv);                     // aloca memorie ptr cuvânt
        if (find (tab,n,pc) < 0) {           // daca nu exista deja
            if (n ==nmax) {                  // daca vector plin
                nmax = nmax+INC;             // mareste capacitate vector
                tab =(char**)realloc(tab,nmax*sizeof(char*)); // realocare
            }
            tab[n++]=pc;                     // adauga la vector adresa cuvânt
        }
    }
}

```

Functia "realloc" primeste ca argumente adresa vectorului ce trebuie extins si noua sa dimensiune si are ca rezultat o altă adresă pentru vector, unde s-au copiat automat si datele de la vechea adresă. Această functie este apelată atunci când se cere adăugarea de noi elemente la un vector plin (în care nu mai există pozitii libere).

Utilizarea functiei "realloc" necesită memorarea următoarelor informatii despre vectorul ce va fi extins: adresă vector, dimensiunea alocată (maximă) si dimensiunea efectivă. Când dimensiunea efectivă ajunge egală cu dimensiunea maximă, atunci devine necesară extinderea vectorului. Extinderea se poate face cu o valoare constantă sau prin dublarea dimensiunii curente sau după altă metodă.

Exemplul următor arată cum se pot încapsula în câteva functii operatiile cu un vector alocat si apoi extins dinamic, fără ca alocarea si realocarea să fie vizibile pentru programul care foloseste aceste subprograme.

```

#define INC 100           // increment de exindere vector
typedef int T;           // tip componente vector
typedef struct {
    T * vec;              // adresa vector (alocat dinamic)
    int dim, max;         // dimensiune efectiva si maxima
} Vector;
// initializare vector v
void initV (Vector & v) {
    v.vec= (T *) malloc (INC*sizeof(T));
    v.max=INC; v.dim=0;
}
// adaugare obiect x la vectorul v
void addV ( Vector & v, T x) {
    if (v.dim == v.max) {
        v.max += INC;     // extindere vector cu o valoare fixa
        v.vec=(T*) realloc (v.vec, (v.max)*sizeof(T));
    }
    v.vec[v.dim]=x; v.dim ++;
}

```

Exemplu de program care generează si afisează un vector de numere:

```

void main() {
    T x; Vector v;
    initV ( v);

```

```
while (scanf("%d",&x) == 1)
    addV ( v,x);
printV (v);
}
```

Timpul necesar pentru căutarea într-un vector neordonat este de ordinul $O(n)$, deci proportional cu dimensiunea vectorului. Într-un vector ordonat timpul de căutare este de ordinul $O(\lg n)$. Adăugarea la sfârșitul unui vector este imediată (are ordinul $O(1)$) iar eliminarea dintr-un vector compact necesită mutarea în medie a $n/2$ elemente, deci este de ordinul $O(n)$.

3.4 APLICATIE : COMPONENTE CONEXE

Aplicatia poate fi formulatã cel puțin în două moduri si a condus la aparitia unui tip abstract de date, numit colectie de multimi disjuncte (“Disjoint Sets”).

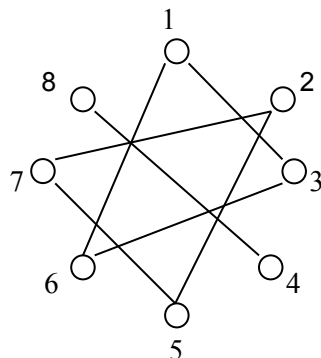
Fiind dată o multime de valori (de orice tip) si o serie de relatii de echivalență între perechi de valori din multime, se cere să se afișeze clasele de echivalență formate cu ele. Dacă sunt n valori, atunci numărul claselor de echivalență poate fi între 1 si n , inclusiv.

Exemplu de date initiale (relatii de echivalentã):

30 ~ 60 / 50 ~ 70 / 10 ~ 30 / 20 ~ 50 / 40 ~ 80 / 10 ~ 60 /

Rezultatul (clasele de echivalenta) : $\{10,30,60\}$, $\{20,50,70\}$, $\{40,80\}$

O altă formulare a problemei cere afisarea componentelor conexe dintr-un graf neorientat definit printr-o listă de muchii. Fiecare muchie corespunde unei relatii de echivalență între vârfurile unite de muchie, iar o componentă conexă este un subgraf (o clasă de noduri) în care există o cale între oricare două vârfuri. Exemplu:



In cazul particular al componentelor conexe dintr-un graf, este suficient un singur vector “cls”, unde cls[k] este componenta în care se află vârful k.

În cazul mai general al claselor de echivalență ce pot conține elemente de orice tip (numere oarecare sau siruri ce reprezintă nume), mai este necesar și un vector cu valorile elementelor. Pentru exemplul anterior cei doi vectori pot arăta în final astfel (numerele de clase pot fi diferite):

val	10	20	30	40	50	60	70	80
cls	1	2	1	3	2	1	2	3

Vectorii val, cls si dimensiunea lor se reunesc într-un tip de date numit “colectie de multimi disjuncte”, pentru că fiecare clasă de echivalență este o multime, iar aceste multimi sunt disjuncte între ele.

```
typedef struct {
    int val[40], cls[40];           // vector de valori si de clase
    int n;                         // dimensiune vectori
} ds;
```

Pentru memorarea unor date agregate într-un vector avem două posibilități:

- Mai multi vectori paraleli, cu aceeași dimensiune; câte un vector pentru fiecare câmp din structură (ca în exemplul anterior).
- Un singur vector de structuri:

```
typedef struct {           // o pereche valoare-clasă
    int val; int cls;
} entry;
typedef struct {
    entry a [40];          // vector de perechi valoare-clasă
    int n;                 // dimensiune vector
} ds;
```

S-au stabilit următoarele operații specifice tipului abstract “Disjoint Sets”:

- Initializare colectie (initDS)
- Găsirea multimii (clasei) care conține o valoare dată (findDS)
- Reunire multimi (clase) ce conțin două valori date (unifDS)
- Afisare colectie de multimi (printDS)

La citirea unei perechi de valori (unei relații de echivalență) se stabilește pentru cele două valori echivalente același număr de clasă, egal cu cel mai mic dintre cele două (pentru a menține ordinea în fiecare clasă).

Dacă valorile sunt chiar numerele 1,2,3...8 atunci evoluția vectorului de clase după fiecare pereche de valori citită va fi

	clase							
initial	1	2	3	4	5	6	7	8
dupa 3-6	1	2	3	4	5	3	7	8
dupa 5-7	1	2	3	4	5	3	5	8
dupa 1-3	1	2	1	4	5	1	5	8
dupa 2-5	1	2	1	4	2	1	2	8
dupa 4-8	1	2	1	4	2	1	2	4
dupa 1-6	1	2	1	4	2	1	2	4

(nu se mai modifică nimic)

Urmează un exemplu de implementare cu un singur vector a tipului “Colectie de multimi disjuncte” și utilizarea sa în problema afișării componentelor conexe.

```
typedef struct {
    int cls[40];           // vector cu numere de multimi
    int n;                 // dimensiune vector
} ds;
// determina multimea in care se afla x
int find ( ds c, int x) {
    return c.cls[x];
}
// reunire multimi ce contin valorile x si y
void unif ( ds & c,int x,int y) {
    int i,cy;
    cy=c.cls[y];
    for (i=1;i<=c.n;i++)    // inlocuieste clasa lui y cu clasa lui x
        if (c.cls[i]==cy)    // daca i era in clasa lui y
            c.cls[i]=c.cls[x]; // atunci i trece in clasa lui x
}
// afisare multimi din colectie
void printDS (ds c) {
    int i,j,m;
    for (i=1;i<=c.n;i++) {    // ptr fiecare multime posibila i
        m=0;                  // numar de valori in multimea i
        for (j=1;j<=c.n;j++)    // cauta valorile din multimea i
```

```

        if (c.cls[j]==i) {
            printf("%d ",j);
            m++;
        }
        if (m)                // daca exista valori in multimea i
            printf("\n");      // se trece pe alta linie
    }
}
// initializare multimi din colectie
void initDS (ds & c, int n) {
    int i;
    c.n=n;
    for (i=1;i<=n;i++)
        c.cls[i]=i;
}
// afisare componente conexe
void main () {
    ds c;                        // o colectie de componente conexe
    int x,y,n;
    printf ("nr. de elemente: "); scanf ("%i",&n);
    initDS(c,n);                // initializare colectie c
    while (scanf("%d%d",&x,&y) > 0) // citeste muchii x-y
        unif(c,x,y);           // reuneste componentele lui x si y
    printDS(c);                 // afisare componente conexe
}

```

În această implementare operația “find” are un timp constant $O(1)$, dar operația de reuniune este de ordinul $O(n)$. Vom arăta ulterior (la discuția despre mulțimi) o altă implementare, mai performantă, dar tot prin vectori a colecției de mulțimi disjuncte.

3.5 VECTORI MULTIDIMENSIONALI (MATRICE)

O matrice bidimensională poate fi memorată în câteva moduri:

- Ca un vector de vectori. Exemplu :
`char a[20][20];` // `a[i]` este un vector
- Ca un vector de pointeri la vectori. Exemplu:
`char* a[20];` // sau `char ** a;`
- Ca un singur vector ce conține elementele matricei, fie în ordinea liniilor, fie în ordinea coloanelor.

Matricele alocate dinamic sunt vectori de pointeri la liniile matricei.

Pentru comparație vom folosi o funcție care ordonează un vector de nume (de siruri) și funcții de citire și afisare a numelor memorate și ordonate.

Prima formă (vector de vectori) este cea clasică, posibilă în toate limbajele de programare, și are avantajul simplității și clarității operațiilor de prelucrare.

De remarcat că numărul de coloane al matricei transmise ca argument trebuie să fie o constantă, aceeași pentru toate funcțiile care lucrează cu matricea.

```

#define M 30                // nr maxim de caractere intr-un sir
// ordonare siruri
void sort ( char vs[][M], int n) {
    int i,j; char tmp[M];
    for (j=1;j<n;j++)
        for (i=0;i<n-1;i++)
            if ( strcmp (vs[i],vs[i+1])>0) {
                strcpy(tmp,vs[i]);      // interschimb siruri (linii din matrice)
                strcpy(vs[i],vs[i+1]);
                strcpy(vs[i+1],tmp);
            }
}

```

```

    }
}
// citire siruri in matrice
int citmat ( char vs[][M] ) {
    int i=0;
    printf ("lista de siruri: \n");
    while ( scanf ("%s", vs[i])!=1 )
        i++;
    return i;          // numar de siruri citite
}
// afisare matrice cu siruri
void scrmat (char vs[][M],int n) {
    int i;
    for (i=0;i<n;i++)
        printf ("%s \n", vs[i]); /* afisare siruri */
}

```

O matrice alocată dinamic este de fapt un vector alocat dinamic ce contine pointeri la vectori alocati dinamic (liniile matricei). Liniile matricei pot avea toate aceeasi lungime sau pot avea lungimi diferite. Exemplu cu linii de lungimi diferite :

```

// ordonare vector de pointeri la siruri
void sort ( char * vp[],int n) {
    int i,j; char * tmp;
    for (j=1;j<n;j++)
        for (i=0;i<n-1;i++)
            if ( strcmp (vp[i],vp[i+1])>0) {
                tmp=vp[i]; vp[i]=vp[i+1];
                vp[i+1]=tmp;
            }
}

```

În exemplul anterior am presupus că vectorul de pointeri are o dimensiune fixă și este alocat în funcția “main”.

Dacă se cunoaște de la început numărul de linii și de coloane, atunci putem folosi o funcție care alocă dinamic memorie pentru matrice. Exemplu:

```

// alocare memorie pentru o matrice de intregi
// rezultat adresa matrice sau NULL
int ** intmat ( int nl, int nc) {          // nl linii si nc coloane
    int i;
    int ** p=(int **) malloc (nl*sizeof (int*)); // vector de pointeri la linii
    if ( p != NULL)
        for (i=0;i<nl;i++)
            p[i] =(int*) calloc (nc,sizeof (int)); // linii ca vectori alocati dinamic
    return p;
}

```

Utilizarea unui singur vector pentru a memora toate liniile unei matrice face mai dificilă programarea unor operații (selectie elemente, sortarea liniilor, s.a.).

3.6 VECTORI DE BITI

Atunci când elementele unui vector sau unei matrice au valori binare este posibilă o memorare mai compactă, folosind câte un singur bit pentru fiecare element din vector. Exemplele clasice sunt multimi realizate ca vectori de biti și grafuri de relații memorate prin matrice de adiacente cu câte un bit pentru fiecare element.

În continuare vom ilustra câteva operații pentru mulțimi realizate ca vectori de 32 de biți (variabile de tipul "long" pentru fiecare mulțime în parte). Operațiile cu mulțimi de biți se realizează simplu și rapid prin operatori la nivel de biți.

```
typedef long Set;    // mulțimi cu max 32 de elemente cu valori între 0 și 31
void initS ( Set & a) {          // inițializare mulțime
    a=0;
}
void addS (Set & a, int x) {      // adaugă element la mulțime
    a = a | (1L<<x);
}
void delS ( Set& a, int x) {      // elimină element din mulțime
    a=a & ~(1L<<x);
}
void retainAll ( Set& a1, Set a2) { // intersecție mulțimi
    a1= a1 & a2;
}
void addAll ( Set& a1, Set a2) {   // reuniune de mulțimi
    a1= a1 | a2;
}
void removeAll (Set& a1, Set a2) { // diferență de mulțimi
    a1 = a1 & ~a2;
}
int findS (Set a,int x) {          // test apartenență la o mulțime
    long b= a & (1L<<x);
    return (b==0) ? 0 : 1;
}
int containsAll (Set a1, Set a2) { // test includere mulțimi
    retainAll (a1,a2);
    if (a1==a2) return 1;
    return 0;
}
int sizeS ( Set a) {              // dimensiune (cardinal) mulțime
    int i, k=0;
    for (i=0;i< 32;i++)
        if ( findS (a,i))
            k++;
    return k;
}
void printS (Set a) {             // afișare mulțime
    int i;
    printf("{ ");
    for (i=0;i<32;i++)
        if( findS(a,i))
            printf("%d,",i);
    printf("\b }\n");
}
```

De observat că operațiile de căutare (findS) și cu două mulțimi (addAll s.a.) nu conțin cicluri și au complexitatea $O(1)$. Mulțimi ca vectori de biți există în Pascal (tipul "Set") și în limbaje cu clase (clasa "BitSet" în Java).

Într-o matrice de adiacențe a unui graf elementul $a[i][j]$ arată prezența (1) sau absența (0) unei muchii între vârfurile i și j .

În exemplul următor matricea de adiacență este un vector de biți, obținut prin memorarea succesivă a liniilor din matrice. Funcția "getbit" arată prezența sau absența unui arc de la nodul i la nodul j (graful este orientat). Funcția "setbit" permite adăugarea sau eliminarea de arce la/din graf.

Nodurile sunt numerotate de la 1.

```

typedef struct {
    int n ;           // nr de noduri in graf (nr de biti folositi)
    char b[256];      // vector de octeti (trebuie alocat dinamic)
} graf;
    // elementul [i][j] din matrice graf primeste valoarea val (0 sau 1)
void setbit (graf & g, int i, int j, int val) {
    int nb = g.n*(i-1) +j;      // nr bit in matrice
    int no = nb/8 +1;           // nr octet in matrice
    int nbo = nb % 8;           // nr bit in octetul no
    int b=0x80;
    int mask = (b >> nbo);      // masca selectare bit nbo din octetul no
    if (val)
        g.b[no] |= mask;
    else
        g.b[no] &= ~mask;
}
    // valoare element [i][j] din matrice graf
int getbit (graf g, int i, int j ) {
    int nb = g.n*(i-1) +j;      // nr bit in matrice
    int no = nb/8 +1;           // nr octet in matrice
    int nbo = nb % 8;           // nr bit in octetul no
    int b=0x80;
    int mask= (b >>nbo);
    return mask==( g.b[no] & mask);
}
    // citire date si creare matrice graf
void citgraf (graf & g ) {
    int no,i,j;
    printf("nr. noduri: "); scanf("%d",&g.n);
    no = g.n*g.n/8 + 1;        // nr de octeti necesari
    for (i=0;i<no;i++)
        g.b[i]=0;
    printf ("perechi de noduri legate prin arce:\n");
    do {
        if (scanf ( "%d%d",&i,&j) < 2) break;
        setbit (g,i,j,1);
    } while (1);
}

```

Ideea marcării prin biti a prezentei sau absentei unor elemente într-o colecție este folosită și pentru arbori binari (parcurși nivel cu nivel, pornind de la rădăcină), fiind generalizată pentru așa-numite structuri de date succinte (compacte), în care relațiile dintre elemente sunt implicate (prin poziția lor în colecție) și nu folosesc pointeri, a căror dimensiune contribuie mult la memoria ocupată de structurile cu pointeri.

Capitolul 4

LISTE CU LEGĂTURI

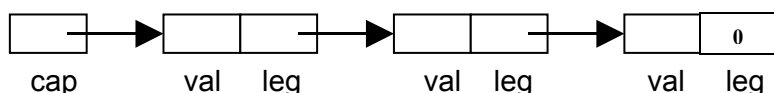
4.1 LISTE ÎNLĂNTUITE

O listă înlântuită ("Linked List") este o colecție de elemente, alocate dinamic, dispersate în memorie, dar legate între ele prin pointeri, ca într-un lant. O listă înlântuită este o structură dinamică, flexibilă, care se poate extinde continuu, fără ca utilizatorul să fie preocupat de posibilitatea depășirii unei dimensiuni estimate inițial (singura limită este mărimea zonei "heap" din care se solicită memorie).

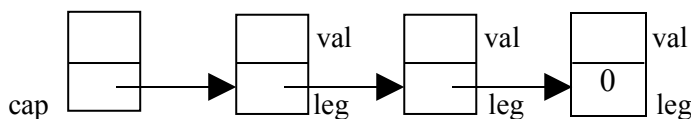
Vom folosi aici cuvântul "listă" pentru o listă liniară, în care fiecare element are un singur succesor și un singur predecesor.

Într-o listă înlântuită simplă fiecare element al listei conține adresa elementului următor din listă. Ultimul element poate conține ca adresă de legătură fie constanta NULL (un pointer către nicăieri), fie adresa primului element din listă (dacă este o listă circulară), fie adresa unui element terminator cu o valoare specială.

Adresa primului element din listă este memorată într-o variabilă pointer cu nume (alocată la compilare) și numită cap de listă ("list head").



Este posibil ca variabila cap de listă să fie tot o structură și nu un pointer:



Un element din listă (numit și nod de listă) este de un tip structură și are (cel puțin) două câmpuri: un câmp de date (sau mai multe) și un câmp de legătură. Exemplu:

```
typedef int T;                // orice tip numeric
typedef struct nod {
    T val ;                   // câmp de date
    struct nod *leg ;         // câmp de legătură
} Nod;
```

Conținutul și tipul câmpului de date depind de informațiile memorate în listă și deci de aplicația care o folosește. Toate funcțiile care urmează sunt direct aplicabile dacă tipul de date nedefinit T este un tip numeric (aritmetic).

Tipul "List" poate fi definit ca un tip pointer sau ca un tip structură:

```
typedef Nod * List;           // listă ca pointer
typedef struct { Nod * cap; } List; // listă ca structură
```

O listă înlântuită este complet caracterizată de variabila "cap de listă", care conține adresa primului nod (sau a ultimului nod, într-o listă circulară).

Operațiile uzuale cu o listă înlântuită sunt :

- Inițializare listă (a variabilei cap de listă): initL (List &)
- Adăugarea unui nou element la o listă: addL (List&, T)
- Eliminarea unui element dintr-o listă: delL (List&, T)

- Căutarea unei valori date într-o listă: `findL (List, T)`
- Test de listă vidă: `emptyL(List)`
- Determinarea dimensiunii listei: `sizeL (List)`
- Parcurgerea tuturor nodurilor din listă (traversare listă).

Accesul la elementele unei liste cu legături este strict secvențial, pornind de la primul element și trecând prin toate nodurile precedente celui căutat, sau pornind din elementul "curent" al listei, dacă se memorează și adresa elementului curent al listei.

Pentru parcurgere se folosește o variabilă cursor, de tip pointer către nod, care se initializează cu adresa cap de listă; pentru a avansa la următorul element din listă se folosește adresa din câmpul de legătură al nodului curent:

```
Nod *p, *prim;
p = prim;          // adresa primului element
...
p = p->leg;        // avans la urmatorul nod
```

Exemplu de afisare a unei liste înlănțuite definite prin adresa primului nod:

```
void printL ( Nod* lst) {
    while (lst != NULL) {          // repeta cat timp exista ceva la adresa lst
        printf ("%d ",lst->val);   // afisare date din nodul de la adresa lst
        lst=lst->leg;              // avans la nodul urmator din lista
    }
}
```

Structura de listă înlănțuită poate fi definită ca o structură recursivă: o listă este formată dintr-un element urmat de o altă listă, eventual vidă. Acest punct de vedere poate conduce la funcții recursive pentru operații cu liste, dar fără nici un avantaj față de funcțiile iterative. Exemplu de afisare recursivă a unei liste:

```
void printL ( Nod* lst) {
    if (lst != NULL) {             // daca (sub)lista nu e vidă
        printf ("%d ",lst->val);   // afisarea primului element
        printL (lst->leg);         // afisare sublistă de după primul element
    }
}
```

Căutarea secvențială a unei valori date într-o listă este asemănătoare operației de afisare, dar are ca rezultat adresa nodului ce conține valoarea căutată .

```
// căutare într-o listă neordonată
Nod* findL (Nod* lst, T x) {
    while (lst!=NULL && x != lst->val)
        lst = lst->leg;
    return lst;                    // NULL dacă x negăsit
}
```

Operația de initializare a unei liste stabilește adresa de început a listei:

```
void initL (Nod* & lst) {
    lst= NULL;                    // lista initial vida (fara santinela)
}
```

Crearea unui nou element de listă necesită alocarea de memorie, prin funcția “malloc” în C sau prin operatorul *new* în C++. Verificarea rezultatului cererii de alocare (NULL, dacă alocare imposibilă) se poate face printr-o instrucțiune “if” sau prin funcția “assert”, dar va fi omisă în continuare. Exemplu de alocare:

```
nou = (Nod*) malloc( sizeof(Nod));    // sau  nou = new Nod;
assert (nou != NULL);                // se include <assert.h>
```

Adăugarea unui element la o listă înlăntuită se poate face:

- Mereu la începutul listei;
- Mereu la sfârșitul listei;
- Intr-o poziție determinată de valoarea noului element.

Dacă ordinea datelor din listă este indiferentă pentru aplicație, atunci cel mai simplu este ca adăugarea să se facă numai la începutul listei. În acest caz lista este de fapt o stivă iar afișarea valorilor din listă se face în ordine inversă introducerii în listă.

Exemplu de creare și afișare a unei liste înlăntuite, cu adăugare la început de listă:

```
typedef Nod* List;                // ptr a permite redefinirea tipului "List"
int main () {
    List lst; int x;  Nod * nou;   // nou=adresa element nou
    lst=NULL;                  // initializare lista vida
    while (scanf("%d",&x) > 0) {
        nou=(Nod*)malloc(sizeof(Nod)); // aloca memorie
        nou->val=x; nou->leg=lst;      // completare element
        lst=nou;                      // noul element este primul
    }
    while (lst != NULL) {          // afisare lista
        printf("%d ",lst->val);      // in ordine inversa celei de adaugare
        lst=lst->leg;
    }
}
```

Operațiile elementare cu liste se scriu ca funcții, pentru a fi reutilizate în diferite aplicații. Exemplu de inserție element la început de listă:

```
void insL ( List & lst, T x) {
    Nod* nou= new (Nod);
    assert ( nou != NULL);
    nou->val=x; nou->leg=lst;
    lst=nou;                      // se modifică parametrul funcției !
}
```

Exemplu de adăugare a unui nou element la sfârșitul unei liste:

```
void addL ( List & lst, T x) {
    Nod *p;                      // un cursor in lista
    Nod* nou = new Nod ;
    assert ( nou != NULL);
    nou->val=x; nou->leg=NULL;    // noul element este si ultimul element
    if ( lst==NULL) {           // daca lista era goala
        lst=nou; return;        // modifica adresa cap de lista
    }
    // leaga nou dupa ultimul element din lista
    p=lst;                      // pentru a nu modifica adresa de inceput a listei !
    while (p->leg !=NULL)        // repeta cat timp exista un element urmator
        p=p->leg;
```

```

    p→leg=nou;                // acum p este adresa ultimului nod din lista
}

```

În exemplele anterioare funcția de adăugare a primit ca argument valoarea ce trebuie introdusă în listă, ceea ce o face dependentă de tipul datelor memorate în listă. O altă idee este de a crea elementul de listă în afara funcției (în programul principal, de exemplu) și de a transmite ca parametru adresa noului element. Exemple:

```

// adaugare nod cu adr. px dupa nodul cu adr. crt
void addL (List crt, List px) {
    px→leg=crt→leg;
    crt→leg=px;
}
// adaugare nod cu adr. px înaintea nodului cu adr. crt
void insL (List crt, List px) {
    T tmp;
    // adauga px dupa crt
    px→leg = crt→leg; crt→leg=px;
    // schimba datele din crt cu cele din px
    tmp=crt→val; crt→val=px→val; px→val=tmp;
}

```

Eliminarea unui nod cu valoare dată x dintr-o listă necesită găsirea nodului p care conține valoarea x , urmată de modificarea legăturii nodului anterior lui p . Într-o listă dublu înălțuită fiecare nod conține și adresa nodului precedent, dar pentru o listă simplu înălțuită trebuie să reținem adresa q a nodului precedent lui p , în timpul operației de căutare a nodului p cu valoarea x . Avem de ales între două variante:

- să folosim doi pointeri q și p astfel că $q→leg=p$ (q este urmat de p);
- să folosim un singur pointer q și să comparăm pe x cu $q→leg→val$ (cu valoarea din nodul următor lui q).

Funcție de eliminare nod dintr-o listă înălțuită, folosind doi pointeri:

```

void delL (List &lst, T x) {    // elimina element cu valoarea x din lista lst
    Nod *p=lst, *q=lst;        // initial q=p
    while ( p != NULL && p→val !=x) {    // cauta pe x in lista (x de tip numeric)
        q=p; p=p→leg;            // q→leg == p (q inainte de p)
    }
    if (p != NULL) {            // daca x găsit
        if (q==p)                // daca p este primul nod din lista
            lst=lst→leg;        // modifica adresa de inceput a listei
        else                    // x găsit la adresa p
            q→leg=p→leg;        // dupa q urmeaza acum succesorul lui p
        free(p);                // eliberare memorie ocupata de elem. eliminat
    }
}

```

Funcție de eliminare nod dintr-o listă înălțuită, folosind un singur pointer:

```

void del (List &lst, T x) {
    Nod* q=lst; Nod* p;        // p = adresa nod cu valoarea x
    if (lst==NULL) return;    // daca lista e vida se iese
    if (x==lst→val)            // daca x este in primul nod din lista
        lst=lst→leg;          // atunci se modifica inceputul listei lst
    while (q→leg !=0 && q→leg→val != x)    // compara x cu q→leg→val
        q=q→leg;
    if (q→leg ==NULL) return;    // daca x negasit in lista se iese
}

```

```

p=q->leg;           // p=adresa nod cu valoarea x;
q->leg = q->leg->leg; // sau: q->leg=p->leg;
free(p);           // eliberare memorie
}

```

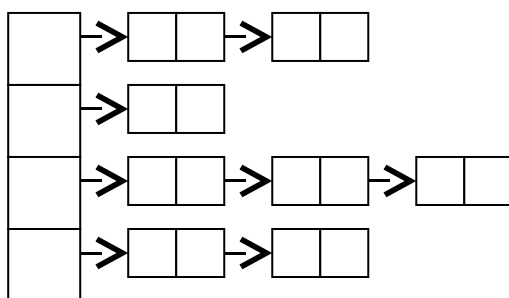
4.2 COLECTII DE LISTE

Listele sunt preferate vectorilor atunci când aplicatia foloseste mai multe liste de lungimi foarte variabile si impredictibile, deoarece asigură o utilizare mai bună a memoriei. Reunirea adreselor de început ale listelor într-o colectie de pointeri se face fie printr-un vector de pointeri la liste, fie printr-o listă înlănțuită de pointeri sau printr-un arbore ce contine în noduri pointeri la liste.

Mentionăm câteva aplicatii clasice care folosesc colectii de liste:

- Sortarea pe compartimente ("Radix Sort" sau "Bin Sort");
- O colectie de multimi disjuncte, în care fiecare multime este o listă;
- Un graf reprezentat prin liste de adiacente (liste cu vecinii fiecărui nod);
- Un dictionar cu valori multiple, în care fiecare cheie are asociată o listă de valori;
- Un tabel de dispersie ("Hashtable") realizat ca vector de liste de coliziuni;

O colectie liniară de liste se reprezintă printr-un vector de pointeri atunci când este necesar un acces direct la o listă printr-un indice (grafuri, sortare pe ranguri, tabele hash) sau printr-o listă de pointeri atunci când numărul de liste variază în limite largi si se poate modifica dinamic (ca într-un dictionar cu valori multiple).



În continuare se prezintă succint sortarea după ranguri (pe compartimente), metodă care împarte valorile de sortat în mai multe compartimente, cărora le corespund tot atâtea liste înlănțuite.

Sortarea unui vector de n numere (cu maxim d cifre zecimale fiecare) se face în d treceri: la fiecare trecere k se distribuie cele n numere în 10 "compartimente" (liste) după valoarea cifrei din poziția k ($k=1$ pentru cifra din dreapta), si apoi se reunesc listele în vectorul de n numere (în care ordinea se modifică după fiecare trecere).

Algoritmul poate fi descris astfel:

```

repetă pentru k de la 1 la d      // pentru fiecare rang
  initializare vector de liste t
  repetă pentru i de la 1 la n      // distribuie elem. din x in 10 liste
    extrage în c cifra din poziția k a lui x[i]
    adaugă x[i] la lista t[c]
  repetă pentru j de la 0 la 9      // reunire liste in vectorul x
    adaugă toată lista j la vectorul x

```

Exemplu cu $n=9$ si $d=3$:

Initial	Trecerea 1		Trecerea 2		Trecerea 3	
Vector	cifra liste	vector	cifra liste	vector	cifra liste	vector
459	0	472	0	432	0	177
254	1	432	1	534	1	177
472	2	254	2	239	2	239,254
534	3	534	3	432,534,239	3	432

649	4	254,534,654	654	4	649	254	4	432,459,472	459
239	5		177	5	254,654,459	654	5	534	472
432	6		459	6		459	6	649,654	534
654	7	177	649	7	472,177	472	7		649
177	8		239	8		177	8		654
	9	459,649,239		9			9		

Cifra din pozitia k a unui număr y se obtine cu relatia: $c = (y / \text{pow}(10, k-1)) \% 10$;

Adăugarea de elemente la o listă (în faza de distribuire) se face mereu la sfârșitul listei, dar extragerea din liste (în faza de colectare a listelor) se face mereu de la începutul listelor, ceea ce face ca fiecare listă să se comporte ca o coadă.

Pentru ordonare de cuvinte formate din litere numărul de compartimente va fi numărul de litere distincte (26 dacă nu contează diferența dintre litere mari și mici).

Funcția următoare implementează algoritmul de sortare pe ranguri:

```
void radsort (int x[ ], int n) {
    int div=1;                // divizor (puteri ale lui 10)
    int i,k,c,d=5;            // d= nr maxim de cifre in numerele sortate
    List t [10];              // vector de pointeri la liste
    // repartizare valori din x in listele t
    for (k=1; k<=d; k++) {    // pentru fiecare rang (cifra zecimala)
        for (c=0; c<10; c++)  // initializare vector pointeri la liste
            initL( t[c] );    // initializare lista care incepe in t[c]
        for (i=0; i<n; i++) {  // distribuie x[i] în liste după cifra k
            c= (x[i] / div) % 10 ; // cifra din pozitia k a lui x[i]
            addL ( t[c], x[i]); // adauga x[i] la lista din pozitia c
        }
        // reuneste liste din t in x
        i=0;
        for (c=0; c<10; c++) { // repeta pentru toate cele 10 liste
            while ( ! emptyL ( t[c] ) ) // cat timp mai sunt elemente in lista t[c]
                x[i++] = delfirstL ( t[c]); // extrage element de la inceputul listei vp[c]
        } // si se adauga la vectorul x
        div=div*10;            // divizor ptr rangul următor
    }
}
```

Tipul abstract “Colectie de multimi disjuncte” poate fi implementat și printr-o colecție de liste, cu câte o listă pentru fiecare multime. Adresele de început ale listelor din colecție sunt reunite într-un vector de pointeri. Numărul de liste se modifică pe măsură ce se reunesc câte două liste într-una singură. Ordinea elementelor în fiecare listă nu este importantă astfel că reunirea a două liste se poate face legând la ultimul element dintr-o listă primul element din cealaltă listă.

Evoluția listelor multimi pentru 6 valori între care există relațiile de echivalență 2~4, 1~3, 6~3, 4~6 poate fi următoarea:

Initial	2~4	1~3	6~3	4~6
→1	→1	→1→3	→1→3→6	→1→3→6→2→4
→2	→2→4	→2→4	→2→4	
→3	→3			
→4				
→5	→5	→5	→5	→5
→6	→6	→6		

În programul următor se consideră că ordinea în fiecare multime nu contează și reuniunea de multimi se face legând începutul unei liste la sfârșitul altei liste.


```

typedef struct sn { // un element de lista
    int nr;          // valoare element multime
    struct sn * leg;
} nod;
typedef struct {
    int n;           // nr de multimii in colectie
    nod* m[M];       // vector de pointeri la liste
} ds;               // tipul "disjoint sets"
// initializare colectie c de n multimii
void initDS ( ds & c, int n) {
    int i; nod* p ;
    c.n=n;
    for (i=1;i<=n;i++) { // pentru fiecare element
        p= (nod*)malloc (sizeof(nod)); // creare un nod de lista
        p->nr = i; p->leg = NULL;       // cu valoarea i si fara succesor
        c.m[i] = p;                   // adresa listei i în pozitia i din vector
    }
}
// cautare într-o lista înlantuita
int inSet (int x, nod* p) {
    while (p != NULL) // cat timp mai exista un nod p
        if (p->nr==x) // daca nodul p contine pe x
            return 1; // gasit
        else // daca x nu este in nodul p
            p= p->leg; // cauta in nodul urmator din lista
    return 0; // negasit
}
// gaseste multimea care-l contine pe x
int findDS (ds c, int x) {
    int i;
    for (i= 1;i<=c.n;i++) // pentru fiecare lista din colectie
        if ( inSet(x,c.m[i]) ) // daca lista i contine pe x
            return i; // atunci x in multimea i
    return 0; // sau -1
}
// reuniune multimii ce contin pe x si pe y
void unifDS (ds & c, int x, int y) {
    int ix,iy ; nod* p;
    ix= find (x,c); iy= find (y,c);
    // adauga lista iy la lista ix
    p= c.m[ix]; // aici incepe lista lui x
    while (p->leg != NULL) // cauta sfarsitul listei lui x
        p=p->leg; // p este ultimul nod din lista ix
    p->leg = c.m[iy]; // leaga lista iy dupa ultimul nod din lista ix
    c.m[iy] = NULL; // si lista iy devine vida
}
    
```

4.3 LISTE ÎNLĂNTUITE ORDONATE

Listele înlantuite ordonate se folosesc în aplicatiile care fac multe operatii de adăugare si/sau stergere la/din listă si care necesită mentinerea permanentă a ordinii în listă. Pentru liste adăugarea cu păstrarea ordinii este mai eficientă decât pentru vectori, dar reordonarea unei liste înlantuite este o operatie ineficientă.

În comparatie cu inserarea într-un vector ordonat, inserarea într-o listă este mai rapidă si mai simplă deoarece nu necesită mutarea unor elemente în memorie. Pe de altă parte, căutarea unei valori

Într-o listă înlănțuită ordonată nu poate fi așa eficientă ca și căutarea într-un vector ordonat (căutarea binară nu se poate aplica și la liste).

Crearea și afisarea unei liste înlănțuite ordonate poate fi considerată și ca o metodă de ordonare a unei colecții de date.

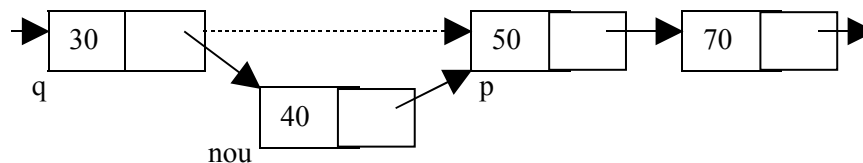
Operația de inserare a unei valori la o listă ordonată este precedată de o căutare a locului unde se face inserarea, adică de găsirea nodului de care se va lega noul element. Mai exact, se caută primul nod cu valoare mai mare decât valoarea care se adaugă. Căutarea folosește o funcție de comparare care depinde de tipul datelor memorate și de criteriul de ordonare al elementelor.

După căutare pot apărea 3 situații:

- Noul element se inserează înaintea primului nod din listă;
- Noul element se adaugă după ultimul element din listă;
- Noul element se intercalează între două noduri existente.

Prima situație necesită modificarea capului de listă și de aceea este tratată separat.

Pentru inserarea valorii 40 într-o listă cu nodurile 30, 50, 70 se caută prima valoare mai mare ca 40 și se inserează 40 înaintea nodului cu 50. Operația presupune modificarea adresei de legătură a nodului precedent (cu valoarea 30), deci trebuie să dispunem și de adresa lui. În exemplul următor se folosește o variabilă pointer *q* pentru a reține mereu adresa nodului anterior nodului *p*, unde *p* este nodul a cărui valoare se compară cu valoarea de adăugat (deci avem mereu $q \rightarrow \text{leg} = p$).



Adăugarea unui nod la o listă ordonată necesită:

- crearea unui nod nou: alocare de memorie și completare câmp de date;
- căutarea poziției din listă unde trebuie legat noul nod;
- legarea efectivă prin modificarea a doi pointeri: adresa de legătură a nodului precedent *q* și legătura noului nod (cu excepția adăugării înaintea primului nod):

$q \rightarrow \text{leg} = \text{nou}; \text{nou} \rightarrow \text{leg} = p;$

```

// inserție în listă ordonată, cu doi pointeri
void insL (List & lst, T x) {
    Nod *p, *q, *nou;
    nou = (Nod*)malloc(sizeof(Nod)); // creare nod nou ptr x
    nou->val = x; // completare cu date nod nou
    if (lst == NULL || x < lst->val) { // dacă lista vidă sau x mai mic ca primul elem
        nou->leg = lst; lst = nou; // dacă nou la început de listă
    } else { // altfel caută locul unde trebuie inserat x
        p = q = lst; // q este nodul precedent lui p
        while (p != NULL && p->val < x) {
            q = p; p = p->leg; // avans cu pointerii q și p
        }
        nou->leg = p; q->leg = nou; // nou se introduce între q și p
    }
}

```

Funcția următoare folosește un singur pointer *q*: căutarea se oprește pe nodul 'q', precedent celui cu valoare mai mare ca *x* ("nou" se leagă între *q* și $q \rightarrow \text{leg}$):

```

void insL (List & lst, T x) {
    Nod* q, *nou;
    nou = (Nod*)malloc(sizeof(Nod)); // creare nod nou
    nou->val = x;

```

```

if ( lst==NULL || x < lst->val ) { // daca lista vida sau x mai mic ca primul
    nou->leg=lst; lst= nou;      // adaugare la inceput de lista
    return;
}
q=lst;                          // ca sa nu se modifice inceputul listei lst
while ( q->leg !=NULL && x > q->leg->val ) // pana cand x < q->leg->val
    q=q->leg;
nou->leg=q->leg; q->leg=nou;     // nou intre q si q->leg
}

```

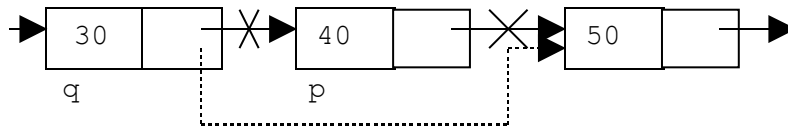
O altă soluție este ca noul element să se adauge după cel cu valoare mai mare (cu adresa p) și apoi să se schimbe între ele datele din nodurile p și nou.

Stergerea unui element cu valoare dată dintr-o listă începe cu căutarea elementului în listă, urmată de modificarea adresei de legătură a nodului precedent celui sters. Fie p adresa nodului ce trebuie eliminat și q adresa nodului precedent. Eliminarea unui nod p (diferit de primul) se realizează prin următoarele operații:

```

q->leg = p->leg;    // succesorul lui p devine succesorul lui q
free(p);

```



Dacă se șterge chiar primul nod, atunci trebuie modificată și adresa de început a listei (primită ca argument de funcția respectivă).

Funcția următoare elimină nodul cu valoarea 'x' folosind doi pointeri.

```

void delL (List & lst, T x) { // elimina element cu valoarea x din lista lst
    Nod* p=lst, *q=lst;
    while ( p != NULL && x > p->val ) { // cauta pe x in lista (x de tip numeric)
        q=p; p=p->leg;                // q->leg == p (q inainte de p)
    }
    if (p->val == x) {                 // daca x găsit
        if (q==p)                     // daca p este primul nod din lista
            lst= lst->leg;              // modifica adresa de inceput a listei
        else                          // x gasit la adresa p
            q->leg=p->leg;              // dupa q urmeaza acum succesorul lui p
        free(p);                      // eliberare memorie ocupata de elem. eliminat
    }
}

```

Funcția următoare de eliminare folosește un singur pointer:

```

void delL (List & lst, T x) {
    Nod* p=lst; Nod* q;                // q= adresa nod eliminat
    if (x==lst->val) {                 // daca x este in primul element
        q=lst; lst=lst->leg;          // q necesar pentru eliberare memorie
        free(q); return;
    }
    while ( p->leg !=NULL && x > p->leg->val )
        p=p->leg;
    if (p->leg ==NULL || x !=p->leg->val) return; // x nu exista in lista
    q=p->leg;                          // adresa nod de eliminat
    p->leg=p->leg->leg;
    free(q);
}

```

}

Inserarea si stergerea într-o listă ordonată se pot exprima si recursiv:

```
// inserare recursiva in lista ordonata
void insL (List & lst, T x) {
    Nod * aux;
    if ( lst !=NULL && x > lst->val) // daca x mai mare ca primul element
        insL ( lst->leg,x);          // se va introduce in sublista de dupa primul
    else {                          // lista vida sau x mai mic decat primul elem
        aux=lst;                    // adresa primului element din lista veche
        lst=(Nod*)malloc(sizeof(Nod));
        lst->val=x; lst->leg= aux;    // noul element devine primul element
    }
}

// eliminare x din lista lst (recursiv)
void delL (List & lst, T x) {
    Nod* q;                        // adresa nod de eliminat
    if (lst != NULL)               // daca lista nu e vida
        if (lst->val != x)          // daca x nu este in primul element
            delL (lst->leg,x);      // elimina x din sublista care urmeaza
        else {                     // daca x in primul element
            q=lst; lst=lst->leg;    // modifica adresa de inceput a listei
            free(q);
        }
}
```

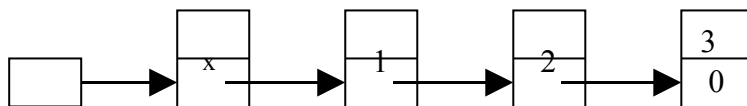
Funcțiile pentru operații cu liste ordonate pot fi simplificate folosind liste cu element sentinelă și alte variante de liste înțânțuite.

4.4 VARIANTE DE LISTE ÎNȚÂNȚUITE

Variantele de liste întâlnite în literatură și în aplicații pot fi grupate în:

- Liste cu structură diferită față de o listă simplă deschisă: liste circulare, liste cu element sentinelă, liste dublu înțânțuite, etc.
- Liste cu elemente comune: un același element aparține la două sau mai multe liste, având câte un pointer pentru fiecare din liste. În felul acesta elementele pot fi parcurse și folosite în ordinea din fiecare listă. Clasa `LinkedHashSet` din Java folosește această idee pentru a menține ordinea de adăugare la mulțime a elementelor dispersate în mai multe liste de coliziuni (sinonime).
- Liste cu auto-organizare, în care fiecare element accesat este mutat la începutul listei ("Splay List"). În felul acesta elementele folosite cel mai frecvent se vor afla la începutul listei și vor avea un timp de regăsire mai mic.
- Liste cu acces mai rapid și/sau cu consum mai mic de memorie.

O listă cu sentinelă conține cel puțin un element (numit sentinelă), creat la initializarea listei și care rămâne la începutul listei indiferent de operațiile efectuate:



Deoarece lista nu este niciodată vidă și adresa de început nu se mai modifică la adăugare sau la stergere de elemente, operațiile sunt mai simple (nu mai trebuie tratat separat cazul modificării primului element din listă). Exemple de funcții:

```
// initializare lista cu sentinela
void initL (List & lst) {
```

```

    lst=(Nod*)malloc(sizeof(Nod));
    lst->leg=NULL;                // nimic în lst->val
}
// afisare lista cu santinela
void printL ( List lst) {
    lst=lst->leg;                  // primul element cu date
    while (lst != NULL) {
        printf("%d ", lst->val);   // afisare element curent
        lst=lst->leg;              // si avans la urmatorul element
    }
}
// inserare in lista ordonata cu santinela
void insL (List lst, int x) {
    Nod *p=lst, *nou ;
    nou= (Nod*)malloc(sizeof(Nod));
    nou->val=x;
    while( p->leg != NULL && x > p->leg->val )
        p=p->leg;
    nou->leg=p->leg; p->leg=nou; // nou dupa p
}
// eliminare din lista ordonata cu santinela
void delL (List lst, int x) {
    Nod*p=lst; Nod*q;
    while ( p->leg !=NULL && x > p->leg->val) // cauta pe x in lista
        p=p->leg;
    if (p->leg ==NULL || x !=p->leg->val) return; // daca x nu exista in lista
    q=p->leg; // adresa nod de eliminat
    p->leg=p->leg->leg;
    free(q);
}

```

Simplificarea introdusă de elementul santinelă este importantă și de aceea se poate folosi la stive liste înlănțuite, la liste “skip” și alte variante de liste.

În elementul santinelă se poate memora dimensiunea listei (numărul de elemente cu date), actualizat la adăugare și la eliminare de elemente. Consecința este un timp $O(1)$ în loc de $O(n)$ pentru operația de obținere a dimensiunii listei (pentru că nu mai trebuie numărate elementele din listă). La compararea a două mulțimi implementate ca liste neordonate pentru a constata egalitatea lor, se reduce timpul de comparare prin compararea dimensiunilor listelor, ca primă operație.

În general se practică memorarea dimensiunii unei colecții și actualizarea ei la operațiile de modificare a colecției, dar într-o structură (sau clasă) care definește colecția respectivă, împreună cu adresa de început a listei.

Prin simetrie cu un prim element (“head”) se folosește uneori și un element terminator de listă (“tail”), care poate conține o valoare mai mare decât oricare valoare memorată în listă. În acest fel se simplifică condiția de căutare într-o listă ordonată crescător. Elementul final este creat la initializarea listei :

```

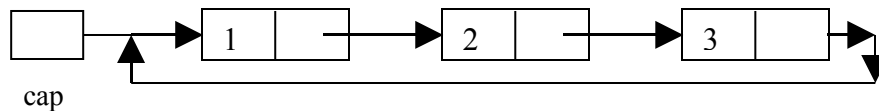
void initL (List & lst) {
    Nod* term;
    term= new Nod; // crearea element terminator de lista
    term->val=INT_MAX; // valoare maxima ptr o lista de numere intregi
    term->leg=NULL;
    lst= new Nod; // creare element santinela
    lst->leg=term; // urmat de element terminator
    lst->val=0; // si care contine lungimea listei
}

```

Exemplu de căutare a unei valori date într-o listă ordonată cu element terminator:

```
Nod* findL (Nod* lst, int x) {
    lst=lst->leg;           // se trece peste santinela
    while ( x > lst->val)    // se opreste cand x < lst->val
        lst=lst->leg;
    return lst->val==x ? lst: NULL; // NULL daca x negasit
}
```

Listele circulare permit accesul la orice element din listă pornind din pozitia curentă, fără a fi necesară o parcurgere de la începutul listei. Într-o listă circulară definită prin adresa elementului curent, nici nu este important care este primul sau ultimul element din listă.



Definiția unui nod de listă circulară este aceeași ca la o listă deschisă. Modificări au loc la initializarea listei și la condiția de terminare a listei: se compară adresa curentă cu adresa primului element în loc de comparație cu constanta NULL.

Exemplu de operații cu o listă circulară cu element sentinelă:

```
// initializare lista circulara cu sentinela
void initL (List & lst) {
    lst = (Nod*) malloc (sizeof(Nod)); // creare element santinela
    lst->leg=lst;                       // legat la el insusi
}

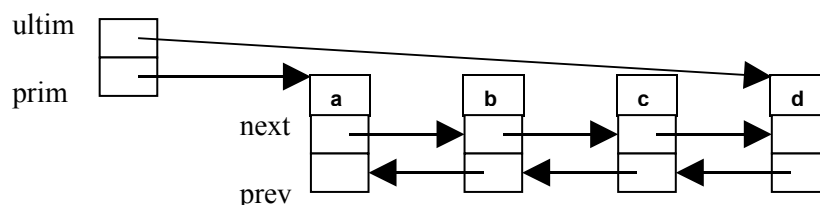
// adaugare la sfarsit de lista
void addL (List & lst, int x) {
    Nod* p=lst; // un cursor in lista
    Nod* nou = (Nod*) malloc(sizeof(Nod));
    nou->val=x;
    nou->leg=lst; // noul element va fi si ultimul
    while (p->leg != lst) // cauta adresa p a ultimului element
        p=p->leg;
    p->leg=nou;
}

// afisare lista
void printL (List lst) { // afisare continut lista
    Nod* p= lst->leg; // primul elem cu date este la adr. p
    while ( p != lst) { // repeta pana cand p ajunge la santinela
        printf ("%d ", p->val); // afisare obiect din pozitia curenta
        p=p->leg; // avans la urmatorul element
    }
}
```

4.5 LISTE DUBLU ÎNLĂNTUITE

Într-o listă liniară dublu înlăntuită fiecare element conține două adrese de legătură: una către elementul următor și alta către elementul precedent. Această structură permite accesul mai rapid la elementul precedent celui curent și parcurgerea listei în ambele sensuri (inclusiv existența unui iterator în sens invers pe listă).

Pentru acces rapid la ambele capete ale listei se poate defini tipul "DList" și ca o structură cu doi pointeri: adresa primului element și adresa ultimului element; acest tip de listă se numește uneori "deque" ("double-ended queue") și este folosită pentru acces pe la ambele capete ale listei.



Exemplu de definire nod de listă dublu înlăntuită:

```
typedef struct nod {          // structura nod
    T val;                    // date
    struct nod * next;        // adresa nod urmator
    struct nod * prev;        // adresa nod precedent
} Nod, * DList;
```

O altă variantă de listă dublu-înlăntuită este o listă circulară cu element santinelă. La crearea listei se creează elementul santinelă. Exemplu de initializare:

```
void initDL (DList & lst) {
    lst = (Nod*)malloc (sizeof(Nod));
    lst->next = lst->prev = lst;
}
```

În funcțiile care urmează nu se transmite adresa de început a listei la operațiile de inserare și de ștergere, dar se specifică adresa elementului șters sau față de care se adaugă un nou element. Exemple de realizare a unor operații cu liste dublu-înlăntuite:

```
void initDL (List & lst) {
    lst = (Nod*)malloc (sizeof(Nod));
    lst->next = lst->prev = NULL;          // lista nu e circulară !
}

// adauga nou dupa pozitia pos
void addDL (Nod* nou, Nod* pos) {
    nou->next = pos->next;
    nou->prev = pos;
    pos->next = nou;
}

// insertie nou inainte de pozitia pos
void insDL ( Nod* nou, Nod * pos) {
    Nod* prec ;
    prec = pos->prev;                    // nod precedent celui din pos
    nou->prev = pos->prev;                // nod precedent lui nou
    nou->next = pos;                     // pos dupa nou
    prec->next = nou;                    // prec inaintea lui nou
    pos->prev = nou;                     // nou inaintea lui pos
}

// stergere element din pozitia pos
void delDL (Nod* pos) {
    Nod * prec, *urm;
    prec = pos->prev;                    // predecesorul nodului de sters
    urm = pos->next;                     // succesorul nodului de sters
    if (pos != prec) {                   // daca nu este sentinela
        prec->next = pos->next;
        urm->prev = prec;
    }
```

```

    free(pos);
}
}

// cauta pozitia unei valori in lista
Nod* pos (DList lst, T x) {
    Nod * p = lst->next;           // primul element cu date
    while ( p != NULL && x != p->val) // cauta pe x in lista
        p=p->next;
    if (p ==NULL) return NULL;      // negasit
    else return p;                  // gasit la adresa p
}

// creare si afisare lista dublu-inlantuita
void main () {
    int x;  Nod *lst, *p, *nou;
    initDL(lst); p= lst;
    for (x=1;x<10;x++) {
        nou= (Nod*) malloc (sizeof(Nod));
        nou->val=x;
        addDL(nou,p); p=nou;        // insDL ( nou ,p); p=nou;
    }
    printDL ( lst);                 // afisare lista
    // sterge valori din lista
    for (x=1;x<10;x++) {
        p= pos(lst,x);              // pozitia lui x in lista
        delDL(p);                   // sterge din pozitia p
    }
}

```

Funcțiile anterioare folosesc un cursor extern listei și pot fi folosite pentru a realiza orice operații cu o listă: inserție în orice poziție, stergere din orice poziție ș.a.

De cele mai multe ori nu se justifică un pointer suplimentar către elementul precedent deoarece poziționarea pe un element din listă se face de obicei printr-o căutare, iar la căutare se poate reține adresa elementului precedent celui găsit:

```

Nod * p,*prev;           // prev este adresa nodului precedent nodului p
prev = p = lst;          // sau p=lst->next ptr liste cu santinela
while ( p != NULL && x != p->val) {      // cauta pe x in lista
    prev=p; p=p->next;
}

```

4.6 COMPARATIE ÎNTRE VECTORI SI LISTE

Un vector este recomandat atunci când este necesar un acces aleator frecvent la elementele listei (complexitate $O(1)$), ca în algoritmi de sortare, sau când este necesară o regăsire rapidă pe baza poziției în listă sau pentru listele al căror conținut nu se mai modifică și trebuie menținute în ordine (fiind posibilă și o căutare binară). Inserția și eliminarea de elemente în interiorul unui vector au însă complexitatea $O(n)$, unde “n” este dimensiunea vectorului.

O listă înlănțuită se recomandă atunci când dimensiunea listei este greu de estimat, fiind posibile multe adăugări și/sau stingeri din listă, sau atunci când sunt necesare inserări de elemente în interiorul listei. Deși este posibil accesul pozițional, printr-un indice întreg, la elementele unei liste înlănțuite, utilizarea sa frecventă afectează negativ performanțele aplicației (complexitatea $O(n)$).

Dacă este necesară o colecție ordonată, atunci se va folosi o listă permanent ordonată, prin procedura de adăugare la listă și nu se face o reordonare a unei liste înlănțuite, așa cum se face ca în cazul vectorilor.

Vectorii au proprietatea de localizare a referințelor, ceea ce permite un acces secvențial mai rapid prin utilizarea unei memorii “cache” (așa cum au procesoarele moderne); memoria “cache” nu ajută în

aceeasi măsură si la reducerea timpului de prelucrare succesivă a elementelor unei liste înlănțuite (mai dispersate în memorie). Din același motiv structura de listă înlănțuită nu se folosește pentru date memorate pe un suport extern (disc magnetic sau optic).

Ideea memoriei “cache” este de a înlocui accesul individual la date dintr-o memorie (cu timp de acces mai mare) prin citirea unor grupuri de date adiacente într-o memorie mai rapidă (de capacitate mai mică), în speranța că programele fac un acces secvențial la date (folosește datele în ordinea în care sunt memorate).

Memorarea explicită de pointeri conduce la un consum suplimentar de memorie, ajungându-se la situații când memoria ocupată de pointeri (și de metadatele asociate cu alocarea dinamică de memorie) să depășească cu mult memoria ocupată de datele necesare aplicației. Prin “metadate” se înțeleg informațiile folosite pentru gestiunea memoriei “heap” dar și faptul că blocurile alocate din “heap” au o dimensiune multiplu de 8, indiferent de numărul de octeți solicitat (poate fi un alt multiplu, dar în orice caz nu pot avea orice dimensiune). Blocurile de memorie alocate dinamic sunt legate împreună într-o listă înlănțuită, la fel ca și blocurile de memorie eliberate prin funcția “free” și care nu sunt adiacente în memorie. Fiecare element din lista spațiului disponibil sau din lista blocurilor alocate este precedat de lungimea sa și de un pointer către următorul element din listă; acestea sunt “metadate” asociate alocării dinamice.

Aceste considerente fac ca de multe ori să se prefere structura de vector în locul unei structuri cu pointeri, tendință accentuată odată cu creșterea dimensiunii memoriei RAM și deci a variabilelor pointer (de la 2 octeți la 4 octeți și chiar la 8 octeți). Se vorbește uneori de structuri de date “succinte” (compacte) atunci când se renunță la structuri de liste înlănțuite sau de arbori cu pointeri în favoarea vectorilor.

La o analiză mai atentă putem deosebi două modalități de eliminare a pointerilor din structurile de date:

- Se păstrează ideea de legare a unor date dispersate fizic dar nu prin pointeri ci prin indici în cadrul unui vector; altfel spus, în locul unor adrese absolute de memorie (pointeri) se folosesc adrese relative în cadrul unui vector pentru legături. Această soluție are și avantajul că face descrierea unor algoritmi independentă de sintaxa utilizării de pointeri (sau de existența tipurilor pointer într-un limbaj de programare) și de aceea este folosită în unele manuale (cea mai cunoscută fiind cartea “Introduction to Algorithms” de T.Cormen, C.Leiserson, R.Rivest, C.Stein , tradusă și în limba română). Ideea se folosește mai ales pentru arbori binari, cum ar fi arbori Huffman sau alți arbori cu număr limitat de noduri.

- Se renunță la folosirea unor legături explicite între date, poziția datelor în vector va determina și legăturile dintre ele. Cel mai bun exemplu în acest sens este structura de vector “heap” (vector parțial ordonat) care memorează un arbore binar într-un vector fără a folosi legături: fiii unui nod aflat în poziția k se află în pozițiile $2*k$ și $2*k+1$, iar părintele unui nod din poziția j se află în poziția $j/2$. Un alt exemplu este soluția cea mai eficientă pentru structura “multimi disjuncte” (componente conexe dintr-un graf): un vector care conține o pădure de arbori, dar în care se memorează numai indici către părintele fiecărui nod din arbore (valoarea nodului este chiar indicele său în vector).

Extinderea acestor idei și la alte structuri conduce în general la un consum mare de memorie, dar poate fi eficientă pentru anumite cazuri particulare; un graf cu număr mare de noduri și arce poate fi reprezentat eficient printr-o matrice de adiacente, dar un graf cu număr mic de arce și număr mare de noduri se va memora mai eficient prin liste înlănțuite de adiacente sau printr-o matrice de biți.

Considerațiile anterioare nu trebuie să conducă la neglijarea studiului structurilor de date care folosesc pointeri (diverse liste înlănțuite și arbori) din câteva motive:

- Structuri cu pointeri sunt folosite în biblioteci de clase (Java, C# s.a.), chiar dacă pointerii sunt mascati sub formă de referințe;
- Listele înlănțuite și arborii cu pointeri pot constitui un “model” de structuri de date (reflectat în operațiile asupra acestor structuri), chiar și atunci când implementarea se face cu vectori. Un exemplu în acest sens îl constituie listele Lisp, care sunt văzute de toată lumea ca liste înlănțuite sau ca arbori, deși unele implementări de Lisp folosesc vectori (numărul de liste într-o aplicație Lisp poate fi foarte mare, iar diferența de memorie necesară pentru vectori sau pointeri poate deveni foarte importantă).

Pentru a ilustra acest ultim aspect vom exemplifica operatiile cu o listă înlănțuită ordonată cu santinelă dar fără pointeri (cu indici în cadrul unui vector). Evolutia listei în cazul secvenței de adăugare a valorilor 5,3,7,1 :

	val	leg	val	leg	val	leg	val	leg	val	leg
0		0		1		2		2		4
1			5	0	5	0	5	3	5	3
2					3		3	1	3	1
3							7	0	7	0
4									1	2

În poziția 0 se află mereu elementul santinelă, care conține în câmpul de legătură indicele elementului cu valoare minimă din listă. Elementul cu valoare maximă este ultimul din listă și are zero ca legătură.

Ca și la listele cu pointeri ordinea fizică (5,3,7,1) diferă de ordinea logică (1,3,5,7)

Lista se definește fie prin doi vectori (vector de valori și vector de legături), fie printr-un vector de structuri (cu câte două câmpuri), plus dimensiunea vectorilor:

```
typedef struct {
    int val[M], leg[M];    // valori elemente si legaturi intre elemente
    int n;                 // nr de elemente in lista = prima pozitie libera
} List;
```

Afisarea valorilor din vector se face în ordinea indicată de legături:

```
void printLV (List a) {
    int i=a.leg[0];        // porneste de la primul element cu date
    while (i>0) {
        printf ("%d ",a.val[i]); // valoare element din pozitia i
        i=a.leg[i];           // indice element urmator
    }
    printf ("\n");
}
```

Insertia în listă ordonată folosește metoda cu un pointer de la listele cu pointeri:

```
void insLV (List & a, int x) {
    // cauta elementul anterior celui cu x
    int i=0;
    while ( a.leg[i] !=0 && x > a.val[a.leg[i]])
        i=a.leg[i];
    // x legat dupa val[i]
    a.leg[a.n]=a.leg[i]; // succesorul lui x
    a.leg[i]= a.n;       // x va fi in pozitia n
    a.val[a.n]=x;        // valoare nod nou
    a.n++;               // noua pozitie libera din vector
}
```

4.7 COMBINATII DE LISTE SI VECTORI

Reducerea memoriei ocupate si a timpului de căutare într-o listă se poate face dacă în loc să memorăm un singur element de date într-un nod de listă vom memora un vector de elemente. Putem deosebi două situatii:

- Vectorii din fiecare nod al listei au acelasi număr de elemente (“unrolled lists”), număr corelat cu dimensiunea memoriilor cache;
- Vectorii din nodurile listei au dimensiuni în progresie geometrică, pornind de la ultimul către primul (“VLists”).

Economia de memorie se obtine prin reducerea numărului de pointeri care trebuie memorati. O listă de n date, grupate în vectori de câte m în fiecare nod necesită numai n/m pointeri, în loc de n pointeri ca într-o listă înlăntuită cu câte un element de date în fiecare nod. Numărul de pointeri este chiar mai mic într-o listă “VList”, unde sunt necesare numai $\log_2(n)$ noduri.

Câstigul de timp rezultă atât din accesul mai rapid după indice (pozitie), cât si din localizarea referintelor într-un vector (folosită de memorii “cache”). Din valoarea indicelui se poate calcula numărul nodului în care se află elementul dorit si pozitia elementului în vectorul din acel nod.

La căutarea într-o listă ordonată cu vectori de m elemente în noduri numărul de comparatii necesar pentru localizarea elementului din pozitia k este k/m în loc de k .

Listele cu noduri de aceeasi dimensiune (“UList”) pot fi si ele de două feluri:

- Liste neordonate, cu noduri complete (cu câte m elemente), în afara de ultimul;
- Liste ordonate, cu noduri având între $m/2$ si m elemente (un fel de arbori B).

Exemple de operatii cu o listă de noduri complete, de aceeasi dimensiune:

```
#define M 4          // nr maxim de elem pe nod (ales mic ptr teste)
typedef struct nod {  // un nod din lista
    int val[M];        // vector de date
    int m;             // nr de elem in fiecare nod ( m <=M)
    struct nod * leg;  // legatura la nodul urmator
} unod;

// initializare lista
void initL (unod * & lst){
    lst = (unod*)malloc (sizeof(unod));
    lst->m=0;  lst->leg=NULL;
}

// adaugare la sfarsit de lista
void addL (unod * lst, int x) {
    unod* nou;
    // cauta ultimul nod din lista
    while (lst->leg !=NULL)
        lst=lst->leg;
    if (lst->m < M)                // daca mai e loc in ultimul nod
        lst->val[lst->m++] = x;
    else {                        // daca ultimul nod e plin
        nou=(unod*) malloc(sizeof(unod)); // creare nod nou
        lst->leg=nou; nou->leg=NULL;      // completare nod nou
        nou->val[0]=x; nou->m=1;
    }
}

// acces pozitional, prin indice
int get (unod* lst, int k){
    int nod = k/M;                // nr nod ce contine datele
    for (; nod>0;nod--)
        lst=lst->leg;
    return lst->val[k%M];
}
```

Numărul de noduri dintr-o astfel de listă crește când se umple vectorul din nodul la care se adaugă, la adăugarea unui nou element la listă. Inițial se porneste cu un singur nod, de dimensiune dată (la "UList") sau de dimensiune 1 (la "VList").

La astfel de liste ori nu se mai elimină elemente ori se elimină numai de la sfârșitul listei, ca la o listă stivă.

În cazul listelor ordonate noile elemente se pot adăuga în orice nod și de aceea se prevede loc pentru adăugări (pentru reducerea numărului de operații). Adăugarea unui element la un nod plin are ca efect crearea unui nou nod și repartizarea elementelor în mod egal între cele două noduri succesive.

La eliminarea de elemente din listă este posibil ca numărul de noduri să scadă, atunci când vectorul dintr-un nod este ocupat mai puțin de jumătate.

Pentru listele cu noduri de dimensiune m , dacă numărul de elemente dintr-un nod scade sub $m/2$, se aduc elemente din nodurile vecine; dacă numărul de elemente din două noduri vecine este sub m atunci se reunesc cele două noduri într-un singur nod.

Exemplu de evoluție a unei liste ordonate cu maxim 3 valori pe nod după ce se adaugă diverse valori (bara '/' desparte noduri succesive din listă):

adaugă	lista
7	7
3	3,7
9	3,7,9
2	2,3 / 7,9
11	2,3 / 7,9,11
4	2,3,4 / 7,9,11
5	2,3 / 4,5 / 7,9,11
8	2,3 / 4,5 / 7,8 / 9,11
6	2,3 / 4,5,6 / 7,8 / 9,11

Algoritm de adăugare a unei valori x la o listă ordonată cu maxim m valori pe nod:

```

cauta nodul p in care va trebui introdus x ( anterior nodului cu valori > x)
dacă mai este loc în nodul p atunci
    adaugă x în nodul p
dacă nodul p este plin atunci {
    creare nod nou si legare nou după nodul p
    copiază ultimele  $m/2$  valori din p în nou
    dacă x trebuie pus în p atunci
        adauga x la nodul p
    altfel
        adauga x la nodul nou
}

```

Exemplu de funcții pentru operații cu o listă ordonată în care fiecare nod conține între $m/2$ și m valori (un caz particular de arbore 2-4):

```

// initializare lista
void initL (unod * & lst){
    lst = (unod*)malloc (sizeof(unod));           // creare nod gol
    lst->m=0; lst->leg=NULL;
}

// cauta locul unei noi valori in lista
unod* find (unod* lst, int x, int & idx) {          // idx=pozitie x in nod
    while (lst->leg !=NULL && x > lst->leg->val[0])
        lst=lst->leg;
    idx=0;
    while (idx < lst->m && x > lst->val[idx])
        idx++;
    // poate fi egal cu m daca x mai mare ca toate din lst
}

```


elemente si legătura la nodul următor. Numai primul nod (de dimensiune maximă) poate fi incomplet. Exemplu de definire:

```
#define M 1          // dimensiunea nodului minim
// def. nod de lista
typedef struct nod {
    int *val;        // vector de date (alocat dinamic)
    int max;         // nr maxim de elem in nod
    int i;           // indicele ultimului element adaugat in val
    struct nod * leg;
} vnod;
```

În cadrul unui nod elementele se adaugă de la sfârșitul vectorului către începutul său, deci valoarea lui *i* scade de la max la 0. Eliminarea primului element dintr-o listă VList se reduce la incrementarea valorii lui *i*.

Exemple de operatii cu o listă VList de numere întregi :

```
// initializare lista
void initL (vnod * & lst){
    lst = (vnod*)malloc (2*sizeof(int)+2*sizeof(void*));
    lst->val= (int*)calloc(M,sizeof(int));
    lst->i=lst->max=M;        // nod initial gol
    lst->leg=NULL;           // este si ultimul nod din lista
}
// adaugare la inceput de lista
void addL (vnod * & lst, int x) {    // se poate modifica inceputul listei !
    vnod* nou;
    if (lst->i == 0) {                // daca primul nod e plin
        nou=(vnod*) malloc(2*sizeof(int)+2*sizeof(void*));
        nou->max= 2* lst->max;        // dimensiune vector din nodul nou
        nou->val=(int*) calloc(nou->max,sizeof(int));
        nou->leg=lst;
        nou->i = nou->max;
        lst=nou;
    }
    lst->val [--lst->i]=x;    // adaug pe x la vectorul din acest nod
}
```

Pentru accesul unui element cu indice dat se compară succesiv valoarea acestui indice cu dimensiunea fiecărui nod, pentru a localiza nodul în care se află. Probabilitatea de a se afla în primul nod este cca $\frac{1}{2}$ (functie de numărul efectiv de elemente în primul nod), probabilitatea de a se afla în al doilea nod este $\frac{1}{4}$, s.a.m.d.

Dezavantajele listelor ce contin ca elemente vectori de valori apar la stergerea de elemente din listă; stergerea este simplă doar pentru elemente de la capetele listei si deci astfel de liste ar fi recomandate pentru implementarea tipului abstract “deque” (coadă cu operatii de adăugare si de stergere la ambele capete).

4.8 TIPUL ABSTRACT LISTĂ (SECVENTĂ)

Vectorii si listele înlănțuite sunt cele mai importante implementări ale tipului abstract “listă”. În literatura de specialitate si în realizarea bibliotecilor de clase există două abordări diferite, dar în esență echivalente, ale tipului abstract “listă”:

1) Tipul abstract “listă” este definit ca o colecție liniară de elemente, cu acces secvențial la elementul următor (si eventual la elementul precedent), după modelul listelor înlănțuite. Se folosește notiunea de element “curent” (poziție curentă în listă) si operatii de avans la elementul următor si respectiv la elementul precedent.

În această abordare, operațiile specifice clasei abstracte "List" sunt: citire sau modificare valoare din poziția curentă, inserare în poziția curentă, avans la elementul următor, poziționare pe elementul precedent, poziționare pe început/sfârșit de listă :

```
T getL (List & lst);           // valoare obiect din pozitia curentă
T setL (List & lst, T x);       // modifica valoare obiect din pozitia curentă
int insL (List & lst, T x);      // inserare x in pozitia curentă
T delL (List & lst);           // scoate si sterge valoare din pozitia curentă
void next (List & lst);         // pozitionare pe elementul următor
void first (List & lst);        // pozitionare la inceput de listă
```

Pot fi necesare două operații de inserție în listă: una după poziția curentă și alta înainte de poziția curentă. Poziția curentă se modifică după inserție.

Pentru detectarea sfârșitului de listă avem de ales între funcții separate care verifică aceste condiții ("end") și modificarea funcției "next" pentru a raporta prin rezultatul ei situația limită (1 = modificare reușită a poziției curente, 0 = nu se mai poate modifica poziția curentă, pentru că s-a ajuns la sfârșitul listei).

2) Tipul abstract listă este definit ca o colecție de elemente cu acces pozițional, printr-un indice întreg, la orice element din listă, după modelul vectorilor. Accesul prin indice este eficient numai pentru vectori, dar este posibil și pentru liste înlănțuite.

În acest caz, operațiile specifice tipului abstract "List" sunt: citire, modificare, inserare, stergere, toate într-o poziție dată (deci acces pozițional):

```
T getP (List & lst, int pos);    // valoare obiect din pozitia pos
int setP (List & lst, int pos, T x); // inlocuieste val din pozitia pos cu x
int insP (List & lst, int pos, T x); // inserare x in pozitia pos
T delP (List & lst, int pos);     // sterge din pos si scoate valoare
int findP (List & lst, Object x); // determina pozitia lui x in lista
```

Diferența dintre utilizarea celor două seturi de operații este aceeași cu diferența dintre utilizarea unui cursor intern tipului listă și utilizarea unui cursor (indice) extern listei și gestionat de programator.

În plus, listele suportă operații comune oricărei colecții:

```
initL (List &), emptyL(List), sizeL(List), addL(List&, T ), delL (List&, T ),
findL (List , T), printL (List).
```

O caracteristică a tipului abstract "Listă" este aceea că într-o listă nu se fac căutări frecvente după valoarea (conținutul) unui element, deși căutarea după conținut poate exista ca operație pentru orice colecție. În general se prelucrează secvențial o parte sau toate elementele unei liste. În orice caz, lista nu este considerată o structură de căutare ci doar o structură pentru memorarea temporară a unor date. Dintr-o listă se poate extrage o sublistă, definită prin indicii de început și de sfârșit.

O listă poate fi folosită pentru a memora rezultatul parcurgerii unei colecții de orice tip, deci rezultatul enumerării elementelor unui arbore, unui tabel de dispersie. Asupra acestei liste se poate aplica ulterior un filtru, care să selecteze numai acele elemente care satisfac o condiție. Elementele listei nu trebuie să fie distincte.

Parcurea (vizitarea) elementelor unei colecții este o operație frecventă, dar care depinde de modul de implementare al colecției. De exemplu, trecerea la elementul următor dintr-un vector se face prin incrementarea unui indice, dar avansul într-o listă se face prin modificarea unui pointer. Pentru a face operația de avans la elementul următor din colecție independentă de implementarea colecției s-a introdus noțiunea de iterator, ca mecanism de parcurgere a unei colecții.

Iteratorii se folosesc mai ales pentru colecții liniare (liste), dar și pentru structuri neliniare (pentru arbori binari, de exemplu).

Conceptul abstract de iterator poate fi implementat prin câteva funcții: initializare iterator (poziționare pe primul sau pe ultimul element din colecție), obținere element din poziția curentă și

avans la elementul următor (sau precedent), verificare sfârșit de colecție. Cursorul folosit de funcții pentru a memora poziția curentă în colecție poate fi o variabilă internă colecției sau o variabilă externă colecției (din aplicație).

Exemplu de afișare a unei liste folosind un iterator care folosește drept cursor o variabilă din structura “List” (cursor intern, invizibil pentru utilizatori):

```
typedef struct {
    Nod* cap, * crt;          // cap lista si pozitia curenta
} List;
// functii ale mecanismului iterator
// pozitionare pe primul element
void first (List & lst) {
    lst.crt=lst.cap→leg;
}
// daca exista un elem urmator in lista
int hasNext (List lst) {
    return lst.crt != NULL;
}
// pozitionare pe urmatorul element
T next (List & lst) {
    T x;
    if (! hasNext(lst)) return NULL;
    x=lst.crt→val;
    lst.crt=lst.crt→leg;
    return x;
}

// utilizare
...
T x; List list;              // List: lista abstracta de elem de tip T
first(list);                 // pozitionare pe primul element din colectie
while ( hasNext(list)) {    // cat timp mai sunt elemente in lista list
    x = next(list);          // x este elementul curent din lista list
    printT (x);              // sau orice operatii cu elementul x din lista
}
```

Un iterator oferă acces la elementul următor dintr-o colecție și ascunde detaliile de parcurgere a colecției, dar limitează operațiile asupra colecției (de exemplu eliminarea elementului din poziția curentă sau inserția unui nou element în poziția curentă nu sunt permise de obicei prin iterator deoarece pot veni în conflict cu alte operații de modificare a colecției și afectează poziția curentă). În plus iterarea pe arbori este dificilă și posibil ineficientă.

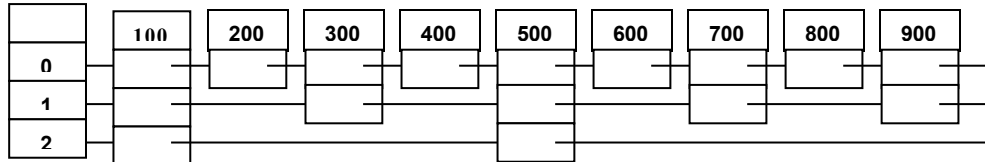
O alternativă este programarea explicită a vizitării elementelor colecției cu apelarea unei funcții de prelucrare la fiecare element vizitat; funcția apelată se numește și “aplicator” pentru că se aplică fiecărui element din colecție. Exemplu:

```
// tip functie aplicator, cu argument adresa date
typedef void (*func)(void *);
// functie de vizitare elemente lista si apel aplicator
void iter ( lista lst, func fp) {
    while ( lst != NULL) { // repeta cat mai sunt elemente in lista
        (*fp) (lst→ptr);   // apel functie aplicator (lst→ptr este adresa datelor)
        lst=lst→leg;       // avans la elementul urmator
    }
}
```

4.9 LISTE “SKIP”

Dezavantajul principal al listelor înlănțuite este timpul de căutare a unei valori date, prin acces secvențial; acest timp este proporțional cu lungimea listei. De aceea s-a propus o soluție de reducere a acestui timp prin utilizarea de pointeri suplimentari în anumite elemente ale listei. Listele denumite “skip list” sunt liste ordonate cu timp de căutare comparabil cu alte structuri de căutare (arbori binari și tabele de dispersie). Timpul mediu de căutare este de ordinul $O(\lg n)$, dar cazul cel mai defavorabil este de ordinul $O(n)$ (spre deosebire de arbori binari echilibrați unde este tot $O(\lg n)$).

O listă skip poate fi privită ca fiind formată din mai multe liste paralele, cu anumite elemente comune.



Adresele de legătură între elemente sunt situate pe câteva niveluri: pe nivelul 0 este legătura la elementul imediat următor din listă, pe nivelul 1 este o legătură la un element aflat la o distanță d_1 , pe nivelul 2 este o legătură la un element aflat la o distanță $d_2 > d_1$ s.a.m.d. Adresele de pe nivelurile 1,2,3 și următoarele permit “salturi” în listă pentru a ajunge mai repede la elementul căutat.

Căutarea începe pe nivelul maxim și se oprește la un element cu valoare mai mică decât cel căutat, după care continuă pe nivelul imediat inferior s.a.m.d. Pentru exemplul din desen, căutarea valorii 800 începe pe nivelul 2, “sare” direct și se oprește la elementul cu valoarea 500; se trece apoi pe nivelul 1 și se sare la elementul cu valoarea 700, după care se trece pe nivelul 0 și se caută secvențial între 700 și 900.

Pointerii pe nivelurile 1,2 etc. împart lista în subliste de dimensiuni apropiate, cu posibilitatea de a sări peste orice sublistă pentru a ajunge la elementul căutat.

Pentru simplificarea operațiilor cu liste skip, ele au un element santinelă (care conține numărul maxim de pointeri) și un element terminator cu o valoare superioară tuturor valorilor din listă sau care este același cu santinela (liste circulare).

Fiecare nod conține un vector de pointeri la elementele următoare de pe câteva niveluri (dar nu și dimensiunea acestui vector) și un câmp de date.

Exemplu de definire a unei liste cu salturi :

```
#define MAXLEVEL 11 // limita sup. ptr nr maxim de pointeri pe nod
typedef struct Nod { // structura care definește un nod de lista
    int val;          // date din fiecare nod
    struct Nod *leg[11]; // legături la nodurile următoare
} Nod;
```

De observat că ordinea câmpurilor în structura Nod este importantă, pentru că vectorul de pointeri poate avea o dimensiune variabilă și deci trebuie să fie ultimul câmp din structură.

Funcțiile următoare lucrează cu o listă circulară, în care ultimul nod de pe fiecare nivel conține adresa primului nod (santinela).

```
// initializare lista
void initL(Nod*& list) {
    int i;
    list = (Node*)malloc(sizeof(Node) + MAXLEVEL*sizeof(Node *));
    for (i = 0; i <= MAXLEVEL; i++) // initializare pointeri din santinela
        list->leg[i] = list;        // listele sunt circulare
    list->val = 0;                  // nivelul curent al listei in santinela
}

// cauta in lista list o valoare data x
Nod findL(Nod* list, int x) {
    int i, level=list->val;
    Nod *p = list;                // lista cu sentinala
```

```

for (i = level; i >= 0; i--)      // se incepe cu nivelul maxim
    while (p->leg[i] != list && x > p->leg[i] ->val)
        p = p->leg[i];
p = p->leg[0];      // cautarea s-a oprit cand x >= p->leg[i]->val
if (p != list && p->val == x) return p; // daca x gasit la adresa p
return NULL;      // daca x negasit
}

```

Nivelul listei (număr maxim de pointeri pe nod) poate crește la adăugarea de noduri și poate scădea la eliminarea de noduri din listă. Pentru a stabili numărul de pointeri la un nod nou (în funcția de adăugare) se folosește un generator de numere aleatoare în intervalul [0,1]: dacă iese 0 nu se adaugă alți pointeri la nod, dacă iese 1 atunci se adaugă un nou pointer și se repetă generarea unui nou număr aleator, până când iese un 0. În plus, mai punem și condiția ca nivelul să nu crească cu mai mult de 1 la o adăugare de element.

Probabilitatea ca un nod să aibă un pointer pe nivelul 1 este $\frac{1}{2}$, probabilitatea să aibă un pointer pe nivelul 2 este $\frac{1}{4}$ s.a.m.d.

Funcția de inserție în listă a unei valori x va realiza următoarele operații:

```

cauta pozitia de pe nivelul 0 unde trebuie inserat x
determina nivelul noului nod (probabilistic)
daca e nevoie se creste nivel maxim pe lista
creare nod nou cu completare legături la nodul următor de pe fiecare nivel

```

Afisarea unei liste skip se face folosind numai pointerii de pe nivelul 0, la fel ca afisarea unei liste simplu înlănțuite.

Pentru a facilita înțelegerea operației de inserție vom exemplifica cu o listă skip în care pot exista maxim două niveluri, deci un nod poate conține unul sau doi pointeri:

```

typedef struct node {
    int val;      // valoare memorata in nod
    struct node *leg[1]; // vector extensibil de legaturi pe fiecare nivel
} Nod;

// initializare: creare nod santinela
void initL(Nod* & hdr) {
    hdr = (Nod*) malloc ( sizeof(Nod)+ sizeof(Nod*)); // nod santinela
    hdr->leg[0] = hdr->leg[1] = NULL;
}

// inserție valoare in lista
void *insL(Nod* head, int x) {
    Nod *p1, *p0, *nou;
    int level= rand()%2;      // determina nivel nod nou (0 sau 1)
    // creare nod nou
    nou = (Nod*) malloc ( sizeof(Nod)+ level*sizeof(Nod*));
    nou->val=x;
    // cauta pe nivelul 1 nod cu valoarea x
    p1 = head;
    while ( p1->leg[1] != NULL && x > p1->leg[1] ->val )
        p1 = p1->leg[1];
    // cauta valoarea x pe nivelul 0
    p0 = p1;
    while ( p0->leg[0] != NULL && x > p0->leg[0] ->val )
        p0 = p0->leg[0];
    // leaga nou pe nivelul 0
    nou->leg[0]=p0->leg[0]; p0->leg[0]=nou;
    if (level == 1) {      // daca nodul nou este si pe nivelul 1
        // leaga nou pe nivelul 1
        nou->leg[1]=p1->leg[1]; p1->leg[1]=nou;
    }
}

```

```
}
}
```

Folosirea unui nod terminal al ambelor liste, cu valoarea maximă posibilă, simplifică codul operațiilor de inserție și de eliminare noduri într-o listă skip.

4.10 LISTE NELINIARE

Într-o listă generală (neliniară) elementele listei pot fi de două tipuri: elemente cu date (cu pointeri la date) și elemente cu pointeri la subliste. O listă care poate conține subliste, pe oricâte niveluri de adâncime, este o listă neliniară.

Limbajul Lisp ("List Processing") folosește liste neliniare, care pot conține atât valori atomice (numere, siruri) cât și alte (sub)liste. Listele generale se reprezintă în limbajul Lisp prin expresii; o expresie Lisp conține un număr oarecare de elemente (posibil zero), încadrate între paranteze și separate prin spații. Un element poate fi un atom (o valoare numerică sau un sir) sau o expresie Lisp.

În aceste liste se pot memora expresii aritmetice, propoziții și fraze dintr-un limbaj natural sau chiar programe Lisp. Exemple de liste ce corespund unor expresii aritmetice în forma prefixată (operatorul precede operandii):

(- 5 3)	5-3	o expresie cu 3 atomi
(+ 1 2 3 4)	1+2+3+4	o expresie cu 5 atomi
(+ 1 (+ 2 (+ 3 4)))	1+2+3+4	o expresie cu 2 atomi și o subexpresie
(/ (+ 5 3) (- 6 4))	(5+3) / (6-4)	o expresie cu un atom și 2 subexpresii

Fiecare element al unei liste Lisp conține două câmpuri, numite CAR (primul element din listă) și CDR (celelalte elemente sau restul listei). Primul element dintr-o listă este de obicei o funcție sau un operator, iar celelalte elemente sunt operanzi.

Imaginea unei expresii Lisp ca listă neliniară (aici cu două subliste):

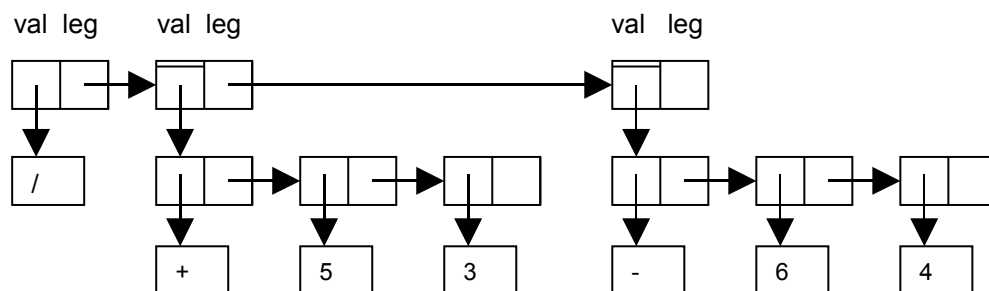
```

/ ----- o ----- o
|           |           |
+ --- 5 --- 3   - --- 6 --- 4

```

O implementare eficientă a unei liste Lisp folosește două tipuri de noduri: noduri cu date (cu pointeri la date) și noduri cu adresa unei subliste. Este posibilă și utilizarea unui singur tip de nod cu câte 3 pointeri: la date, la nodul din dreapta (din aceeași listă) și la nodul de jos (sublista nodului).

În figura următoare am considerat că elementele atomice memorează un pointer la date și nu chiar valoarea datelor, pentru a permite siruri de caractere ca valori.



Structura anterioară corespunde expresiei fără paranteze exterioare $/(+53)(-64)$ iar prezenta parantezelor pe toată expresia (cum este de fapt în Lisp) necesită adăugarea unui element initial de tip 1, cu NULL în câmpul "leg" și cu adresa elementului atomic '/' în câmpul "val".

Urmează o definiție posibilă a unui nod de listă Lisp cu doi pointeri și un câmp care arată cum trebuie interpretat primul pointer: ca adresă a unui atom sau ca adresă a unei subliste:

```

struct nod {
    char tip;           // tip nod (interpretare camp "val")
    void* val;          // pointer la o valoare (atom) sau la o sublista
    struct nod* leg;    // succesorul acestui nod in lista
};

```

Din cauza alinierii la multiplu de 4 octeti (pentru procesoare cu acces la memorie pe 32 de biti), structura anterioară va ocupa 12 octeti (în Dev-Cpp). Folosind câmpuri de biti în structuri putem face ca un nod să ocupe numai 8 octeti:

```

typedef struct nod {
    unsigned int tip:1 ;      // tip nod (0=atom,1=lista)
    unsigned int val:31;      // adresa atom sau sublista
    struct nod* leg;          // adresa urmatorului element
} nod;

```

Interpretarea adresei din câmpul "val" depinde de câmpul "tip" și necesită o conversie înainte de a fi utilizată. Exemplu de funcție care afișează o listă Lisp cu atomi și siruri, sub forma unei expresii cu paranteze (în sintaxa limbajului Lisp):

```

// afisare lista de liste
void printLisp (nod* p) {      // p este adresa de început a listei
    if (p ==NULL) return;      // iesire din recursivitate
    if (p->tip==0)              // daca nod atom
        printf("%s ",(char*)p->val); // scrie valoare atom
    else {                     // daca nod sublista
        printf("(");           // atunci scrie o expresie intre paranteze
        printLisp ((nod*)p->val); // scrie sublista nod p
        printf(")");
    }
    printLisp(p->leg);          // scrie restul listei (dupa nodul p )
}

```

Expresiile Lisp ce reprezintă expresii aritmetice sunt cazuri particulare ale unor expresii ce reprezintă apeluri de funcții (în notatia prefixată) : (f x y ...). Mai întâi se evaluează primul element (funcția f), apoi argumentele funcției (x,y,...), și în final se aplică funcția valorilor argumentelor.

Exemplu de funcție pentru evaluarea unei expresii aritmetice cu orice număr de operanzi de o singură cifră:

```

// evaluare expr prefixata cu orice numar de operanzi
int eval ( nod* p ) {
    int x,z; char op;
    // evaluarea primului element al listei (functie/operator)
    op= *(char*)p->val;      // primul element este operator aritmetic
    p=p->leg; z=eval1(p);     // primul operand
    while (p->leg !=NULL){    // repeta cat timp mai sunt operanzi
        p=p->leg;
        x=eval1(p);          // urmatorul operand
        z=calc (op, z, x ); // aplica operator op la operanzii x si y
    }
    return z;
}

```

Funcția eval1 evaluează un singur operand (o cifră sau o listă între paranteze):

```

int eval1 (nod* p) {
    int eval(nod*);
}

```

```

if (p→tip==0)                // daca e un atom
    return *(char*)p→val -'0'; // valoare operand (o cifra) in x
else                          // daca este o sublista
    return eval ((nod*)p→val); // rezultat evaluare sublista in x
}

```

Cele două functii (eval si eval1) pot fi reunite într-una singură.

Pentru crearea unei liste dintr-o expresie Lisp vom defini mai întâi două functii auxiliare folosite la crearea unui singur nod de listă:

```

// creare adresa ptr un sir de un caracter
char * cdup (char c) {
    char* pc=(char*) malloc (2);
    *pc=c; *(pc+1)=0;    // sir terminat cu zero
    return pc;
}

// creare nod de lista
nod* newnode (char t, void* p, nod* cdr) {
    nod * nou = new nod;    // nou= (nod*)malloc( sizeof(nod));
    nou→tip= t; nou→val=(unsigned int) p; nou→leg=cdr;
    return nou;
}

```

Exemplu de functie recursivă pentru crearea unei liste dintr-o expresie Lisp, cu rezultat adresa noului nod creat (care este si adresa listei care începe cu acel nod), după ce s-au eliminat parantezele exterioare expresiei:

```

nod* build ( char * & s) {    // adresa 's' se modifica in functie !
    while (*s && isspace(*s) ) // ignora spatii albe
        ++s;
    if (*s==0 ) return 0;      // daca sfarsit de expresie
    char c= *s++;              // un caracter din expresie
    if (c==' ') return 0;      // sfarsit subexpresie
    if(c=='(') {               // daca inceput sublista
        nod* val=build(s);     // sublista de jos
        nod* leg =build(s);    // sublista din dreapta
        return newnode (1,val,leg); // creare nod sublista
    }
    else                       // daca c este atom
        return newnode (0,cdup(c),build(s)); // creare nod atom
}

```

Orice listă Lisp se poate reprezenta si ca arbore binar având toate nodurile de acelasi tip (cu un pointer la date si doi pointeri la succesori), dar pentru expresii se foloseste o altă reprezentare prin arbori binari (descrișă în capitolul de arbori).

Capitolul 5

MULTIMI SI DICTIONARE

5.1 TIPUL ABSTRACT "MULTIME"

Tipul abstract multime ("Set") poate fi definit ca o colectie de valori distincte (toate de aceasi tip) , cu toate operatiile asociate colectiilor. Fată de alte colectii abstracte, multimea are drept caracteristică definitorie căutarea unui element după conținut, căutare care este o operatie frecventă si de aceea trebuie să necesite un timp cât mai mic. Principalele operatii cu o multime sunt:

```
void initS ( Set & s );      // creare multime vidă (initializare )
int emptyS ( Set s );      // test de multime vidă : 1 daca s multime vida
int findS (Set s ,T x);     // 1 daca x apartine multimii s , 0 altfel
void addS ( Set & s, T x);  // adauga pe x la multimea s
void delS ( Set & s, T x);  // elimina valoarea x din multimea s
void printS ( Set s );     // afisarea conținutului unei multimi s
int sizeS( Set s);        // dimensiune multime
```

Pentru anumite aplicatii sunt necesare si operatii cu două multimi:

```
void addAll (Set & s1, Set s2);    // reuniunea a două multimi
void retainAll (Set & s1, Set s2); // intersectia a două multimi
void removeAll (Set & s1, Set s2); // diferență de multimi s1-s2
int containsAll (Set s1, Set s2);  // 1 daca s1 contine pe s2
```

Multimea nouă (reuniune, intersectie, diferență) înlocuieste primul operand (multimea s1). Nu există operatia de copiere a unei multimi într-o altă multime, dar ea se poate realiza prin initializare si reuniune multime vidă cu multimea sursă :

```
initS (s1); addAll (s1,s2); // copiere s2 in s1
```

Nu există comparatie de multimi la egalitate, dar se poate compara diferenta simetrică a două multimi cu multimea vidă, sau se poate scrie o functie mai performantă pentru această operatie.

Tipul "multime" poate fi implementat prin orice structură de date: vector, listă cu legături sau multime de biti dacă sunt putine elemente în multime. Cea mai simplă implementare a tipului abstract multime este un vector neordonat cu adăugare la sfârșit. Realizarea tipului multime ca o listă înlăntuită se recomandă pentru colectii de mai multe multimi, cu conținut variabil.

Dacă sunt multe elemente atunci se folosesc acele implementări care realizează un timp de căutare minim: tabel de dispersie si arbore de căutare echilibrat.

Anumite operatii se pot realiza mai eficient dacă multimele sunt ordonate: căutare element în multime, reuniune de multimi, afisare multime în ordinea cheilor s.a.

Pentru cazul particular al unei multimi de numere întregi cu valori într-un domeniu cunoscut si restrâns se foloseste si implementarea printr-un vector de biti, în care fiecare bit memorează prezenta sau absenta unui element (potential) în multime. Bitul din pozitia k este 1 dacă valoarea k apartine multimii si este 0 dacă valoarea k nu apartine multimii. Această reprezentare ocupă puțină memorie si permite cel mai bun timp pentru operatii cu multimi (nu se face o căutare pentru verificarea apartenentei unei valori x la o multime, ci doar se testează bitul din pozitia x).

Pentru multimi realizate ca vectori sau ca liste înlăntuite, operatiile cu o singură multime se reduc la operatii cu un vector sau cu o listă: initializare, căutare, adăugare, eliminare, afisare colectie, dimensiune multime si/sau test de multime vidă.

O multime cu valori multiple ("Multiset") poate contine elemente cu aceeasi valoare, dar nu este o listă (abstractă) pentru că nu permite accesul direct la elemente. Justificarea existentei unei clase Multiset în limbaje ca Java si C++ este aceea că prin implementarea acestui tip cu un dictionar se

reduce timpul necesar anumitor operatii uzuale cu multimi: compararea la egalitate a două multimi cu elemente multiple si eventual neordonate, obtinerea numărului de aparitii a unui element cu valoare dată si eliminarea tuturor aparitiilor unui element dat.

Ideea este de a memora fiecare element distinct o singură dată dar împreună cu el se memorează si numărul de aparitii în multime; acesta este un dictionar având drept chei elemente multimii si drept valori asociate numărul de aparitii (un întreg pozitiv).

5.2 APLICATIE: ACOPERIRE OPTIMĂ CU MULTIMI

Problema acoperirii optime cu multimi ("set cover") este o problemă de optimizare si se formulează astfel: Se dă o multime scop S si o colectie C de n multimi candidat, astfel că orice element din S apartine cel puțin unei multimi candidat; se cere să se determine numărul minim de multimi candidat care acoperă complet pe S (deci reuniunea acestor multimi candidat contine toate elementele lui S).

Exemplu de date si rezultate :

$S = \{ 1,2,3,4,5 \}$, $n=4$

$C[1]= \{ 2 \}$, $C[2]=\{1,3,5\}$, $C[3]= \{ 2,3 \}$, $C[4]= \{2,4\}$

Solutia optimă este :

$\{ C[2], C[4] \}$

Algoritmul "greedy" pentru această problemă selectează, la fiecare pas, acea multime $C[k]$ care acoperă cele mai multe elemente neacoperite încă din S (intersectia lui $C[k]$ cu S contine numărul maxim de elemente). După alegerea unei multimi $C[k]$ se modifică multimea scop S, eliminând din S elementele acoperite de multimea $C[k]$ (sau se reunesc candidatii selectati într-o multime auxiliară A). Ciclul de selectie se opreste atunci când multimea S devine vidă (sau când A contine pe S).

Exemplu de date pentru care algoritmul "greedy" nu determină solutia optimă :

$S = \{ 1,2,3,4,5,6 \}$, $n=4$;

$C[1]= \{ 2,3,4 \}$, $C[2]=\{ 1,2,3 \}$, $C[3]= \{ 4,5,6 \}$, $C[4]=\{1\}$

Solutia greedy este $\{ C[1], C[3], C[2] \}$, dar solutia optimă este $\{ C[2], C[3] \}$

În programul următor se alege, la fiecare pas, candidatul optim, adică cel pentru care intersectia cu multimea scop are dimensiunea maximă. După afisarea acelei multimi se elimină din multimea scop elementele acoperite de multimea selectată.

Colectia de multimi este la rândul ei o multime (sau o listă) de multimi, dar pentru simplificare vom folosi un vector de multimi. Altfel spus, pentru multimea C am ales direct implementarea printr-un vector. Pentru fiecare din multimile $C[i]$ si pentru multimea scop S putem alege o implementare prin liste înlântuite sau prin vectori, dar această decizie poate fi amânată după programarea algoritmului greedy:

```
Set cand[100], scop, aux;          // multimi candidat, scop si o multime de lucru
int n;                             // n= nr. de multimi candidat
void setcover () {
    int i,imax,dmax,k,d;
    do {
        dmax=0;                     // dmax = dim. maxima a unei intersectii
        for (i=1 ;i<=n ; i++) {
            initS (aux); addAll (aux,scop); // aux = scop
            retainAll (aux,cand[i]);        // intersectie aux cu cand[i]
            d= size (aux);                  // dimensiune multime intersectie
            if (dmax < d) {                 // retine indice candidat cu inters. maxima
                dmax=d; imax=i;
            }
        }
    }
    printf ("%d ", imax); printS (cand[imax]); // afiseaza candidat
```

```

    removeAll (scop,cand[imax]); // elimina elemente acoperite de candidat
} while ( ! emptyS(scop));
}

```

Se poate verifica dacă problema admite soluție astfel: se reunesc multimele candidat și se verifică dacă mulțimea scop este conținută în reuniunea candidaților:

```

void main () {
    int i;
    getData();           // citește multimi scop și candidat
    initS (aux);         // crează multime vidă "aux"
    for (i=1;i<=n;i++)   // reuniune multimi candidat
        addAll (aux,cand[i]);
    if (! containsAll(aux,scop))
        printf (" nu există soluție \n");
    else
        setcover();
}

```

5.3 TIPUL "COLECȚIE DE MULTIMI DISJUNCTE"

Unele aplicații necesită gruparea elementelor unei multimi în mai multe submultimi disjuncte. Conținutul și chiar numărul multimilor din colecție se modifică de obicei pe parcursul execuției programului. Astfel de aplicații sunt determinarea componentelor (subgrafurilor) conexe ale unui graf și determinarea claselor de echivalență pe baza unor relații de echivalență.

O multime din colecție nu este identificată printr-un nume sau un număr, ci printr-un element care aparține multimei. De exemplu, o componentă conexă dintr-un graf este identificată printr-un număr de nod aflat în componenta respectivă.

În literatură se folosesc mai multe nume diferite pentru acest tip de date: "Disjoint Sets", "Union and Find Sets", "Merge and Find Sets".

Operațiile asociate tipului abstract "colecție de multimi disjuncte" sunt:

- Inițializare colecție c de n multimi, fiecare multime k cu o valoare k: init (c,n)
- Găsirea multimei dintr-o colecție c care conține o valoare dată x: find (c,x)
- Reuniunea multimilor din colecția c ce conțin valorile x și y : union (c,x,y)

În aplicația de componente conexe se creează inițial în colecție câte o multime pentru fiecare nod din graf, iar apoi se reduce treptat numărul de multimi prin analiza muchiilor existente. După epuizarea listei de muchii fiecare multime din colecție reprezintă un subgraf conex. Dacă graful dat este conex, atunci în final colecția va conține o singură multime.

Cea mai bună implementare pentru "Disjoint Sets" folosește tot un singur vector de întregi, dar acest vector reprezintă o pădure de arbori. Elementele vectorului sunt indici (adrese) din același vector cu semnificația de pointeri către nodul părinte. Fiecare multime este un arbore în care fiecare nod (element) conține o legătură la părintele său, dar nu și legături către fii săi. Rădăcina fiecărui arbore poate conține ca legătură la părinte fie chiar adresa sa, fie o valoare nefolosită ca indice (-1).

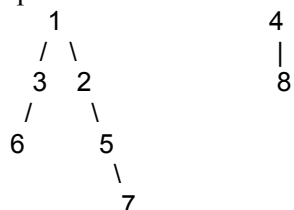
Pentru datele folosite anterior (8 vârfuri în 3 componente conexe), starea finală a vectorului ce reprezintă colecția și arborii corespunzători arată astfel:

valoare	1	2	3	4	5	6	7	8
legătura	-1	-1	1	-1	2	3	5	4

1		2		4
3		5		8
6		7		

În funcție de codul folosit sunt posibile și alte variante, dar tot cu trei arbori și cu aceleași noduri (se modifică doar rădăcina și structura arborilor).

Dacă se mai adaugă o muchie 3-7 atunci se reunesc arborii cu rădăcinile în 1 și 2 într-un singur arbore, iar în vectorul ce reprezintă cei doi arbori rămași se modifică legătura lui 2 ($p[2]=1$).



Găsirea multimii care conține o valoare dată x se reduce la aflarea rădăcinii arborelui în care se află x , mergând în sus de la x către rădăcină. Reunirea arborilor ce conțin un x și un y se face prin legarea rădăcinii arborelui y ca fiu al rădăcinii arborelui x (sau al arborelui lui x la arborele lui y).

Urmează funcțiile ce realizează operațiile specifice tipului “Disjoint Sets”:

```

typedef struct {
    int p[M]; // legături la noduri părinte
    int n;    // dimensiune vector
} ds;
// initializare colecție
void init (ds & c, int n) {
    int i;
    c.n=n;
    for (i=1; i<=n; i++)
        c.p[i]=-1; // rădăcina conține legătura -1
}
// determină mulțimea care conține pe x
int find ( ds c, int x) {
    int i=x;
    while ( c.p[i] > 0)
        i=c.p[i];
    return i;
}
// reunire clase ce conțin valorile x și y
void unif ( ds & c, int x, int y) {
    int cx, cy;
    cx=find(c,x); cy=find(c,y);
    if (cx != cy)
        c.p[cy]=cx;
}
    
```

În această variantă operația de căutare are un timp proporțional cu adâncimea arborelui, iar durata operației de reunire este practic aceeași cu durata lui “find”. Pentru reducerea în continuare a duratei operației “find” s-au propus metode pentru reducerea adâncimii arborilor. Modificările au loc în algoritm, dar structura de date rămâne practic neschimbată (tot un vector de indici către noduri părinte).

Prima idee este ca la reunirea a doi arbori în unul singur să se adauge arborele mai mic (cu mai puține noduri) la arborele mai mare (cu mai multe noduri). O soluție simplă este ca numărul de noduri dintr-un arbore să se păstreze în nodul rădăcină, ca număr negativ. Funcția de reunire de mulțimi va arăta astfel:

```

void unif ( ds & c, int x, int y) {
    int cx, cy;
    cx=find(c,x); cy=find(c,y); // indici noduri rădăcina
    if (cx == cy) return;       // dacă x și y în același arbore
    if ( c.p[cx] <= c.p[cy]) {  // dacă arborele cx este mai mic ca cy
        c.p[cy]=cx;
    }
    else {
        c.p[cx]=cy;
    }
}
    
```

```

    c.p[cx] += c.p[cy];           // actualizare nr de noduri din cx
    c.p[cy]=cx;                  // leaga radacina cy ca fiu al nodului cx
} else {                         // daca arborele cy este mai mic ca cx
    c.p[cy] += c.p[cx];          // actualizare nr de noduri din cy
    c.p[cx]=cy;                  // cy devine parintele lui cx
}
}

```

A doua idee este ca în timpul căutării într-un arbore să se modifice legăturile astfel ca toate nodurile din arbore să fie legate direct la rădăcina arborelui:

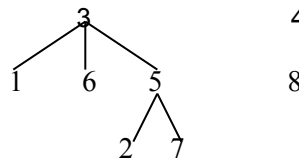
```

int find ( ds c, int x) {
    if( c.p[x] < 0 )
        return x;
    return c.p[x]=find (c, c.p[x]);
}

```

În cazul unui graf cu 8 vârfuri și muchiile 3-6, 5-7, 1-3, 2-5, 4-8, 1-6, 3-7 vor fi doi arbori cu 6 și respectiv 2 noduri, iar vectorul p de legături la părinți va arăta astfel:

i	1	2	3	4	5	6	7	8
p[i]	3	5	-6	-2	3	3	5	4



Dacă se mai adaugă o muchie 2-4 atunci înălțimea arborelui rămas va fi tot 2 iar nodul 4 va avea ca părinte rădăcina 3.

Reuniunea după dimensiunea arborilor are drept efect proprietatea că nici un arbore cu n noduri nu are înălțime mai mare ca $\log(n)$. Prin reunirea a doi arbori numărul de noduri din arborele rezultat crește cel puțin de două ori (se dublează), dar înălțimea sa crește numai cu 1. Deci raportul dintre înălțimea unui arbore și numărul său de noduri va fi mereu de ordinul $\log_2(n)$. Rezultă că și timpul mediu de căutare într-un arbore cu n noduri va crește doar logaritmice în raport cu dimensiunea sa.

Ca soluție alternativă se poate păstra înălțimea fiecărui arbore în locul numărului de noduri, pentru a adăuga arborele cu înălțime mai mică la arborele cu înălțime mai mare.

5.4 TIPUL ABSTRACT "DICTIONAR"

Un dicționar ("map"), numit și tabel asociativ, este o colecție de perechi cheie - valoare, în care cheile sunt distincte și sunt folosite pentru regăsirea rapidă a valorilor asociate. Un dicționar este o structură pentru căutare rapidă (ca și mulțimea) având aceleași implementări: vector sau listă de înregistrări dacă sunt puține chei și tabel de dispersie ("hash") sau arbore binar echilibrat de căutare dacă sunt multe chei și timpul de căutare este important. Cheia poate fi de orice tip.

Un dicționar poate fi privit ca o mulțime de perechi cheie-valoare, iar o mulțime poate fi privită ca un dicționar în care cheia și valoarea sunt egale. Din acest motiv și implementările principale ale celor două tipuri abstracte sunt aceleași.

Operațiile principale specifice unui dicționar, după modelul Java, sunt :

Introducerea unei perechi cheie-valoare într-un dicționar:

```
int putD (Map & M, Tk key, Tv val);
```

Extragerea dintr-un dicționar a valorii asociate unei chei date:

```
Tv getD ( Map M, Tk key);
```

Eliminarea unei perechi cu cheie dată dintr-un dicționar:

```
int delD (Map & M, Tk key);
```

Am notat cu "Map" tipul abstract dictionar, cu Tk tipul cheii si cu Tv tipul valorii asociate; ele depind de datele folosite în fiecare aplicatie si pot fi diferite sau identice. Putem înlocui tipurile Tk si Tv cu tipul generic "void*", cu pretul unor complicatii în programul de aplicatie care foloseste functiile "getD" si "putD".

Rezultatul functiilor este 1 (adevărat) dacă cheia "key" este găsită sau 0 (fals) dacă cheia "key" nu este găsită. Functia "putD" modifică dictionarul, prin adăugarea unei noi perechi la dictionar sau prin modificarea valorii asociate unei chei existente.

Functiile "getD" si "putD" compară cheia primită cu cheile din dictionar, iar realizarea operatiei de comparare depinde de tipul cheilor. Adresa functiei de comparare poate fi transmisă direct acestor functii sau la initializarea dictionarului.

La aceste operatii trebuie adăugate si cele de initializare dictionar (initD) si de afisare dictionar (printD).

Este importantă precizarea că executia functiei "putD" cu o cheie existentă în dictionar nu adaugă un nou element (nu pot exista mai multe perechi cu aceeasi cheie) ci doar modifică valoarea asociată cheii existente. De aici si numele functiei "put" (pune în dictionar) în loc de "add" (adăugare), ca la multimi. In functie de implementare, operatia se poate realiza prin înlocuirea valorii asociate cheii existente, sau prin eliminarea perechii cu aceeasi cheie, urmată de adăugarea unei noi perechi.

Operatiile "getD" si "putD" necesită o căutare în dictionar a cheii primite ca argument, iar această operatie poate fi realizată ca o functie separată.

Implementările cele mai bune pentru dictionare sunt:

- Tabel de dispersie ("Hash table")
- Arbori binari echilibrati de diferite tipuri
- Liste "skip"

Ultimele două solutii permit si mentinerea dictionarului în ordinea cheilor, ceea ce le recomandă pentru dictionare ordonate.

Pentru un dictionar cu număr mic de chei se poate folosi si o implementare simplă printr-o listă înlântuită, sau prin doi vectori (de chei si de valori) sau printr-un singur vector de structuri, care poate fi si ordonat dacă este nevoie.

De cele mai multe ori fiecare cheie are asociată o singură valoare, dar există si situatii când o cheie are asociată o listă de valori. Un exemplu este un index de termeni de la finalul unei cărți tehnice, în care fiecare cuvânt important (termen tehnic) este trecut împreună cu numerele paginilor unde apare acel cuvânt. Un alt exemplu este o listă de referinte încrucisate, cu fiecare identificator dintr-un program sursă însoțit de numerele liniilor unde este definit si folosit acel identificator.

Un astfel de dictionar este numit dictionar cu valori multiple sau dictionar cu chei multiple sau multi-dictionar ("Multimap"). Un exemplu este crearea unei liste de referinte încrucisate care arată în ce linii dintr-un text sursă este folosit fiecare identificator. Exemplu de date initiale:

unu / doi / unu / doi / doi / trei / doi / trei / unu

Rezultatele programului pot arăta astfel (ordinea cuvintelor poate fi alta):

```
unu  1, 3, 9
doi   2, 4, 5, 7
trei  6, 8
```

Cuvintele reprezintă cheile iar numerele de linii sunt valorile asociate unei chei.

Putem privi această listă si ca un dictionar cu chei multiple, astfel:

unu 1 / doi 2 / unu 3 / doi 4 / doi 5 / trei 6 / doi 7 / trei 8

Oricare din implementările unui dictionar simplu poate fi folosită si pentru un multidictionar, dacă se înlocuieste valoarea asociată unei chei cu lista valorilor asociate acelei chei (un pointer la o listă înlântuită, în limbajul C).

O variantă de dictionar este dictionarul bidirectional (reversibil), numit "BiMap", în care si valorile sunt distincte putând fi folosite drept chei de căutare într-un dictionar "invers". La încercarea de adăugare a unei perechi cheie-valoare ("putD") se poate elimina o pereche anterioară cu aceeasi valoare si deci dimensiunea dictionarului BiMap poate creste, poate rămâne neschimbată (dacă există

o pereche cu aceeași cheie dar cu valoare diferită) sau poate să scadă (dacă există o pereche cu aceeași cheie și o pereche cu aceeași valoare). Structurile folosite de un BiMap nu diferă de cele pentru un dicționar simplu, dar diferă funcția de adăugare la dicționar.

5.5 IMPLEMENTARE DICȚIONAR PRIN TABEL DE DISPERSIE

În expresia "tabel de dispersie", cuvântul "tabel" este sinonim cu "vector".

Un tabel de dispersie ("hash table") este un vector pentru care poziția unde trebuie introdus un nou element se calculează din valoarea elementului, iar aceste poziții rezultă în general dispersate, fiind determinate de valorile elementelor și nu de ordinea în care ele au fost adăugate. O valoare nouă nu se adaugă în prima poziție liberă ci într-o poziție care să permită regăsirea rapidă a acestei valori (fără căutare).

Ideea este de a calcula poziția unui nou element în vector în funcție de valoarea elementului. Același calcul se face atât la adăugare cât și la regăsire :

- Se reduce cheia la o valoare numerică (dacă nu este deja un număr întreg pozitiv);
- Se transformă numărul obținut (codul "hash") într-un indice corect pentru vectorul respectiv; de regulă acest indice este egal cu restul împărțirii prin lungimea vectorului (care e bine să fie un număr prim). Se pot folosi și alte metode care să producă numere aleatoare uniform distribuite pe mulțimea de indici în vector.

Procedura de calcul a indicelui din cheie se numește și metodă de dispersie, deoarece trebuie să asigure dispersia cât mai uniformă a cheilor pe vectorul alocat pentru memorarea lor.

Codul hash se calculează de obicei din valoarea cheii. De exemplu, pentru siruri de caractere codul hash se poate calcula după o relație de forma:

$$(\sum s[k] * (k+1)) \% m \quad \text{suma ptr } k=0, \text{ strlen}(s)$$

unde $s[k]$ este caracterul k din sir, iar m este valoarea maximă pentru tipul întreg folosit la reprezentarea codului (int sau long). În esență este o sumă ponderată cu poziția în sir a codului caracterelor din sir (sau a primelor n caractere din sir).

O variantă a metodei anterioare este o sumă modulo 2 a caracterelor din sir.

Orice metodă de dispersie conduce inevitabil la apariția de "sinonime", adică chei (obiecte) diferite pentru care rezultă aceeași poziție în vector. Sinonimele se numesc și "coliziuni" pentru că mai multe obiecte își dispută o aceeași adresă în vector.

Un tabel de dispersie se poate folosi la implementarea unei multimi sau a unui dicționar; diferențele apar la datele continute și la funcția de punere în dicționar a unei perechi cheie-valoare (respectiv funcția de adăugare la mulțime).

Pentru a exemplifica să considerăm un tabel de dispersie de 5 elemente în care se introduc următoarele chei: 2, 3, 4, 5, 7, 8, 10, 12. Resturile împărțirii prin 5 ale acestor numere conduc la indicii: 2, 3, 4, 0, 2, 3, 0, 2. După plasarea primelor 4 chei, în pozițiile 2,3,4,0 rămâne liberă poziția 1 și vor apărea coliziunile 7 cu 2, 8 cu 3, 10 și 12 cu 5. Se observă că este importantă și ordinea de introducere a cheilor într-un tabel de dispersie, pentru că ea determină conținutul acestuia.

O altă dimensiune a vectorului (de exemplu 7 în loc de 5) ar conduce la o altă distribuție a cheilor în vector și la alt număr de coliziuni.

Metodele de redistribuire a sinonimelor care pot fi grupate în:

- 1) Metode care calculează o nouă adresă în același vector pentru sinonimele ce găsesc ocupată poziția rezultată din calcul : fie se caută prima poziție liberă ("open-hash"), fie se aplică o a doua metodă de dispersie pentru coliziuni ("rehash"), fie o altă soluție. Aceste metode folosesc mai bine memoria dar pot necesita multe comparații. Pentru exemplul anterior, un tabel hash cu 10 poziții ar putea arăta astfel:

poz	0	1	2	3	4	5	6	7	8	9
val	5	10	2	3	4	5	12		8	

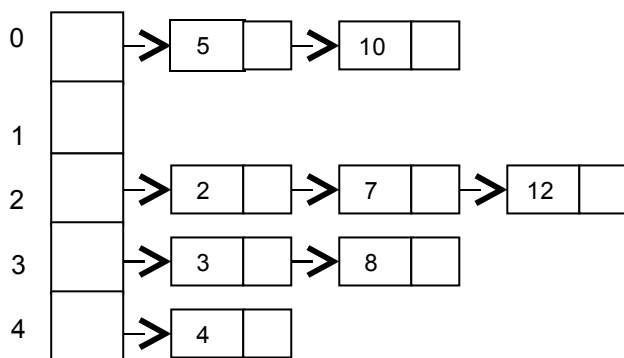
Pentru regăsirea sirului 12 se calculează adresa 2 ($12\%10$) și apoi se mai fac 5 comparații pentru a găsi sirul în una din următoarele poziții. Numărul de comparații depinde de dimensiunea vectorului și poate fi foarte mare pentru anumite coliziuni.

O variantă este utilizarea a doi vectori: un vector în care se pun cheile care au găsit liberă poziția calculată și un vector cu coliziuni (chei care au găsit poziția ocupată):

Vector principal : 5 - 2 3 4
Vector coliziuni : 7 8 10 12

2) Metode care plasează coliziunile în liste înlănțuite de sinonime care pleacă din poziția rezultată din calcul pentru fiecare grup de sinonime. Această metodă asigură un timp mai bun de regăsire, dar folosește mai multă memorie pentru pointeri. Este metoda preferată în clasele multime sau dictionar pentru că nu necesită o estimare a numărului maxim de valori (chei și valori) ce vor introduse în multime sau dictionar.

În acest caz tabelul de dispersie este un vector de pointeri la liste de sinonime, iar câștigul de timp provine din faptul că nu se caută într-o listă a tuturor cheilor și se caută numai în lista de sinonime care poate conține cheia căutată. Este de dorit ca listele de sinonime să fie de dimensiuni cât mai apropiate. Dacă listele devin foarte lungi se va reorganiza tabelul prin extinderea vectorului și mărirea numărului de liste.



Avantajele structurii anterioare sunt timpul de căutare foarte bun și posibilitatea de extindere nelimitată (dar cu degradarea performanțelor). Timpul de căutare depinde de mai mulți factori și este greu de calculat, dar o estimare a timpului mediu este $O(1)$, cu cazul cel mai defavorabil $O(n)$.

Un dezavantaj al tabelor de dispersie este acela că datele nu sunt ordonate și că se pierde ordinea de adăugare la tabel. O soluție este adăugarea unei liste cu toate elementele din tabel, în ordinea introducerii lor.

De observat că în liste sau în vectori de structuri se poate face căutare după diverse chei dar în tabele de dispersie și în arbori această căutare este posibilă numai după o singură cheie, stabilită la crearea structurii și care nu se mai poate modifica sau înlocui cu o altă cheie (cu un alt câmp).

Ideea înlocuirii unui sir de caractere printr-un număr întreg (operația de "hashing") are și alte aplicații: un algoritm eficient de căutare a unui sir de caractere într-un alt sir (algoritmul Rabin-Karp), în metode de criptare a mesajelor ș.a.

În exemplul următor se folosește un dictionar tabel de dispersie în problema afișării numărului de apariții al fiecărui cuvânt distinct dintr-un text; cheile sunt siruri de caractere iar valorile asociate sunt numere întregi (număr de repetări cuvânt):

```

#define H 13                // dimensiune tabel hash
typedef struct nod {        // un nod din lista de sinonime
    char * cuv;             // adresa unui cuvânt
    int nr;                 // numar de aparitii cuvânt
    struct nod * leg;       // legatura la nodul urmator
} nod;

```

```

typedef nod* Map [H];    // tip dictionar
// functie de dispersie
int hash ( char * s ) {
    int i,sum=0;
    for (i=0;i< strlen(s);i++)
        sum=sum+(i+1)*s[i];
    return sum % H;
}
// initializare tabel hash
void initD (Map d) {
    int i;
    for (i=0;i<H;i++)
        d[i]=NULL;
}
// afisare dictionar (lista dupa lista)
void printD (Map d) {
    int i; nod* p;
    for (i= 0;i<H;i++) {
        p=d[i];
        while (p != NULL) {
            printf ("%20s %4d\n", p->cuv,p->nr);
            p=p->leg;
        }
    }
}
// cautare (localizare) cuvânt în dictionar
nod* locD (Map d, char * c) {
    nod* p; int k;
    k=hash(c);           // pozitie cheie c in vector
    p=d[k];              // adresa listei de sinonime
    while ( p != NULL && strcmp(p->cuv,c)) // cauta cheia c in lista
        p=p->leg;
    return p;            // p=NULL daca c negasit
}
// adauga o pereche cheie-val la dictionar
void putD ( Map d, char * c, int nr) {
    nod *p, *pn; int k;
    k= hash (c);
    if ( (p=locD (d,c)) !=NULL) // daca c exista in nodul p
        p->nr=nr;                // modifica valoarea asociata cheii c
    else {                      // daca cheia nu era in dictionar
        pn= new nod;            // creare nod nou
        pn->cuv=c; pn->nr=nr;    // completare nod cu cheie si valoare
        pn->leg= d[k]; d[k]=pn; // adaugare la inceputul listei de sinonime
    }
}
// extrage valoarea asociata unei chei date
int getD (Map d, char* c) {
    nod *p;
    if ( (p=locD (d,c)) != NULL)
        return p->nr;           // daca nu exista anterior
    else
        return 0;
}
// citire cuvinte, creare si afisare dictionar
int main () {
    char numef[20], buf[128], * q;
    Map dc; int nra;

```

```

FILE *f;
printf ("nume fisier: "); scanf ("%s", numef);
f=fopen (numef, "r"); // assert (f != NULL);
initD (dc);
while (fscanf(f, "%s", buf) > 0) {
    q= strdup(buf); // creare adresa ptr sirul citit
    nra =getD (dc,q); // obtine numar de aparitii cuvant
    if (nra ==0) // daca e prima aparitie
        putD (dc, q,1); // pune cuvant in dictionar
    else // daca nu e prima aparitie
        putD(dc,q,nra+1); // modifica numar de aparitii cuvant
}
printD(dc); // afisare dictionar
}

```

Pentru a face mai general tabelul de dispersie putem defini tipul “Map” ca o structură care să includă vectorul de pointeri, dimensiunea lui (stabilită la initializarea dictionarului) si functia folosită la compararea cheilor (un pointer la o functie).

5.6 APLICATIE: COMPRESIA LZW

Metoda de compresie LZW (Lempel-Ziv-Welch), în diferite variante, este cea mai folosită metodă de compresie a datelor deoarece nu necesită informatii prelabile despre datele comprimate (este o metodă adaptivă) si este cu atât mai eficace cu cât fisierul initial este mai mare si contine mai multă redundanță.

Pentru texte scrise în engleză sau în română rezultatele sunt foarte bune deoarece ele folosesc în mod repetat anumite cuvinte uzuale, care vor fi înlocuite printr-un cod asociat fiecărui cuvânt.

Metoda LZW este folosită de multe programe comerciale (gzip, unzip, s.a.) precum si în formatul GIF de reprezentare (compactă) a unor imagini grafice.

Metoda foloseste un dictionar prin care asociază unor siruri de caractere de diferite lungimi coduri numerice întregi si înlocuieste secvente de caractere din fisierul initial prin aceste numere. Acest dictionar este cercetat la fiecare nou caracter extras din fisierul initial si este extins de fiecare dată când se găseste o secvență de caractere care nu exista anterior în dictionar.

Pentru decompresie se reface dictionarul construit în etapa de compresie; deci dictionarul nu trebuie transmis împreună cu fisierul comprimat.

Dimensiunea uzuală a dictionarului este 4096, dintre care primele 256 de pozitii contin toate caracterele individuale ce pot apare în fisierele de comprimat.

Din motive de eficiență pot exista diferente importante între descrierea principială a metodei LZW si implementarea ei în practică; astfel, sirurile de caractere se reprezintă tot prin numere, iar codurile asociate pot avea lungimi diferite.

Se poate folosi un dictionar format dintr-un singur vector de siruri (pointeri la siruri), iar codul asociat unui sir este chiar pozitia în vector unde este memorat sirul.

Sirul initial (de comprimat) este analizat si codificat într-o singură trecere, fără revenire. La stânga pozitiei curente sunt subsiruri deja codificate, iar la dreapta cursorului se caută cea mai lungă secvență care există deja în dictionar. Odată găsită această secvență, ea este înlocuită prin codul asociat deja si se adaugă la dictionar o secvență cu un caracter mai lungă.

Pentru exemplificare vom considera că textul de codificat contine numai două caractere (‘a’ si ‘b’) si arată astfel (sub text sunt trecute codurile asociate secventelor respective):

```

a b b a a b b a a b b a a a b a a b b a
0 | 1 | 1 | 0 | 2 | 4 | 2 | 6 | 5 | 5 | 7 | 3 | 0

```

Dictionarul folosit în acest exemplu va avea în final următorul continut:

0=a / 1=b / 2=0b (ab) / 3=1b (bb) / 4=1a (ba) / 5=0a (aa) / 6=2b (abb) / 7=4a (baa)
8=2a (aba) / 9=6a (abba) / 10=5a (aaa) / 11=5b (aab) / 12=7b (baab) / 13=3a (bba)

Intr-o variantă puțin modificată se asociază codul 0 cu sirul nul, după care toate secvențele de unul sau mai multe caractere sunt codificate printr-un număr întreg și un caracter:

1=0a / 2=0b / 3=1b (ab) / 4=2b (bb) / 5=2a (ba) / ...

Urmează o descriere posibilă pentru algoritmul de compresie LZW:

```

initializare dictionar cu n coduri de caractere individuale
w = NIL; k=n // w este un cuvânt (o secvență de caractere)
repetă cât timp mai există caractere neprelucrate
    citește un caracter c
    dacă w+c există în dictionar // '+' pentru concatenare de siruri
        w = w+c // prelungeste secvența w cu caracterul c
    altfel
        adaugă wc la dictionar cu codul k=k+1
        scrie codul lui w
        w = c

```

Este posibilă și următoarea formulare a algoritmului de compresie LZW:

```

initializare dictionar cu toate secvențele de lungime 1
repetă cât timp mai sunt caractere
    caută cea mai lungă secvență de car. w care apare în dictionar
    scrie poziția lui w în dictionar
    adaugă w plus caracterul următor la dictionar

```

Aplicarea acestui algoritm pe sirul "abbaabbaababbaaabaabba" conduce la secvența de pași rezumată în tabelul următor:

w	c	w+c	k	scrie (cod w)
nul	a	a		
a	b	ab	2=ab	0 (=a)
b	b	bb	3=bb	1 (=b)
b	a	ba	4=ba	1 (=b)
a	a	aa	5=aa	0 (=a)
a	b	ab		
ab	b	abb	6=abb	2 (=ab)
b	a	ba		
ba	a	baa	7=baa	4 (=ba)
a	b	ab		
ab	a	aba	8=aba	2 (=ab)
a	b	ab		
ab	b	abb		
abb	a	abba	9=abba	6 (=abb)
a	a	aa		
aa	a	aaa	10=aaa	5 (=aa)
a	a	aa		
aa	b	aab	11=aab	5 (=aa)
b	a	ba		
ba	a	baa		
baa	b	baab	12=baab	7 (=baa)
b	b	bb		
bb	a	bba	13=bba	3 (=bb)
a	-	a		0 (=a)

În exemplul următor codurile generate sunt afișate pe ecran:


```

// cauta un sir in vector de siruri
int find (char w[], char d[][8], int n) {
    int i;
    for (i=0;i<n;i++)
        if (strcmp(w,d[i])==0)
            return i;
    return -1;
}

// functie de codificare a unui sir dat
void compress (char * in) {
    char dic[200][8]; // max 200 de elemente a cate 7 caractere
    char w[8]="", w1[8], c[2]={0};
    int k; char * p=in; // p =adresa caracter in sirul de codificat
    // initializare dictionar
    strcpy(dic[0],"a"); strcpy(dic[1],"b");
    // ciclul de cautare-adaugare in dictionar
    k=2; // dimensiune dictionar (si prima pozitie libera)
    while (*p) { // cat timp mai sunt caractere in sirul initial
        c[0]=*p; // un sir de un singur caracter
        strcpy(w1,w); // w1=w
        strcat(w,c); // w= w + c
        if( find(w,dic,k) < 0 ) { // daca nu exista in dictionar
            strcpy(dic[k],w); // adauga in prima pozitie libera din dictionar
            printf("%d | ",find(w1,dic,k)); // scrie codul lui w
            k++; // creste dimensiune dictionar
            strcpy(w,c); // in continuare w=c
        }
        p++; // avans la caracterul urmator din sir
    }
}

```

Dimensiunea dictionarului se poate reduce dacă folosim drept chei ‘w’ întregi mici (“short”) obtinuti din codul k si caracterul ‘c’ adăugat la secventa cu codul k.

Timpul de căutare în dictionar se poate reduce folosind un tabel “hash” sau un arbore în locul unui singur vector, dar cu un consum suplimentar de memorie.

Rezultatul codificării este un sir de coduri numerice, cu mai putine elemente decât caractere în sirul initial, dar câstigul obtinut depinde de mărimea acestor coduri; dacă toate codurile au aceeasi lungime (de ex 12 biti pentru 4096 de coduri diferite) atunci pentru un număr mic de caractere în sirul initial nu se obtine nici o compresie (poate chiar un sir mai lung de biti). Compresia efectivă începe numai după ce s-au prelucrat câteva zeci de caractere din sirul analizat.

La decompresie se analizează un sir de coduri numerice, care pot reprezenta caractere individuale sau secvente de caractere. Cu ajutorul dictionarului se decodifică fiecare cod întâlnit. Urmează o descriere posibilă pentru algoritmul de decompresie LZW:

```

initializare dictionar cu codurile de caractere individuale
citeste primul cod k;
w = sirul din pozitia k a dictionarului;
repetă cat timp mai sunt coduri
    citeste urmatorul cod k
    cauta pe k in dictionar si extrage valoarea asociata c
    scrie c in fisierul de iesire
    adauga w + c[0] la dictionar
    w = c

```

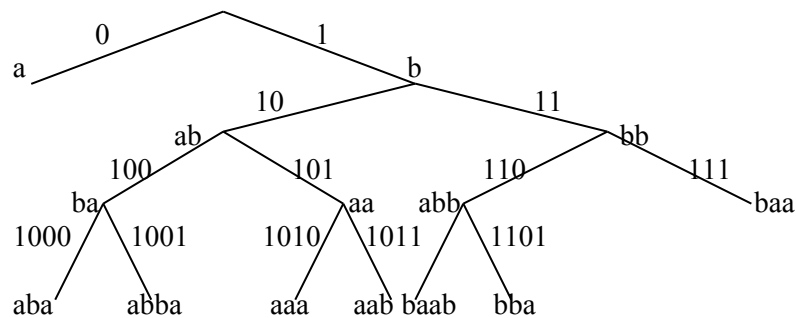
Dictionarul are aceeasi evolutie ca si în procesul de compresie (de codificare).

```

void decompress (int cod[], int n) {
    char dic[100][8];           // max 100 de elemente a cate 7 caractere
    char w[8]="", e[8]={0}, c[2]={0};
    int i,k;
    // initializare dictionar
    strcpy(dic[0],"a"); strcpy(dic[1],"b");
    k=2;
    printf("%s|",dic[0]); // caracterul cu primul cod
    strcpy(w,dic[0]); // w=dic[k]
    for (i=1;i<n;i++) {
        strcpy(e,dic[cod[i]]); // sirul cu codul cod[i]
        printf("%s|",e);
        c[0]=e[0];
        strcpy(dic[k++],strcat(w,c)); // adauga la dictionar un nou sir w+c
        strcpy(w,e);
    }
}

```

Codurile generate de algoritmul LZW pot avea un număr variabil de biti, iar la decompresie se poate determina numărul de biti în funcția de dimensiunea curentă a dictionarului. Dictionarul creat poate fi privit ca un arbore binar completat nivel cu nivel, de la stânga la dreapta:



Notând cu k nivelul din arbore, același cu dimensiunea curentă a dictionarului, se observă că numărul de biti pe acest nivel este $\log_2(k) + 1$.

Capitolul 6

STIVE SI COZI

6.1 LISTE STIVĂ

O stivă este o listă cu acces la un singur capăt, numit “vârful” stivei. Singurele operatii permise sunt inserare în prima pozitie si stergere din prima pozitie (eventual si citire din prima pozitie). Aceste operatii sunt denumite traditional “push” (pune pe stivă) si “pop” (scoate din stivă) si nu mai specifică pozitia din listă, care este implicită. O stivă mai este numită si listă LIFO (‘Last In First Out’), deoarece ultimul element pus este primul care va fi extras din stivă.

Operatiile asociate tipului abstract "stivă" sunt:

- initializare stivă vidă (initSt)
- test stivă vidă (emptySt)
- pune un obiect pe stivă (push)
- extrage obiectul din vârful stivei (pop)
- obtine valoare obiect din vârful stivei, fără scoatere din stivă (top)

Operatiile cu o stivă pot fi privite ca niste cazuri particulare de operatii cu liste oarecare, dar este mai eficientă o implementare directă a operatiilor "push" si "pop".

În STL operatia de scoatere din stivă nu are ca rezultat valoarea scoasă din stivă, deci sunt separate operatiile de citire vârful stivă si de micșorare dimensiune stivă.

O solutie simplă este folosirea directă a unui vector, cu adăugare la sfârșit (în prima pozitie liberă) pentru "push" si extragerea ultimului element, pentru "pop".

Exemplu de afisare a unui număr întreg fără semn în binar, prin memorarea în stivă a resturilor împărțirii prin 2, urmată de afisarea continutului stivei.

```
void binar (int n) {
    int st[100],vs, b;           // stiva "st" cu varful in "vs"
    vs=0 ;                       // indice varf stiva
    while (n > 0) {
        b= n % 2 ; n= n /2;      // b = rest impartire prin 2
        st[vs++]=b ;            // memoreaza b in stiva
    }
    while (vs > 0) {             // cat timp mai e ceva in stiva
        b=st[--vs];             // scoate din stiva in b
        printf ("%d ",b);       // si afiseaza b
    }
    printf ("\n");
}
```

Vârful stivei (numit si "stack pointer") poate fi definit ca fiind pozitia primei pozitii libere din stivă sau ca pozitie a ultimului element pus în stivă. Diferenta de interpretare are efect asupra secventei de folosire si modificare a vârfului stivei:

```
void binar (int n) {
    int st[100],vs, b;           // stiva "st" cu varful in "vs"
    vs= -1 ;                     // indice varf stiva (ultima valoare pusa in stiva)
    while (n > 0) {
        b= n % 2 ; n= n /2;      // b = rest impartire prin 2
        st[++vs]=b ;            // memoreaza b in stiva
    }
    while (vs >= 0) {           // cat timp mai e ceva in stiva
        b=st[vs--];             // scoate din stiva in b
        printf ("%d ",b);       // si afiseaza b
    }
}
```

```

}
printf ("\n");
}

```

Ca și pentru alte colecții de date, vom prefera definirea unor funcții pentru operații asociate structurii de stivă. Vom exemplifica cu o stivă realizată ca un vector, cu adăugare și stergere la sfârșitul vectorului.

```

#define M 100                // dimensiune maxima stiva
typedef struct {
    T st[M];                  // stiva vector
    int sp;                   // virful stivei
} Stack;
// initializare stiva
void initSt (Stack & s) {
    s.sp = 0;
}
// test stiva goala
int emptySt (Stack s) {
    return (s.sp == 0);
}
// pune in stiva pe x
void push (Stack & s, T x) {
    assert (s.sp < M-1);      // verifica umplere stiva
    s.st [s.sp++] = x;
}
// scoate in x din stiva
T pop (Stack & s) {
    assert (s.sp >= 0);        // verifica daca stiva nu e vida
    return s.st [s.sp--];
}
T top (Stack s) {             // valoare obiect din varful stivei
    assert (s.sp >= 0);        // verifica daca stiva nu e vida
    return s.st [s.sp];
}

```

Dimensionarea vectorului stivă este dificilă în general, dar putem folosi un vector extensibil dinamic (alocat și realocat dinamic). Modificările apar numai la initializarea stivei și la punere în stivă.

De asemenea, se poate folosi o listă înlănțuită cu adăugare și extragere numai la începutul listei (mai rapid și mai simplu de programat). Exemplu:

```

typedef struct s {
    T val;
    struct s * leg;
} nod;
typedef nod * Stack;          // tipul Stack este un tip pointer
// initializare stiva
void initSt (Stack & s) {
    s = NULL;
}
// test stiva goala
int emptySt (Stack s) {
    return (s == NULL);
}
// pune in stiva un obiect
void push (Stack & s, T x) {
    nod * nou;
    nou = (nod*)malloc(sizeof(nod));
}

```

```

nou→val = x; nou→leg = s;
s = nou;
}
// scoate din stiva un obiect
T pop (Stack & s) {
    nod * aux; T x;
    assert (s != NULL);
    x = s→val;
    aux=s→leg; free (s) ;
    s = aux;
    return x;
}
// obiect din varful stivei
T top (Stack s) {
    assert ( s != NULL);
    return s→val;
}

```

Dacă sunt necesare stive cu conținut diferit în același program sau dacă în aceeași stivă trebuie memorate date de tipuri diferite vom folosi o stivă de pointeri "void*".

Prima și cea mai importantă utilizare a unei stive a fost în traducerea apelurilor de funcții, pentru revenirea corectă dintr-o secvență de apeluri de forma:

```

int main ( ) {      void f1 ( ) {      void f2 ( ) {      void f3 ( ) {
    ...              ...              ...              ...
    f1( );           f2( );           f3( );           ...
    a: ...           a1: ...           a2: ...           ...
}                   }                   }                   }

```

În stivă se pun succesiv adresele a,a1 și a2 pentru ca la ieșirea din f3 să se sară la a2, la ieșirea din f2 se sare la a1, și la ieșirea din f1 se revine la adresa 'a'.

Pentru execuția corectă a funcțiilor recursive se vor pune în aceeași stivă și valorile argumentelor formale și ale variabilelor locale.

Aplicațiile stivelor sunt cele în care datele memorate temporar în lista stivă se vor utiliza în ordine inversă punerii lor în stivă, cum ar fi în memorarea unor comenzi date sistemului de operare (ce pot fi readuse spre execuție), memorarea unor modificări asupra unui text (ce pot fi anulate ulterior prin operații de tip "undo"), memorarea paginilor Web afișate (pentru a se putea reveni asupra lor) sau memorarea marcărilor initiale ("start tags") dintr-un fișier XML, pentru verificarea utilizării lor corecte, împreună cu marcărele finale ("end tags").

Cealaltă categorie importantă de aplicații sunt cele în care utilizarea stivei este soluția alternativă (iterativă) a unor funcții recursive (direct sau indirect recursive).

6.2 APLICATIE : EVALUARE EXPRESII ARITMETICE

Evaluarea expresiilor aritmetice este necesară într-un program interpretor BASIC, într-un program de calcul tabelar (pentru formulele care pot apărea în celulele foi de calcul) și în alte programe care admit ca intrări expresii (formule) și care trebuie să furnizeze rezultatul acestor expresii.

Pentru simplificare vom considera numai expresii cu operanzi numerici întregi de o singură cifră, la care rezultatele intermediare și finale sunt tot întregi de o cifră.

Problema evaluării expresiilor este aceea că ordinea de aplicare a operatorilor din expresie (ordinea de calcul) este diferită în general de ordinea apariției acestor operatori în expresie (într-o parcurgere de la stânga la dreapta). Exemplu:

(5 – 6 / 2) * (1+ 3)

Evaluarea acestei expresii necesită calculele următoare (în această ordine):

$$6 / 2 = 3, \quad 5 - 3 = 2, \quad 1 + 3 = 4, \quad 2 * 4 = 8$$

Ordinea de folosire a operatorilor este determinată de importanta lor (înmulțirea și împărțirea sunt mai importante ca adunarea și scăderea) și de parantezele folosite.

Una din metodele de evaluare a expresiilor necesită două etape și fiecare din cele două etape utilizează câte o stivă :

- Transformarea expresiei în forma postfixată, folosind o stivă de operatori.
- Evaluarea expresiei postfixate, folosind o stivă de operanzi (de numere).

În forma postfixată a unei expresii nu mai există paranteze, iar un operator (binar) apare după cei doi operanzi folosiți de operator. Exemple de expresii postfixate:

Expresie infixată	Expresie postfixată
1+2	1 2 +
1+2+3	1 2 + 3 +
1+ 4/2	1 4 2 / +
(5-6/2)*(1+3)	5 6 2 / - 1 3 + *

Ambele etape pot folosi același tip de stivă sau stive diferite ca tip de date.

Comparând cele două forme ale unei expresii se observă că ordinea operanzilor se păstrează în sirul postfixat, dar operatorii sunt rearanjați în funcție de importanta lor și de parantezele existente. Deci operanzii trec direct din sirul infixat în sirul postfixat, iar operatorii trec în sirul postfixat numai din stivă. Stiva memorează temporar operatorii până când se decide scoaterea lor în sirul postfixat.

Algoritmul de trecere la forma postfixată cu stivă de operatori arată astfel:

```

repetă până la terminarea sirului infixat
  extrage următorul caracter din sir în ch
  dacă ch este operand atunci trece ch în sirul postfixat
  dacă ch este '(' atunci se pune ch în stiva
  dacă ch este ')' atunci
    repeta până la '('
    extrage din stiva și trece în sirul postfixat
    scoate '(' din stiva
  dacă ch este operator atunci
    repeta cât timp stiva nu e goală și prior(ch) <= prior(operator din varful stivei)
    scoate operatorul din stiva în sirul postfixat
    pune ch în stiva
  scoate operatori din stiva în sirul postfixat

```

Funcția următoare folosește o stivă de caractere:

```

void topostf (char * in, char * out) {
    Stack st;           // stiva de operatori
    char ch, op;
    initSt (st);        // initializare stiva
    while (*in != 0) {   // repeta până la sfârșit sir infixat
        while (*in == ' ') ++in; // ignora spații dintre elementele expresiei
        ch = *in++;      // următorul caracter din sirul infixat
        if (isdigit(ch)) // dacă ch este operand
            *out++ = ch; // trece ch în sir postfixat
        if (ch == '(')
            push(st, ch); // pune paranteze deschise în stiva
        if (ch == ')')
            // scoate din stiva toți operatorii până la o paranteză deschisă
            while (!emptySt(st) && (op = pop(st)) != '(')
                *out++ = op; // și trece operatori în sirul postfixat
        else {           // dacă este un operator aritmetic

```

```

        while (!emptySt(st) && pri(ch) <= pri(top(st))) // compara prioritati op.
            *out++=pop(st); // trece in sirul postfixat operator de prior. mare
            push(st,ch); // pune pe stiva operator din sirul infixat
    }
}
while (! empty(st) ) // scoate din stiva in sirul postfixat
    *out++=pop(st);
*out=0; // ptr terminare sir rezultat
}
    
```

Functia "pri" are ca rezultat prioritatea operatorului primit ca argument:

```

int pri (char op) {
    int k,nop=6; // numar de operatori
    char vop[ ] = { '(', '+', '-', '*', '/' }; // tabel de operatori
    int pr[ ]={ 0, 1, 1, 2, 2 }; // tabel de prioritati
    for (k=0;k<nop;k++)
        if (op==vop[k]) // cauta operator in tabel
            return pr[k]; // prioritate operator din pozitia k
    return -1; // operator negasit in tabel
}
    
```

Evolutia stivei de operatori la transformarea expresiei $8/(6-2) + 3*1$

infix	8	/	(6	-	2)	+	3	*	1
	↓			↓		↓			↓		↓
stiva de operatori	↓	/	(↓	(↓	/	+	↓	+	↓
postfix	8			6		2	-	/	3		1 * +

La terminarea expresiei analizate mai pot rămâne în stivă operatori, care trebuie scosi în sirul postfixat. O altă soluție este să se pună de la început în stivă un caracter folosit și ca terminator de expresie (';'), cu prioritate minimă. Altă soluție adaugă paranteze în jurul expresiei primite și repetă ciclul principal până la golirea stivei (ultima paranteză din sirul de intrare va scoate din stivă toți operatorii rămași).

Evaluarea expresiei postfixate parcurge expresia de la stânga la dreapta, pune pe stivă toți operanzii întâlniți, iar la găsirea unui operator aplică acel operator asupra celor doi operanzi scosi din vârful stivei și pune în stivă rezultatul parțial obținut.

Evolutia stivei la evaluarea expresiei postfixate $8\ 6\ 2\ -\ /\ 3\ 1\ *\ +$ va fi:

8	
8 6	
8 6 2	
8 4	(4=6-2)
2	(2=8/4)
2 3	
2 3 1	
2 3	(3=1*3)
5	(5=2+3)

Functie de evaluare a unei expresii postfixate cu operanzi de o singură cifră:

```

int eval ( char * in) {
    Stack st;                // stiva operanzi
    int t1,t2,r; char ch;
    initSt (st);             // initializare stiva
    while (*in != 0) {
        ch=*in++;            // un caracter din sirul postfixat
        if (isdigit(ch))     // daca este operand
            push(st,ch-'0'); // pune pe stiva un numar intreg
        else {               // daca este operator
            t2=pop (st); t1=pop (st); // scoate operanzi din stiva
            r=calc (ch,t1,t2); // evaluare subexpresie (t1 ch t2)
            push (st,r);      // pune rezultat partial pe stiva
        }
    }
    return pop(st);          // scoate rezultat final din stiva
}

```

Funcția "calc" calculează valoarea unei expresii cu numai doi operanzi:

```

int calc ( char op, int x, int y, char op) {
    switch (op) {
        case '+': return x+y;
        case '-': return x-y;
        case '*': return x*y;
        case '/': return x/y;
        default: return 0;    // nu ar trebui sa ajunga aici !
    }
}

```

Evaluarea unei expresii postfixate se poate face și printr-o funcție recursivă, fără a recurge la o stivă. Ideea este aceea că orice operator se aplică asupra rezultatelor unor subexpresii, deci se poate aplica definiția recursivă următoare:

$$\langle pf \rangle ::= \langle val \rangle \mid \langle pf \rangle \langle pf \rangle \langle op \rangle$$

unde: $\langle pf \rangle$ este o expresie postfixată, $\langle val \rangle$ este o valoare (un operand numeric) și $\langle op \rangle$ este un operator aritmetic.

Expresia postfixată este analizată de la dreapta la stânga:

```

void main () {
    char postf[40]; // sir postfixat
    printf ("sir postfixat: ");
    gets (postf);
    printf ("%d \n", eval(postf, strlen(postf)-1));
}

```

Funcția recursivă de evaluare poate folosi indici sau pointeri în sirul postfixat.

```

int eval (char p[], int& n) { // n=indicele primului caracter analizat
    int x,y; char op;
    if (n<0) return 0;       // daca expresie vida, rezultat zero
    if (isdigit(p[n]))        // daca este operand
        return p[n--] - '0'; // rezultat valoare operand
    else {                    // daca este operator
        op=p[n--];           // retine operator
        y=eval(p,n);         // evaluare operand 2
        x=eval(p,n);         // evaluare operand 1
    }
}

```



```

    return calc (op, x, y);
}
}

```

Eliminarea stivei din algoritmul de trecere la forma postfixată se face prin așa-numita analiză descendent recursivă, cu o recursivitate indirectă de forma:

$A \rightarrow B \rightarrow A$ sau $A \rightarrow B \rightarrow C \rightarrow A$

Regulile gramaticale folosite în analiza descendent recutsivă sunt următoarele:

```

expr ::= termen | expr + termen | expr - termen
termen ::= factor | termen * factor | termen / factor
factor ::= numar | ( expr )

```

Funcțiile următoare realizează analiza și interpretarea unor expresii aritmetice corecte sintactic. Fiecare funcție primește un pointer ce reprezintă poziția curentă în expresia analizată, modifică acest pointer și are ca rezultat valoarea (sub) expresiei. Funcția "expr" este apelată o singură dată în programul principal și poate apela de mai multe ori funcțiile "term" și "fact", pentru analiza subexpresiilor continute de expresie

Exemplu de implementare:

```

// valoare (sub)expresie
double expr ( char * & p ) {           // p= inceput (sub)expresie in sirul infixat
    double term(char*&);               // prototip functie apelată
    char ch; double t,r;               // r = rezultat expresie
    r=term(p);                         // primul (singurul) termen din expresie
    if (*p==0 ) return r;              // daca sfarsit de expresie
    while ( (ch=*p)=='+' || ch=='-' ) { // pot fi mai multi termeni succesivi
        t= term (++p);                 // urmatorul termen din expresie
        if(ch=='+') r +=t;             // aduna la rezultat partial
        else r-= t;                   // scade din rezultat partial
    }
    return r;
}

// valoare termen
double term (char * & p) {             // p= inceput termen in sirul analizat
    double fact(char*&);               // prototip functie apelată
    char ch; double t,r;
    r=fact(p);                         // primul (singurul) factor din termen
    if(*p==0) return r;               // daca sfarsit sir analizat
    while ( (ch=*p)=='*' || ch=='/' ) { // pot fi mai multi factori succesivi
        t= fact (++p);                 // valoarea factorului urmator
        if(ch=='*') r *=t;             // modifica rezultat partial cu acest factor
        else r/= t;
    }
    return r;
}

// valoare factor
double fact (char * & p) {             // p= inceputul unui factor
    double r;                         // r = rezultat (valoare factor)
    if ( *p=='(' ) {                  // daca incepe cu paranteza '('
        r= expr (++p);                 // valoarea expresiei dintre paranteze
        p++;                          // peste paranteza ')'
        return r;
    }
    else                             // este un numar
        return strtod(p,&p);          // valoare numar
}

```

Deși se bazează pe definiții recursive, funcțiile de analiză a subexpresiilor nu sunt direct recursive, folosind o rescriere iterativă a definițiilor după cum urmează:

```
opad ::= + |-
expr ::= termen | termen opad termen [opad termen]...
```

6.3 ELIMINAREA RECURSIVITĂȚII FOLOSIND O STIVĂ

Multe aplicații cu stive pot fi privite și ca soluții alternative la funcții recursive pentru aceleași aplicații.

Eliminarea recursivității este justificată atunci când dimensiunea maximă a stivei utilizate de compilator limitează dimensiunea problemei care trebuie rezolvată prin algoritmul recursiv. În stiva implicită se pun automat parametri formali, variabilele locale și adresa de revenire din funcție.

Funcțiile cu un apel recursiv urmat de alte operații sau cu mai multe apeluri recursive nu pot fi rescrise iterativ fără a utiliza o stivă.

În exemplul următor se afișează un număr întreg n în binar (în baza 2) după următorul raționament: sirul de cifre pentru n este format din sirul de cifre pentru $(n/2)$ urmat de o cifră 0 sau 1 care se obține ca $n \% 2$. De exemplu, numărul $n = 22$ se afișează în binar ca 10110 ($16+4+2$)

```
10110 este sirul binar pentru 22  ( 22 = 11*2 +0)
1011  este sirul binar pentru 11  ( 11 = 5*2 + 1)
101   este sirul binar pentru 5   ( 5  = 2*2 +1)
10    este sirul binar pentru 2   ( 2  = 1*2 +0)
1     este sirul binar pentru 1   ( 1  = 0*2 +1)
```

Forma recursivă a funcției de afișare în binar:

```
void binar (int n) {
    if (n>0) {
        binar (n/2);      // afiseaza in binar n/2
        printf("%d", n%2); // si apoi o cifra binara
    }
}
```

Exemplul următor arată cum se poate folosi o stivă pentru rescrierea iterativă a funcției recursive de afișare în binar.

```
void binar (int n) {
    int b ; Stack st;      // st este stiva pentru cifre binare
    initSt (st);
    while (n > 0) {        // repeta cat se mai pot face impartiri la 2
        b= n % 2 ; n= n / 2; // b este restul unei impartiri la 2 (b=0 sau 1)
        push(st,b);        // memoreaza rest in stiva
    }
    while (! emptySt(st)) { // repeta pana la golirea stivei
        b=pop(st);          // scoate din stiva in b
        printf ("%d ",b);   // si afiseaza
    }
}
```

În cazul funcțiilor cu mai multe argumente se va folosi fie o stivă de structuri (sau de pointeri la structuri), fie o stivă matrice, în care fiecare linie din matrice este un element al stivei (dacă toate argumentele sunt de același tip).

Vom exemplifica prin functii nerecursive de sortare rapidă (“quick sort”), în care se pun în stivă numai argumentele care se modifică între apeluri (nu si vectorul ‘a’).

Funcția următoare folosește o stivă de numere întregi:

```
void qsort (int a[], int i, int j) {
    int m; Stack st;
    initSt (st);
    push (st,i); push(st,j);      // pune argumente initiale in stiva
    while (! empty(st)) {        // repeta cat timp mai e ceva in stiva
        if (i < j) {              // daca se mai poate diviza partitia (i,j)
            m=pivot(a,i,j);       // creare subpartitii cu limita m
            push(st,i); push(st,m); // pune i si m pe stiva
            i=m+1;                // pentru a doua partitie
        }
        else {                    // daca partitie vida
            j=pop (st); i=pop(st); // refacere argumente din stiva (in ordine inversa !)
        }
    }
}
```

Dezavantajul acestei solutii este acela că argumentele trebuie scoase din stivă în ordine inversă introducerii lor, iar când sunt mai multe argumente se pot face erori.

În funcția următoare se folosește o stivă realizată ca matrice cu două coloane, iar punerea pe stivă înseamnă adăugarea unei noi linii la matrice:

```
typedef struct {
    int st[M][2];      // stiva matrice
    int sp;
}Stack;
// operatii cu stiva matrice
void push ( Stack & s, int x, int y) {
    s.st [s.sp][0]=x;
    s.st [s.sp][1]=y;
    s.sp++;
}
void pop ( Stack & s, int &x, int & y) {
    assert ( ! emptySt(s));
    s.sp--;
    x= s.st [s.sp][0];
    y= s.st [s.sp][1];
}
// utilizare stiva matrice
void qsort (int a[], int i, int j) {
    int m; Stack st;
    initSt (st);
    push (st,i,j);          // pune i si j pe stiva
    while (! emptySt(st)) {
        if (i < j) {
            m=pivot(a,i,j);
            push(st,i,m);    // pune i si m pe stiva
            i=m+1;
        }
        else {
            pop (st,i,j);    // scoate i si j din stiva
        }
    }
}
```

Atunci când argumentele (care se modifică între apeluri) sunt de tipuri diferite se va folosi o stivă de structuri (sau de pointeri la structuri), ca în exemplul următor:

```
typedef struct {    // o structura care grupeaza parametrii de apel
    int i,j;        // pentru qsort sunt doi parametrii intregi
} Pair;
// operatii cu stiva
typedef struct {
    Pair st[M];      // vector de structuri
    int sp;          // varful stivei
} Stack;
void push ( Stack & s, int x, int y) { // pune x si y pe stiva
    Pair p;
    p.i=x; p.j=y;
    s.st [s.sp++]= p;
}
void pop ( Stack & s, int & x, int & y) { // scoate din stiva in x si y
    assert ( ! emptySt(s));
    Pair p = s.st [--s.sp];
    x=p.i; y=p.j;
}
```

Utilizarea acestei stive de structuri este identică cu utilizarea stivei matrice, adică funcțiile “push” și “pop” au mai multe argumente (valorile puse sau scoase din stivă), în aceeași ordine pentru ambele funcții.

6.4 LISTE COADĂ

O coadă ("Queue"), numită și listă FIFO ("First In First Out") este o listă la care adăugarea se face pe la un capăt (de obicei la sfârșitul cozii), iar extragerea se face de la celălalt capăt (de la începutul cozii). Ordinea de extragere din coadă este aceeași cu ordinea de introducere în coadă, ceea ce face utilă o coadă în aplicațiile unde ordinea de servire este aceeași cu ordinea de sosire: procese de tip "vânzător - client" sau "producător - consumator". În astfel de situații coada de așteptare este necesară pentru a acoperi o diferență temporară între ritmul de servire și ritmul de sosire, deci pentru a memora temporar cereri de servire (mesaje) care nu pot fi încă prelucrate.

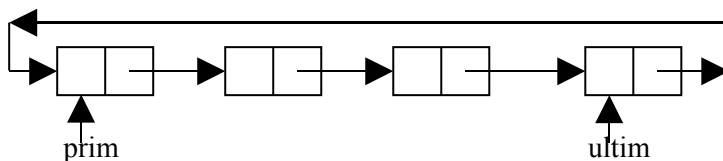
Operațiile cu tipul abstract "coadă" sunt:

- initializare coadă (initQ)
- test coadă goală (emptyQ)
- adaugă un obiect la coadă (addQ, insQ, enqueue)
- scoate un obiect din coadă (delQ, dequeue)

În STL și va exista în plus operația de citire din coadă, fără eliminare din coadă. Ca și alte liste abstracte, cozile pot fi realizate ca vectori sau ca liste înlănțuite, cu condiția suplimentară ca durata operațiilor addQ și delQ să fie minimă ($O(1)$).

O coadă înlănțuită poate fi definită prin :

- Adresa de început a cozii, iar pentru adăugare să se parcurgă toată coada (listă) pentru a găsi ultimul element (durata operației addQ va fi $O(n)$);
- Adresele primului și ultimului element, pentru a elimina timpul de parcurgere a listei la adăugare;
- Adresa ultimului element, care conține adresa primului element (coadă circulară).



Programul următor folosește o listă circulară definită prin adresa ultimului element din coadă, fără element santinelă:

```
typedef struct nod {
    int val;
    struct nod * leg;
} nod, *coada;
// initializare coada
void initQ ( coada & q) {
    q=NULL;
}
// scoate primul element din lista (cel mai vechi)
int delQ ( coada & q) {
    nod* prim; int x;
    if ( q!=NULL) {           // daca coada nu e vida
        prim= q->leg;         // adresa primului element
        x = prim->val;         // valoarea din primul element
        if (q==q->leg)         // daca era si ultimul element
            q=NULL;           // coada ramane goala
        else                   // daca nu era ultimul element
            q->leg=prim->leg;   // succesorul lui prim devine primul
        free(prim);           // eliberare memorie
        return x;             // rezultat extragere din coada
    }
}
// adaugare x la coada, ca ultim element
void addQ (coada & q, int x) {
    nod* p = (nod*) malloc(sizeof(nod)); // creare nod nou
    p->val=x;                             // completare nod nou
    if (q==NULL) {                         // daca se adauga la o coada goala
        q=p; p->leg=p;                     // atunci se creeaza primul nod
    }
    else {                                 // daca era ceva in coada
        p->leg=q->leg;                     // se introduce p intre q si q->leg
        q->leg=p;                          // si si noul nod devine ultimul
    }
}
// afisare coada, de la primul la ultimul
void printQ (coada q) {
    if (q==NULL) return; // daca coada e goala
    nod* p = q->leg;      // p= adresa primului nod
    do {                  // un ciclu while poate pierde ultimul nod
        printf ("%d ",p->val);
        p=p->leg;
    } while (p !=q->leg);
    printf ("\n");
}
```

Implementarea unei cozi printr-un vector circular (numit si buffer circular) limitează numărul maxim de valori ce pot fi memorate temporar în coadă. Caracterul circular permite reutilizarea locațiilor eliberate prin extragerea unor valori din coadă.

Câmpul "ultim" conține indicele din vector unde se va adăuga un nou element, iar "prim" este indicele primului (celui mai vechi) element din coadă. Deoarece "prim" si "ultim" sunt egale si când coada e goală si când coada e plină, vom memora si numărul de elemente din coadă. Exemplu de coadă realizată ca vector circular:

```

#define M 100          // capacitate vector coada
typedef struct {
    int nel;            // numar de elemente in coada
    T elem [M];         // coada vector
    int prim, ultim ;   // indici in vector
} Queue ;
// operatii cu coada vector
void initQ (Queue & q) { // initializare coada
    q.prim=q.ultim=0; q.nel=0;
}
int fullQ (Queue q) {    // test coada plina
    return (q.nel==M);
}
int emptyQ (Queue q) {   // test coada goala
    return (q.nel==0);
}
void addQ (Queue & q, T x ) { // introducere element in coada
    q.nel++;
    q.elem[q.ultim]=x;
    q.ultim=(q.ultim+1) % M ;
}
T delQ (Queue & q) {      // extrage element dintr-o coada
    T x;
    q.nel--;
    x=q.elem[q.prim];
    q.prim=(q.prim+1) % M ;
    return x;
}

```

Exemplu de secvență de operatii cu o coadă de numai 3 elemente :

Operatie	x	prim	ultim	nel	elem	fullQ	emptyQ
initial	1	0	0	0	0 0 0		T
addQ	1	0	1	1	1 0 0		
addQ	2	0	2	2	1 2 0		
addQ	3	0	0	3	1 2 3	T	
delQ	1	1	0	2	0 2 3		
addQ	4	1	1	3	4 2 3	T	
delQ	2	2	1	2	4 0 3		
addQ	5	2	2	3	4 5 3	T	
delQ	3	0	2	2	4 5 0		
delQ	4	1	2	1	0 5 0		
addQ	6	1	0	2	0 5 6		
delQ	5	2	0	1	0 0 6		
delQ	6	0	0	0	0 0 0		T

O coadă poate prelua temporar un număr variabil de elemente, care vor fi folosite în aceeași ordine în care au fost introduse în coadă. În sistemele de operare apar cozi de procese aflate într-o anumită stare (blocate în așteptarea unor evenimente sau gata de executie dar cu prioritate mai mică decât procesul în executie). Simularea unor procese de servire folosește de asemenea cozi de clienți în așteptarea momentului când vor putea fi serviti (prelucrați). Exemplul următor este o simulare simplă a unei cozi de mesaje între un emitător (producător de mesaje) și un receptor:

```

void main () {
    Queue q; int i; T x,y;
    q=initQ (); printf("\n");
    randomize();
    x=1; // mesaje puse in coada (numere intregi)
    for(i=1;i<30;i++)
        if (rand()%2 ==1) // daca e randul emitatorului

```

```

    addQ (q,x);x++;      // adauga mesaj la coada
else                    // daca e randul receptorului
if (!emptyQ(q)) {      // daca mai sunt mesaje in coada
    y=delQ (q);         // scoate mesaj
    printf ("%d ",y);   // si afiseaza
}
while (! emptyQ(q))    // in final scoate toate mesajele din coada
    printf ("%d ", delQ(q));
}

```

Secventa aleatoare de 1 si 0 generată simulează intervale variabile aleator între generarea de mesaje (activare emitor) si consumarea de mesaje (activare receptor).

Intr-un proces de servire există una sau mai multe statii de servire ("server") care satisfac cererile unor clienti. Intervalul dintre sosirile unor clienti succesivi, ca si timpii de servire pentru diversi clienti sunt variabile aleatoare în intervale cunoscute. Scopul simulării este obtinerea unor date statistice, cum ar fi timpul mediu si maxim dintre sosire si plecare client, numărul mediu si maxim de clienti în coada de asteptare la statie, în vederea îmbunătățirii procesului de servire (prin adăugarea altor statii de servire sau prin reducerea timpului de servire).

Vom considera cazul unei singure statii ; clientii care sosesc când statia e ocupată intră într-o coadă de asteptare si sunt serviti în ordinea sosirii si/sau în functie de anumite priorități ale clientilor. Imediat după sosirea unui client se generează momentul de sosire al unui nou client, iar când începe servirea unui client se generează momentul plecării acelui client. Simularea se face într-un interval dat de timp tmax.

Vom nota cu ts timpul de sosire a unui client la statie si cu tp timpul de servire a unui client (sau de prelucrare a unei cereri). Acesti timp si se calculează cu un generator de numere aleatoare, într-un interval dat de valori (functie de timpul mediu dintre doi clienti si respectiv de servire client).

Simularea se poate face în două moduri:

- Intervalul de simulare este împărțit în intervale egale (secunde, minute); scurgerea timpului este simulată printr-un ciclu, iar valorile variabilei contor reprezintă timpul curent din proces. Durata simulării este în acest caz proporțională cu mărimea intervalului de timp simulat. In fiecare pas se compară timpul curent cu timpul de producere a unor evenimente generate anterior (sosire client si plecare client).
- Se foloseste o coadă ordonată de evenimente (evenimente de sosire si de plecare clienti), din care evenimentele se scot în ordinea timpului de producere. Durata simulării depinde de numărul de evenimente produse într-un interval dat si mai puțin de mărimea intervalului.

Algoritmul de simulare care foloseste o coadă prioritară de evenimente poate fi descris astfel:

```

pune in coada de evenimente un eveniment "sosire" cu ts=0
se face server liber
repetă cat timp coada de evenim nu e goala {
    scoate din coada un eveniment
    dacă timpul depaseste durata simulării se termina
    dacă este un eveniment "sosire" {
        dacă server liber {
            se face server ocupat
            calculeaza alt timp tp
            pune in coada un eveniment "plecare" cu tp
        }
        altfel {
            pune client in coada de asteptare
            calculeaza alt timp ts
            pune in coada un eveniment "sosire" cu ts
        }
    }
    dacă eveniment "plecare" {
        dacă coada de clienti e goala
    }
}

```

```

        se face server liber
    altfel {
        scoate client din coada de asteptare
        pune in coada un eveniment "plecare" cu tp
    }
}

```

Algoritmul de simulare cu coadă de clienti poate fi descris astfel:

```

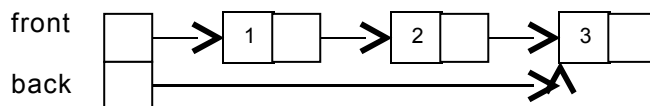
initializari (server liber)
calcul timp de sosire si plecare primul client ts, tp
repetă pentru valori t de la 0 la tmax cu pasul 1 {
    dacă t=ts (dacă sosește un client) {
        dacă server liber // începe servire client
        calcul timp de servire client tp
        altfel
        pune client in coada
        calcul timp sosire următorul client ts
    }
    dacă t=tp (dacă s-a terminat servirea unui client) {
        dacă coada nu e goală {
            scoate client din coada
            calcul timp de servire client tp
        }
        altfel // coada de clienti goală
        se face server liber
    }
}

```

Calitatea procesului de servire este determinată de lungimea cozii de clienti și deci de diferența dintre momentul sosirii și momentul plecării unui client (compus din timpul de așteptare în coadă și timpul de servire efectivă). Pentru obținerea de statistici se pot memora timpul de sosire și timpul de plecare al fiecărui client (împreună cu numărul clientului), precum și lungimea cozii de clienti când intră sau ies clienți din coadă.

Uneori se definește o coadă cu posibilități de adăugare și de extragere de la ambele capete ale cozii, numită "deque" ("double ended queue"), care are drept cazuri particulare stiva și coada, așa cum au fost definite aici. Operațiile caracteristice se numesc "pushfront", "pushback", "popfront", "popback" (în biblioteca STL).

O implementare adecvată pentru structura "deque" este o listă înlănțuită definită printr-o pereche de pointeri: adresa primului și adresa ultimului element din listă:



```

typedef struct nod { // nod de lista
    void* val; // cu pointer la date de orice tip
    struct nod * leg;
} nod;
typedef struct {
    nod* front; // adresa primului element
    nod* back; // adresa ultimului element
} deque;
// initializare lista
void init (deque & q){

```



```

    q.front = q.back=NULL; // lista fara santinela
}
int empty (deque q) {      // test lista voda
    return q.front==NULL;
}
// adaugare la inceput
void push_front (deque & q, void* px) {
    nod* nou = (nod*) malloc(sizeof(nod));
    nou->val=px;
    nou->leg= q.front;    // nou inaintea primului nod
    q.front=nou;
    if (q.back==NULL)    // daca este singurul nod
        q.back=nou;      // atunci devine si ultimul nod
}
// adaugare la sfarsit
void push_back (deque & q, void* px) {
    nod* nou = (nod*) malloc(sizeof(nod));
    nou->val=px;
    nou->leg= NULL;
    if (q.back==NULL)    // daca se adauga la lista vida
        q.front=q.back=nou; // este si primul si ultimul nod
    else {                // daca lista nu era goala
        q.back->leg=nou;    // nou se adauga dupa ultimul nod
        q.back=nou;        // si devine ultimul nod din lista
    }
}
// scoate de la inceput
void* pop_front (deque & q) {
    nod* t = q.front;
    void *r =t->val;      // rezultat functie
    if (q.front==q.back)  // daca era singurul nod din lista
        q.front=q.back=NULL; // lista devine goala
    else
        q.front=q.front->leg; // succesul lui front devine primul nod
    free(t);
    return r;
}
// scoate de la sfarsit de lista
void* pop_back (deque & q) {
    nod* t = q.back;
    void *r =t->val;
    int k;
    if (q.back==q.front)  // daca era singurul nod din lista
        q.back=q.front=NULL; // lista ramane goala
    else {                // daca nu era ultimul
        nod*p= q.front;    // cauta predecesorul ultimului nod
        while (p->leg != q.back)
            p=p->leg;
        p->leg=NULL;      // predecesorul devine ultimul
        q.back=p;
    }
    free(t);
    return r;
}

```

Se observă că numai ultima operatie (pop_back) contine un ciclu si deci necesită un timp ce depinde de lungimea listei $O(n)$; complexitatea acestei operatii poate deveni tot $O(1)$ dacă se foloseste o listă dublu-înlăntuită, cu legături înapoi.

6.5 TIPUL "COADĂ CU PRIORITĂȚI"

O coadă cu priorități ("Priority Queue") este o colecție din care se extrage mereu elementul cu prioritate maximă (sau minimă). Prioritatea este dată de valoarea elementelor memorate sau de o cheie numerică asociată elementelor memorate în coadă. Dacă există mai multe elemente cu aceeași prioritate, atunci ordinea de extragere este aceeași cu ordinea de introducere.

Algoritmii de tip "greedy" folosesc în general o coadă cu priorități pentru listele de candidați; la fiecare pas se extrage candidatul optim din listă.

O coadă cu priorități este o structură dinamică, la care au loc alternativ introduceri și extrageri din coadă. Dacă nu se mai fac inserări în coadă, atunci putem folosi un simplu vector, ordonat la început și apoi parcurs succesiv de la un cap la altul.

Operațiile specifice cozii cu priorități sunt:

- Adăugare element cu valoarea x la coada q : $\text{addPQ}(q, x)$
- Extrage în x și șterge din coada q elementul cu cheia maximă (minimă): $\text{delPQ}(q)$
- Citire (fără extragere) valoare minimă sau maximă: $\text{minPQ}(q)$
- Initializare coadă: $\text{initPQ}(q)$.
- Test coadă vidă: $\text{emptyPQ}(q)$

Sunt posibile diverse implementări pentru o coadă cu priorități (vector ordonat, listă ordonată, arbore binar ordonat), dar cele mai bune performante le are un vector "heap", din care se extrage mereu primul element, dar se face o rearanjare parțială după fiecare extragere sau inserție.

O aplicație simplă pentru o coadă cu priorități este un algoritm greedy pentru interclasarea mai multor vectori ordonați cu număr minim de operații (sau pentru reuniunea mai multor mulțimi cu număr minim de operații).

Interclasarea a doi vectori cu n_1 și respectiv n_2 elemente necesită $n_1 + n_2$ operații. Fie vectorii 1,2,3,4,5,6 cu dimensiunile următoare: 10,10,20,20,30,30.

Dacă ordinea de interclasare este ordinea crescătoare a vectorilor, atunci numărul de operații la fiecare interclasare va fi:

$$10+10=20, 20+20=40, 40+20=60, 60+30=90, 90+30=120.$$

Numărul total de operații va fi $20+40+60+90+120=330$

Numărul total de operații depinde de ordinea de interclasare a vectorilor și are valoarea minimă 300.

Ordinea de interclasare poate fi reprezentată printr-un arbore binar sau printr-o expresie cu paranteze. Modul de grupare care conduce la numărul minim de operații este $((1+2)+3)+6+(4+5)$ deoarece la fiecare pas se execută operațiile:

$$10+10=20, 20+20=40, 40+30=70, 20+30=50, 70+50=120 \quad (20+40+70+50+120=300)$$

Algoritmul de interclasare optimă poate fi descris astfel:

```

creare coadă ordonată crescător cu lungimile vectorilor
repetă
    scoate valori minime din coada în  $n_1$  și  $n_2$ 
     $n = n_1 + n_2$ 
    dacă coada e goală
        scrie  $n$  și stop
    altfel
        pune  $n$  în coada

```

Evoluția cozii cu priorități pentru exemplul anterior cu 6 vectori va fi:

```

10,10,20,20,30,30
20,20,20,30,30
20,30,30,40
30,40,50
50,70
120

```

Urmează aplicatia de interclasare vectori cu număr minim de operatii, folosind o coadă de numere întregi reprezentând lungimile vectorilor.

```
void main () {
    PQ pq; int i,p1,p2,s ;
    int n=6, x[ ]={10,10,20,20,30,30}; // dimensiuni vectori
    initpq (pq,n); // creare coada cu datele initiale
    for (i=0;i<n;i++)
        addpq (pq, &x[i]); // adauga adrese la coada
    do { // scoate si pune in coada ordonata
        p1=delpq (pq); p2=delpq (pq); // adrese dimensiuni minime de vectori
        s=p1 +p2; // dimensiune vector rezultat prin interclasare
        if ( emptypq(pq)) { // daca coada goala
            printf ("%d ", s); // afiseaza ultima suma (dimens vector final)
            break;
        }
        addpq(pq,s); // adauga suma la coada
    } while (1);
    printf ("\n");
}
```

Programul anterior nu permite afisarea modului de grupare optimă a vectorilor si nici operatia de interclasare propriu-zisă, deoarece nu se memorează în coadă adresele vectorilor, dar se poate extinde cu memorarea numerelor (adreselor) vectorilor.

6.6 VECTORI HEAP (ARBORI PARTIAL ORDONATI)

Un "Heap" este un vector care reprezintă un arbore binar partial ordonat de înălțime minimă, completat de la stânga la dreapta pe fiecare nivel. Un max-heap are următoarele proprietăți:

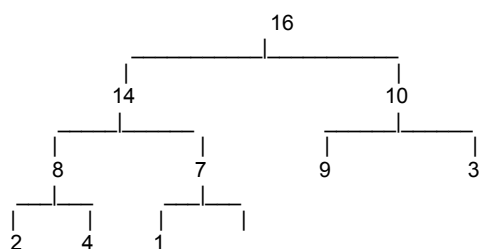
- Toate nivelurile sunt complete, cu posibila exceptie a ultimului nivel, completat de la stânga spre dreapta.
- Valoarea oricărui nod este mai mare sau egală cu valorile succesorilor săi.

O definiție mai scurtă a unui (max)heap este: un arbore binar complet în care orice fiu este mai mic decât părintele său.

Rezultă de aici că rădăcina arborelui conține valoarea maximă dintre toate valorile din arbore (pentru un max-heap).

Vectorul conține valorile nodurilor, iar legăturile unui nod cu succesorii săi sunt reprezentate implicit prin pozițiile lor în vector :

- Rădăcina are indicele 1 (este primul element din vector).
 - Pentru nodul din poziția k nodurile vecine sunt:
 - Fiul stânga în poziția $2*k$
 - Fiul dreapta în poziția $2*k + 1$
 - Părintele în poziția $k/2$
- Exemplu de vector max-heap :



Indice	1	2	3	4	5	6	7	8	9	10
Valoare	16	14	10	8	7	9	3	2	4	1

De observat că valorile din noduri depind de ordinea introducerii lor în heap, dar structura arborelui cu 10 valori este aceeași (ca repartizare pe fiecare nivel). Altfel spus, cu aceleași n valori se pot construi mai mulți vectori max-heap (sau min-heap).

Intr-un min-heap prima poziție (rădăcină) conține valoarea minimă, iar fiecare nod are o valoare mai mică decât valorile din cei doi fii ai săi.

Vectorii heap au cel puțin două utilizări importante:

- (Max-Heap) o metodă eficientă de sortare ("HeapSort");
- (Min-Heap) o implementare eficientă pentru tipul "Coadă cu priorități";

Operațiile de introducere și de eliminare dintr-un heap necesită un timp de ordinul $O(\log n)$, dar citirea valorii minime (maxime) este $O(1)$ și nu depinde de mărimea sa.

Operațiile de bază asupra unui heap sunt :

- Transformare heap după apariția unui nod care nu este mai mare ca succesorii săi, pentru menținerea proprietății de heap ("heapify", "percolate");
- Crearea unui heap dintr-un vector oarecare;
- Extragere valoare maximă (minimă);
- Inserare valoare nouă în heap, în poziția corespunzătoare.
- Modificarea valorii dintr-o poziție dată.

Primul exemplu este cu un max-heap de numere întregi, definit astfel:

```
typedef struct {
    int v[M];          // vector heap (cu maxim M numere)
    int n;             // dimensiune efectiva
} heap;
```

Operația "heapify" reface un heap dintr-un arbore la care elementul k nu respectă condiția de heap, dar subarborii săi respectă această condiție; la această situație se ajunge după înlocuirea sau după modificarea valorii din rădăcina unui arbore heap. Aplicată asupra unui vector oarecare funcția "heapify(k)" nu creează un heap, dar aduce în poziția k cea mai mare dintre valorile subarborului cu rădăcina în k : se mută succesiv în jos pe arbore valoarea $v[k]$, dacă nu este mai mare decât fii săi. Funcția recursivă "heapify" din programul următor face această transformare propagând în jos pe arbore valoarea din nodul " i ", astfel încât arborele cu rădăcina în " i " să fie un heap. În acest scop se determină valoarea maximă dintre $v[i]$, $v[st]$ și $v[dr]$ și se aduce în poziția " i ", pentru ca să avem $v[i] \geq v[st]$ și $v[i] \geq v[dr]$, unde " st " și " dr " sunt adresele (indicii) succesorilor la stânga și la dreapta ai nodului din poziția " i ". Valoarea coborâtă din poziția " i " în " st " sau " dr " va fi din nou comparată cu succesorii săi, la un nou apel al funcției "heapify".

Funcția următoare schimbă între ele valorile $v[i]$ și $v[j]$

```
void swap (heap h, int i, int j) {
    int t;
    t=h.v[i]; h.v[i]=h.v[j]; h.v[j]=t;
}

// ajustare max-heap
void heapify (heap & h,int i) {
    int st,dr,m;
    int aux;
    st=2*i; dr=st+1;          // succesorii nod i
    // determin maxim dintre valorile din pozitiile i, st, dr
    if (st<= h.n && h.v[st] > h.v[i] )
        m=st;                 // maxim in stanga lui i
    else
        m=i;                  // maxim in pozitia i
    if (dr<= h.n && h.v[dr]> h.v[m] )
        m=dr;                 // maxim in dreapta lui i
    if (m !=i) {               // daca e necesar
        swap(h,i,m);          // schimba maxim cu v[i]
        heapify (h,m);        // ajustare din pozitia m
    }
```

```
}
}
```

Urmează o variantă iterativă pentru funcția “heapify”:

```
void heapify (heap& h, int i) {
    int st,dr,m=i;           // m= indice val. maxima
    while (2*i <= h.n) {
        st=2*i; dr=st+1;     // succesori nod i
        if (st<= n && h.v[st]>h.v[m] ) // daca v[m] < v[st]
            m=st;
        if (dr<= n && h.v[dr]>h.v[m]) // daca v[m] < v[dr]
            m=dr;
        if ( i==m) break;    // gata daca v[i] nemodificat
        swap (h, i,m);       // interschimb v[i] cu v[m]
        i=m;
    }
}
```

Transformarea unui vector dat într-un vector heap se face treptat, pornind de la frunze spre rădăcină, cu ajustare la fiecare element:

```
void makeheap (heap & h) {
    int i;
    for (i=h.n/2; i>=1;i--) // parintele ultimului element este in pozitia n/2
        heapify (h,i);
}
```

Vom ilustra acțiunea funcției "makeheap" pe exemplul următor:

operatie	vector	arbore
initializare	1 2 3 4 5 6	<pre> 1 / \ 2 3 / \ / \ 4 5 6</pre>
heapify(3)	1 2 6 4 5 3	<pre> 1 / \ 2 6 / \ / \ 4 5 3</pre>
heapify(2)	1 5 6 4 2 3	<pre> 1 / \ 5 6 / \ / \ 4 2 3</pre>
heapify(1)	6 5 3 4 2 1	<pre> 6 / \ 5 3 / \ / \ 4 2 1</pre>

Programul de mai jos arată cum se poate ordona un vector prin crearea unui heap și interschimb între valoarea maximă și ultima valoare din vector.

```
// sortare prin creare si ajustare heap
void heapsort (int a[],int n) {
    int i, t; heap h;
    h.n=n;           // copiaza in heap valorile din vectorul a
    for (i=0;i<n;i++)
        h.v[i+1]=a[i];
    makeheap(h);     // aducere vector la structura de heap
    for (i=h.n;i>=2;i--) { // ordonare vector heap
```

```

    t=h.v[1]; h.v[1]=h.v[h.n]; h.v[h.n]=t;
    h.n--;
    heapify (h,1);
}
for (i=0;i<n;i++)      // scoate din heap in vectorul a
    a[i]= h.v[i+1];
}

```

În funcția de sortare se repetă următoarele operații:

- schimbă valoarea maximă $a[1]$ cu ultima valoare din vector $a[n]$,
- se reduce dimensiunea vectorului
- se "ajustează" vectorul rămas

Vom arăta acțiunea procedurii "heapSort" pe următorul exemplu:

după citire vector	4,2,6,1,5,3
după makeheap	6,5,4,1,2,3
după schimbare 6 cu 3	3,5,4,1,2,6
după heapify(1,5)	5,3,4,1,2,6
după schimbare 5 cu 2	2,3,4,1,5,6
după heapify(1,4)	4,3,2,1,5,6
după schimbare 4 cu 1	1,3,2,4,5,6
după heapify(1,3)	3,1,2,4,5,6
după schimbare 3 cu 2	2,1,3,4,5,6
după heapify(1,2)	2,1,3,4,5,6
după schimbare 2 cu 1	1,2,3,4,5,6

Extragerea valorii maxime dintr-un heap se face eliminând rădăcina (primul element din vector), aducând în prima poziție valoarea din ultima poziție și aplicând funcția "heapify" pentru menținerea vectorului ca heap:

```

int delmax ( heap & h) { // extragere valoare maxima din coada
    int hmax;
    if (h.n <= 0) return -1;
    hmax = h.v[1];      // maxim in prima pozitie din vector
    h.v[1] = h.v[h.n];  // se aduce ultimul element in prima pozitie
    h.n --;             // scade dimensiune vector
    heapify (h,1);      // ajustare ptr mentinere conditii de heap
    return hmax;
}

```

Adăugarea unei noi valori la un heap se poate face în prima poziție liberă (de la sfârșitul vectorului), urmată de deplasarea ei în sus cât este nevoie, pentru menținerea proprietății de heap:

```

// introducere in heap
void insH (heap & h, int x ) {
    int i ;
    i=++h.n;                // prima pozitie libera in vector
    h.v[i]=x;               // adauga noua valoare la sfarsit
    while (i > 1 && h.v[i/2] < x ) { // cat timp x este prea mare pentru pozitia sa
        swap (h, i, i/2);      // se schimba cu parintele sau
        i = i/2;              // si se continua din noua pozitie a lui x
    }
}

```

Modul de lucru al funcției insH este arătat pe exemplul de adăugare a valorii $val=7$ la vectorul $a=[8,5,6,3,2,4,1]$

i=8, a[8]=7	a= [8,5,6,3,2,4,1,7]
i=8, a[4]=3 < 7 , a[8] cu a[4]	a= [8,5,6,7,2,4,1,3]
i=4, a[2]=5 < 7 , a[4] cu a[2]	a= [8,7,6,5,2,4,1,3]
i=2, a[1]=8 > 7	a= [8,7,6,5,2,4,1,3]

Intr-un heap folosit drept coadă cu priorități se memorează obiecte ce contin o cheie, care determină prioritatea obiectului, plus alte date asociate acestei chei. De exemplu, în heap se memorează arce dintr-un graf cu costuri, iar ordonarea lor se face după costul arcului. In limbajul C avem de ales între un heap de pointeri la *void* si un heap de structuri. Exemplu de min-heap generic folosit pentru arce cu costuri:

```
typedef struct {
    int v,w,cost ;
} Arc;
typedef Arc T;           // tip obiecte puse in heap
typedef int Tk;          // tip cheie
typedef int (* fcomp)(T,T); // tip functie de comparare
typedef struct {
    T h[M];              // vector heap
    int n;
    fcomp comp;
} heap;
// compara arce dupa cost
int cmparc (Arc a, Arc b) {
    return a.cost - b.cost;
}
// ajustare heap
void heapify (heap & h,int i) {
    int st,dr,min;
    T aux;
    st=2*i; dr=st+1;      // succesori nod i
    // determin minim între valorile din pozitiile i, st, dr
    if (st<= h.n && h.comp(h.v[st], h.v[i]) < 0 )
        min=st;
    else
        min=i;
    if (dr<= h.n && h.comp(h.v[dr],h.v[min])<0 )
        min=dr;
    if (min !=i) {         // schimba minim cu elementul i
        aux=h.v[i]; h.v[i] = h.v[min]; h.v[min]=aux;
        heapify (h,min);
    }
}
```

La utilizarea unei cozi cu priorități apare uneori situatia când elementele din coadă au acelasi număr, aceleasi date memorate dar prioritatea lor se modifică în timp. Un exemplu este algoritmul Dijkstra pentru determinarea drumurilor minime de la un nod sursă la toate celelalte noduri dintr-un graf; în coadă se pun distantele calculate de la nodul sursă la celelalte noduri, dar o parte din aceste distante se modifică la fiecare pas din algoritmul (se modifică doar costul dar nu si numărul nodului). Pentru astfel de cazuri este utilă operatia de modificare a priorității, cu efect asupra pozitiei elementului respectiv în coadă (fără adăugări sau eliminări de elemente din coadă).

La implementarea cozii printr-un vector heap operatia de modificare a priorității unui element are ca efect propagarea elementului respectiv în sus (diminuare prioritate la un min-heap) sau în jos (crestere prioritate într-un max-heap). Operatia este simplă dacă se cunoaste pozitia elementului în heap pentru că seamănă cu adăugarea unui nou element la heap (se compară repetat noua prioritate cu prioritatea nodului părinte si se mută elementul dacă e necesar, pentru a mentine un heap).

Capitolul 7

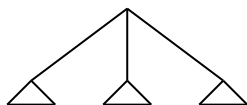
ARBORI

7.1 STRUCTURI ARBORESCENTE

Un arbore cu rădăcină ("rooted tree") este o structură neliniară, în care fiecare nod poate avea mai multi succesori, dar un singur predecesor, cu excepția unui nod special, numit rădăcină și care nu are nici un predecesor.

Structura de arbore se poate defini recursiv astfel: Un arbore este compus din:

- nimic (arbore vid)
- un singur nod (rădăcina)
- un nod care are ca succesori un număr finit de (sub)arbori.



Altfel spus, dacă se elimină rădăcina unui arbore rezultă mai multi arbori, care erau subarbori în arborele inițial (dintre care unii pot fi arbori fără nici un nod).

Definiția recursivă este importantă pentru că multe operații cu arbori pot fi descompuse recursiv în câteva operații componente:

- prelucrare nod rădăcină
- prelucrare subarbori pentru fiecare fiu.

Un arbore poate fi privit ca o extindere a listelor liniare. Un arbore binar în care fiecare nod are un singur succesor, pe aceeași parte, este de fapt o listă liniară.

Structura de arbore este o structură ierarhică, cu noduri așezate pe diferite niveluri, cu relații de tip părinte - fiu între noduri. Nodurile sunt de două feluri:

- Nodurile terminale, fără succesori, se numesc și "frunze";
- Noduri interne (interioare), cu unul sau doi succesori.

Fiecare nod are două proprietăți:

- Adâncimea ("depth") este egală cu numărul de noduri de pe calea (unică) de la rădăcină la acel nod;
- Înălțimea ("height") este egală cu numărul de noduri de pe cea mai lungă cale de la nod la un descendent (calea de la nod la cel mai îndepărtat descendent).

Înălțimea unui arbore este înălțimea rădăcinii sale, deci de calea cea mai lungă de la rădăcină la o frunză. Un arbore cu un singur nod (rădăcină) are înălțimea zero.

Un arbore este perfect echilibrat dacă înălțimile fiilor oricărui nod diferă între ele cel mult cu 1. Un arbore este echilibrat dacă înălțimea sa este proporțională cu $\log(N)$, ceea ce face ca durata operațiilor de căutare, inserție, eliminare să fie de ordinul $O(\log(N))$, unde N este numărul de noduri din arbore.

În fiecare nod dintr-un arbore se memorează valoarea nodului (sau un pointer către informații asociate nodului), pointeri către fii săi și eventual alte date: pointer la nodul părinte, adâncimea sa înălțimea nodului s.a. De observat că adresa nodului părinte, înălțimea sau adâncimea nodului pot fi determinate prin apelarea unor funcții (de obicei recursive), dacă nu sunt memorate explicit în fiecare nod.

După numărul maxim de fii ai unui nod arborii se împart în:

- Arbori multicăi (general), în care un nod poate avea orice număr de succesori;
- Arbori binari, în care un nod poate avea cel mult doi succesori.

În general construirea unui arbore începe cu rădăcina, la care se adaugă noduri fii, la care se adaugă alți fii în mod recursiv, cu creșterea adâncimii (înălțimii) arborelui. Există însă și câteva excepții (arbori Huffman, arbori pentru expresii aritmetice), care se construiesc de la frunze către rădăcină.

Cuvântul “arbore” se folosește și pentru un caz particular de grafuri fără cicluri, la care orice vârf poate fi privit ca rădăcină. Diferența dintre arborii cu rădăcină (din acest capitol) și arborii liberi (grafuri aciclice) este că primii conțin în noduri date importante pentru aplicații, iar arborii grafuri nu conțin date în noduri (dar arcele ce unesc aceste noduri pot avea asociate valori sau costuri).

Structurile arborescente se folosesc în programare deoarece:

- Reprezintă un model natural pentru o ierarhie de obiecte (entități, operații etc).
- Sunt structuri de căutare cu performanțe foarte bune, permițând și menținerea în ordine a unei colecții de date dinamice (cu multe adăugări și stergeri).

De cele mai multe ori legăturile unui nod cu succesorii săi se reprezintă prin pointeri, dar sunt posibile și reprezentări fără pointeri ale arborilor, prin vectori.

De obicei se înțelege prin arbore o structură cu pointeri, deoarece aceasta este mai eficientă pentru arbori multicai și pentru arbori binari cu structură imprevizibilă.

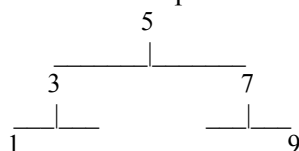
O reprezentare liniară posibilă a unui arbore este o expresie cu paranteze complete, în care fiecare nod este urmat de o paranteză ce grupează succesorii săi. Exemple:

1) a (b,c)

este un arbore binar cu 3 noduri: rădăcina 'a', având la stânga pe 'b' și la dreapta pe 'c'

2) 5 (3 (1,), 7(9))

este un arbore binar ordonat cu rădăcina 5. Nodul 3 are un singur succesor, la stânga, iar nodul 7 are numai succesor la dreapta.



Afișarea arborilor binari sau multicai se face de obicei prefixat și cu indentare diferită la fiecare nivel (fiecare valoare pe o linie, iar valorile de pe același nivel în aceeași coloană). Exemplu de afișare prefixată, cu indentare, a arborelui de mai sus:

```

5
 3
  1
  -
 7
  -
  9
  
```

Uneori relațiile dintre nodurile unui arbore sunt impuse de semnificația datelor memorate în noduri (ca în cazul arborilor ce reprezintă expresii aritmetice sau sisteme de fisiere), dar alteori distribuția valorilor memorate în noduri nu este impusă, fiind determinată de valorile memorate (ca în cazul arborilor de căutare, unde structura depinde de ordinea de adăugare și poate fi modificată prin reorganizarea arborelui).

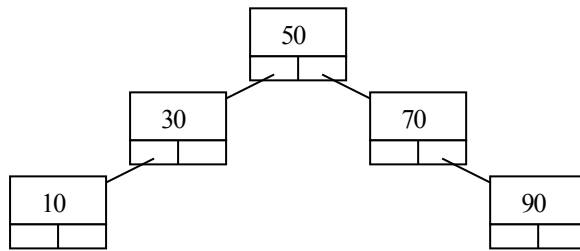
7.2 ARBORI BINARI NEORDONAȚI

Un caz particular important de arbori îl constituie arborii binari, în care un nod poate avea cel mult doi succesori: un succesor la stânga și un succesor la dreapta.

Arborii binari pot avea mai multe reprezentări, folosind pointeri sau vectori. Structura de arbore cu pointeri permite o bună utilizare a memoriei și un timp de prelucrare bun pentru arbori cu multe noduri și cu o repartizare neuniformă a lor pe diverse niveluri din arbore.

Mentionăm câteva reprezentări folosite pentru arbori binari:

a) Noduri (structuri) cu pointeri pentru legături părinte-fii. Exemplu:



b) Reprezentare prin 3 vectori: valoare, indice fiu stânga, indice fiu dreapta. Exemplu:

indici	1	2	3	4	5
val	50	70	30	10	90
st	3	0	4	0	0
dr	2	5	0	0	0

c) Reprezentare prin 3 vectori: valoare, valoare fiu stânga, valoare fiu dreapta (mai compact, fără frunze, dar necesită căutarea fiecărui fiu). Exemplu:

val	50	70	30
st	30	-1	10
dr	70	90	-1

d) Reprezentare printr-un singur vector, nivel cu nivel din arbore :

val	50	30	70	10	-1	-1	90
-----	----	----	----	----	----	----	----

Un arbore relativ echilibrat poate fi reprezentat eficient printr-un singur vector, după ideea unui vector heap, chiar dacă nu este complet fiecare nivel din arbore (valorile lipsă fiind marcate printr-o valoare specială); în acest caz relațiile dintre noduri părinte-fiu nu mai trebuie memorate explicit (prin indici sau valori noduri), ele rezultă implicit din poziția fiecărui element în vector (se pot calcula).

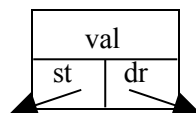
Această reprezentare devine ineficientă pentru arbori cu înălțime mare dar cu număr de noduri relativ mic, deoarece numărul de noduri într-un arbore complet crește exponențial cu înălțimea sa. De aceea s-au propus soluții bazate pe vectori de biti: un vector de biti conține 1 pentru un nod prezent și 0 pentru un nod absent într-o liniarizare nivel cu nivel a arborelui, iar valorile din noduri sunt memorate separat dar în aceeași ordine de parcurgere a nodurilor (în lărgime). Pentru arborele folosit ca exemplu vectorul de biti va fi 1111001, iar vectorul de valori va fi 50,30,70,10,90.

Definiția unui nod dintr-un arbore binar, cu pointeri către cei doi succesori posibili

```

typedef struct tnod {
    T val;           // valoare memorata in nod, de tipul T
    struct tnod * st; // succesor la stânga
    struct tnod * dr; // succesor la dreapta
} tnod;

```



Uneori se memorează în fiecare nod și adresa nodului părinte, pentru a ajunge repede la părintele unui nod (pentru parcurgere de la frunze către rădăcină sau pentru modificarea structurii unui arbore). Nodurile terminale pot conține valoarea NULL sau adresa unui nod sentinelă. Adresa către nodul părinte și utilizarea unui nod unic sentinelă sunt utile pentru arborii echilibrati, care își modifică structura.

Un arbore este complet definit printr-o singură variabilă pointer, care conține adresa nodului rădăcină; pornind de la rădăcină se poate ajunge la orice nod.

Operațiile cu arbori, considerați drept colecții de date, sunt:

- Initializare arbore (creare arbore vid);
 - Adăugarea unui nod la un arbore (ca frunză);
 - Căutarea unei valori date într-un arbore;
 - Eliminarea (stergere) unui nod cu valoare dată;
 - Enumerarea tuturor nodurilor din arbore într-o anumită ordine.
- Alte operații cu arbori, utile în anumite aplicații :
- Determinarea valorii minime sau maxime dintr-un arbore
 - Determinarea valorii imediat următoare valorii dintr-un nod dat
 - Determinarea rădăcinii arborelui ce conține un nod dat
 - Rotatii la stânga sau la dreapta noduri

Enumerarea (afisarea) nodurilor unui arbore cu N noduri necesită $O(N)$ operații. Durata operațiilor de adăugare și de eliminare noduri depinde de înălțimea arborelui.

Initializarea unui arbore vid se poate reduce la atribuirea valorii NULL pentru variabila rădăcină, sau la crearea unui nod sentinelă, fără date. Poate fi luată în considerare și o initializare a rădăcinii cu prima valoare introdusă în arbore, astfel ca adăugările ulterioare să nu mai modifice rădăcina (dacă nu se face modificarea arborelui pentru reechilibrare, după fiecare adăugare).

Funcțiile pentru operații cu arbori binari sunt natural recursive, pentru că orice operație (afisare, căutare etc) se reduce la operații similare cu subarborii stânga și dreapta, plus operația asupra rădăcinii. Reducerea (sub)arborilor continuă până se ajunge la un (sub)arbore vid.

Vizitarea nodurilor unui arbore binar înseamnă de fapt liniarizarea arborelui, adică stabilirea unei secvențe liniare de noduri. În funcție de ordinea în care se iau în considerare rădăcina, subarboarele stânga și subarboarele dreapta putem vizita în:

- Ordine prefixată (preordine sau RSD) : rădăcină, stânga, dreapta
- Ordine infixată (inordine sau SRD) : stânga, rădăcină, dreapta
- Ordine postfixată (postordine sau SDR): stânga, dreapta, rădăcină

Fie arborele binar descris prin expresia cu paranteze:

5 (2 (1,4(3,)), 8 (6 (,7),9))

Traversarea prefixată produce secvența de valori: 5 2 1 4 3 8 6 7 9

Traversarea infixată produce secvența de valori: 1 2 3 4 5 6 7 8 9

Traversarea postfixată produce secvența de valori: 1 3 4 2 7 6 9 8 5

Exemplu de funcție pentru afisare infixată a valorilor dintr-un arbore binar:

```
void infix (tnod * r) {
    if ( r == NULL) return;    // nimic daca (sub)arbore vid
    infix (r->st);             // afisare subarbore stânga
    printf ("%d ",r->val);      // afisare valoare din radacina
    infix (r->dr);             // afisare subarbore dreapta
}
```

Funcția "infix" poate fi ușor modificată pentru o altă strategie de vizitare. Exemplu

```
// traversare prefixata arbore binar
void prefix (tnod * r) {
    if ( r == NULL) return;
    printf ("%d ",r->val);      // radacina
    prefix (r->st);             // stânga
    prefix (r->dr);             // dreapta
}
```

Pornind de la functia minimală de afisare se pot scrie si alte variante de afisare: ca o expresie cu paranteze sau cu evidentierea structurii de arbore:

```
// afisare structura arbore (cu indentare)
void printT (tnod * r, int ns) {          // ns = nr de spatii la inceput de linie
    if ( r != NULL) {
        printf ("%*c%d\n",ns,' ',r->val);    // scrie r->val dupa ns spatii
        printT (r->st,ns+3);                // subarbore stanga, decalat cu 3 spatii
        printT (r->dr,ns+3);                // subarbore dreapta, decalat cu 3 sp.
    }
}
```

Majoritatea operatiilor cu arbori pot fi considerate drept cazuri de vizitare (parcurgere, traversare) a tuturor nodurilor din arbore; diferenta constă în operatia aplicată nodului vizitat: afisare, comparare, adunare nod sau valoare la o sumă, etc.

Un exemplu este determinarea înălțimii unui arbore; înălțimea unui arbore este egală cu cea mai mare dintre înălțimile subarborilor săi, plus unu pentru rădăcină, pentru un arbore nevid si zero pentru arborele vid.

```
int maxpath (tnod * r)
    if (r == NULL)    return 0;          // inaltime arbore vid
    else return max (maxpath (r->st), maxpath(r->dr))+1;
}
```

În mod similar se poate scrie o functie care să numere nodurile dintr-un arbore.

Căutarea unei valori într-un arbore binar se reduce la căutarea succesivă în fiecare din cei doi subarbori:

```
tnod * find ( tnod * r, int x) { // cauta x in arborele cu radacina r
    tnod * p;
    if (r==NULL || x == r->val) // daca arbore vid sau x in nodul r
        return r;             // poate fi si NULL
    p= find (r->st,x);          // rezultat cautare in subarbore stanga
    if (p != NULL)              // daca s-a gasit in stanga
        return p;
    else                        // daca nu s-a gasit in stanga
        return find (r->dr,x);  // rezultat cautare in subarbore dreapta
}
```

Explorarea unui arbore sau unui graf în lățime (pe niveluri succesive) necesită memorarea succesorilor (fiilor) unui nod. După vizitarea nodului de plecare (rădăcina în cazul unui arbore) se pun în coadă toti succesorii lui, care vor fi apoi extrasi în ordinea în care au fost pusi. După ce se extrage un nod se adaugă la sfârșitul cozii succesorii lui. În felul acesta, fii unui nod sunt prelucrați după frații nodului respectiv.

Exemplu de evolutie a cozii de pointeri la noduri pentru arborele binar următor:

5 (3 (2 , 4) , 7 (6 , 8))

```
5
3 7      (scrie 5)
7 2 4    (scrie 3)
2 4 6 8  (scrie 7)
4 6 8    (scrie 2)
6 8      (scrie 4)
8         (scrie 6)
-         (scrie 8)
```

```
// vizitare arbore binar nivel cu nivel folosind o coadă
void bfs_bin ( tnod * r) { // vizitare nivel cu nivel
    Queue q; // q este o coada de pointeri void*
    initQ(q); // initial coada vida
    addQ (q,r); // adauga radacina la coada
    while (!emptyQ(q)) { // cat timp mai e ceva in coada
        r=(tnod*) delQ (q); // scoate adresa nod din coada
        printf ("%d ", r->val); // pune valoare din nod in vectorul v
        if (r->st) addQ (q, r->st); // adauga la coada fiu stanga
        if (r->dr) addQ (q, r->dr); // adauga la coada fiu dreapta
    }
    printf("\n");
}
```

In varianta prezentată am considerat $r \neq \text{NULL}$ si nu s-au mai pus în coadă si pointerii egali cu NULL, dar este posibilă si varianta următoare:

```
void bfs_bin ( tnod * r) {
    Queue q; // o coada de pointeri void*
    initQ(q); // initial coada vida
    addQ (q,r); // adauga radacina la coada
    while (!emptyQ(q)) { // cat timp mai e ceva in coada
        r= (tnod*) delQ (q); // scoate adresa nod din coada
        if ( r !=NULL) { // daca pointer nenul
            printf ("%d ", r->val); // scrie valoare din nod
            addQ (q, r->st); // adauga la coada fiu stanga (chiar NULL)
            addQ (q, r->dr); // adauga la coada fiu dreapta (chiar NULL)
        }
    }
    printf ("\n");
}
```

Vizitarea nodurilor unui arbore binar în adâncime, prefixat, se poate face în mod asemănător, dar folosind o stivă în loc de coadă pentru memorarea adreselor nodurilor prin care s-a trecut dar fără prelucrarea lor, pentru o revenire ulterioară.

```
void dfs_bin (tnod * r) {
    Stack s; // o stiva de pointeri void*
    initSt(s); // initializare stiva vida
    push (s, r); // pune adresa radacina pe stiva
    while ( ! emptySt (s)) { // repeta cat timp e ceva in stiva
        r=(tnod*)pop(s); // scoate din stiva adresa nod
        printf ("%d ",r->val); // afisare valoare din nod
        if ( r->dr != NULL) // daca exista fiu dreapta
            push (s,r->dr); // pune pe stiva fiu dreapta
        if ( r->st != NULL) // daca exista fiu stanga
            push (s, r->st); // pune pe stiva fiu stanga
    }
    printf ("\n");
}
```

De observat ordinea punerii pe stivă a fiilor unui nod (fiu dreapta si apoi fiu stânga), pentru ca la scoatere din stivă si afisare să se scrie în ordinea stânga-dreapta.

Evolutia stivei la afisarea infixată a arborelui binar: 5(3 (2,4) , 7(6,8))

Operatie	Stiva	Afisare
----------	-------	---------

initSt	-	
push (&5)	&5	
pop	-	5
push (&7)	&7	
push (&3)	&7,&3	
pop	&7	3
push(&4)	&7,&4	
push(&2)	&7,&4,&2	
pop	&7,&4	2
pop	&7	4
pop	-	7
push (&8)	&8	
push (&6)	&8,&6	
pop	&8	6
pop	-	8

După modelul afisării prefixate cu stivă se pot scrie nerecursiv si operatiile de căutare nod cu valoare dată, numărare noduri s.a. Exemplu de căutare iterativă:

```
tnod* find (tnod* r, int x) { // cauta valoarea x in arborele cu radacina r
    Stiva s; // o stiva de pointeri void*
    initS(s); // initializare stiva
    if (r==NULL) return NULL; // daca arbore vid atunci x negasit
    push (s,r); // pune pe stiva adresa radacinii
    while ( ! emptyS(s)) { // repeta pana la golirea stivei
        r= (tnod*) pop(s); // scoate adresa nod din stiva
        if (x==r->val) return r; // daca x gasit in nodul cu adresa r
        if (r->st) push(s,r->st); // daca exista fiu stanga, se pune pe stiva
        if (r->dr) push(s,r->dr); // daca exista fiu dreapta, se pune pe stiva
    }
    return NULL; // daca x negasit in arbore
}
```

Adăugarea de noduri la un arbore binar se poate face în mai multe moduri:

- adăugarea la un arbore de căutare plasează noul nod în funcție de valoarea sa;
- adăugarea la un arbore completat nivel cu nivel (ca un “heap”) poate folosi o coadă;
- adăugarea la un arbore cu orice structură folosește funcții de felul următor:

```
void addLeft (tnod* p, tnod* left); // adauga lui p un fiu stanga
void addRight (tnod* p, tnod* right); // adauga lui p un fiu dreapta
```

În exemplul următor se consideră că datele folosite la construirea arborelui se dau sub forma unor tripleți de valori: valoare nod părinte, valoare fiu stânga, valoare fiu dreapta. O valoare zero marchează absența fiului respectiv. Exemplu de date:

5 3 7 / 7 6 8 / 3 2 4 / 2 1 0 / 8 0 9

```
// creare si afisare arbore binar
int main () {
    int p,s,d; tnod* w, *r=NULL;
    while (scanf("%d%d%d",&p,&s,&d) == 3) {
        if (r==NULL) // daca arbore vid
            r=build(p); // primul nod (radacina)
        w=find (r,p); // adresa nodului parinte (cu valoarea p)
        if (s!=0)
            addLeft (w,s); // adauga s ca fiu stanga a lui w
        if (d!=0)
            addRight (w,d); // adauga d ca fiu dreapta a lui w
    }
}
```

```

infix (r);          // afisare infixata
}
    
```

7.3 ARBORI BINARI PENTRU EXPRESII

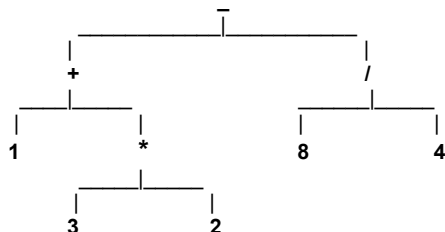
Reprezentarea unei expresii (aritmetice, logice sau de alt tip) în compilatoare se poate face fie printr-un sir postfixat, fie printr-un arbore binar; arborele permite si optimizări la evaluarea expresiilor cu subexpresii comune. Un sir postfixat este de fapt o altă reprezentare, liniară, a unui arbore binar.

Reprezentarea expresiilor prin arbori rezolvă problema ordinii efectuării operațiilor prin poziția operatorilor în arbore, fără a folosi paranteze sau priorități relative între operatori: operatorii sunt aplicati începând de la frunze către rădăcină, deci în ordine postfixată.

Construcția arborelui este mai simplă dacă se porneste de la forma postfixată sau prefixată a expresiei deoarece nu există problema priorității operatorilor și a parantezelor; construirea progresa de la frunze spre rădăcină. Un algoritm recursiv este mai potrivit dacă se pleacă de la sirul prefixat, iar un algoritm cu stivă este mai potrivit dacă se pleacă de la sirul postfixat.

Pentru simplificarea codului vom considera aici numai expresii cu operanzi dintr-o singură cifră, cu operatorii aritmetici binari '+', '-', '*', '/' și fără spații albe între operanzi și operatori. Eliminarea acestor restricții nu modifică esența problemei și nici soluția discutată, dar complică implementarea ei.

Pentru expresia $1+3*2-8/4$ arborele echivalent arată astfel:



Operanzii se află numai în noduri terminale iar operatorii numai în noduri interne. Evaluarea expresiei memorate într-un arbore binar este un caz particular de vizitare postfixată a nodurilor arborelui și se poate face fie recursiv, fie folosind o stivă de pointeri la noduri. Nodurile sunt interpretate diferit (operanzi sau operatori), fie după conținutul lor, fie după poziția lor în arbore (terminale sau neterminale).

Evaluarea recursivă a unui arbore expresie se poate face cu funcția următoare.

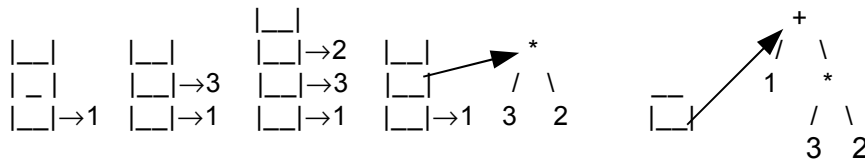
```

int eval (tnod * r) {
    int vst, vdr ;          // valoare din subarbore stanga si dreapta
    if (r == NULL)
        return 0;
    if ( isdigit(r->val))    // daca este o cifra
        return r->val - '0'; // valoare operand
    // operator
    vst = eval(r->st);        // valoare din subarbore stanga
    vdr = eval(r->dr);        // valoare din subarbore dreapta
    switch (r->val) {         // r->val este un operator
        case '+': return vst + vdr;
        case '*': return vst * vdr;
        case '-': return vst - vdr;
        case '/': return vst / vdr;
    }
    return 0;
}
    
```

Algoritmul de creare arbore pornind de la forma postfixată sau prefixată seamănă cu algoritmul de evaluare a unei expresii postfixate (prefixate). Funcția următoare folosește o stivă de pointeri la noduri și creează (sub)arbori care se combină treptat într-un singur arbore final.

```
tnod * buidtree ( char * exp) {           // exp= sir postfixat terminat cu 0
Stack s ; char ch;                       // s este o stiva de pointeri void*
tnod* r=NULL;                            // r= adresa radacina subarbore
initSt(s);                               // initializare stiva goala
while (ch=*exp++) {                       // repeta pana la sfarsitul expresiei exp
    r=new tnode;                           // construire nod de arbore
    r->val=ch;                             // cu operand sau operator ca date
    if (isdigit(ch))                       // daca ch este operand
        r->st=r->dr=NULL;                  // atunci nodul este o frunza
    else {                                 // daca ch este operator
        r->dr =(tnod*)pop (s);              // la dreapta un subarbore din stiva
        r->st= (tnod*)pop (s);              // la stanga un alt subarbore din stiva
    }
    push (s,r);                           // pune radacina noului subarbore in stiva
}
return r;                                 // radacina arbore creat { return(tnod*)pop(s);}
}
```

Pentru expresia postfixată 132*+84/- evoluția stivei după 5 pași va fi următoarea:



Funcția următoare creează un arbore binar pornind de la o expresie prefixată:

```
tnod* build ( char p[], int & i) {         // p este sirul prefixat, terminat cu zero
tnod* nou= (tnod*) malloc(sizeof (tnod)); // creare nod nou
if (p[i]==0) return NULL;                 // daca sfarsit sir prefixat
if ( isdigit(p[i])) {                     // daca este o cifra
    nou->val=p[i++];                       // se pune operand in nod
    nou->st=nou->dr=NULL;                   // nodul este o frunza
}
else {                                     // daca este operator
    nou->val=p[i++];                       // se pune operator in nod
    nou->st= build(p,i);                    // primul operand
    nou->dr= build (p,i);                   // al doilea operand
}
return nou;                               // nod creat (in final, radacina)
}
```

Crearea unui arbore dintr-o expresie infixată, cu paranteze (forma uzuală) se poate face modificând funcțiile mutual recursive care permit evaluarea acestei expresii.

7.4 ARBORI HUFFMAN

Arborii Huffman sunt arbori binari folosiți într-o metodă de compresie a datelor care atribuie fiecărui caracter (octet) un cod binar a cărui lungime depinde de frecvența octetului codificat; cu cât un caracter apare mai des într-un fișier cu atât se folosesc mai puțini biți pentru codificarea lui. De exemplu, într-un fișier apar 6 caractere cu următoarele frecvențe:

a (45), b(13), c(12), d(16), e(9), f(5)

Codurile Huffman pentru aceste caractere sunt:

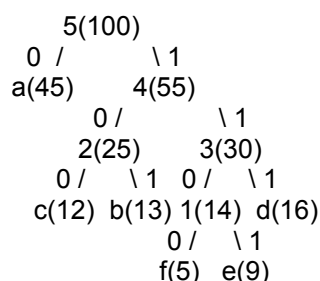
a= 0, b=101, c=100, d=111, e=1101, f=1100

Numărul de biti necesari pentru un fisier de 1000 caractere va fi 3000 în cazul codificării cu câte 3 biti pentru fiecare caracter si 2240 în cazul folosirii de coduri Huffman, deci se poate realiza o compresie de cca. 25% (în cele 1000 de caractere vor fi 450 de litere 'a', 130 de litere 'b', 120 litere 'c', s.a.m.d).

Fiecare cod Huffman începe cu un prefix distinct, ceea ce permite recunoasterea lor la decompresie; de exemplu fisierul comprimat 001011101 va fi decodificat ca 0/0/101/1101 = aabe.

Problema este de a stabili codul fiecărui caracter functie de probabilitatea lui de aparitie astfel încât numărul total de biti folositi în codificarea unui sir de caractere să fie minim. Pentru generarea codurilor de lungime variabilă se foloseste un arbore binar în care fiecare nod neterminal are exact doi succesori.

Pentru exemplul dat arborele de codificare cu frecventele de aparitie în nodurile neterminale si cu literele codificate în nodurile terminale este :



Se observă introducerea unor noduri intermediare notate cu cifre.

Pentru codificare se parcurge arborele începând de la rădăcină si se adaugă câte un bit 0 pentru un succesor la stânga si câte un bit 1 pentru un succesor la dreapta.

Construirea unui arbore Huffman seamănă cu construirea arborelui echivalent unei expresii aritmetice: se construiesc treptat subarbori cu număr tot mai mare de noduri până când rezultă un singur arbore. Diferenta este că în cazul expresiilor se foloseste o stivă pentru memorarea rădăcinilor subarborilor, iar în algorimul Huffman se foloseste o coadă cu priorități de subarbori binari care se combină treptat.

Algoritmul generează arborele de codificare începând de jos în sus, folosind o coadă cu priorități, ordonată crescător după frecventa de aparitie a caracterelor. La fiecare pas se extrag primele două elemente din coadă (cu frecvente minime), se creează cu ele un subarbore si se introduce în coadă un element a cărui frecvență este egală cu suma frecventelor elementelor extrase.

Coadă poate memora adrese de noduri de arbore sau valori din nodurile rădăcină (dar atunci mai este necesară o căutare în arbore pentru aflarea adresei nodului).

Evoluția cozii de caractere si frecvente pentru exemplul dat este :

```

f(5), e(9), c(12), b(13), d(16), a(45)
c(12), b(13), 1(14), d(16), a(45)
1(14), d(16), 2(25), a(45)
2(25), 3(30), a(45)
a(45), 4(55)
5(100)
    
```

Elementele noi adăugate la coadă au fost numerotate în ordinea producerii lor.

La început se introduc în coadă toate caracterele, sau pointeri la noduri de arbore construite cu aceste caractere si frecventa lor. Apoi se repetă n-1 pasi (sau până când coada va contine un singur element) de forma următoare:

- extrage si sterge din coadă primele două elemente (cu frecventa minimă)

- construiește un nou nod cu suma frecvențelor și având ca subarbori adresele scoase din coadă
- introduce în coadă adresa noului nod (rădăcină a unui subarbor)

Exemple de definire a unor tipuri de date utilizate în continuare:

```
typedef struct hnod {      // un nod de arbore Huffman
    char ch ; int fr;      // un caracter si frecventa lui de utilizare
    struct hnod *st,*dr;   // adrese succesori
} hnod;
```

Funcția următoare construiește arborele de codificare:

```
// creare arbore de codificare cu radacina r
int build (FILE* f, hnod* & r ) {    // f= fisier cu date (caractere si frecvente)
    hnod *t1,*t2,*t3;
    int i,n=0; char ch, s[2]={0};
    int fr2,fr;
    pq q;                          // coada cu prioritati de pointeri hnod*
    initPQ (q);                    // initial coada e vida
    // citire date din fisier si adaugare la coada
    while ( fscanf(f,"%1s%d",s,&fr) != EOF){
        addPQ (q, make(s[0], fr, NULL,NULL));    // make creeaza un nod
        n++;                                     // n= numar de caractere distincte
    }
    // creare arbore
    i=0;                                // folosit la numerotare noduri interne
    while ( ! emptyPQ(q)) {
        t1= delPQ(q);                    // extrage adresa nod in t1
        if (emptyPQ(q)) break;
        t2= delPQ(q);                    // extrage adresa nod in t2
        fr2 = t1->fr + t2->fr;            // suma frecvențelor din cele doua noduri
        ch= i+'1'; i++;                  // ch este caracterul din noul nod
        t3 = make(ch,fr2,t1,t2);         // creare nod cu ch si fii t1 si t2
        addPQ (q, t3);                   // adauga adresa nod creat la coada q
    }
    r=t1;                                // ultimul nod din coada este radacina arborelui
    return n;                             // numar de caractere
}
```

Determinarea codului Huffman al unui caracter c înseamnă aflarea căii de la rădăcină la nodul ce conține caracterul c, prin căutare în arbore. Pentru simplificarea programării și verificării vom genera siruri de caractere '0' și '1' și nu configurații binare (siruri de biți 0).

Funcția următoare produce codul Huffman al unui caracter dat ca sir de cifre binare (terminat cu zero), dar în ordine inversă (se poate apoi inversa cu "strrev"):

```
// codificare caracter ch pe baza arborelui a; hc=cod Huffman
char* encode (hnod* r, char ch, char* hc) {
    if (r==NULL) return r;
    if (r->val.ch==ch) return hc;        // daca s-a gasit nodul cu caracterul ch
    if (encode (r->st, ch, hc))          // cauta in subarbori stanga
        return strcat(hc,"0");          // si adauga cifra 0 la codul hc
    if (encode (r->dr, ch, hc))          // cauta in subarbori dreapta
        return strcat(hc,"1");          // si adauga cifra 1 la codul hc
    else                                 // daca ch negasit in arborele a
        return NULL;
}
```

Un program pentru decompresie Huffman trebuie să primească atât fișierul codificat cât și arborele folosit la compresie (sau datele necesare pentru reconstruirea sa). Arborele Huffman (și orice arbore binar) poate fi serializat într-o formă fără pointeri, prin 3 vectori care să conțină valoarea (caracterul) din fiecare nod, valoarea fiului stânga și valoarea fiului dreapta. Exemplu de arbore serializat:

```
car  5  4  2  3  1
st   a  2  c  1  f
dr   4  3  b  d  e
```

Pentru decodificare se parcurge arborele de la rădăcină spre stânga pentru o cifră zero și la dreapta pentru o cifră 1; parcurgerea se reia de la rădăcină pentru fiecare secvență de biți arborele Huffman. Funcția următoare folosește tot arborele cu pointeri pentru afisarea caracterelor codificate Huffman într-un sir de cifre binare:

```
void decode (hnod* r, char* ht) {    // ht = text codificat Huffman (cifre 0 si 1)
    hnod* p;
    while ( *ht != 0) {              // cat timp nu e sfarsit de text Huffman
        p=r;                         // incepe cu radacina arborelui
        while (p->st!=NULL) {         // cat timp p nu este nod frunza
            if (*ht=='0')              // daca e o cifra 0
                p= p->st;              // spre stanga
            else                       // daca e o cifra 1
                p=p->dr;               // spre dreapta
            ht++;                      // si scoate alta cifra din ht
        }
        putchar(p->ch);               // scrie sau memoreaza caracter ASCII
    }
}
```

Deoarece pentru decodificare este necesară trimiterea arborelui Huffman împreună cu fișierul codificat, putem construi de la început un arbore fără pointeri, cu vectori de indici către succesorii fiecărui nod. Putem folosi un singur vector de structuri (o structură corespunde unui nod) sau mai multi vectori reuniti într-o structură. Exemplu de definire a tipurilor de date folosite într-un arbore Huffman fără pointeri, cu trei vectori: de date, de indici la fii stânga și de indici la fii dreapta.

```
#define M 100          // dimensiune vectori (nr. maxim de caractere)
typedef struct {
    int ch;             // cod caracter
    int fr;             // frecventa de aparitie
} cf;                  // o pereche caracter-frecventa
typedef struct {
    cf c[M];            // un vector de structuri
    int n;              // dimensiune vector
} pq;                  // coada ca vector ordonat de structuri
typedef struct {
    int st[M], dr[M];   // vectori de indici la fii
    cf v[M];            // valori din noduri
    int n;              // nr de noduri in arbore
} ht;                  // arbore Huffman
```

Vom exemplifica cu funcția de codificare a caracterelor ASCII pe baza arborelui:

```
char* encode (bt a, int k, char ch, char* hc) {    // hc initial un sir vid
    if (k<0) return 0;
    if (a.v[k].ch==ch)                             // daca s-a gasit caracterul ch in arbore
        return hc ;                                // hc contine codul Huffman inversat
    if (encode (a,a.st[k],ch, hc))                  // daca ch e la stanga
```

```

    return strcat(hc,"0");           // adauga zero la cod
    if (encode (a,a.dr[k],ch,hc))    // daca ch e la dreapta
        return strcat(hc,"1");      // adauga 1 la cod
    else return 0;
}

```

Arborele Huffman este de fapt un dictionar care asociază fiecărui caracter ASCII (cheia) un cod Huffman (valoarea asociată cheii); la codificare se caută după cheie iar la decodificare se caută după valoare (este un dictionar bidirecțional). Implementarea ca arbore permite căutarea rapidă a codurilor de lungime diferită, la decodificare.

7.5 ARBORI GENERALI (MULTICĂI)

Un arbore general ("Multiway Tree") este un arbore în care fiecare nod poate avea orice număr de succesori, uneori limitat (arbori B și 2-4) dar de obicei nelimitat.

Arborii multicăi pot fi clasificați în două grupe:

- Arbori de căutare, echilibrați folosiți pentru mulțimi și dictionare (arbori B);
- Arbori care exprimă relațiile dintre elementele unei colecții și a căror structură nu mai poate fi modificată pentru reechilibrare (nu se pot schimba relațiile părinte-fiu).

Multe structuri arborescente "naturale" (care modelează situații reale) nu sunt arbori binari, iar numărul succesorilor unui nod nu este limitat. Exemplele cele mai cunoscute sunt: arborele de fișiere și subdirectoare care reprezintă conținutul unui volum disc și arborele ce reprezintă conținutul unui fișier XML.

În arborele XML (numit și arbore DOM) nodurile interne corespund marcajelor de început ("start tag"), iar nodurile frunză conțin textele dintre marcaje pereche.

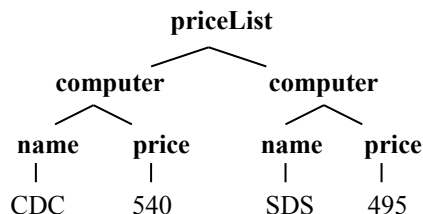
În arborele creat de un parser XML (DOM) pe baza unui document XML fiecare nod corespunde unui element XML. Exemplu de fișier XML:

```

<priceList>
  <computer>
    <name> CDC </name>
    <price> 540 </price>
  </ computer >
  <computer>
    <name> SDS </name>
    <price> 495 </price>
  </ computer >
</priceList>

```

Arborele DOM (Document Object Model) corespunzător acestui document XML:



Sistemul de fișiere de pe un volum are o rădăcină cu nume constant, iar fiecare nod corespunde unui fișier; nodurile interne sunt subdirectoare, iar nodurile frunză sunt fișiere "normale" (cu date). Exemplu din sistemul MS-Windows:

```

\
  Program Files
    Adobe
      Acrobat 7.0
        Reader
      ...
    Internet Explorer
      ...
      iexplorer.exe
    WinZip
      winzip.txt
      wz.com
      wz.pif
      ...
  WINDOWS
    ...
  
```

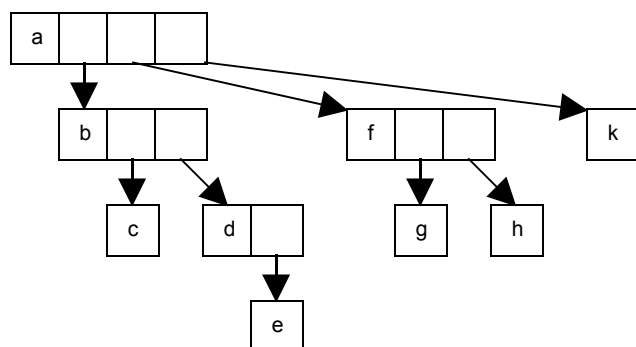
Un arbore multicăi cu rădăcină se poate implementa în cel puțin două moduri:

a) - Fiecare nod conține un vector de pointeri la nodurile fii (succesori directi) sau adresa unui vector de pointeri, care se extinde dinamic.

De exemplu, arborele descris prin expresia cu paranteze următoare:

a (b (c, d (e)), f (g, h), k)

se va reprezenta prin vectori de pointeri la fii ca în figura următoare:



În realitate numărul de pointeri pe nod va fi mai mare decât cel strict necesar (din motive de eficiență vectorul de pointeri nu se extinde prin mărirea capacității cu 1 ci prin dublarea capacității sau prin adunarea unui increment constant):

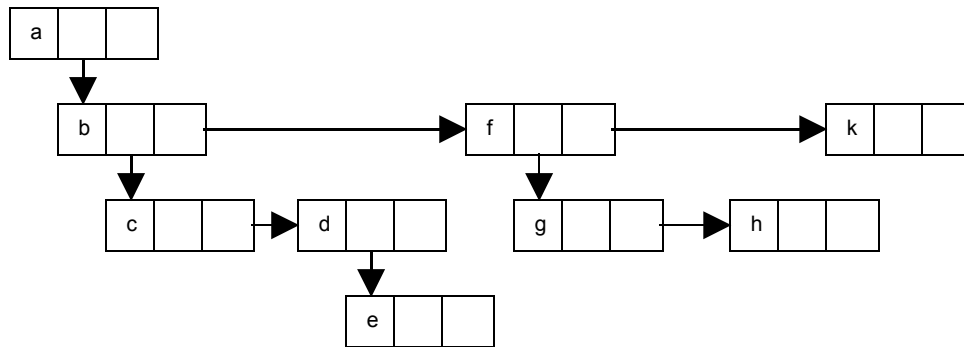
```

// definitia unui nod de arbore cu vector extensibil de fii
typedef struct tnod {
    int val;                // valoare (date) din nod
    int nc;                 // numar de fii ai acestui nod
    struct tnod ** kids;    // vector cu adrese noduri fii
} tnod;
  
```

b) - Fiecare nod conține 2 pointeri: la fiul stânga și la fratele următor ("left son, right sibling"). În acest fel un arbore multicăi este redus la un arbore binar. Putem considera și că un nod conține un pointer la lista de fii și un pointer la lista de frați. De exemplu, arborele

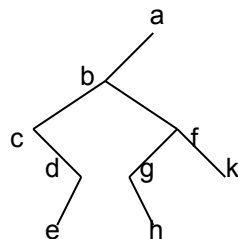
a (b (c,d (e)), f (g, h), k)

se va reprezenta prin legături la fiul stânga și la fratele dreapta astfel:



Structura unui astfel de arbore este similară cu structura unei liste Lisp: “car” corespunde cu adresa primului fiu iar “cdr” cu adresa primului frate al nodului curent.

Desenul următor arată arborele general anterior ca arbore binar:



Succesorul din stânga al unui nod reprezintă primul fiu, iar succesorul din dreapta este primul frate. În reprezentarea fiu-frate un nod de arbore poate fi definit astfel:

```
typedef struct tnod {
    int val;
    struct tnod *fiu, *frate;
} tnod;
```

Exemple de funcții pentru operații cu arbori ce contin adrese către fiu și frate :

```
// afisare arbore multicai fiu-frate
void print (tnod* r, int ns) {          // ns= nr de spatii ptr acest nivel
    if (r != NULL) {
        printf ("%*c%d\n", ns, ' ', r->val); // valoare nod curent
        print (r->fiu, ns+2);              // subarbore cu radacina in primul fiu
        r=r->fiu;
        while ( r != NULL) {              // cat mai sunt frati pe acest nivel
            print (r->frate, ns+2);         // afisare subarbore cu radacina in frate
            r=r->frate;                    // si deplasare la fratele sau
        }
    }
}
```

```
// adaugare fiu la un nod crt
void addChild (tnod* crt, tnod* child) {
    tnod* p;
    if ( crt->fiu == NULL)                // daca este primul fiu al nodului crt
        crt->fiu=child;                   // child devine primul din lista
    else {                                // daca child nu este primul fiu
        p=crt->fiu;                        // adresa listei de fii
        while (p->frate != NULL)           // mergi la sfarsitul listei de fii ai lui crt
            p=p->frate;
```

```

        p=p→frate;
    p→frate=child;          // adauga child la sfarsitul listei
}
}

```

Pentru afisare, căutare și alte operații putem folosi funcțiile de la arbori binari, față de care adresa primului fiu corespunde subarborelui stânga iar adresa fratelui corespunde subarborelui dreapta.

Exemple de funcții pentru arbori generali văzuți ca arbori binari:

```

// afisare prefixata cu indentare (ns=nivel nod r)
void print (tnod* r, int ns) {
    if (r !=NULL) {
        printf("%*c%d\n",ns,' ',r→val);
        print(r→fiu,ns+3);
        print (r→frate,ns);
    }
}

// cautare x in arbore
tnod* find (tnod*r, int x) {
    tnod* p;
    if (r==NULL) return r;          // daca arbore vid atunci x negasit
    if (x==r→val)                   // daca x in nodul r
        return r;
    p=find(r→fiu,x);               // cauta in subarbore stanga (in jos)
    return p? p: find(r→frate, x); // sau cauta in subarbore dreapta
}

#define max(a,b) ( (a)>(b)? (a): (b) )
// inaltime arbore multicaie (diferita de inaltime arbore binar)
int ht (tnod* r) {
    if (r==NULL)
        return 0;
    return max ( 1+ ht(r→fiu), ht(r→frate));
}

```

Pentru arborii ce contin vectori de fii în noduri vom considera că vectorul de pointeri la fii se extinde cu 1 la fiecare adăugare a unui nou fiu, deși în acest fel se poate ajunge la o fragmentare excesivă a memoriei alocate dinamic.

```

// creare nod frunza
tnod* make (int v) {
    tnod* nou=(tnod*) malloc( sizeof(tnod));
    nou→val=v;
    nou→nc=0; nou→kids=NULL;
    return nou;
}

// adaugare fiu la un nod p
void addChild (tnod*& p, tnod* child) {
    p→kids =(tnod**) realloc (p→kids, (p→nc + 1)*sizeof(tnod*)); // extindere
    p→kids[p→nc]=child;      // adauga un nou fiu
    (p→nc)++;                // marire numar de fii
}

// afisare prefixată (sub)arbore cu radacina r
void print (tnod* r, int ns) {
    int i;

```

```

    if (r != NULL) {
        printf ("%*c%d\n", ns, ' ', r->val); // afisare date din acest nod
        for (i=0; i< r->nc; i++)             // repeta pentru fiecare fiu
            print ( r->kids[i], ns+2);        // afisare subarbore cu radacina in fiul i
    }
}

// cauta nod cu valoare data x in arbore cu radacina r
tnod* find (tnod * r, int x) {
    int i; tnod* p;
    if (r==NULL) return NULL; // daca arbore vid atunci x negasit
    if (r->val==x)             // daca x este in nodul r
        return r;
    for (i=0; i<r->nc; i++) {   // pentru fiecare subarbore i al lui r
        p=find (r->kids[i], x); // cauta pe x in subarboarele i
        if ( p != NULL)        // daca x gasit in subarboarele i
            return p;
    }
    return NULL;               // x negasit in toti subarborii lui r
}

```

Pentru ambele reprezentări de arbori multicăi adăugarea unui pointer către părinte în fiecare nod permite afisarea rapidă a căii de la rădăcină la un nod dat si simplificarea altor operatii (eliminarea nod, de exemplu), fiind o practică curentă.

În multe aplicatii relatiile dintre nodurile unui arbore multicăi nu pot fi modificate pentru a reduce înălțimea arborelui (ca în cazul arborilor binari de căutare), deoarece aceste relatii sunt impuse de aplicatie si nu de valorile din noduri.

Crearea unui arbore nebinar se face prin adăugarea de noduri frunză, folosind functiile “addChild” si “find”.

Nodul fiu este un nod nou creat cu o valoare dată (citită sau extrasă dintr-un fisier sau obținută prin alte metode). Nodul părinte este un nod existent anterior în arbore; el poate fi orice nod din arbore (dat prin valoarea sa) sau poate fi nodul “curent”, atunci când există un astfel de cursor care se deplasează de la un nod la altul.

Datele pe baza cărora se construiește un arbore pot fi date în mai multe forme, care reprezintă descrieri liniare posibile ale relatiilor dintre nodurile unui arbore. Exemple de date pentru crearea arborelui: 1 (1.1 (1.1.1, 1.1.2), 1.2 (1.2.1), 1.3)

- perechi de valori tată-fiu, în orice ordine:
1 1.1 ; 1 1.2 ; 1.2 1.2.1 ; 1.1 1.1.1 ; 1 1.3 ; 1.1 1.1.2
- liste cu fiii fiecărui nod din arbore:
1 1.1 1.2 1.3 ; 1.1 1.1.1 1.1.2 ; 1.2 1.2.1
- secvențe de valori de pe o cale ce pleacă de la rădăcina si se termină la o frunză:
1/1.1/1.1.1 ; 1/1.1/1.1.2 ; 1/1.2 /1.2.1 ; 1/1.3

Ultima formă este un mod de identificare a unor noduri dintr-un arbore si se folosește pentru calea completă la un fisier si în XPath pentru noduri dintr-un arbore (dintr-o structură) XML.

Algoritmul de construire a unui arbore cu fisierele dintr-un director si din subdirectoarele sale este recursiv: la fiecare apel primește un nume de fisier; dacă acest fisier este un subdirector atunci creează noduri pentru fisierele din subdirector si repetă apelul pentru fiecare din aceste fisiere. Din fisierele normale se creează frunze.

```

void filetree ( char* name, tnode* r ) {           // r= adresa nod curent
    daca "name" nu e director atunci return
    repeta pentru fiecare fisier "file" din "name" {
        creare nod "nou" cu valoarea "file"
        adauga nod "nou" la nodul r
        daca "file" este un director atunci

```



```

        filetree (file, nou);
    }
}

```

Pozitia curentă în arbore coboară după fiecare nod creat pentru un subdirector și urcă după crearea unui nod frunză (fișier normal).

Nodul rădăcină este construit separat, iar adresa sa este transmisă la primul apel.

Standardul DOM (Document Object Model), elaborat de consorțiul W3C, stabilește tipurile de date și operațiile (funcțiile) necesare pentru crearea și prelucrarea arborilor ce reprezintă structura unui fișier XML. Standardul DOM are ca obiectiv separarea programelor de aplicații de modul de implementare a arborelui și unificarea accesului la arborii creați de parser-ul XML de tip DOM (pentru că există și alte programe parser XML, care nu au ca rezultat un arbore).

DOM este un model de tip arbore general (multicăi) în care fiecare nod are un nume, o valoare și un tip. Numele și valoarea sunt (pointeri la) siruri de caractere iar tipul nodului este un întreg scurt cu valori precizate în standard. Exemple de tipuri de noduri (ca valori numerice și simbolice):

1 (ELEMENT_NODE)	nod ce conține un marcaj (tag)
3 (TEXT_NODE)	nod ce conține un text delimitat de marcaje
9 (DOCUMENT_NODE)	nod rădăcină al unui arbore document

Un nod element are drept nume marcajul corespunzător și ca valoare unică pentru toate nodurile de tip 1 un pointer NULL. Toate nodurile text au același nume (“#text”), dar valoarea este sirul dintre marcaje. Tipul “Node” (sau “DOMNode”) desemnează un nod de arbore DOM și este asociat cu operații de creare/modificare sau de acces la noduri dintr-un arbore DOM.

Implementarea standardului DOM se face printr-un program de tip “parser XML” care oferă programatorilor de aplicații operații pentru crearea unui arbore DOM prin program sau pe baza analizei unui fișier XML, precum și pentru acces la nodurile arborelui în vederea extragerii informațiilor necesare în aplicație. Programul parser face și o verificare a utilizării corecte a marcajelor de început și de sfârșit (de corectitudine formală a fișierului XML analizat).

Construirea unui arbore XML se poate face printr-o funcție recursivă sau folosind o stivă de pointeri la noduri (ca și în cazul arborelui de fișiere), dar există și o a treia soluție care folosește un pointer pentru poziția curentă în arbore: în cazul unui marcaj de început (de forma <tag>) se coboară un nivel, iar în cazul unui marcaj de sfârșit (de forma </tag>) se urcă un nivel în arbore. Algoritmul de creare a unui arbore DOM pe baza unui fișier XML poate fi descris astfel:

```

creare nod radacina r cu valoarea "Document"
crt=r // pozitie curenta in arbore
repetă cat timp nu e sfarsit de fisier xml {
    extrage urmatorul simbol din fisier in token
    dacă token este marcaj de inceput atunci {
        creare nod "nou" având ca nume marcaj
        adauga la crt pe nou
        crt=nou // coboara un nivel
    }
    dacă token este marcaj de sfarsit atunci
        dacă token difera de nume din crt atunci eroare
        altfel
            crt = parent(crt) // urca un nivel, la nod parinte
    dacă token este text atunci {
        creare nod "nou" cu valoare text
        adauga la crt pe nou // si ramane pe acelasi nivel
    }
}

```

Acest algoritm necesită în fiecare nod un pointer către nodul părinte și se bazează pe observația că ridicarea în arbore se face întotdeauna la nivelul imediat inferior și nu la orice nivel cu număr mai mic (ca în cazul arborelui de fisiere).

7.6 ARBORI “ROPE”

Reprezentarea sirurilor de caractere prin vectori conduce la performante slabe pentru anumite operații asupra unor siruri (texte) foarte lungi, așa cum este cazul editării unor documente mari. Este vorba de durata unor operații cum ar fi intercalarea unui text într-un document mare, eliminarea sau înlocuirea unor porțiuni de text, concatenarea de texte, s.a., dar și de memoria necesară pentru operații cu siruri nemodificabile (“immutable”), sau pentru păstrarea unei istorii a operațiilor de modificare a textelor necesară pentru anularea unor operații anterioare (“undo”).

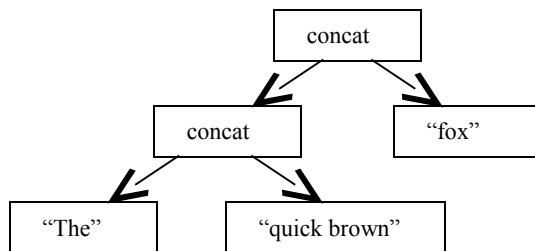
Structura de date numită “rope” (ca variantă a cuvântului “string”, pentru a sugera o însiruire de caractere) a fost propusă și implementată (în diferite variante) pentru a permite operații eficiente cu texte foarte lungi (de exemplu clasa “rope” din biblioteca STL).

Un “rope” este un arbore multicai, realizat de obicei ca arbore binar, în care numai nodurile frunză conțin (sub)siruri de caractere (ca pointeri la vectori alocati dinamic).

Dacă vrem să scriem conținutul unui “rope” într-un fisier atunci se vor scrie succesiv sirurile din nodurile frunză, de la stânga la dreapta.

Nodurile interne (și rădăcina) sunt doar puncte de reunire a unor subsiruri, prin concatenarea cărora a rezultat textul reprezentat printr-un “rope”. Anumite operații de modificare a textului dintr-un “rope” sunt realizate prin modificarea unor noduri din arbore, fără deplasarea în memorie a unor blocuri mari și fără copierea inutilă a unor siruri dintr-un loc în altul (pentru a păstra intacte sirurile concatenate, de exemplu).

Figura următoare este preluată din articolul care a lansat ideea de “rope”:



Crearea de noduri intermediare de tip “concat” la fiecare adăugare de caractere la un text ar putea mări înălțimea arborelui “rope”, și deci timpul de căutare a unui caracter (sau subsir) într-un “rope”. Din acest motiv concatenarea unor siruri scurte se face direct în nodurile frunză, fără crearea de noduri noi. S-au mai propus și alte optimizări pentru structura de “rope”, inclusiv reechilibrarea automată a arborelui, care poate deveni un arbore B sau AVL. Pentru a stabili momentul când devine necesară reechilibrarea se poate impune o înălțime maximă și se poate memora în fiecare nod intern înălțimea (sau adâncimea) sa.

Determinarea poziției unui caracter (subsir) dat într-un text “rope” necesită memorarea în fiecare nod a lungimii subsirului din fiecare subarbore. Algoritmul care urmează extrage un subsir de lungime “len” care începe în poziția “start” a unui rope:

```

substr(rope,start,len)
  // partea stanga din subsir
  if start <=0 and len >= length(rope.left)
    left= rope.left           // subsirul include subarboarele stanga
  else
    left= substr(rope.left,start,len)  // subsirul se afla numai in subarboarele stanga
  // partea dreapta din subsir
  if start <=length(rope.left) and start + len >= length(rope.left) + length(rope.right)
    right=rope.right          // subsirul include subarboarele dreapta
  
```

```

else
    right=substr(rope.right,start-length(rope.left), len- length(left))
    concat(left,right)          // concatenare subsir din stanga cu subsir din dreapta
    
```

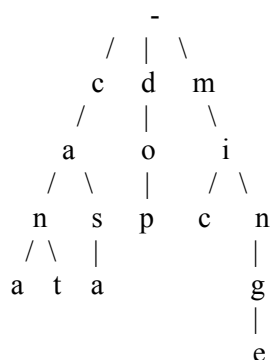
Implementarea în limbajul C a unui nod de arbore “rope” se poate face printr-o uniune de două structuri: una pentru noduri interne si una pentru noduri frunză.

7.7 ARBORI TRIE

Un arbore “Trie” (de la “retrieve” = regăsire) este un arbore folosit pentru memorarea unor siruri de caractere sau unor siruri de biti de lungimi diferite, dar care au în comun unele subsiruri, ca prefixe.

Nodurile unui trie nu contin date, iar un sir este o cale de la rădăcină la un nod frunză sau la un nod interior. Pentru siruri de biti arborele trie este binar, dar pentru siruri de caractere arborele trie nu mai este binar (numărul de succesori ai unui nod este egal cu numărul de caractere distincte din sirurile memorate).

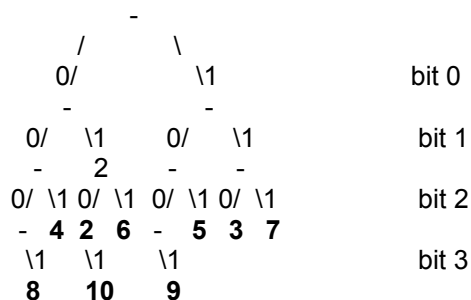
În exemplul următor este un trie construit cu sirurile (cuvintele): cana, cant, casa, dop, mic, minge. Nu toate nodurile interioare din arbore au asociate valori.



Avantajele unui arbore trie sunt:

- Regăsirea rapidă a unui sir dat sau verificarea apartenenței unui sir dat la dictionar; numărul de comparatii este determinat numai de lungimea sirului căutat, indiferent de numărul de siruri memorate în dictionar (deci este un timp constant $O(1)$ în raport cu dimensiunea colecției). Acest timp poate fi important într-un program “spellchecker” care verifică dacă fiecare cuvânt dintr-un text apartine sau nu unui dictionar.
- Determinarea celui mai lung prefix al unui sir dat care se află în dictionar (operatie necesară în algoritmul de compresie LZW).
- O anumită reducere a spatiului de memorare, dacă se folosesc vectori în loc de arbori cu pointeri.

Exemplul următor este un trie binar în care se memorează numerele 2,3,4,5,6,7,8,9,10 care au următoarele reprezentări binare pe 4 biti : 0010, 0011, 0100, 0101, 0110,... 1010



Este de remarcat că structura unui arbore trie nu depinde de ordinea în care se adaugă valorile la arbore, iar arborele este în mod natural relativ echilibrat. Înălțimea unui arbore trie este determinată de lungimea celui mai lung sir memorat și nu depinde de numărul de valori memorate.

Arborele Huffman de coduri binare este un exemplu de trie binar, în care codurile sunt căi de la rădăcină la frunzele arborelui (nodurile interne nu sunt semnificative).

Pentru arbori trie este avantajoasă memorarea lor ca vectori (matrice) și nu ca arbori cu pointeri (un pointer ocupă uzual 32 biti, un indice de 16 biti este suficient pentru vectori de 64 k elemente). O soluție și mai compactă este un vector de biti, în care fiecare bit marchează prezenta sau absenta unui nod, la parcurgerea în lățime.

Dictionarul folosit de algoritmul de compresie LZW poate fi memorat ca un "trie". Exemplul următor este arborele trie, reprezentat prin doi vectori "left" și "right", la compresia sirului "abbaabbaababbaaaabaabba" :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
w	-	a	b	ab	bb	ba	aa	abb	baa	aba	abba	aaa	aab	baab	bba
left	1	6	5	9	14	7	11	-	-	-	-	-	-	-	-
right	2	3	4	7	-	-	12	13	-	-	-	-	-	-	-

În acest arbore trie toate nodurile sunt semnificative, pentru că reprezintă secvențe codificate, iar codurile sunt chiar pozițiile în vectori (notate cu 'i'). În poziția 0 se află nodul rădăcină, care are la stânga nodul 1 ('a') și la dreapta nodul 2 ('b'), s.a.m.d.

Căutarea unui sir 'w' în acest arbore arată astfel:

```
// cautare sir in trie
int get ( short left[], short right[],int n, char w[]) {
    int i,j,k;
    i=k=0; // i = pozitie curenta in vectori (nod)
    while ( i >= 0 && w[k] !=0 ) { // cat timp mai exista noduri si caractere in w
        j=i; // j este nodul parinte al lui i
        if (w[k]=='a') // daca este 'a'
            i=left[j]; // continua la stanga
        else // daca este 'b'
            i=right[j]; // continua la dreapta
        k++; // caracterul urmator din w
    }
    return j; // ultimul nivel din trie care se potriveste
}
```

Adăugarea unui sir 'w' la arborele trie începe prin căutarea poziției (nodului) unde se termină cel mai lung prefix din 'w' aflat în trie și continuă cu adăugarea la trie a caracterelor următoare din 'w'.

Pentru reducerea spațiului de memorare în cazul unor cuvinte lungi, cu prea puține caractere comune cu alte cuvinte în prefix, este posibilă comasarea unei subcăi din arbore ce conține noduri cu un singur fiu într-un singur nod; acești arbori trie comprimați se numesc și arbori Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric).

Într-un arbore Patricia nu există noduri cu un singur succesor și în fiecare nod se memorează indicele elementului din sir (sau caracterul) folosit drept criteriu de ramificare.

Un arbore de sufixe (suffix tree) este un trie format cu toate sufixele cu sens ale unui sir dat; el permite verificarea rapidă (într-un timp proportional cu lungimea lui q) a condiției ca un sir dat q să fie un suffix al unui sir dat s.

7.8 ARBORI KD SI ARBORI QUADTREE

Arborii kD sunt un caz special de arbori binari de căutare, iar arborii QuadTree (QT) sunt arbori multicăi, dar utilizarea lor este aceeași: pentru descompunerea unui spațiu k-dimensional în regiuni dreptunghiulare (hiperdreptunghiulare pentru $k > 2$).

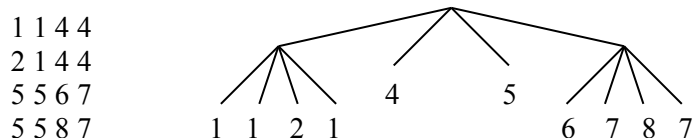
Vom exemplifica cu cazul unui spatiu bidimensional ($k=2$) deoarece arborii "QuadTree" (QT) reprezintă alternativa arborilor 2D (pentru arborii 3D alternativa multicăi se numeste "OctTree", pentru că fiecare nod are 8 succesori). Intr-un arbore QT fiecare nod care nu e o frunză are exact 4 succesori.

Arborii QT sunt folositi pentru reprezentarea compactă a unor imagini fotografice care contin un număr mare de puncte diferit colorate, dar în care există regiuni cu puncte de aceeași culoare. Fiecare regiune apare ca un nod frunză în arborele QT.

Construirea unui arbore QT se face prin împărțire succesivă a unui dreptunghi în 4 dreptunghiuri egale (stânga, dreapta, sus, jos) printr-o linie verticală și una orizontală. Cei 4 succesori ai unui nod corespund celor 4 dreptunghiuri (celule) componente. Operatia de divizare este aplicată recursiv până când toate punctele dintr-un dreptunghi au aceeași valoare.

O aplicatie pentru arbori QT este reprezentarea unei imagini colorate cu diferite culori, încadrată într-un dreptunghi ce corespunde rădăcinii arborelui. Dacă una din celulele rezultate prin partitionare contine puncte de aceeași culoare, atunci se adaugă un nod frunză etichetat cu acea culoare. Dacă o celulă contine puncte de diferite culori atunci este împărțită în alte 4 celule mai mici, care corespund celor 4 noduri fii.

Exemplu de imagine și de arbore QT asociat acestei imagini.



Nodurile unui arbore QT pot fi identificate prin numere întregi (indici) și/sau prin coordonatele celulei din imagine pe care o reprezintă în arbore.

Reprezentarea unui quadtree ca arbore cu pointeri necesită multă memorie (în cazul unui număr mare de noduri) și de aceea se folosesc și structuri liniare cu legături implicite (vector cu lista nodurilor din arbore), mai ales pentru arbori statici, care nu se modifică în timp.

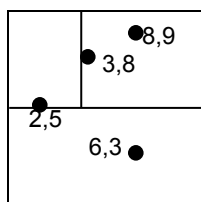
Descompunerea spatiului 2D pentru un quadtree se face simultan pe ambele directii (printr-o linie orizontală și una verticală), iar în cazul unui arbore 2D se face succesiv pe fiecare din cele două directii (sau pe cele k directii, pentru arbori kD).

Un arbore kD este un arbore binar care reprezintă o descompunere ierarhică a unui spatiu cu k dimensiuni (de unde și numele) în celule, fiecare celulă conținând un singur punct sau un număr redus de puncte dintr-o porțiune a spatiului k -dimensional. Împărțirea spatiului se face prin (hiper)plane paralele cu axele.

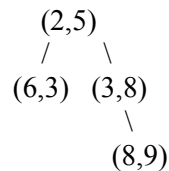
Arborii kD se folosesc pentru memorarea coordonatelor unui număr relativ redus de puncte, folosite la decuparea spatiului în subregiuni. Intr-un arbore 2D fiecare nod din arbore corespunde unui punct sau unei regiuni ce contine un singur punct.

Fie punctele de coordonate întregi : (2,5), (6,3), (3,8), (8,9)

O regiune plană dreptunghiulară delimitată de punctele (0,0) și (10,10) va putea fi descompusă astfel:



Prima linie a fost orizontala la $y=5$ prin punctul (2,5), iar a doua linie a fost semi-dreapta verticală la $x=3$, prin punctul (3,8). Arborele 2D corespunzător acestei împărțiri a spatiului este următorul:



Punctul (6,3) se află în regiunea de sub (2,5) iar (3,8) în regiunea de deasupra lui (2,5); față de punctul (3,8) la dreapta este punctul (8,9) dar la stânga nu e nici un alt punct (dintre punctele aflate peste orizontala cu $y=5$).

Altă secvență de puncte sau de orizontale și verticale ar fi condus la un alt arbore, cu același număr de noduri dar cu altă înălțime și altă rădăcină. Dacă toate punctele sunt cunoscute de la început atunci ordinea în care sunt folosite este importantă și ar fi de dorit un arbore cu înălțime cât mai mică.

În ceea ce privește ordinea de “tăiere” a spațiului, este posibilă fie o alternanță de linii orizontale și verticale (preferată), fie o secvență de linii orizontale, urmată de o secvență de linii verticale, fie o altă secvență.

Este posibilă și o variantă de împărțire a spațiului în celule egale (ca la arborii QT) în care caz nodurile arborelui kD nu ar mai conține coordonatele unor puncte date.

Fiecare nod dintr-un arbore kD conține un număr de k chei, iar decizia de continuare de pe un nivel pe nivelul inferior (la stânga sau la dreapta) este dictată de o altă cheie (sau de o altă coordonată). Dacă se folosesc mai întâi toate semidreptele ce trec printr-un punct și apoi se trece la punctul următor, atunci nivelul următor celui cu numărul j va fi $(j+1)\%k$ unde k este numărul de dimensiuni.

Pentru un arbore 2D fiecare nod conține ca date 2 întregi (x,y) , iar ordinea de tăiere în ceea ce urmează va fi $y_1, x_1, y_2, x_2, y_3, x_3, \dots$. Căutarea și inserarea într-un arbore kD seamănă cu operațiile corespunzătoare dintr-un arbore binar de căutare BST, cu diferența că pe fiecare nivel se folosește o altă cheie în luarea deciziei.

```

typedef struct kdNode {          // definire nod arbore 2D
    int x[2];                    // int x[3] pentru arbori 3D (coordonate)
    struct kdNode *left, *right; // adrese succesori
} kdNode;

// insertie in arbore cu radacina t a unui vector de chei d pe nivelul k
void insert( kdNode* &t, int d[], int k ) {
    if( t == NULL ) {            // daca arbore vid (nod frunza)
        t = (kdNode*) malloc( sizeof(kdNode) ); // creare nod nou
        t->x[0]=d[0]; t->x[1]=d[1]; // initializare vector de chei (coord.)
    }
    else if( d[k] < t->x[k] )      // dacă se continuă spre stânga sau spre dreapta
        insert(t->left,d,(k+1)%2 ); // sau 1-k ptr 2D
    else
        insert(t->right,d,(k+1)%2 ); // sau 1-k ptr 2D
    }

// creare arbore cu date citite de la tastatura
void main() {
    kdNode *r; int x[2];
    int k=0; // indice cheie folosita la adaugare
    initkd (r); // initializare arbore vid
    while (scanf("%d%d",&x[0],&x[1])!=2) {
        insert(r,x,k); // cheile x[0] si x[1]
        k=(k+1)%2; // utilizare alternata a cheilor
    }
}
  
```

Un arbore kD poate reduce mult timpul anumitor operații de căutare într-o imagine sau într-o bază de date (unde fiecare cheie de căutare corespunde unei dimensiuni): localizarea celulei în care se află

un anumit punct, căutarea celui mai apropiat vecin, căutare regiuni (ce puncte se află într-o anumită regiune), căutare cu informații parțiale (se cunosc valorile unor chei dar nu se știe nimic despre unul sau câteva atribute ale articolelor căutate).

Exemplu cu determinarea punctelor care se află într-o regiune dreptunghiulară cu punctul de minim “low” și punctul de maxim “high”, folosind un arbore 2D:

```
void printRange( kdNode* t, int low[], int high[], int k ) {
    if( t == NULL ) return;
    if( low[ 0 ] <= t->x[ 0 ] && high[ 0 ] >= t->x[ 0 ] &&
        low[ 1 ] <= t->x[ 1 ] && high[ 1 ] >= t->x[ 1 ] )
        printf( "( %d , %d )\n", t->x[ 0 ], t->x[ 1 ] );
    if( low[ k ] <= t->x[ k ] )
        printRange( t->left, low, high, (k+1)%2 );
    if( high[ k ] >= t->x[ k ] )
        printRange( t->right, low, high, (k+1)%2 );
}
```

Căutarea celui mai apropiat vecin al unui punct dat folosind un arbore kD determină o primă aproximație ca fiind nodul frunză care ar putea conține punctul dat. Funcție de căutare a punctului în a cărei regiune s-ar putea găsi un punct dat.

```
// cautare (nod) regiune care (poate) conține punctul (c[0],c[1])
// t este nodul (punctul) posibil cel mai apropiat de (c[0],c[1])
int find ( kdNode* r, int c[], kdNode * & t ) {
    int k;
    for (k=1; r!= NULL; k=(k+1)%2) {
        t=r; // retine în t nod curent înainte de avans în arbore
        if (r->x[0]==c[0] && r->x[1]==c[1])
            return 1; // găsit
        else
            if (c[k] <= r->x[k])
                r=r->left;
            else
                r=r->right;
    }
    return 0; // negăsit când r==NULL
}
```

De exemplu, într-un arbore cu punctele (2,5),(6,3),(3,9),(8,7), cel mai apropiat punct de (8,8) este (8,7), dar cel mai apropiat punct de (4,6) este (2,5) și nu (8,7), care este indicat de funcția “find”; la fel (2,4) este mai apropiat de (2,5) decât este conținut în regiunea definită de punctul (6,3).

De aceea, după ce se găsește nodul cu “find”, se caută în apropierea acestui nod (în regiunile vecine), până când se găsește cel mai apropiat punct. Nu vom intra în detaliile acestui algoritm, dar este sigur că timpul necesar va fi mult mai mic decât timpul de căutare a celui mai apropiat vecin într-o mulțime de N puncte, fără a folosi arbori kD. Folosind un vector de puncte (o matrice de coordonate) timpul necesar este de ordinul $O(n)$, dar în cazul unui arbore kD este de ordinul $O(\log(n))$, adică este cel mult egal cu înălțimea arborelui.

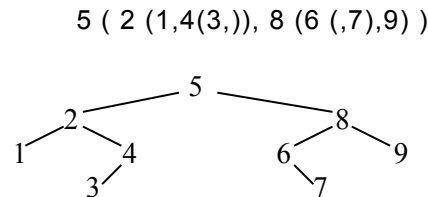
Reducerea înălțimii unui arbore kD se poate face alegând la fiecare pas tăierea pe dimensiunea maximă în locul unei alternanțe regulate de dimensiuni; în acest caz mai trebuie memorat în fiecare nod și indicele cheii (dimensiunii) folosite în acel nod.

Capitolul 8

ARBORI DE CAUTARE

8.1 ARBORI BINARI DE CĂUTARE

Un arbore binar de căutare (BST=Binary Search Tree), numit si arbore de sortare sau arbore ordonat, este un arbore binar cu proprietatea că orice nod interior are valoarea mai mare decât orice nod din subarborele stânga si mai mică decât orice nod din subarborele dreapta. Exemplu de arbore binar de căutare:



Arborii BST permit mentinerea datelor în ordine si o căutare rapidă a unei valori, ceea ce îi recomandă pentru implementarea de multimi si dictionare ordonate (afisarea infixată a unui arbore binar de căutare produce un vector ordonat cu valorile din noduri).

Intr-un arbore ordonat, de căutare, este importantă ordinea memorării succesorilor fiecărui nod, deci este important care este fiul stânga si care este fiul dreapta.

Căutarea într-un arbore BST este comparabilă cu căutarea binară pentru vectori ordonati: după ce se compară valoarea căutată cu valoarea din rădăcină se poate decide în care din cei doi subarbori se află (dacă există) valoarea căutată. Fiecare nouă comparatie elimină un subarbore din căutare si reduce cu 1 înălțimea arborelui în care se caută. Procesul de căutare într-un arbore binar ordonat poate fi exprimat recursiv sau nerecursiv.

```

// căutare recursivă în arbore ordonat
tnod * find ( tnod * r, int x) {
  if (r==NULL) return NULL;          // x negasit in arbore
  if (x == r->val) return r;          // x gasit in nodul r
  if ( x < r->val)
    return find (r->st,x);            // cauta in subarb stanga
  else
    return find (r->dr,x);            // cauta in subarb. dreapta
}

// căutare nerecursivă în arbore ordonat
tnod * find ( tnod * r, int x) {
  while (r!=NULL) {                  // cat timp se mai poate cobora in arbore
    if (x == r->val) return r;        // x gasit la adresa r
    if ( x < r->val)
      r=r->st;                        // cauta spre stanga
    else
      r=r->dr;                        // cauta spre dreapta
  }
  return NULL;
}

```

Timpul minim de căutare se realizează pentru un arbore BST echilibrat (cu înălțime minimă), la care înălțimile celor doi subarbori sunt egale sau diferă cu 1. Acest timp este de ordinul $\log_2 n$, unde n este numărul total de noduri din arbore. Valoarea maximă dintr-un arbore binar de căutare se află în nodul din extremitatea dreaptă, iar valoarea minimă în nodul din extremitatea stângă. Exemplu :


```

int maxABC ( tnod * r) {      // maxim dintr-un arbore ordonat cu rădăcina r
    if (r==NULL) return -MAXINT; // dacă arbore vid
    while ( r->dr != NULL)      // mergi la dreapta cât se poate
        r=r->dr;
    return r->val;              // r este ultimul nod din dreapta
}
    
```

Adăugarea unui nod la un arbore BST seamănă cu căutarea, pentru că se caută nodul frunză cu valoarea cea mai apropiată de valoarea care se adaugă. Nodul nou se adaugă ca frunză (arborele crește prin frunze).

```

void add (tnod *& r, int x) {    // adaugare x la arborele cu radacina r
    tnod * nou ;                // adresa nod cu valoarea x
    if (r == NULL) {            // daca este primul nod
        nou =(tnod*) malloc (sizeof(tnod)); // creare nod la adresa n
        nou->val =x; nou->st = nou->dr = NULL;
        r = nou;                // nou devine radacina arborelui
        return;
    }                            // daca arbore nevid
    if (x < r->val)               // daca x mai mic ca valoarea din radacina
        add (r->st,x);           // se adauga la subarborele stanga
    else                         // daca x mai mare ca valoarea din radacina
        add (r->dr,x);           // se adauga la subarborele dreapta
}
    
```

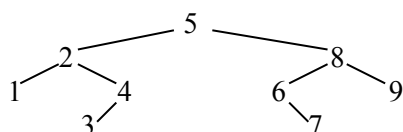
Eliminarea unui nod cu valoare dată dintr-un arbore BST trebuie să considere următoarele situații:

- Nodul de sters nu are succesori (este o frunză);
- Nodul de sters are un singur succesori;
- Nodul de sters are doi succesori.

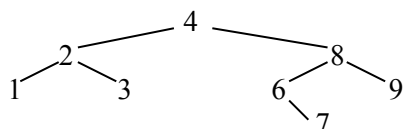
Eliminarea unui nod cu un succesori sau fără succesori se reduce la înlocuirea legăturii la nodul sters prin legătura acestuia la succesoriul său (care poate fi NULL).

Eliminarea unui nod cu 2 succesori se face prin înlocuirea sa cu un nod care are cea mai apropiată valoare de cel sters; acesta poate fi nodul din extremitatea dreaptă a subarborelui stânga sau nodul din extremitatea stânga a subarborelui dreapta (este fie predecesorul, fie succesoriul în ordine infixată). Acest nod are cel mult un succesori

Fie arborele BST următor



Eliminarea nodului 5 se face fie prin înlocuirea sa cu nodul 4, fie prin înlocuirea sa cu nodul 6. Același arbore după înlocuirea nodului 5 prin nodul 4 :



Operația de eliminare nod se poate exprima nerecursiv sau recursiv, iar funcția se poate scrie ca funcție de tip "void" cu parametru referință sau ca funcție cu rezultat pointer (adresa rădăcinii arborelui se poate modifica în urma stergerii valorii din nodul rădăcină). Exemplu de funcție nerecursivă pentru eliminare nod:

```

void del (tnod*& r, int x) { // sterge nodul cu valoarea x din arborele r
    tnod *p, *pp, *q, *s, *ps;
    // cauta valoarea x in arbore si pune in p adresa sa
    p=r; pp=0; // pp este parintele lui p
    while ( p !=0 && x != p->val) {
        pp=p; // retine adr. p inainte de modificare
        p= x < p->val ? p->st : p->dr;
    }
    if (p==0) return; // nu exista nod cu val. x
    if (p->st != 0 && p->dr != 0) { // daca p are 2 fii
        // reducere la cazul cu 1 sau 0 succesori
        // s= element maxim la stanga lui p
        s=p->st; ps=p; // ps = parintele lui s
        while (s->dr != 0) {
            ps=s; s=s->dr;
        }
        // muta valoarea din s in p
        p->val=s->val;
        p=s; pp=ps; // p contine adresa nodului de eliminat
    }
    // p are cel mult un fiu q
    q= (p->st == 0)? p->dr : p->st;
    // elimina nodul p
    if (p==r) r=q; // daca se modifica radacina
    else {
        if (p == pp->st) pp->st=q; // modifica parintele nodului eliminat
        else pp->dr=q; // prin inlocuirea fiului p cu nepotul q
    }
    free (p);
}

```

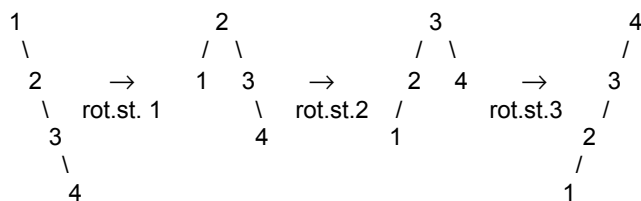
8.2 ARBORI BINARI ECHILIBRATI

Căutarea într-un arbore binar ordonat este eficientă dacă arborele este echilibrat. Timpul de căutare într-un arbore este determinat de înălțimea arborelui, iar această înălțime este cu atât mai mică cu cât arborele este mai echilibrat. Înălțimea minimă este $O(\lg n)$ și se realizează pentru un arbore echilibrat în înălțime.

Structura și înălțimea unui arbore binar de căutare depinde de ordinea în care se adaugă valori în arbore, ordine impusă de aplicație și care nu poate fi modificată.

În funcție de ordinea adăugărilor de noi noduri (și eventual de stergeri) se poate ajunge la arbori foarte dezechilibrați; cazul cel mai defavorabil este un arbore cu toate nodurile pe aceeași parte, cu un timp de căutare de ordinul $O(n)$.

Ideea generală este ajustarea arborelui după operații de adăugare sau de ștergere, dacă aceste operații strică echilibrul existent. Structura arborelui se modifică, dar se mențin relațiile dintre valorile continute în noduri. Exemple de arbori binari de căutare echivalenți (cu același conținut), dar cu structuri și înălțimi diferite:



De cele mai multe ori se verifică echilibrul și se modifică structura după fiecare operație de adăugare sau de eliminare, dar în cazul arborilor Scapegoat modificările se fac numai din când în când (după un număr oarecare de operații asupra arborelui).

Criteriile de apreciere a echilibrului pot fi diferite:

- diferența dintre înălțimile celor doi subarbori ai fiecărui nod (arbore echilibrat în înălțime), criteriu folosit de arborii AVL;
- diferența dintre cea mai lungă și cea mai scurtă cale de la rădăcină la frunze, criteriu folosit de arborii RB (Red-Black);
- limite maxime pentru înălțimea arborelui, funcție de numărul de noduri (dintre care unele stabilite probabilistic), ca în cazul Treap, Splay Tree sau Scapegoat Tree.

Există mai multe variante de arbori de căutare pentru noduri cu probabilități egale de căutare. Principalele tipuri de arbori cu autoechilibrare sunt:

- Arbori Treap și Splay
- Arbori AVL
- Arbori cu noduri colorate : RB și AA.
- Arbori "scapegoat"
- Arbori 2-3 (sau 2-3-4)

Diferențele dintre acești arbori se manifestă prin criteriul folosit pentru aprecierea gradului de echilibru și prin valoarea suplimentară memorată în fiecare nod.

În cazul arborilor AVL criteriul este diferența dintre înălțimilor subarborilor ce au ca rădăcină acel nod. Dacă acest număr, memorat în fiecare nod, are o valoare absolută mai mare ca 1, se consideră că arborele s-a dezechilibrat. Arborii AVL au fost primii în ordine istorică, dar își păstrează importanța și în prezent.

În cazul arborilor RB nodurile sunt colorate fie cu roșu, fie cu negru. Criteriul de echilibru este ca fiecare cale de la rădăcină la o frunză să aibă același număr de noduri negre. Deoarece un nod roșu trebuie să aibă fii colorați negru, rezultă că pe fiecare cale există cel puțin jumătate din noduri negre; de aici și criteriul de echilibru: cea mai lungă cale nu poate avea mai mult decât de două ori mai multe noduri decât cea mai scurtă cale. Arborii RB asigură performanțe bune și în cazul mediu și în cazul cel mai defavorabil, ceea ce explică de ce sunt utilizați în clase C++ și Java.

În cazul arborilor Treap se memorează în fiecare nod și o prioritate (număr întreg generat aleator), iar arborele de căutare (ordonat după valorile din noduri) este obligat să respecte și condiția de heap relativ la prioritățile nodurilor. Un treap nu este un heap deoarece nu are toate nivelurile complete, dar în medie înălțimea sa nu depășește dublul înălțimii minime ($2 \cdot \lg(n)$).

Arborii "scapegoat" memorează în fiecare nod atât înălțimea cât și numărul de noduri din subarboarele cu rădăcina în acel nod. Ideea este de a nu face restructurarea arborelui prea frecvent, ea se va face numai după un număr de adăugări sau de stergeri de noduri. Stergerea unui nod nu este efectivă ci este doar o marcarea a nodurilor respective ca invalidate. Eliminarea efectivă și restructurarea se va face numai când în arbore sunt mai mult de jumătate de noduri marcate ca sterse. La adăugarea unui nod se actualizează înălțimea și numărul de noduri pentru nodurile de pe calea ce conține nodul nou și se verifică (pornind de la nodul adăugat în sus, spre rădăcină) dacă există un arbore prea dezechilibrat (cu înălțime mai mare ca logaritmul numărului de noduri: $h(v) > m + \log(|v|)$). Se va restructura numai acel subarbore găsit vinovat de dezechilibrarea întregului arbore ("scapegoat"=tap ispășitor).

Fie următorul subarbore dintr-un arbore BST:

```

    15
   /  \
  10   20

```

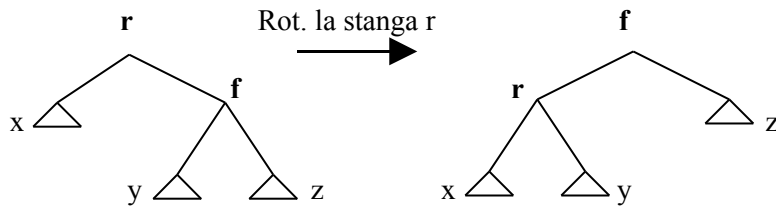
După ce se adaugă valoarea 8 nu se face nici o modificare, desi subarboarele devine "putin" dezechilibrat. Dacă se adaugă și valoarea 5, atunci subarboarele devine "mult" dezechilibrat și se va restructura, fără a fi nevoie să se propage în sus modificarea (părintele lui 15 era mai mare ca 15, deci va fi mai mare și ca 10). Exemplu:



Costul amortizat al operațiilor de inserție și ștergere într-un arbore “scapegoat” este tot $O(\log(n))$.

Restructurarea unui arbore binar de căutare se face prin rotații; o rotație modifică structura unui (sub)arbore, dar menține relațiile dintre valorile din noduri.

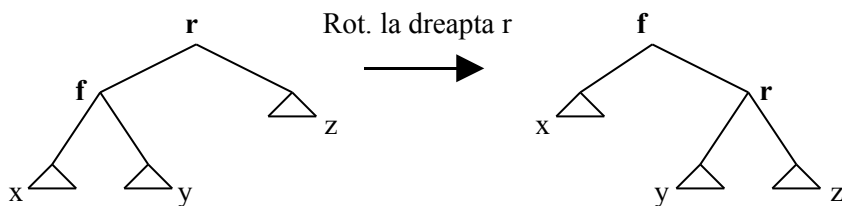
Rotatia la stânga în subarboarele cu rădăcina r coboară nodul r la stânga și aduce în locul lui fiul său dreapta f , iar r devine fiu stânga al lui f ($\text{val}(f) > \text{val}(r)$).



Prin rotații se mențin relațiile dintre valorile nodurilor:

$$x < r < f < z ; \quad r < y < f ;$$

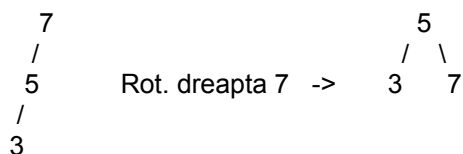
Rotatia la dreapta a nodului r coboară pe r la dreapta și aduce în locul lui fiul său stânga f ; r devine fiu dreapta al lui f .



Se observă că la rotație se modifică o singură legătură, cea a subarboarelui y în figurile anterioare.

Rotatiile au ca efect ridicarea (și coborârea) unor noduri în arbore și pot reduce înălțimea arborelui. Pentru a ridica un nod (f în figurile anterioare) se rotește părintele nodului care trebuie ridicat (notat cu r aici), fie la dreapta, fie la stânga.

Exemplul următor arată cum se poate reduce înălțimea unui arbore printr-o rotație (nodul 7 coboară la dreapta iar nodul 5 urcă în rădăcină):



Codificarea rotațiilor depinde de utilizarea funcțiilor respective și poate avea o formă mai simplă sau mai complexă.

În forma simplă se consideră că nodul rotit este rădăcina unui (sub)arbore și nu are un nod părinte (sau că părintele se modifică într-o altă funcție):

```
// Rotatie dreapta prin inlocuire cu fiul din stanga
void rotRight ( tnode*& r) {
    tnode* f = r->st; // f este fiul stanga al lui r
    r->st = f->dr;     // se modifica numai fiul stanga
    f->dr = r;         // r devine fiu dreapta al lui f
}
```

```

    r = f;                // adresa primită se modifică
}

// Rotatie stanga prin inlocuire cu fiul din dreapta
void rotLeft ( tnode*& r) {
    tnode* f = r->dr;      // f este fiul dreapta al lui r
    r->dr = f->st;          // se modifica fiul din dreapta
    f->st = r;              // r devine fiu stanga al lui f
    r = f;                 // f ia locul lui r
}

```

Dacă nodul rotit p este un nod interior (cu părinte) atunci trebuie modificată și legătura de la părintele lui p către nodul adus în locul lui p. Părintele nodului p se poate afla folosind un pointer păstrat în fiecare nod, sau printr-o căutare pornind din rădăcina arborelui. Exemplu de funcție pentru rotație dreapta a unui nod interior p într-un arbore cu legături în sus (la noduri părinte):

```

void rotateRight(tnode* & root, tnode *p) {
    tnode *f = p->st;      // f este fiu stanga al lui p
    if (f==NULL) return;   // nimic daca nu are fiu stanga
    p->st = f->dr;          // inlocuieste fiu stanga p cu fiu dreapta f
    if (f->dr != NULL)
        f->dr->parent = p;  // si legatura la parinte
    f->parent = p->parent;   // noul parinte al lui f
    if (p->parent) {        // daca p are parinte
        if (p == p->parent->dr)
            p->parent->dr = f;
        else
            p->parent->st = f;
    } else                 // daca p este radacina arborelui
        root = f;          // atunci modifica radacina
    f->dr = p;              // p devine fiu dreapta al lui f
    p->parent = f;          // p are ca parinte pe f
}

```

Dacă nu se memorează în noduri legături în sus atunci se poate folosi o funcție care caută nodul părinte al unui nod dat pornind din rădăcină (cu costul de timp aferent):

```

tnod * parent ( tnod * r, int x) {    // x= valoare nod
    tnod * p;
    if (r==NULL || r->val==x)         // daca x in radacina r
        return NULL;
    if ( r->st && x==r->st->val)        // daca x este fiu stanga r
        return r;
    if ( r->dr && x==r->dr->val)        // daca x este fiu dreapta r
        return r;
    if (p= parent(r->st,x))            // cauta x in subarbore stanga r
        return p;
    else
        return parent (r->dr,x);      // cauta x in subarbore dreapta r
}

```

Rotatiile simple, care se aplică numai rădăcinii unui (sub)arbore, se folosesc în funcții recursive de adăugare de noduri, unde adăugarea și rotația se aplică recursiv unui subarbore tot mai mic, identificat prin rădăcina sa. Subarborii sunt afectați succesiv (de adăugare și rotație), de la cel mai mic la cel mai mare (de jos în sus), astfel încât modificarea legăturilor dintre noduri se propagă treptat în sus.

8.3 ARBORI SPLAY SI TREAP

Arborii binari de căutare numiti “Splay” si “Treap” nu au un criteriu determinist de mentinere a echilibrului, iar înălțimea lor este mentinută în limite acceptabile în mod probabilistic.

Desi au utilizări diferite, arborii Splay si Treap folosesc un algoritm asemănător de ridicare în arbore a ultimului nod adăugat (sau căutat, la arbori Splay); acest nod este ridicat mereu în rădăcină (arbori Splay) sau până când este îndeplinită o conditie (Treap).

În anumite aplicatii acelasi nod face obiectul unor operatii succesive de căutare, insertie, stergere. Altfel spus, probabilitatea căutării aceleasi valori dintr-o colectie este destul de mare, după un prim acces la acea valoare. Aceasta este si ideea care stă la baza memoriilor “cache”. Pentru astfel de cazuri este utilă modificarea automată a structurii după fiecare operatie de căutare, de adăugare sau de stergere, astfel ca valorile căutate cel mai recent să fie cât mai aproape de rădăcină.

Un arbore “splay” este un arbore binar de căutare, care se modifică automat pentru aducerea ultimei valori accesate în rădăcina arborelui, prin rotatii, după căutarea sau după adăugarea unui nou nod, ca frunză. Pentru stergere, se aduce întâi nodul de eliminat în rădăcină si apoi se sterge.

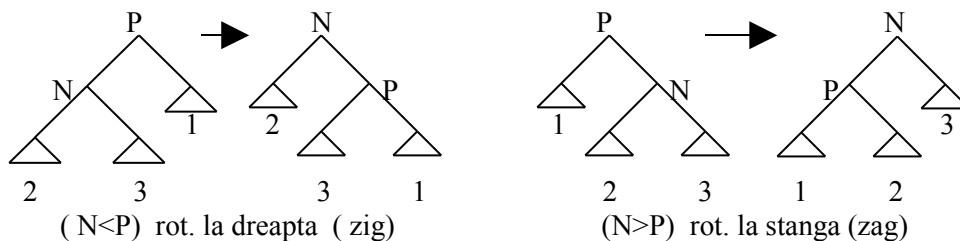
Timpul necesar aducerii unui nod în rădăcină depinde de distanta acestuia față de rădăcină, dar în medie sunt necesare $O(n \log(n) + m \log(n))$ operatii pentru m adăugări la un arbore cu n noduri, iar fiecare operatie de “splay” costă $O(n \log(n))$.

Operatia de ridicare a unui nod N se poate realiza în mai multe feluri:

- Prin ridicarea treptată a nodului N , prin rotatii simple, repetate, functie de relatia dintre N si părintele său (“move-to-root”);
- Prin ridicarea părintelui lui N , urmată de ridicarea lui N (“splay”).

Cea de a doua metodă are ca efect echilibrarea mai bună a arborelui “splay”, în anumite cazuri de arbori foarte dezechilibrati, dar este ceva mai complexă.

Dacă N are doar părinte P si nu are “bunic” (P este rădăcina arborelui) atunci se face o singură rotatie pentru a-l aduce pe N în rădăcina arborelui (nu există nici o diferență între “move-to-root” si “splay”):



Dacă N are si un bunic B (părintele lui P) atunci se deosebesc 4 cazuri, functie de pozitia nodului (nou) accesat N față de părintele său P si a părintelui P față de “bunicul” B al lui N :

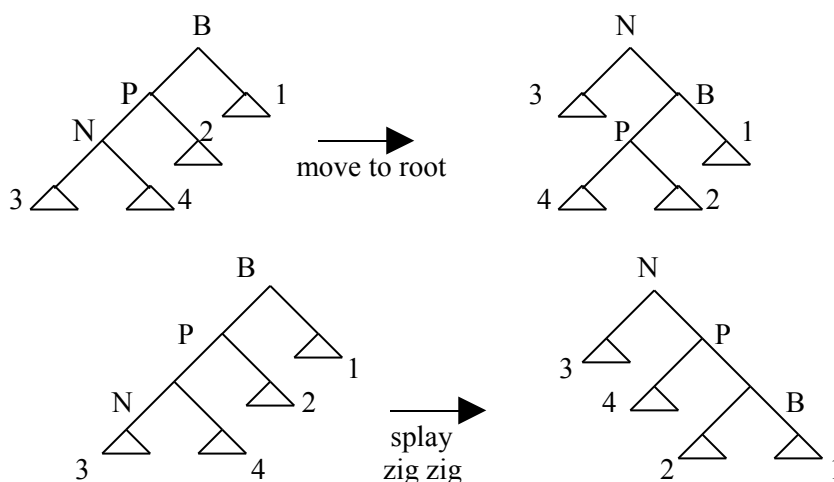
Cazul 1(zig zig): $N < P < B$ (N si P fii stânga) - Se ridică mai întâi P (rotatie dreapta B) si apoi se ridică N (rotatie dreapta P)

Cazul 2(zag zag): $N > P > B$ (N si P fii dreapta), simetric cu cazul 1 - Se ridică P (rotatie stânga B) si apoi se ridică N (rotatie stânga P)

Cazul 3(zig zag): $P < N < B$ (N fiu dreapta, P fiu stânga) - Se ridică N de două ori, mai întâi în locul lui P (rotatie stânga P) si apoi în locul lui B (rotatie dreapta B).

Cazul 4(zag zig): $B < N < P$ (N fiu stânga, P fiu dreapta) - Se ridică N de două ori, locul lui P (rotatie dreapta P) si apoi în locul lui B (rotatie stânga B)

Diferenta dintre operatiile “move-to-root” si “splay” apare numai în cazurile 1 si 2



Exemplu de funcție pentru adăugarea unei valori la un arbore Splay:

```
void insertS (tnode* &t, int x){
    insert (t,x);      // adaugare ca la orice arbore binar de căutare
    splayr (t,x);      // ridicare x în rădăcina arborelui
}
```

Urmează două variante de funcții “move-to-root” pentru ridicare în rădăcină:

```
// movetoroot recursiv
void splayr( tnode* & r, int x ) {
    tnode* p;
    p=find(r,x);
    if (p==r) return;
    if (x > p->parent->val)
        rotateLeft(r,p->parent);
    else
        rotateRight(r,p->parent);
    splayr(r,x);
}

// movetoroot iterativ
void splay( Node* & r, int x ) {
    Node* p;
    while ( x != r->val ) {
        p=find(r,x);
        if (p==r) return;
        if (x > p->parent->val)
            rotateLeft(r,p->parent);
        else
            rotateRight(r,p->parent);
    }
}
```

Funcția “splay” este apelată și după căutarea unei valori x în arbore. Dacă valoarea căutată x nu există în arbore, atunci se aduce în rădăcină nodul cu valoarea cea mai apropiată de x , ultimul pe calea de căutare a lui x . După eliminarea unui nod cu valoarea x se aduce în rădăcină valoarea cea mai apropiată de x .

Deși nu sunt dintre cei mai cunoscuți arbori echilibrați (înălțimea medie este mai mare ca pentru alți arbori), arborii Treap folosesc numai rotații simple și prezintă analogii cu structura “Heap”, ceea ce îi face mai ușor de înțeles.

S-a arătat că pentru o secvență de chei generate aleator și adăugate la un arbore binar de căutare, arborele este relativ echilibrat; mai exact, calea de lungime minimă este $1.4 \lg(n)-2$ iar calea de lungime maximă este $4.3 \lg(n)$.

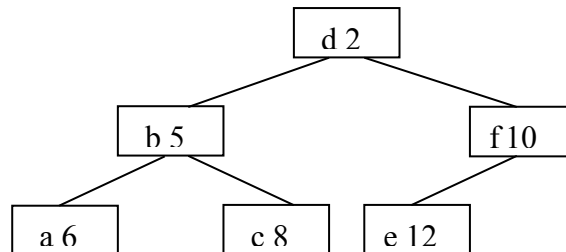
Numele “Treap” provine din “Tree Heap” și desemnează o structură care combină caracteristicile unui arbore binar de căutare cu caracteristicile unui Heap. Ideea este de a asocia fiecărui nod o prioritate, generată aleator și folosită la restructurare.

Fiecare nod din arbore conține o valoare (o cheie) și o prioritate. În raport cu cheia nodurile unui treap respectă condiția unui arbore de căutare, iar în raport cu prioritatea este un min-heap. Prioritățile sunt generate aleator.

```
typedef struct th {      // un nod de arbore Treap
    int val;             // valoare (cheie)
    int pri;             // prioritate
    struct th* st, *dr;   // adrese succesori (subarbori)
    struct th * parent;   // adresa nod parinte
} tnode;
```

Exemplu de arbore treap construit cu următoarele chei și priorități:

Cheie	a	b	c	d	e	f
Prior	6	5	8	2	12	10



În lipsa acestor priorități arborele ar fi avut înălțimea 6, deoarece cheile vin în ordinea valorilor. Echilibrarea se asigură prin generarea aleatoare de priorități și rearanjarea arborelui binar de căutare pentru a respecta și condiția de min-heap.

În principiu, adăugarea unei valori într-un treap se face într-o frunză (ca la orice arbore binar de căutare) după care se ridică în sus nodul adăugat pentru a respecta condiția de heap pentru prioritate.

În detaliu, insertia și corectia se pot face în două moduri:

- Corectia după insertie (care poate fi iterativă sau recursivă);
- Corectie și insertie, în mod recursiv (cu funcții de rotație scurte).

Varianța de adăugare cu corectie după ce se termină adăugarea:

```
// insertie nod in Treap
void insertT( tnode*& r, int x, int pri) {
    insert(r,x,pri);
    tnode* p= find(r,x);          // adresa nod cu valoarea x
    fixup (r,p);                  // fixup(r,p);
}

// corectie Treap functie de prioritate (recursiv)
void fixupr ( tnode* & r, tnode* t){
    tnode* p;                     // nod parinte al lui t
    if ( (p=t->parent)==NULL ) return; // daca s-a ajuns la radacina
    if ( t->pri < p->pri)           // daca nodul t are prioritate mica
        if (p->st == t)            // daca t e fiu stanga al lui p
            rotateRight(r,p);      // rotatie dreapta p
        else                      // daca t e fiu dreapta al lui p
            rotateLeft(r,p);       // rotatie stanga p
    fixupr(r,p);                  // continua recursiv in sus (p s-a modificat)
}
```

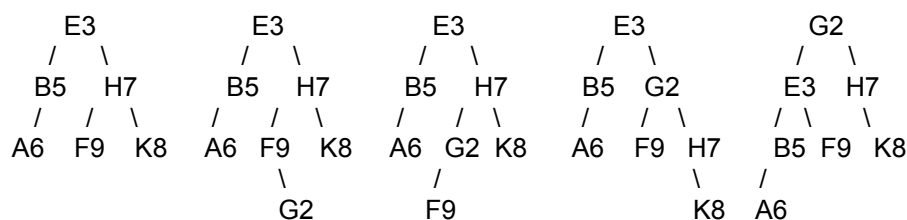

Funcție iterativă de corecție după inserție, pentru mentinere ca heap după prioritate:

```
void fixup ( tnode* & r, tnode* t) {
    tnode* p;
    while ((p=t->parent)!=NULL ) { // cat timp nu s-a ajuns la radacina
        if ( t->pri < p->pri) // daca nodul t are prioritate mica
            if (p->st == t) // daca t e fiu stanga al lui p
                rotateRight(r,p); // rotatie: se aduce t in locul lui p
            else // daca t e fiu dreapta al lui p
                rotateLeft(r,p); // rotatie pentru inlocuire p cu t
            t=p; // muta comparatia mai sus un nivel
    }
}
```

Varianta cu inserție recursivă și rotație (cu funcții scurte de rotație):

```
void insert( tnode*& r, int x, int p) {
    if( r == NULL)
        r = make( x, p); // creare nod cu valoarea x si prioritatea p
    else
        if( x < r->val ) {
            insert( r->st, x,p );
            if( r->st->pri < r->pri )
                rotLeft( r );
        }
        else if( x > r->val ) {
            insert( r->dr, x, p );
            if( r->dr->pri < r->pri )
                rotRight( r );
        }
    // else duplicat; nu se adauga
}
```

La adăugarea unui nod se pot efectua mai multe rotații (dreapta și/sau stânga), dar numărul lor nu poate depăși înălțimea arborelui. Exemplul următor arată etapele prin care trece un treap cu rădăcină E3 la adăugarea cheii G cu prioritatea 2:



initial adauga G2 dupa rot.st. F9 dupa rot.dr. H7 dupa rot.st. E3

Eliminarea unui nod dintr-un treap nu este mult mai complicată decât eliminarea dintr-un arbore binar de căutare; numai după eliminarea unui nod cu doi succesori se compară prioritățile fiilor nodului sters și se face o rotație în jurul nodului cu prioritate mai mare (la stânga pentru fiul stânga și la dreapta pentru fiul dreapta).

O altă utilizare posibilă a unui treap este ca structură de căutare pentru chei cu probabilități diferite de căutare; prioritatea este în acest caz determinată de frecvența de căutare a fiecărei chei, iar rădăcina are prioritatea maximă (este un max-heap).

8.4 ARBORI AVL

Arborii AVL (Adelson-Velski, Landis) sunt arbori binari de căutare în care fiecare subarbore este echilibrat în înălțime. Pentru a recunoaște rapid o dezechilibrare a arborelui s-a introdus în fiecare nod un câmp suplimentar, care să arate fie înălțimea nodului, fie diferența dintre înălțimile celor doi subarbori pentru acel nod (dreapta minus stânga). Acest câmp poate conține valorile -1, 0, 1 pentru noduri "echilibrate".

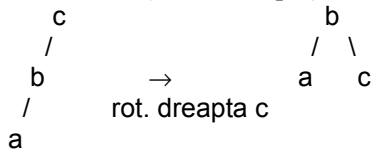
La adăugarea unui nou nod (ca frunză) factorul de echilibru al unui nod interior se poate modifica la -2 (adăugare la subarboarele stânga) sau la +2 (adăugare la subarboarele dreapta), ceea ce va face necesară modificarea structurii arborelui.

Reechilibrarea se face prin rotații simple sau duble, însoțite de recalcularea înălțimii fiecărui nod întâlnit parcurgând arborele de jos în sus, spre rădăcină.

Fie arborele AVL următor:



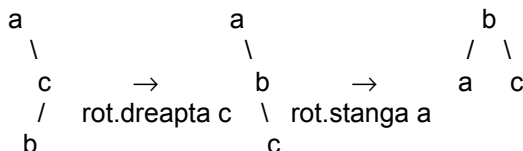
După adăugarea valorii 'a' arborele devine dezechilibrat spre stânga și se rotește nodul 'c' la dreapta pentru reechilibrare (rotatie simplă):



Rotatia dublă este necesară în cazul adăugării valorii 'b' la arborele AVL următor:



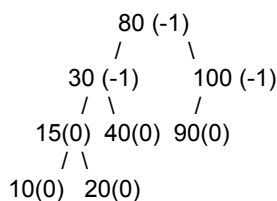
Pentru reechilibrare se rotește c la dreapta și apoi a la stânga (rotatie dublă stânga):



Putem generaliza cazurile anterioare astfel:

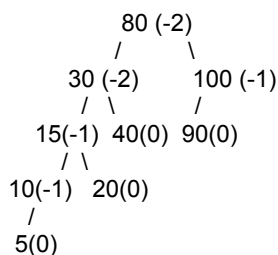
- Insertia în subarboarele dreapta al unui fiu dreapta necesită o rotație simplă la stânga
- Insertia în subarboarele stânga al unui fiu stânga necesită o rotație simplă la dreapta
- Insertia în subarboarele stânga al unui fiu dreapta necesită o rotație dublă la stânga
- Insertia în subarboarele dreapta al unui fiu stânga necesită o rotație dublă la dreapta

Exemplu de arbore AVL (în paranteze factorul de echilibru al nodului):

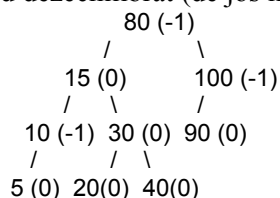


Adăugarea valorilor 120 sau 35 sau 50 nu necesită nici o ajustare în arbore pentru că factorii de echilibru rămân în limitele [-1,+1].

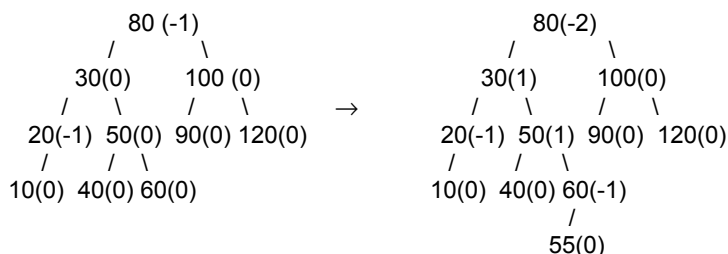
După adăugarea unui nod cu valoarea 5, arborele se va dezechilibra astfel:



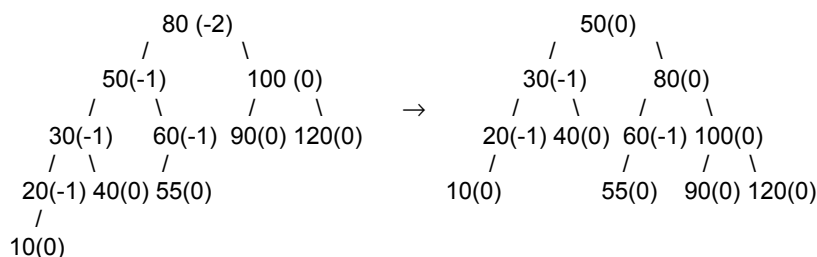
Primul nod dezechilibrat (de jos în sus) este 30, iar soluția este o rotație la dreapta a acestui nod:



Exemplu de rotație dublă (stânga, dreapta) pentru corectia dezechilibrului creat după adăugarea valorii 55 la arborele AVL următor:



Mai întâi se face o rotație la stânga a nodului 30 și apoi o rotație la dreapta a nodului 80 :



Înălțimea maximă a unui arbore AVL este $1.44 \cdot \log(n)$, deci în cazul cel mai rău căutarea într-un arbore AVL nu necesită mai mult de 44% comparații față de cele necesare într-un arbore perfect echilibrat. În medie, este necesară o rotație (simplă sau dublă) cam la 46,5% din adăugări și este suficientă o singură rotație pentru refacere.

Implementarea care urmează memorează în fiecare nod din arbore înălțimea sa, adică înălțimea subarborelui cu rădăcina în acel nod. Un nod vid are înălțimea -1, iar un nod frunză are înălțimea zero.

```

typedef struct node {
    int value;           // valoare din nod
    int h;               // inaltime nod
    struct node *left, *right; // adrese succesori
} Node;
    
```

```
// determina inaltime nod cu adresa t
int th (Node* t) { return t==NULL? -1: t->h; }
```

Operatiile de rotatie simplă recalculează în plus și înălțimea:

```
// rotatie simpla la stanga
void rotateLeft( Node * & b ) {
    Node *a = b->left;
    b->left = a->right;
    a->right = b;
    b->h = max( th(b->left), th(b->right))+1;
    a->h=max( th(a->left),b->h)+1;
    b = a;
}

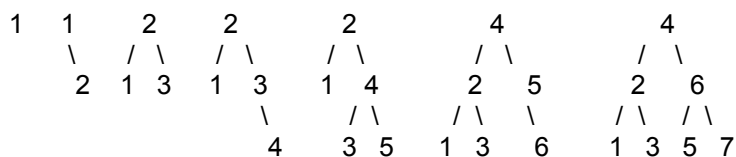
// rotatie simpla la dreapta
void rotateRight( Node * & a ) {
    Node *b = a->right;
    a->right = b->left;
    b->left = a;
    a->h = max( th(a->left), th(a->right))+1;
    b->h=max( th(b->right),a->h)+1;
    a = b;
}
```

Pentru arborii AVL sunt necesare și următoarele rotații duble:

```
// rotatie dubla la stanga (RL)
void doubleLeft( Node * & c ){
    rotateRight( c->left );
    rotateLeft( c );
}

// rotatie dubla la dreapta (LR)
void doubleRight( Node * & c ) {
    rotateLeft( c->right );
    rotateRight( c );
}
```

Evoluția unui arbore AVL la adăugarea valorilor 1,2,3,4,5,6,7 este următoarea:



Exemplu de funcție recursivă pentru adăugarea unei noi valori la un arbore AVL :

```
// Insertie x in arbore AVL
void add ( tnode* & t, int x ) {
    if( t == NULL ) { // daca arbore vid
        t = (tnode*) malloc(sizeof(tnode)); // atunci se creeaza radacina t
        t->val=x; t->st=t->dr=NULL;
        t->h = 0; // inaltime nod unic
        return;
    }
    if (x==t->val) // daca x exista deja in arbore
        return; // atunci nu se mai adauga
}
```

```

if( x < t->val ) {           // daca x este mai mic
    add ( t->st,x );          // atunci se adauga in subarbore stanga
    if( th( t->st ) - th( t->dr ) == 2 ) // daca subarbore dezechilibrat
        if( x < t->st->val ) // daca x mai mic ca fiul stanga
            rotRight( t );   // rotatie dreapta t (radacina subarbore)
        else                 // daca x mai mare ca fiul stanga
            doubleRight( t ); // atunci dubla rotatie la dreapta t
    }
    else {                  // daca x este mai mare
        add ( t->dr,x );      // atunci se adauga la subarboarele dreapta
        if( th( t->dr ) - th( t->st ) == 2 ) // daca subarbore dezechilibrat
            if( x > t->dr->val ) // daca x mai mare ca fiul dreapta
                rotLeft( t );   // atunci rotatie stanga t
            else                 // daca x mai mic ca fiul dreapta
                doubleLeft( t ); // atunci dubla rotatie la stanga
    }
    t->h = max( th( t->st ), th( t->dr ) ) + 1; // recalculeaza inaltime nod t
}
    
```

8.5 ARBORI RB

Arborii de căutare cu noduri colorate ("Red Black Trees") realizează un bun compromis între gradul de dezechilibru al arborelui și numărul de operații necesare pentru menținerea acestui grad. Un arbore RB are următoarele proprietăți:

- Orice nod este colorat fie cu negru fie cu roșu.
- Succesorii (inexistenți) ai nodurilor frunză se consideră colorați în negru
- Un nod roșu nu poate avea decât succesori negri
- Nodul rădăcină este negru
- Orice cale de la rădăcină la o frunză are același număr de noduri negre.

Se consideră că toate frunzele au ca succesor un nod sentinelă negru.

De observat că nu este necesar ca pe fiecare cale să alterneze noduri negre și roșii.

Consecința acestor proprietăți este că cea mai lungă cale din arbore este cel mult dublă față de cea mai scurtă cale din arbore; cea mai scurtă cale poate avea numai noduri negre, iar cea mai lungă are noduri negre și roșii care alternează.

O definiție posibilă a unui nod dintr-un arbore RB:

```

typedef struct Node {
    struct Node *left, *right, *parent; // fii stanga, dreapta si parinte
    char color;                        // culoare nod ('B', 'R')
    int value;                         // date memorate in nod
} Node;
#define NIL &sentinel                // adresa memorata in nodurile frunza
Node sentinel = { NIL, NIL, 0, 'B', 0}; // santinela este un nod negru
    
```

Orice nod nou primește culoarea roșie și apoi se verifică culoarea nodului părinte și culoarea "unchiului" său (frate cu părintele său, pe același nivel). La adăugarea unui nod (roșu) pot apărea două situații care să necesite modificarea arborelui:

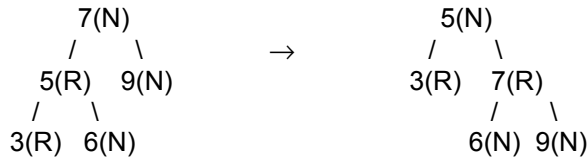
- Părinte roșu și unchi roșu.
- Părinte roșu dar unchi negru.

Prima situație este ilustrată în figura următoare:



După ce se adaugă nodul roșu cu valoarea 3 se modifică culorile nodurilor cu valorile 5 (părinte) și 9 (unchi) din roșu în negru și culoarea nodului 7 din negru în roșu. Dacă 7 nu este rădăcina atunci modificarea culorilor se propagă în sus.

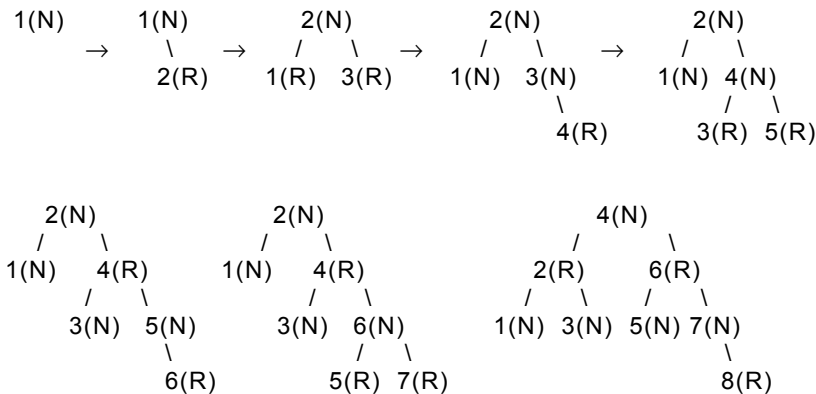
A doua situație este ilustrată de figura următoare (se adaugă nodul 3):



În acest caz se rotește la dreapta nodul 7, dar modificarea nu se propagă în sus deoarece rădăcina subarborelui are aceeași culoare dinainte (negru).

Dacă noul nod se adaugă ca succesor dreapta (de ex. valoarea 6, dacă nu ar fi existat deja), atunci se face mai întâi o rotație la stânga a nodului 5, astfel ca 6 să ia locul lui 5, iar 5 să devină fiu stânga a lui 6.

Pentru a înțelege modificările suferite de un arbore RB vom arăta evoluția sa la adăugarea valorilor 1, 2, ..., 8 (valori ordonate, cazul cel mai defavorabil):



Exemplu de funcție pentru corectare după adăugare (și la primul nod) la arbore RB:

```
// ajustare după inserție nod x
void insertFixup(Node* & root, Node *x) {
    while (x != root && x->parent->color == 'R') { // dacă x are părinte roșu
        if (x->parent == x->parent->parent->left) { // dacă părintele este fiu stânga
            Node *y = x->parent->parent->right; // y este unchi al lui x
            if (y->color == 'R') { // dacă unchi roșu
                x->parent->color = 'B'; // colorează părinte cu negru
                y->color = 'B'; // colorează unchi cu negru
                x->parent->parent->color = 'R'; // colorează bunic x cu roșu
                x = x->parent->parent; // și continuă în sus
            } else { // dacă unchi negru
                if (x == x->parent->right) { // și dacă x este fiu dreapta
                    x = x->parent; // pentru rotație părinte x
                    rotateLeft(root, x); // rotație stânga părinte x
                }
                // recolorare și rotație
                x->parent->color = 'B'; // părintele devine negru
                x->parent->parent->color = 'R'; // iar bunicul roșu
                rotateRight(root, x->parent->parent); // rotație dreapta bunic
            }
        }
    }
}
```

```

} else {          // daca parintele lui x este fiu dreapta al bunicului
    // simetric cu codul anterior
    Node *y = x→parent→parent→left;      // y este unchi al lui x
    if (y→color == 'R') {                  // daca unchi rosu
        x→parent→color = 'B';              // parintele lui x devine negru
        y→color = 'B';                    // unchiul lui x devine negru
        x→parent→parent→color = 'R';      // bunicul lui x devine rosu
        x = x→parent→parent;              // continua in sus cu parintele lui x
    } else {                                //daca unchi negru
        if (x == x→parent→left) {          // daca x este fiu stanga
            x = x→parent;                  // pentru rotatie
            rotateRight(root, x);          // rotatie dreapta parinte x
        }
        x→parent→color = 'B';              // parintele devine negru
        x→parent→parent→color = 'R';      // bunicul devine rosu
        rotateLeft(root, x→parent→parent); // rotatie stanga bunic x
    }
}
}
root→color = 'B';                          // nodul radacina este negru dupa corectie
}

```

Si după operatia de eliminare a unui nod se apelează o functie de ajustare pentru mentinerea conditiilor de arbore RB.

Este posibilă si o variantă de arbori RBT fără legături în sus, la noduri părinte, pentru că la căutarea locului unde se va adăuga noul nod (functia “insert”) se trece prin nodurile părinte si “bunic” ale noului nod si se pot retine adresele lor.

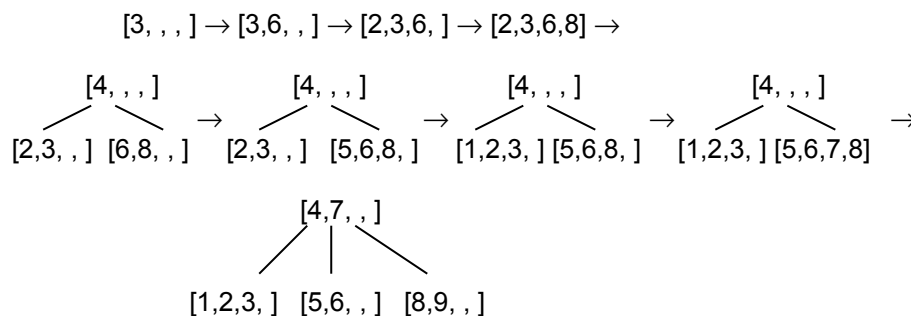
8.6 ARBORI DE CĂUTARE MULTICĂI

Arborii 2-4 (2-3-4) sunt arbori ordonati si echilibrati, cu toate căile de aceeasi lungime si cu următoarele caracteristici:

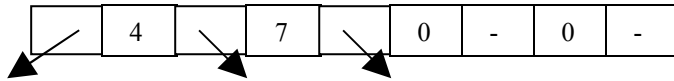
- Un nod contine n valori (n între 2 si 4) si n+1 pointeri către noduri fii (subarbori);
- Valorile dintr-un nod sunt ordonate crescător;
- Toate valorile dintr-un nod sunt mai mari decât valorile din subarborele stânga si mai mici decât valorile aflate în subarborele din dreapta sa.
- Valorile noi pot fi adăugate numai în noduri frunză.
- Prin adăugarea unei noi valori la un nod plin, acesta este spart în alte două noduri cu câte două valori, iar valoarea mediană este trimisă pe nivelul superior;
- Arborele poate creste numai în sus, prin crearea unui nou nod rădăcină.

Să considerăm următoarea secvență de chei (valori) adăugate la un arbore 2-4:
3, 6, 2, 8, 4, 5, 1, 7, 9.

Evolutia arborelui după fiecare valoare adăugată este prezentată mai jos:



Fiecare nod dintr-un arbore 2-4 contine un vector de 4 chei si un vector de 5 pointeri; exemplu pentru nodul rădăcină din arborele de mai sus:



Exemplu de definire a unui nod de arbore 2-4 cu valori întregi:

```
typedef struct bnod {
    int n; // Numar de chei dintr-un nod
    int keys[4]; // Cheile (valorile) din nod
    struct bnod* link[5]; // Legaturi la noduri fii (la subarbori)
} bnod;
```

Căutarea unei valori si căutarea nodului la care trebuie adăugată o nouă valoare se face la fel ca la arborii binari de căutare: se pleacă de la rădăcină si se compară cheia căutată cu cheile din nod; se continuă căutarea pe calea indicată de adresa de la stânga valorii imediat următoare celei căutate (sau de la dreapta ultimei valori din nod dacă valoarea căutată este mai mare decât ultima valoare din nod):

```
// cauta nodul unde trebuie plasat "key": "moveUp" este 1 daca
// "newKey" trebuie plasat in nodul parinte (datorita spargerii unui nod)
// "moveUp" este 0 daca este loc in nodul gasit
int find( bnod* bt, int key, int & moveUp, int & newKey, bnod* & newLink) {
    int idx, gasit;
    moveUp=0;
    gasit=search(bt, key, idx); // cauta key in nodul bt
    if (gasit) return 1; // daca cheia exista nu se mai introduce
    if ( bt == NULL) { // daca arbore vid se creeaza alt nod
        moveUp = 1;
        newKey = key;
        newLink = NULL;
    }
    else { // continua cautarea in noduri fii
        if( find(bt->link[idx + 1],key, moveUp, newKey, newLink))
            return 1; // daca cheie gasita
        if (moveUp) { // daca nod fiu al lui bt este plin
            if ( bt->n < 4) { // si daca e loc in bt
                moveUp = 0;
                add (newKey, newLink, bt, idx + 1); // adauga key la nodul bt
            }
            else { // daca nodul bt este plin
                moveUp = 1; // atunci o cheie se propaga in sus
                split(bt, newKey, newLink, idx, newKey, newLink); // sparge bt
            }
        }
    }
    return 0; // daca cheie negasita
}
```

Funcția “search” cauta o cheie în cadrul unui nod dat si determină indicele unde trebuie introdusă cheia:

```
// cauta "key" in bt si pune in "idx" pozitia sa in vectorul keys
// "idx" este -1 daca "key" este mai mica decat prima cheie din bloc
int search (bnod* bt, int key, int & idx) {
    int gasit=0;
```



```

if (bt==0) return 0; // negasit in arbore vid
if (key < bt->keys[0]) // daca mai mica ca prima cheie din nod
    idx = -1;
else { // cautare secventiala in vectorul de chei
    idx = bt->n - 1;
    while (key < bt->keys[idx] && idx > 0 )
        idx--; // idx = pozitia unde trebuie introdusa cheia
    if (key == bt->keys[idx])
        gasit = 1;
}
return gasit;
}

```

Functia "add" adaugă o cheie la un nod dat:

```

// adauga key la nodul bt in pozitia "idx"
void add (int key, bnod* newRight, bnod * bt, int idx) {
    int j;
    for (j = bt->n; j > idx; j--) { // deplasare dreapta intre idx si n
        bt->keys[j] = bt->keys[j - 1];
        bt->link[j + 1] = bt->link[j];
    }
    bt->keys[idx] = key; // pune cheie in pozitia idx
    bt->link[idx + 1] = newRight; // adresa ultimului fiu al nodului bt
    bt->n++; // creste numarul de fii din bt
}

```

Functia "split" sparge vectorul de chei dintr-un nod dat: jumătate rămân în nodul initial, jumătate se mută într-un nod nou creat, iar cheia mediană se duce pe nivelul superior:

```

// sparge nodul bt si creeaza nodul r, care va fi la dreapta lui newKey
// newKey se muta in nodul parinte
void split( bnod * bt, int key, bnod* right, int loc, int & newKey, bnod* & newLink) {
    int j, median;
    bnod* r= new bnod; // creare nod nou
    if (loc < 2) // loc = pozitia unde se pune key
        median = 2; // median = indice median intre cheile din bt si key
    else
        median = 3;
    for (j = median; j < 4; j++) {
        // muta jumătate din chei in nodul r
        r->keys[j - median] = bt->keys[j];
        r->link[j - median + 1] = bt->link[j + 1];
    }
    r->n = 4 - median; // numar de chei in nodul r
    bt->n = median; // numar de chei ramase in bt
    // adauga cheia key in nodul bt sau r
    if (loc < 2)
        add(key, right, bt, loc+1); // adauga key la bt
    else
        add(key, right, r, loc-median+1); // adauga key la r
    newKey = bt->keys[bt->n - 1]; // se ridica ultima cheie din bt
    r->link[0] = bt->link[bt->n];
    bt->n--;
    newLink = r; // nodul nou va fi la dreapta lui newKey
}

```

Funcția care adaugă o nouă cheie la un arbore 2-4:

```
// insertie "key" in arborele cu radacina bt
int insert(bnod * & bt, int key ) {
    int moveUp, gasit;
    bnod* newLink; bnod* nou;
    int newKey;
    // cauta nodul ce trebuie sa contine "item"
    gasit=find(bt, key, moveUp, newKey, newLink);
    if (gasit) return 1;
    if (moveUp) {          // daca e nevoie se creeaza alta radacina
        nou = new bnod;
        nou->n = 1;
        nou->keys[0] = newKey;
        nou->link[0] = bt;
        nou->link[1] = newLink;
        bt = nou;
    }
    return 0;
}
```

Funcție de afisare infixată (în ordine crescătoare) a valorilor dintr-un arbore 2-3:

```
void infix (bnod* r) {
    int i;
    if (r==0) return;
    // repeta pentru fiecare fiu
    for (i=0; i<r->n; i++){
        infix (r->link[i]);
        printf("%d ", r->keys[i]);
    }
    infix (r->link[r->n]);          // exista n+1 subarbori la n chei
}
```

Din cauza numărului mare de pointeri pe nod arborii 2-4 sunt mai puțin folosiți decât arborii binari echilibrați, dar varianta numită arbore B este cea mai folosită structură de căutare memorată pe un suport extern. Într-un arbore B numărul de chei pe nod este mult mai mare (de ordinul sutelor), fiecare nod se memorează într-un articol pe disc, iar în loc de pointeri se folosesc adrese în cadrul unui fisier disc.

Capitolul 9

STRUCTURI DE GRAF

9.1 GRAFURI CA STRUCTURI DE DATE

Operațiile cu grafuri pot fi considerate:

- Ca un capitol de matematică (teoria grafurilor a fost dezvoltată de matematicieni);
- Ca o sursă de algoritmi interesanți, care pot ilustra diferite clase de algoritmi, soluții alternative pentru o aceeași problemă și metode de analiză a complexității lor;
- Ca probleme de programare ce folosesc diverse structuri de date.

Aici ne interesează acest ultim aspect – probleme de grafuri ca studii de caz în folosirea unor structuri de date, cu implicații asupra performanțelor aplicațiilor, mai ales că unele probleme practice cu grafuri au dimensiuni foarte mari.

Graful este un model abstract (matematic) pentru multe probleme reale, concrete, a căror rezolvare necesită folosirea unui calculator. În matematică un graf este definit ca o pereche de două mulțimi $G = (V, M)$, unde V este mulțimea (nevidă) a vârfurilor (nodurilor), iar M este mulțimea muchiilor (arcelor). O muchie din M uneste o pereche de două vârfuri din V și se notează cu (v, w) .

De obicei nodurile unui graf se numerotează începând cu 1 și deci mulțimea V este o submulțime a mulțimii numerelor naturale N .

Termenii “vârf” și “muchie” provin din analogia unui graf cu un poliedru și se folosesc mai ales pentru grafuri neorientate. Termenii “nod” și “arc” se folosesc mai ales pentru grafuri orientate.

Într-un graf orientat, numit și digraf, arcul (v, w) pleacă din nodul v și intră în nodul w ; el este diferit de arcul (w, v) care pleacă de la w la v . Într-un graf neorientat poate exista o singură muchie între două vârfuri date, notată (v, w) sau (w, v) .

Deoarece în mulțimea M nu pot exista elemente identice înseamnă că între două noduri dintr-un graf orientat pot exista cel mult două arce, iar între două vârfuri ale unui graf neorientat poate exista cel mult o muchie. Două noduri între care există un arc se numesc și noduri vecine sau adiacente. Într-un graf orientat putem vorbi de succesorii și de predecesorii unui nod, respectiv de arce care ies și de arce care intră într-un nod.

Un drum (o cale) într-un graf uneste o serie de noduri $v[1], v[2], \dots, v[n]$ printr-o secvență de arce $(v[1], v[2]), (v[2], v[3]), \dots$. Între două noduri date poate să nu existe un arc, dar să existe o cale, ce trece prin alte noduri intermediare.

Un graf este conex dacă, pentru orice pereche de noduri (v, w) există cel puțin o cale de la v la w sau de la w la v .

Un digraf este tare conex (puternic conectat) dacă, pentru orice pereche de noduri (v, w) există (cel puțin) o cale de la v la w și (cel puțin) o cale de la w la v . Un exemplu de graf tare conex este un graf care conține un ciclu care trece prin toate nodurile: $(1, 2), (2, 3), (3, 4), (4, 1)$.

O componentă conexă a unui graf (V, M) este un subgraf conex (V', M') unde V' este o submulțime a lui V , iar M' este o submulțime a lui M . Împărțirea unui graf neorientat în componente conexe este unică, dar un graf orientat poate fi partitionat în mai multe moduri în componente conexe. De exemplu, graful $(1, 2), (1, 4), (3, 2), (3, 4)$ poate avea componentele conexe $\{1, 2, 4\}$ și $\{3\}$ sau $\{3, 2, 4\}$ și $\{1\}$.

Un ciclu în graf (un circuit) este o cale care porneste și se termină în același nod. Un ciclu hamiltonian este un ciclu complet, care uneste toate nodurile dintr-un graf.

Un graf neorientat conex este ciclic dacă numărul de muchii este mai mare sau egal cu numărul de vârfuri.

Un arbore liber este un graf conex fără cicluri și poate fi neorientat sau orientat.

Putem deosebi trei categorii de grafuri:

a) Grafuri de relație (simple), în care se modelează doar relațiile dintre entități, iar arcele nu au alte atribute.

- b) Grafuri cu costuri (rețele), în care fiecare arc are un cost asociat (o distanță geometrică, un timp de parcurgere, un cost exprimat în bani). Intre costurile arcelor nu există nici o relație.
- c) Rețele de transport, în care fluxul (debitul) prin fiecare arc (tronsoane de rețea) este corelat cu fluxul prin arcele care vin sau pleacă din același nod.

Anumite probleme reale sugerează în mod natural modelarea lor prin grafuri: probleme asociate unor rețele de comunicație, unor rețele de transport de persoane sau de mărfuri, rețele de alimentare cu apă, cu energie electrică sau termică, s.a.

Alteori asocierea obiectelor din lumea reală cu nodurile și arcele unui graf este mai puțin evidentă. Arcele pot corespunde unor relații dintre persoane (persoana x cunoaște persoana y) sau dintre obiecte (piesa x conține piesa y) sau unor relații de condiționare (operația x trebuie precedată de operația y).

Un graf poate fi privit și ca un tip de date abstract, care permite orice relații între componentele structurii. Operațiile uzuale asociate tipului "graf" sunt:

- Inițializare graf cu număr dat de noduri: `initG (Graph & g, int n);`
- Adăugare muchie (arc) la un graf: `addArc (Graph & g, int x, int y);`
- Verifică existența unui arc de la un nod x la un nod y : `int arc(Graph g, int x, int y);`
- Eliminare arc dintr-un graf: `delArc (Graph & g, int x, int y);`
- Eliminare nod dintr-un graf: `delNod (Graph & g, int x);`

Mai mulți algoritmi pe grafuri necesită parcurgerea vecinilor (succesorilor) unui nod dat, care poate folosi funcția "arc" într-un ciclu repetat pentru toți vecinii posibili (deci pentru toate nodurile din graf). Pentru grafuri reprezentate prin liste de vecini este suficientă parcurgerea listei de vecini a unui nod, mult mai mică decât numărul de noduri din graf (egală cu numărul de arce asociate acelui nod).

De aceea se consideră uneori ca operații elementare cu grafuri următoarele:

- Pozitionare pe primul succesor al unui nod dat ("firstSucc");
- Pozitionare pe următorul succesor al unui nod dat ("nextSucc").

Exemplu de afișare a succesorilor unui nod dat k dintr-un graf g :

```
p=firstSucc(g,k);           // p= adresa primului succesor
if (p !=NULL) {             // daca exista un succesor
    printf ("%d ",p->nn);    // atunci se afiseaza
    while ( (p=nextSucc(p)) != NULL) // p=adresa urmatorului succesor
        printf ("%d ",p->nn); // afiseaza urmatorul succesor
}
```

Pentru un graf cu costuri (numit și "rețea") apar câteva mici diferențe la funcțiile "arc" (costul unui arc) și "addArc" (mai are un argument care este costul arcului) :

```
typedef struct {              // tip rețea (graf cu costuri)
    int n,m;                  // nr de noduri și nr de arce
    int **c;                  // matrice de costuri
} Net;
void addArc (Net & g, int v,int w,int cost) { // adauga arcul (v,w) la g
    g.c[v][w]=cost; g.m++;
}
int arc (Net & g, int v, int w) {             // cost arc (v,w)
    return g.c[v][w];
}
```

9.2 REPREZENTAREA GRAFURILOR PRIN ALTE STRUCTURI

Reprezentarea cea mai directă a unui graf este printr-o matrice de adiacente (de vecinătăți), pentru grafuri de relație respectiv printr-o matrice de costuri, pentru rețele. Avantajele reprezentării unui graf printr-o matrice sunt:

- Simplitatea și claritatea programelor.
- Aceeași reprezentare pentru grafuri orientate și neorientate, cu sau fără costuri.

- Se pot obtine usor si repede succesorii sau predecesorii unui nod dat v (coloanele nenule din linia v sunt succesorii, iar liniile nenule din coloana v sunt predecesorii).
- Timp constant pentru verificarea existentei unui arc între două noduri date (nu necesită căutare, deci nu depinde de dimensiunea grafului).

Reprezentarea matricială este preferată în determinarea drumurilor dintre oricare două vârfuri (tot sub formă de matrice), în determinarea drumurilor minime dintre oricare două vârfuri dintr-un graf cu costuri, în determinarea componentelor conexe ale unui graf orientat (prin transpunerea matricei se obtine graful cu arce inversate, numit si graf dual al grafului initial), si în alte aplicatii cu grafuri.

O matrice este o reprezentare naturală pentru o colectie de puncte cu attribute diferite: un labirint (puncte accesibile si puncte inaccesibile), o suprafată cu puncte de diferite înălțimi, o imagine formată din puncte albe si negre (sau colorate diferit), s.a.

Dezavantajul matricei de adiacente apare atunci când numărul de noduri din graf este mult mai mare ca numărul de arce, iar matricea este rară (cu peste jumătate din elemente nule). În astfel de cazuri se preferă reprezentarea prin liste de adiacente.

Matricea de adiacente "a" este o matrice pătratică cu valori întregi, având numărul de linii si de coloane egal cu numărul de noduri din graf. Elementele $a[i][j]$ sunt:

1 (true) dacă există arc de la i la j sau 0 (false) dacă nu există arc de la i la j

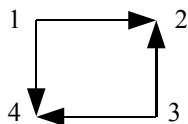
Exemplu de definire a unui tip graf printr-o matrice de adiacente alocată dinamic:

```
// cu matrice alocata dinamic
typedef struct {
    int n,m;           // n=nr de noduri, m=nr de arce
    int ** a;          // adresa matrice de adiacente
} Graf;
```

În general numărul de noduri dintr-un graf poate fi cunoscut de program încă de la început si matricea de adiacente poate fi alocată dinamic.

Matricea de adiacente pentru graful (1,2),(1,4),(3,2),(3,4) este:

	1	2	3	4
1	0	1	0	1
2	0	0	0	0
3	0	1	0	1
4	0	0	0	0



Succesorii unui nod dat v sunt elementele nenule din linia v , iar predecesorii unui nod v sunt elementele nenule din coloana v . De obicei nu există arce de la un nod la el însuși si deci $a[i][i]=0$.

Exemple de functii cu grafuri în cazul utilizării matricei de adiacente.

```
void initG (Graf & g, int n) {           // initializare graf
    int i;
    g.n=n; g.m=0;
    g.a=(int**) malloc( (n+1)*sizeof(int*)); // varfuri numerotate 1..n
    for (i=1;i<=n;i++)
        g.a[i]= (int*) calloc( (n+1),sizeof(int)); // linia 0 si col. 0 nefolosite
}

void addArc (Graf & g, int x,int y) {     // adauga arcul (x,y) la g
    g.a[x][y]=1; g.m++;
}

int arc (Graf & g, int x, int y) {        // daca exista arcul (x,y) in g
    return g.a[x][y];
}

void delArc (Graf& g,int x,int y) {       // elimina arcul (x,y) din g
    g.a[x][y]=0; g.m--;
}
```

Eliminarea unui nod din graf ar trebui să modifice și dimensiunile matricei, dar vom elimina doar arcele ce pleacă și vin în acel nod:

```
void delNode (Graf & g, int x) {           // elimina nodul x din g
    int i;
    for (i=1;i<=g.n;i++) {
        delArc(g,x,i); delArc(g,i,x);
    }
}
```

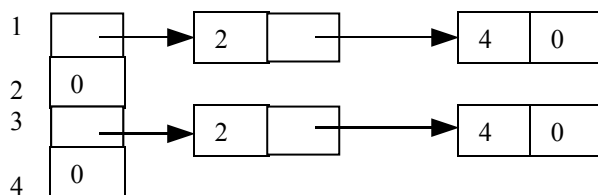
Pentru un graf cu costuri vom înlocui funcția “arc” cu o funcție “carc” care are ca rezultat costul unui arc, iar acolo unde nu există arc vom pune o valoare foarte mare (mai mare ca orice cost din graf), care corespunde unui cost infinit.

```
typedef struct {
    int n,m;           // nr de noduri si nr de arce
    int **c;           // matrice de costuri
} Net;                // retea (graf cu costuri)
void addArc (Net & g, int v,int w,int cost) {
    g.c[v][w]=cost; g.m++;
}
void delArc (Net& g,int v, int w) {
    g.c[v][w]=MARE; g.m--;
}
int arc (Net & g, int v, int w) {
    return g.c[v][w];
}
```

Constanta MARE va fi în general mai mică decât jumătate din cea mai mare valoare pentru tipul de date folosit la costul arcelor, deoarece altfel poate apare depășire la adunare de costuri (de un tip întreg).

Vom aborda acum reprezentarea grafurilor printr-un vector de pointeri la liste de noduri vecine (liste de adiacente).

Lista tuturor arcelor din graf este împărțită în mai multe subliste, câte una pentru fiecare nod din graf. Listele de noduri vecine pot avea lungimi foarte diferite și de aceea se preferă implementarea lor prin liste înlănțuite. Reunirea listelor de succesori se poate face de obicei într-un vector, deoarece permite accesul direct la un nod pe baza numărului său (fără căutare). Figura următoare arată cum se poate reprezenta graful (1,2),(1,4),(3,2),(3,4) printr-un vector de pointeri la liste de adiacente.



Ordinea nodurilor într-o listă de adiacente nu este importantă și de aceea putem adăuga mereu la începutul listei de noduri vecine.

Exemple de operații elementare cu grafuri în cazul folosirii listelor de adiacente:

```
typedef struct nod {
    int val;           // numar nod
    struct nod * leg;  // adresa listei de succesori ptr nodul nr
} * pnod;             // ptr este un tip pointer
typedef struct {
    int n;             // numar de noduri in graf
    pnod * v;          // vector de pointeri la liste de succesori
} Graf;
```

```

void initG (Graf & g, int n) {           // initializare graf
    g.n=n; // nr de noduri
    g.v= (pnod*) calloc(n+1,sizeof(pnod)); // initializare pointeri cu 0 (NULL)
}
void addArc (Graf & g, int x, int y) {    // adauga arcul x-y
    pnod nou = (pnod) malloc (sizeof(nod));
    nou->val=y; nou->leg=g.v[x]; g.v[x]=nou; // adauga la inceput de lista
}
int arc (Graf g,int x,int y) { // test daca exista arcul (x,y) in graful g
    pnod p;
    for (p=g.v[x]; p !=NULL ;p=p->leg)
        if ( y==p->val) return 1;
    return 0;
}
    
```

Reprezentarea unui graf prin liste de vecini ai fiecărui vârf asigură cel mai bun timp de explorare a grafurilor (timp proporțional cu suma dintre numărul de vârfuri și numărul de muchii din graf), iar explorarea apare ca operație în mai mulți algoritmi pe grafuri.

Pentru un graf neorientat fiecare muchie (x,y) este memorată de două ori: y în lista de vecini a lui x și x în lista de vecini a lui y.

Pentru un graf orientat listele de adiacente sunt de obicei liste de succesori, dar pentru unele aplicații interesează predecesorii unui nod (de ex. în sortarea topologică). Lipsa de simetrie poate fi un dezavantaj al listelor de adiacente pentru reprezentarea grafurilor orientate.

Pe lângă reprezentările principale ale structurilor de graf (matrice și liste de adiacente) se mai folosesc uneori și alte reprezentări:

- O listă de arce (de perechi de noduri) este utilă în anumiți algoritmi (cum este algoritmul lui Kruskal), dar mărește timpul de căutare: timpul de execuție al funcției "arc" crește liniar cu numărul de arce din graf.
- O matrice de biti este o reprezentare mai compactă a unor grafuri de relație cu un număr foarte mare de noduri.
- Un vector de pointeri la vectori (cu vectori în locul listelor de adiacente) necesită mai puțină memorie și este potrivit pentru un graf static, care nu se mai modifică.
- Pentru grafuri planare care reprezintă puncte și distanțe pe o hartă poate fi preferabilă o reprezentare geometrică, printr-un vector cu coordonatele vârfurilor.

Anumite cazuri particulare de grafuri pot fi reprezentate mai simplu.

Un arbore liber este un graf neorientat aciclic; într-un arbore liber nu există un nod special rădăcină. Într-un arbore fiecare vârf are un singur părinte (predecesor), deci am putea reprezenta arborele printr-un vector de noduri părinte.

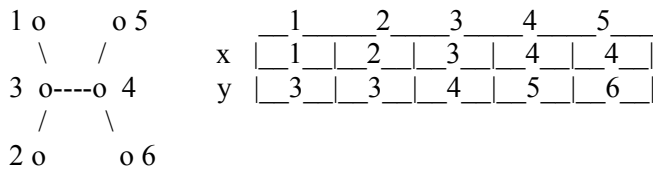
Rezultatul mai multor algoritmi este un arbore liber și acesta se poate reprezenta compact printr-un singur vector. Exemple: arbori de acoperire de cost minim, arborele cu drumurile minime de la un punct la toate celelalte (Dijkstra), s.a.

Un graf conex se poate reprezenta printr-o singură listă - lista arcelor, iar numărul de noduri este valoarea maximă a unui nod prezent în lista de arce (toate nodurile din graf apar în lista de arce). Lista arcelor poate fi un vector sau o listă de structuri, sau doi vectori de noduri:

```

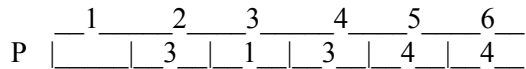
typedef struct { int x,y } arc; // un arc
typedef arc graf[M] ; // lista de arce
    
```

Figura următoare arată un arbore liber și lista lui de arce.



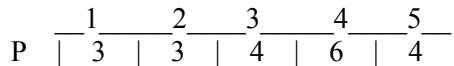
Pentru arbori liberi această reprezentare poate fi simplificată și mai mult, dacă vom impune ca poziția în vector să fie egală cu unul dintre noduri. Vom folosi deci un singur vector P, în care P[k] este perechea (predecesorul) nodului k. Este posibil întotdeauna să notăm arcele din arbore astfel încât fiecare nod să aibă un singur predecesor (sau un singur succesor).

Pentru arborele anterior vectorul P va fi:



Lista arcelor (k, P[k]) este deci: (2,3),(3,1),(4,3),(5,4),(6,4).

Am considerat că nodul 1 nu are nici un predecesor, dar putem să considerăm că nodul ultim nu are nici un predecesor:



Un astfel de vector este chiar vectorul soluție într-o abordare backtracking a unor probleme de grafuri.

9.3 METODE DE EXPLORARE A GRAFURILOR

Explorarea unui graf înseamnă vizitarea sistematică a tuturor nodurilor din graf, folosind arcele existente, astfel încât să se treacă o singură dată prin fiecare nod.

Rezultatul explorării unui graf este o colecție de arbori de explorare, numită și "pădure" de acoperire. Dacă se pot atinge toate nodurile unui graf pornind dintr-un singur nod, atunci rezultă un singur arbore de acoperire. Explorarea unui graf neorientat conex conduce la un singur arbore, indiferent care este nodul de pornire.

Rezultatul explorării unui graf orientat depinde mult de nodul de plecare. Pentru graful orientat cu arcele (1,4),(2,1),(3,2),(3,4),(4,2) numai vizitarea din nodul 3 poate atinge toate celelalte noduri.

De obicei se scrie o funcție care primește un nod de start și încearcă să atingă cât mai multe noduri din graf. Această funcție poate fi apelată în mod repetat, pentru fiecare nod din graf considerat ca nod de start. Astfel se asigură vizitarea tuturor nodurilor pentru orice graf. Fiecare apel generează un arbore de acoperire a unei submultimi de noduri.

Explorarea unui graf poate fi văzută și ca o metodă de enumerare a tuturor nodurilor unui graf, sau ca o metodă de căutare a unui drum către un nod dat din graf.

Transformarea unui graf (structură bidimensională) într-un vector (structură liniară) se poate face în multe feluri, deoarece fiecare nod are mai mulți succesori și trebuie să alegem numai unul singur pentru continuarea explorării.

Algoritmii de explorare dintr-un nod dat pot folosi două metode:

- Explorare în adâncime (DFS = Depth First Search)
- Explorare în lărgime (BFS = Breadth First Search)

Explorarea în adâncime folosește, la fiecare nod, un singur arc (către nodul cu număr minim) și astfel se pătrunde cât mai repede în adâncimea grafului. Dacă rămân noduri nevizitate, se revine treptat la nodurile deja vizitate pentru a lua în considerare și alte arce, ignorate în prima fază. Explorarea DFS din nodul 3 a grafului anterior produce secvența de noduri 3, 2, 1, 4 iar arborele de acoperire este format din arcele 3-2, 2-1 și 1-4.

Vizitarea DFS a unui graf aciclic corespunde vizitării prefixate de la arbori binari.

Explorarea în lărgime folosește, la fiecare nod, toate arcele care pleacă din nodul respectiv și după aceea trece la alte noduri (la succesorii nodurilor vizitate). În felul acesta se explorează mai întâi nodurile adiacente, din "lățimea" grafului și apoi se coboară mai adânc în graf. Explorarea BF din nodul 3 a grafului anterior conduce la secvența de noduri 3,2,4,1 și la arborele de acoperire 3-2, 3-4, 2-1, dacă se folosesc succesorii în ordinea crescătoare a numerelor lor.

Este posibil ca pentru grafuri diferite să rezulte o aceeași secvență de noduri, dar lista de arce este unică pentru fiecare graf (dacă se aplică același algoritm).

De asemenea este posibil ca pentru anumite grafuri să rezulte același arbore de acoperire atât la explorarea DF cât și la explorarea BF; exemple sunt grafuri liniare (1-2, 2-3, 3-4) sau graful 1-2, 1-3, 1-4.

Algoritmul de explorare DFS poate fi exprimat recursiv sau iterativ, folosind o stivă de noduri. Ambele variante trebuie să țină evidența nodurilor vizitate până la un moment dat, pentru a evita vizitarea repetată a unor noduri. Cea mai simplă implementare a multimii de noduri vizitate este un vector "văzut", initializat cu zerouri și actualizat după vizitarea fiecărui nod x ($vazut[x]=1$).

Exemplul următor conține o funcție recursivă de explorare DF dintr-un nod dat v și o funcție pentru vizitarea tuturor nodurilor.

```
void dfs (Graf g, int v, int vazut[]) {    // explorare DF dintr-un nod dat v
    int w, n=g.n;                        // n= nr noduri din graful g
    vazut[v]=1;                          // marcare v ca vizitat
    printf ("%d ",v);                    // afisare (sau memorare)
    for (w=1;w<=n;w++)                  // repeta ptr fiecare posibil vecin w
        if ( arc(g,v,w) && vazut[w]==0 ) // daca w este un vecin nevizitat al lui v
            dfs (g,w,vazut);             // continua explorarea din w
}

// explorare graf in adancime
void df (Graf g) {
    int vazut[M]={0};                    // multime noduri vizitate
    int v;
    for (v=1;v<=g.n;v++)
        if ( !vazut[v] ) {
            printf("\n explorare din nodul %d \n", v);
            dfs(g,v,vazut);
        }
}
```

Pentru afisarea de arce în loc de noduri se modifică puțin funcția, dar ea nu va afișa nimic dacă nu se poate atinge nici un alt nod din nodul de plecare. Este posibil ca un arbore de explorare să conțină un singur nod și nici un arc.

Un algoritm DFS nerecursiv trebuie să folosească o stivă pentru a memora succesorii (vecinii) neprelucrați ai fiecărui nod vizitat, astfel ca să putem reveni ulterior la ei. Algoritmul DFS nerecursiv poate fi descris astfel:

```
pune nodul de plecare în stivă
repetă cât timp stiva nu e goală
    scoate din stivă în x
    afisare și marcare x
    pune în stivă orice succesori nevizitați y al lui x
```

Pentru ca funcția DFS nerecursivă să producă aceleași rezultate ca și funcția DFS recursivă, succesorii unui nod sunt puși în stivă în ordinea descrescătoare a numerelor lor (extragerea lor din stivă și afisarea lor se va face în ordine inversă).

```
void dfs (Graf g,int v, int vazut[]) {
    int x,y;
```

```

Stack s;                // o stiva de intregi
initSt (s);             // initializare stivă
push (s,v);             // pune nodul v pe stiva
while (!emptySt(s)) {
    x=pop (s);           // scoate din stivă în x
    vazut[x]=1;          // marcare x ca vizitat
    printf ("%d ",x);    // si afisare x
    for (y=g.n; y >=1; y--) // cauta un vecin cu x nevizitat
        if ( arc (g,x,y) && ! vazut[y]) {
            vazut[y]=1; push (s,y);    // pune y pe stivă
        }
    }
}

```

Evolutia stivei si variabilelor x,y pentru graful (1,2)(1,4),(2,3),(2,4),(3,4) va fi:

stiva s	x	y	afisare
1			
-	1		1
-	1	4	
4	1	2	
2,4	1		
4	2		2
4	2	4	
4,4	2	3	
3,4,4	2		
4,4	3		3
4,4	3	4	
4,4,4	3		
4,4	4		4
4	4		
-	4		

Algoritmul de explorare în lățime afișează si memorează pe rând succesorii fiecărui nod. Ordinea de prelucrare a nodurilor memorate este aceeași cu ordinea de introducere în listă, deci lista este de tip “coadă”. Algoritmul BFS este foarte asemănător algoritmului DFS nerecursiv, diferența apare numai la tipul listei folosite pentru memorarea temporară a succesorilor fiecărui nod: stivă la DFS si coadă la BFS

```

// explorare în lățime dintr-un nod dat v
void bfs ( Graf g, int v, int vazut[]) {
    int x,y ;
    Queue q;                // o coada de intregi
    initQ (q);
    vazut[v]=1;             // marcare v ca vizitat
    addQ (q,v);             // pune pe v în coadă
    while (! emptyQ(q)) {
        x=delQ (q);         // scoate din coadă în x
        for (y=1;y <=g.n; y++) // repeta ptr fiecare potential vecin cu x
            if ( arc(g,x,y) && vazut[y]==0) { // daca y este vecin cu x si nevizitat
                printf ("%d - %d \n",x,y);    // scrie muchia x-y
                vazut[y]=1;                  // y vizitat
                addQ(q,y);                  // pune y in coada
            }
        }
    }
}

```

Evolutia cozii q la explorarea BF a grafului cu arcele (1,2),(1,4),(2,3),(2,4),(3,4):

coada q	x	y	afisare
1			
-	1		
	1	2	1 - 2
2	1	4	1 - 4
2,4	1		
4	2		
4	2	3	2 - 3
4,3	2		
4	3		
-	4		

Un drum minim între două vârfuri este drumul care folosește cel mai mic număr de muchii. Drumurile minime de la un vârf v la toate celelalte noduri pot fi găsite prin explorare în lărgime din nodul v , cu actualizare distante față de v , la fiecare coborâre cu un nivel în graf. Vom folosi un vector d cu $d[y]$ =distanța vârfului y față de "rădăcina" v și un vector p , cu $p[y]$ =număr vârf predecesor pe calea de la v la y .

```
// distante minime de la v la toate celelalte noduri din g
void bfs (Graph g, int v,int vazut[],int d[], int p[]) {
    int x,y;
    Queue q;
    initQ (q);
    vazut[v]=1; d[v]=0; p[v]=0;
    addQ (q,v);                // pune v in coada
    while (! emptyQ(q)) {
        x=delQ (q);            // scoate din coadă în x
        for (y=1;y <=g.n;y++)
            if ( arc(g,x,y) && vazut[y]==0) { // test dacă arc între x și y
                vazut[y]=1; d[y]=d[x]+1;      // y este un nivel mai jos ca x
                p[y]=x;                        // x este predecesorul lui x pe drumul minim
                addQ(q,y);
            }
    }
}
```

Pentru afisarea vârfurilor de pe un drum minim de la v la x trebuie parcurs în sens invers vectorul p (de la ultimul element la primul):

$x \rightarrow p[x] \rightarrow p[p[x]] \rightarrow \dots \rightarrow v$

9.4 SORTARE TOPOLOGICĂ

Problema sortării topologice poate fi formulată astfel: între elementele unei multimi A există relații de condiționare (de precedentă) de forma $a[i] \ll a[j]$, exprimate în cuvinte astfel: $a[i]$ precede (condiționează) pe $a[j]$, sau $a[j]$ este condiționat de $a[i]$. Se mai spune că $a[i]$ este un predecesor al lui $a[j]$ sau că $a[j]$ este un succesor al lui $a[i]$. Un element poate avea oricâți succesori și predecesori. Multimea A supusă unor relații de precedentă poate fi văzută ca un graf orientat, având ca noduri elementele $a[i]$; un arc de la $a[i]$ la $a[j]$ arată că $a[i]$ precede pe $a[j]$.

Exemplu : $A = \{ 1,2,3,4,5 \}$

$2 \ll 1 \quad 1 \ll 3 \quad 2 \ll 3 \quad 2 \ll 4 \quad 4 \ll 3 \quad 3 \ll 5 \quad 4 \ll 5$

Scopul sortării topologice este ordonarea (afisarea) elementelor multimii A într-o succesiune liniară astfel încât fiecare element să fie precedat în această succesiune de elementele care îl conditionează.

Elementele multimii A pot fi privite ca noduri dintr-un graf orientat, iar relațiile de conditionare ca arce în acest graf. Sortarea topologică a nodurilor unui graf orientat nu este posibilă dacă graful conține cel puțin un ciclu. Dacă nu există nici un element fără conditionări atunci sortarea nici nu poate începe. Uneori este posibilă numai o sortare topologică parțială, pentru o parte din noduri.

Pentru exemplul dat există două secvențe posibile care satisfac condițiile de precedentă :

2, 1, 4, 3, 5 și 2, 4, 1, 3, 5

Determinarea unei soluții de ordonare topologică se poate face în câteva moduri:

- Incepând cu elementele fără predecesori (neconditionate) și continuând cu elementele care depind de acestea (nodul 2 este un astfel de element în exemplul dat);
- Incepând cu elementele fără succesori (finale) și mergând către predecesori, din aproape în aproape (nodul 5 în exemplu).
- Algoritmul de explorare în adâncime a unui graf orientat, completat cu afisarea nodului din care începe explorarea, după ce s-au explorat toate celelalte noduri.

Aceste metode pot folosi diferite structuri de date pentru reprezentarea relațiilor dintre elemente; în cazul (a) trebuie să putem găsi ușor predecesorii unui element, iar în cazul (b) trebuie să putem găsi ușor succesorii unui element,

Algoritmul de sortare topologică cu liste de predecesori este:

repetă

caută un nod nemarcat și fără predecesori

dacă s-a găsit atunci

afisează nod și marchează nod

sterge nod marcat din graf

până când nu mai sunt noduri fără predecesori

dacă rămân noduri nemarcate atunci

nu este posibilă sortarea topologică

Pentru exemplul dat evoluția listelor de predecesori este următoarea:

1 - 2	1 -	1 -	1 -	1 -
2 -	2 -	2 -	2 -	2 -
3 - 1,2,4	3 - 1,4	3 - 4	3 -	3 -
4 - 2	4 -	4 -	4 -	4 -
5 - 3,4	5 - 3,4	5 - 3	5 - 3	5 -
scrie 2	scrie 1	scrie 4	scrie 3	scrie 5

Programul următor ilustrează acest algoritm .

```
int nrcond (Graf g, int v ) { // determina nr de conditionari nod v
int j,cond=0; // cond = numar de conditionari
for (j=1;j<=g.n;j++)
if ( arc(g,j,v))
cond++;
return cond;
}

// sortare topologica si afisare
void topsort (Graf g) {
int i,j,n=g.n,ns,gasit, sortat[50]={0};
ns=0; // noduri sortate si afisate
do {
gasit=0;
// cauta un nod nesortat, fara conditionari
for (i=1;i<= n && !gasit; i++)
```

```

    if ( ! sortat[i] && nrcond(g,i)==0) { // i fara conditionari
        gasit =1;
        sortat[i]=1; ns++; // noduri sortate
        printf ("%d ",i);    // scrie nod gasit
        delNod(g,i);        // elimina nodul i din graf
    }
} while (gasit);
if (ns != n) printf ("\n nu este posibila sortarea topologica! ");
}

```

Algoritmul de sortare topologică cu liste de succesori este:

```

repetă
    caută un nod fără succesori
    pune nod găsit în stivă si marchează ca sortat
    elimină nod marcat din graf
până când nu mai există noduri fără succesori
dacă nu mai sunt noduri nemarcate atunci
    repetă
        scoate nod din stivă si afisare nod
    până când stiva goală

```

Evolutia listelor de succesori pentru exemplul dat este:

1 - 3	1 - 3	1 -	1 -	1 -
2 - 1,3,4	2 - 1,3,4	2 - 1,4	2 - 4	2 -
3 - 5	3 -	3 -	3 -	3 -
4 - 3,5	4 - 3	4 -	4 -	4 -
5 -				
pune 5	pune 3	pune 1	pune 4	pune 2

La extragerea din stivă se afisează: 2, 4, 1, 3, 5

9.5 APLICATII ALE EXPLORĂRII ÎN ADÂNCIME

Explorarea în adâncime stă la baza altor algoritmi cu grafuri, cum ar fi: determinarea existentei ciclurilor într-un graf, găsirea componentelor puternic conectate dintr-un graf, sortare topologică, determinare puncte de articulare s.a.

Determinarea componentelor conexe ale unui graf se poate face prin repetarea explorării DF din fiecare nod nevizitat în explorările anterioare. Un apel al funcției “dfs” afisează o componentă conexă. Pentru grafuri neorientate există un algoritm mai performant de aflare a componentelor conexe, care folosește tipul abstract de date “colecție de mulțimi disjuncte”.

Algoritmul de sortare topologică derivat din explorarea DF se bazează pe faptul că explorarea în adâncime vizitează toți succesorii unui nod. Explorarea DF va fi repetată până când se vizitează toate nodurile din graf. Funcția “ts” este derivată din funcția “dfs”, în care s-a înlocuit afisarea cu punerea într-o stivă a nodului cu care a început explorarea, după ce s-au memorat în stivă succesorii săi. În final se scoate din stivă si se afisează tot ce a pus funcția “ts”.

Programul următor realizează sortarea topologică ca o variantă de explorare în adâncime a unui graf g si folosește o stivă s pentru memorarea nodurilor.

```

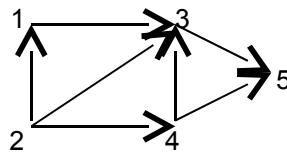
Stack s;          // stiva folosita in doua functii
// sortare topologica dintr-un nod v
void ts (Graf g,int v) {
    int w;
    vazut[v]=1;
    for (w=1;w<=g.n;w++)

```

```

if ( arc (g,v,w) && ! vazut[w])
    ts(g,w);
push (s,v);
}
// sortare topologica graf
void main () {
    int i,j,n; Graf g;
    readG(g); n=g.n;
    for (j=1;j<=n;j++)
        vazut[j]=0;
    initSt(s);
    for (i=1;i<=n;i++)
        if ( vazut[i]==0 )
            ts(g,i);
    // scoate din stiva si afiseaza
    while( ! emptySt (s)) {
        pop(s,i);
        printf("%d ",i);
    }
}

```



Secventa de apeluri si evolutia stivei pentru graful 2-1, 1-3, 2-3, 2-4, 4-3, 3-5, 4-5 :

Apel	Stiva	Din
ts(1)		main()
ts(3)		ts(1)
ts(5)		ts(3)
push(5)	5	ts(5)
push(3)	5,3	ts(3)
push(1)	5,3,1	ts(1)
ts(2)		main()
ts(4)		ts(2)
push(4)	5,3,1,4	ts(4)
push(2)	5,3,1,4,2	ts(2)

Numerotarea nodurilor în ordinea de vizitare DF permite clasificarea arcelor unui graf orientat în patru clase:

- Arce de arbore, componente ale arborilor de explorare în adâncime (de la un nod în curs de vizitare la un nod nevizitat încă).
- Arce de înaintare, la un succesor (la un nod cu număr de vizitare mai mare).
- Arce de revenire, la un predecesor (la un nod cu număr de vizitare mai mic).
- Arce de traversare, la un nod care nu este nici succesor, nici predecesor .

Fie graful cu 4 noduri si 6 arce:

(1,2), (1,3), (2,3), (2,4), (4,1), (4,3)

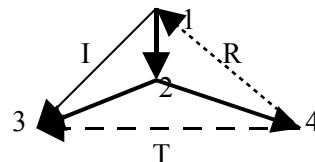
După explorarea DF cele 6 arce se împart în:

Arce de arbore (dfs): (1,2), (2,3), (2,4)

Arce înainte : (1,3)

Arce înapoi : (4,1)

Arce transversale : (4,3)



Numerele de vizitare DF pentru nodurile 1,2,3,4 sunt: 1,2,3,4 iar vectorul P contine numerele 0,1,2,2 (în 3 si 4 se ajunge din 2).

Dacă există cel puțin un arc de revenire la explorarea DF a unui graf orientat atunci graful conține cel puțin un ciclu, iar un graf orientat fără arce de revenire este aciclic.

Pentru a diferenția arcele de revenire de arcele de traversare se memorează într-un vector P nodurile din arborele de explorare DF; un arc de revenire merge către un nod din P , dar un arc de traversare nu are ca destinație un nod din P .

```
// clasificare arce la explorare în adâncime dintr-un nod dat v
void dfs (int v, int t1[]) { // t1[k]= tip arc k
    int w,k;
    nv[v]=++m; // nv[k]= numar de vizitare nod k
    for (w=1;w<=n;w++)
        if ( (k=arc(v,w)) >= 0 ) // k= numar arc de la v la w
            if (nv[w]==0) { // daca w nevizitat
                t1[k]='A'; p[w]=v; // atunci v-w este arc de arbore
                dfs (w,t1); // continua explorarea din w
            }
            else // daca w este deja vizitat
                if ( nv[v] < nv[w]) // daca w vizitat dupa v
                    t1[k]='I'; // atunci v-w este arc inainte
                else // daca w vizitat inaintea lui v
                    if ( precede(w,v) ) // daca w precede pe v in arborele DFS
                        t1[k]='R'; // atunci v-w este arc inapoi (de revenire)
                    else // daca w nu precede pe v in arborele DFS
                        t1[k]='T'; // atunci v=w este arc transversal
    }
    // daca v precede pe w in arborele DFS
    int precede (int v, int w) {
        while ( (w=p[w]) > 0)
            if (w==v)
                return 1;
        return 0;
    }
}
```

Funcția de explorare DF poate fi completată cu numerotarea nodurilor atât la primul contact cu nodul, cât și la ultimul contact (la ieșirea din funcția dfs). Funcția dfs care urmează folosește variabila externă 't', inițializată cu zero în programul principal și incrementată la fiecare intrare în funcția "dfs". Vectorul t1 este actualizat la intrarea în funcția dfs, iar vectorul t2 la ieșirea din dfs.

```
int t; // var externa, implicit zero
void dfs (Graph g, int v, int t1[], int t2[]) {
    int w;
    t1[v] = ++t; // descoperire nod v
    for(w=1; w<=g.n; w++) // g.n= nr de noduri
        if ( arc(g,v,w) && t1[w]==0 ) // daca w este succesor nevizitat
            dfs (g,w,t1,t2); // continua vizitare din w
    t2[v] = ++t; // parasire nod v
}
```

Pentru digraful cu 4 noduri și cu arcele (1,3), (1,4), (2,1), (2,3), (3,4), (4,2) arborele dfs este secvența 1→3→4→2, iar vectorii t1 și t2 vor conține următoarele valori după apelul dfs(1):

nod k	1	2	3	4	
t1[k]	1	4	2	3	(intrare in nod)
t2[k]	8	5	7	6	(iesire din nod)

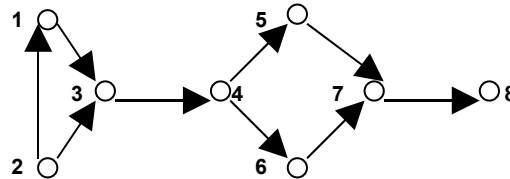
Se observă că ultimul nod vizitat (2) este și primul părăsit, după care este părăsit nodul vizitat anterior; numerele $t1(k)$ și $t2(k)$ pot fi privite ca paranteze în jurul nodului k , iar structura de paranteze a grafului la vizitare dfs este :

(1 (3 (4 (2))))

O componentă puternic conectată (tare conexă) dintr-un digraf este o submultime maximală a nodurilor astfel încât există o cale între oricare două noduri din cpc.

Pentru determinarea componentelor puternic conectate (cpc) dintr-un graf orientat vom folosi următorul graf orientat ca exemplu:

$1 \rightarrow 3$, $3 \rightarrow 2$, $2 \rightarrow 1$, $3 \rightarrow 4$, $4 \rightarrow 5$, $5 \rightarrow 7$, $7 \rightarrow 6$, $6 \rightarrow 4$, $7 \rightarrow 8$



Vizitarea DFS dintr-un nod oarecare v produce o multime cu toate nodurile ce pot fi atinse plecând din v . Repetând vizitarea din v pentru graful cu arce inversate ca sens obținem o altă multime de noduri, din care se poate ajunge în v . Intersecția celor două mulțimi reprezintă componenta tare conexă care conține nodul v . După eliminarea nodurilor acestei componente din graf se repetă operația pentru nodurile rămase, până când nu mai sunt noduri în graf.

Pentru graful anterior vizitarea DFS din 1 produce mulțimea $\{1,3,2,4,5,7,6,8\}$ iar vizitarea grafului inversat din 1 produce mulțimea $\{1,2,3\}$. Intersecția celor două mulțimi $\{1,2,3\}$ reprezintă componenta tare conexă care conține nodul 1. După eliminarea nodurilor 1,2 și 3, vizitarea grafului rămas din nodul 4 produce mulțimea $\{4,5,7,6,8\}$, iar vizitarea din 4 a grafului cu arce inversate produce mulțimea $\{4,6,7,5\}$, deci componenta tare conexă care-l conține pe 4 este $\{4,5,6,7\}$. Ultima componentă cpc conține doar nodul 8.

Este posibilă îmbunătățirea acestui algoritm pe baza observației că s-ar putea determina toate componentele cpc la o singură vizitare a grafului inversat, folosind ca puncte de plecare nodurile în ordine inversă vizitării DFS a grafului initial. Algoritmul folosește vectorul $t2$ cu timpii de părăsire ai fiecărui nod și repetă vizitarea grafului inversat din nodurile considerate în ordinea inversă a numerelor $t2$.

Pentru graful anterior vectorii $t1$ și $t2$ la vizitarea DFS din 1 vor fi:

nod i	1	2	3	4	5	6	7	8
$t1[i]$	1	3	2	5	6	8	7	10
$t2[i]$	16	4	15	14	13	9	12	11

Vizitarea grafului inversat se va face din 1 ($t2=16$) cu rezultat $\{1,2,3\}$, apoi din 4 ($t2=14$) cu rezultat $\{4,6,7,5\}$ și din 8 (singurul nod rămas) cu rezultat $\{8\}$.

Graful cu arce inversate se obține prin transpunerea matricei initiale.

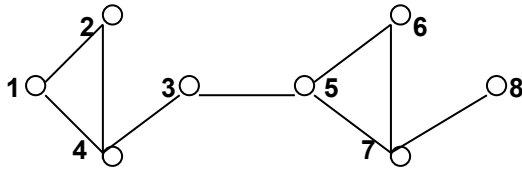
Pentru grafuri neorientate ce reprezintă rețele de comunicații sunt importante problemele de conectivitate. Un punct de articulare (un punct critic) dintr-un graf conex este un vârf a cărui eliminare (împreună cu muchiile asociate) face ca graful să nu mai fie conex. O "punte" (o muchie critică) este o muchie a cărei eliminare face ca graful rămas să nu mai fie conex. O componentă biconexă este o submultime maximală de muchii astfel că oricare două muchii se află pe un ciclu simplu. Fie graful conex cu muchiile:

(1,2), (1,4), (2,4), (3,4), (3,5), (5,6), (5,7), (6,7), (7,8)

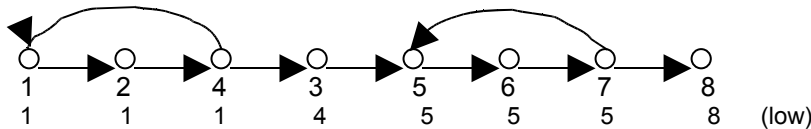
Puncte de articulare: 3, 4, 5, 7

Muchi critici: 3-4, 3-5

Componente biconexe: (1,2,4), (5,6,7)



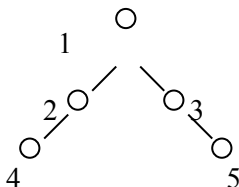
Un algoritm eficient pentru determinarea punctelor critice dintr-un graf conex folosește vizitarea în adâncime dintr-un vârf rădăcină oarecare. Arborele de vizitare al unui graf neorientat conține numai două tipuri de arce: de explorare (de arbore) și arce de revenire (înapoi). Pentru graful anterior arborele produs de vizitarea DFS din vârful 1 conține arcele $1 \rightarrow 2$, $2 \rightarrow 4$, $4 \rightarrow 3$, $3 \rightarrow 5$, $5 \rightarrow 6$, $6 \rightarrow 7$, $7 \rightarrow 8$, iar arcele înapoi sunt $4 \rightarrow 1$ și $7 \rightarrow 5$.



Un vârf terminal (o frunză) din arbore nu poate fi punct de articulare, deoarece eliminarea lui nu întrerupe accesul la alte vârfuri, deci vârful 8 nu poate fi un punct critic. Rădăcina arborelui DFS poate fi punct de articulare numai dacă are cel puțin doi fii în arborele DFS, căci eliminarea ei ar întrerupe legătura dintre fiii săi. Deci 1 nu este punct de articulare. Un vârf interior v din arborele DFS nu este punct de articulare dacă există în graf o muchie înapoi de la un vârf u următor lui v în arbore la un vârf w anterior lui v în arbore, pentru că eliminarea lui v din graf nu ar întrerupe accesul de la w la u . O muchie înapoi este o muchie de la un vârf cu low mare la un vârf cu low mic. Un nod u următor lui v în arbore are $low[u] > low[v]$, adică u este vizitat după v . De exemplu, $low[1]=1$, $low[4]=3$, deci 4 este un descendent al lui 1 în arbore.

Pentru graful anterior vârful 2 nu este punct critic deoarece există muchia $(4,1)$ care permite accesul de la predecesorul lui 2 (1) la succesorul lui 2 (4); la fel 6 nu este punct critic deoarece există muchia $(7,5)$ de la fiul 7 la părintele 5. Vârful 3 este punct de articulare deoarece nu există o muchie de la fiul 5 la părintele său 4 și deci eliminarea sa ar întrerupe accesul către vârful 5 și următoarele. La fel 4, 5 și 7 sunt puncte critice deoarece nu există muchie înapoi de la un fiu la un părinte.

Un alt exemplu este graful cu 5 vârfuri și muchiile 1-2, 1-3, 2-4, 3-5:



Arborele de explorare dfs din 1 este același cu graful; vârfurile 4 și 5 sunt frunze în arbore, iar 1 este rădăcină cu doi fii. Punctele de articulare sunt 1, 2, 3.

Dacă se adaugă muchiile 1-4 și 1-5 la graful anterior atunci 2 și 3 nu mai sunt puncte critice (există arce înapoi de la succesori la predecesori).

Implementarea algoritmului folosește 3 vectori de noduri:

$d[v]$ este momentul vizitării (descoperirii) vârfului v la explorarea dfs

$p[v]$ este predecesorul vârfului v în arborele de explorare dfs

$low[v]$ este cel mai mic $d[w]$ al unui nod w anterior lui v în arborele dfs, către care urcă un arc înapoi de la un succesor al lui v .

Vectorul "low" se determină la vizitarea dfs, iar funcția ce determină punctele de articulare verifică pe rând fiecare vârf din graf ce statut are în arborele dfs:

```
// numara fii lui v in arborele descris prin vectorul de predecesori p
int fii (int v, int p[], int n) {
    int i,m=0;
    for (i=1;i<=n;i++)
```

```

    if ( i !=v && p[i]==v) // daca i are ca parinte pe v
        m++;
    return m;
}
// vizitare in adancime g din varful v, cu creare vectori d,p,low
void dfs (Graf g,int v,int t,int d[],int p[],int low[]) {
    int w;
    low[v]=d[v]=++t;
    for (w=1;w<=g.n;w++) {
        if ( g.a[v][w]) // daca w este vecin cu v
            if( d[w]==0) { // daca w nevizitat
                p[w]=v; // w are ca predecesor pe v
                dfs(g,w,t,d,p,low); // continua vizitarea din w
                low[v]=min (low[v],low[w]); // actualizare low[v]
            }
        else // daca w deja vizitat
            if ( w != p[v]) // daca arc inapoi v-w
                low[v]=min(low[v],d[w]); // actualizare low[v]
    }
}
// gasire puncte de articulare
void artic (Graf g, int d[],int p[],int low[] ) {
    int v,w,t=0; // t= moment vizitare (descoperire varf)
    dfs(g,1,t,d,p,low); // vizitare din 1 (graf conex)
    for (v=1;v<=g.n;v++){
        if (p[v]==0){
            if( fii(v,p,g.n)>1) // daca radacina cu cel putin 2 fii
                printf("%d ",v); // este pct de artic
        }
        else // daca nu e radacina
            for (w=1;w <=g.n;w++) { // daca v are un fiu w in arborele DFS
                if ( p[w]==v && low[w] >= d[v]) // cu low[w] > d[v]
                    printf("%d ",v); // atunci v este pct de artic
            }
    }
}
}

```

9.6 DRUMURI MINIME IN GRAFURI

Problema este de a găsi drumul de cost minim dintre două noduri oarecare i și j dintr-un graf orientat sau neorientat, cu costuri pozitive.

S-a arătat că această problemă nu poate fi rezolvată mai eficient decât problema găsirii drumurilor minime dintre nodul i și toate celelalte noduri din graf. De obicei se consideră ca nod sursă i chiar nodul 1 și se determină lungimile drumurilor minime $d[2], d[3], \dots, d[n]$ până la nodurile 2, 3, ..., n . Pentru memorarea nodurilor de pe un drum minim se folosește un singur vector P , cu $p[i]$ egal cu nodul precedent lui i pe drumul minim de la 1 la i (multimea drumurilor minime formează un arbore, iar vectorul P reprezintă acest arbore de căi în graf).

Cel mai eficient algoritm cunoscut pentru problema drumurilor optime cu o singură sursă este algoritmul lui Dijkstra, care poate fi descris în mai multe moduri: ca algoritm de tip “greedy” cu o coadă cu priorități, ca algoritm ce folosește operația de “relaxare” (comună și altor algoritmi), ca algoritm cu mulțimi de vârfuri sau ca algoritm cu vectori. Diferențele de prezentare provin din structurile de date utilizate.

În varianta următoare se folosește un vector D astfel că $d[i]$ este distanța minimă de la 1 la i , dintre drumurile care trec prin noduri deja selectate. O variabilă S de tip mulțime memorează numerele nodurilor cu distanță minimă față de nodul 1, găsite până la un moment dat. Inițial $S=\{1\}$ și $d[i]=\text{cost}[1][i]$, adică se consideră arcul direct de la 1 la i ca drum minim între 1 și i . Pe măsură ce algoritmul evoluează, se actualizează D și S .

```

S = {1}    // S =multime noduri ptr care s-a determinat dist. minima fata de 1
repetă cât timp S contine mai puțin de n noduri {
    găsește muchia (x,y) cu x în S și y nu în S care face minim d[x]+cost(x,y)
    adaugă y la S
    d[y] = d[x] + cost(x,y)
}
    
```

La fiecare pas din algoritmul Dijkstra:

- Se găsește dintre nodurile j care nu aparțin lui S acel nod "jmin" care are distanța minimă față de nodurile din S;
- Se adaugă nodul "jmin" la mulțimea S;
- Se recalculează distanțele de la nodul 1 la nodurile care nu fac parte din S, pentru că distanțele la nodurile din S rămân neschimbate;
- Se reține în p[j] numărul nodului precedent cel mai apropiat de nodul j (de pe drumul minim de la 1 la j).

Pentru a ilustra modul de lucru al algoritmului Dijkstra considerăm un graf orientat cu următoarele costuri de arce:

```

(1,2)=5; (1,4)=2; (1,5)=6;
(2,3)=3;
(3,2)=4; (3,5)=4;
(4,2)=2; (4,3)=7; (4,5)=3;
(5,3)=3;
    
```

Drumurile posibile între 1 și 3 și costul lor :

1-2-3 = 8 ; 1-4-3 = 9; 1-4-2-3 = 7; 1-4-5-3 = 8; 1-5-3 = 9;

Drumurile minime de la 1 la celelalte noduri sunt în acest graf:

```

1.4.2      de cost 4
1.4.2.3    de cost 7
1.4        de cost 2
1.4.5      de cost 5
    
```

De observat că într-un drum minim fiecare drum parțial este minim; astfel în drumul 1-4-2-3, drumurile parțiale 1-4-2 și 1-4 sunt și ele minime.

Evoluția vectorilor D și S pentru acest graf în cazul algoritmului Dijkstra :

S	d[2]	d[3]	d[4]	d[5]	nod sel.
1	5	M	2	6	4
1,4	4	9	2	5	2
1,4,2	4	7	2	5	5
1,4,2,5	4	7	2	5	3

Vectorul P va arăta în final astfel:

p[2]	p[3]	p[4]	p[5]
4	2	1	4

Exemplu de funcție pentru algoritmul Dijkstra:

```

void dijkstra (Net g,int p[]) { // Net este tipul abstract "graf cu costuri"
    int d[M],s[M];              // s= noduri ptr care se stie distanta minima
    int dmin; int jmin,i,j;
    for (i=2;i<=g.n;i++) {
        p[i]=1; d[i]=carc(g,1,i); // distante initiale de la 1 la alte noduri
    }
    s[1]=1;
}
    
```

```

for (i=2;i<=g.n;i++) {          // repeta de n-1 ori
    // caută nodul j ptr care d[j] este minim
    dmin =MARE;
    for (j=2;j<=g.n;j++)        // determina minimul dintre distantele d[j]
        if (s[j]==0 && dmin > d[j]) { // dacă j nu e in S si este mai aproape de S
            dmin =d[j]; jmin=j;
        }
    s[jmin]=1;                  // adauga nodul jmin la S
    for (j=2;j<=g.n;j++)        // recalculare distante noduri fata de 1
        if ( d[j] >d[jmin] + carc(g,jmin,j) ) {
            d[j] =d[jmin] + carc(g,jmin,j);
            p[j] =jmin;          // predecesorul lui j pe drumul minim
        }
    }
}
}

```

În programul principal se apelează repetat funcția "drum":

```

for(j=2;j<=n;j++)
    drum (p,1, j);          // afisare drum minim de la 1 la j

```

Afisarea drumului minim pe baza vectorului "p" se poate face recursiv sau iterativ:

```

// drum minim intre i si j - recursiv
void drum (int p[], int i,int j) {
    if (j != i)
        drum (p,i,p[j]);
    printf ("%d ",j);
}

// drum minim intre i si j - iterativ
void drum (int p[], int i,int j){
    int s[M], sp=0;          // s este o stiva vector cu varful in sp
    printf ("%d ",i);        // primul nod de pe calea i~j
    while (j != i) {         // pune pe stiva nodurile precedente lui j
        s[++sp]=j;
        j=p[j];              // precesorul lui j
    }
    for( ; sp>=1;sp--)       // afisare continut stiva
        printf("%d ",s[sp]);
}

```

De observat că valoarea constantei MARE, folosită pentru a marca în matricea de costuri absența unui arc, nu poate fi mai mare ca jumătate din valoarea maximă pentru tipul întreg, deoarece la însumarea costurilor a două drumuri se poate depăși cel mai mare întreg (se pot folosi pentru costuri și numere reale foarte mari).

Metoda de ajustare treptată a lungimii drumurilor din vectorul D este o metodă de "relaxare", folosită și în alți algoritmi pentru drumuri minime sau maxime: algoritmul Bellmann-Ford pentru drumuri minime cu o singură sursă în grafuri cu costuri negative, algoritmul Floyd pentru drumuri minime între oricare pereche de noduri s.a.

Prin relaxarea unei muchii (v,w) se înțelege ajustarea costului anterior al drumului către nodul w tinându-se seama și de costul muchiei v-w, deci considerând și un drum către w care trece prin v. Un pas de relaxare pentru drumul minim către nodul w se poate exprima prin secvența următoare:

```

// d[w] = cost drum minim la w fara a folosi si v
if (d[w] > d[v] + carc(g,v,w) ) { // dacă drumul prin v este mai scurt
    d[w]= d[v]+ carc(g,v,w);      // atunci se retine in d[w] acest cost
    p[w]=v;                      // si in p[w] nodul din care s-a ajuns la w
}

```

}

Deci luarea în considerare a muchiei v-w poate modifica sau nu costul stabilit anterior pentru a ajunge în nodul w, prin alte noduri decât v.

Complexitatea algoritmului Dijkstra este $O(n^2)$ și poate fi redusă la $O(m \lg(n))$ prin folosirea unei cozi ordonate (min-heap) cu operație de diminuare a cheii.

În coadă vom pune distanța cunoscută la un moment dat de la 1 până la un alt nod: inițial sunt costurile arcelor directe, după care se pun costurile drumurilor de la 1 prin nodurile determinate ca fiind cele mai apropiate de 1. Ideea cozii cu diminuarea priorității este că în coadă vor fi mereu aceleași elemente (noduri), dar cu priorități (distanțe) modificate de la un pas la altul. În loc să adăugăm la coadă distanțe tot mai mici (la aceleași noduri) vom modifica numai costul drumului deja memorat în coadă. Vom exemplifica cu graful orientat următor: 1-2=4, 1-3=1, 1-4=7, 2-4=1, 3-2=2, 3-4=5, 4-1=7

În coadă vom pune nodul destinație și distanța de la 1 la acel nod.

În cazul cozii cu priorități numai cu operații de adăugare și eliminare vom avea următoarea evoluție a cozii cu distanțe la noduri:

(3,1), (2,4), (4,7)	// costuri initiale (arce directe)
(2,3), (4,6), (2,4), (4,7)	// plus costuri drumuri prin 3
(2,4), (4,4), (4,7), (4,6)	// plus costuri drumuri prin 2
(4,6), (4,7)	// elemente ramase in coada

În cazul cozii cu diminuarea costului drumurilor (priorităților) coada va evolua astfel:

pas	coada ordonata	nod proxim (fata de 1)	distanța de la 1
initial	(3,1), (2,4), (4,7)	3	1
prin 3	(2,3), (4,6)	2	3
prin 3 și 2	(4,4)	4	4

Funcția următoare folosește operația de diminuare a priorității într-un min-heap și actualizează în coadă distanțele recalculat:

```

void dijkstra (Net g, int n[], int d[]) {
    // d[k] = distanța minimă de la 1 la nodul n[k]
    // pq= Coada cu distanțe minime de la 1 la alte noduri
    heap pq; dist min, a ;    // "dist" este o structura cu 2 intregi
    int i, nn;
    initpq(&pq);
    for (i=2; i<=g.n; i++) {    // pune in coada cost arce de la 1 la 2,3,...n
        a.n=i; a.d= cost(g,1,i); // numar nod si distanța in variabila a
        adpq( &pq, a);         // adauga a la coada pq
    }
    for (j=2; j<=g.n; j++) {    // repeta de n-1 ori
        min= delpq(&pq);        // scoate din coada nodul cel mai apropiat
        nn=min.n;               // numar nod proxim
        *d++=min.d;             // distanța de la 1 la nn
        *n++=nn;               // retine nn in vectorul n
        // ptr fiecare vecin al nodului nn
        for (i=2; i<=g.n; i++) {
            a.n=i; a.d=min.d+cost(g,nn,i); // recalculeaza distanța ptr fiecare nod i
            decrpq( &pq,a);
        }
    }
}
    
```

9.7 ARBORI DE ACOPERIRE DE COST MINIM

Un arbore de acoperire ("Spanning Tree") este un arbore liber ce contine o parte dintre arcele grafului cu care se acoperă toate nodurile grafului. Un arc "acoperă" nodurile pe care le uneste. Un graf conex are mai multi arbori de acoperire, numărul acestor arbori fiind cu atât mai mare cu cât numărul de cicluri din graful initial este mai mare. Pentru un graf conex cu n vârfuri, arborii de acoperire au exact $n-1$ muchii.

Problema este de a găsi pentru un graf dat arborele de acoperire cu cost total minim (MST=Minimum Spanning Tree) sau unul dintre ei, dacă sunt mai multi.

Exemplu: graful neorientat cu 6 noduri si următoarele arce si costuri:

(1,2)=6; (1,3)=1; (1,4)=5;
(2,3)=5; (2,5)=3;
(3,4)=5; (3,5)=6; (3,6)=4;
(4,6)=2;
(5,6)=6;

Arborele minim de acoperire este format din arcele: (1,3),(3,6),(6,4),(3,2),(2,5) si are costul total $1+5+3+4+2=15$.

Pentru determinarea unui arbore de acoperire de cost minim se cunosc doi algoritmi eficienti având ca autori pe Kruskal si Prim.

Algoritmul Kruskal foloseste o listă ordonată de arce (după costuri) si o colectie de multimi disjuncte pentru a verifica dacă următorul arc scos din listă poate fi sau nu adăugat arcelor deja selectate (dacă nu formează un ciclu cu arcele din MST).

Algoritmul lui Prim seamănă cu algoritmul Dijkstra pentru drumuri minime si foloseste o coadă cu priorități de arce care leagă vârfuri din MST cu alte vârfuri (coada se modifică pe măsură ce algoritmul evoluează).

Algoritmul lui Prim se bazează pe observatia următoare: fie S o submultime a vârfurilor grafului si R submultimea $V-S$ (vârfuri care nu sunt în S); muchia de cost minim care uneste vârfurile din S cu vârfurile din R face parte din MST.

Se poate folosi notiunea de "tăietură" în graf: se taie toate arcele care leagă un nod k de restul nodurilor din graf si se determină arcul de cost minim dintre arcele tăiate; acest arc va face parte din MST si va uni nodul k cu MST al grafului rămas după îndepărtarea nodului k . La fiecare pas se face o nouă tăietură în graful rămas si se determină un alt arc din MST; proces repetat de $n-1$ ori (sau până când S este vidă).

Fiecare tăietură în graf împarte multimea nodurilor din graf în două submultimi S (noduri incluse în MST) si R (restul nodurilor, încă acoperite cu arce). Initial $S=\{1\}$ dacă se porneste cu nodul 1, iar în final S va contine toate nodurile din graf. Tăieturile succesive pentru exemplul considerat sunt:

S (mst)	arce între S si R (arce tăiate)	minim	y
1	(1,2)=6; (1,3)=1; (1,4)=5;	(1,3)=1	3
1,3	(1,2)=6; (1,4)=5; (3,2)=5; (3,4)=5; (3,5)=6; (3,6)=4	(3,6)=4	6
1,3,6	(1,2)=6; (1,4)=5; (3,2)=5; (3,4)=5; (3,5)=6; (6,4)=2; (6,5)=6;	(6,4)=2	4
1,3,6,4	(1,2)=6; (3,2)=5; (3,5)=6; (6,5)=6	(3,2)=5	2
1,3,6,4,2	(2,5)=3; (3,5)=6; (6,5)=6	(2,5)=3	5

Solutia problemei este o multime de arce, deci un vector de perechi de noduri, sau doi vectori de întregi X si Y , cu semnificatia că o pereche $x[i]-y[i]$ reprezintă un arc din MST. Este posibilă si folosirea unui vector de întregi pentru arborele MST.

Algoritmul Prim este un algoritm greedy, la care lista de candidati este lista arcelor “tăiate”, deci arcele care unesc noduri din U cu noduri din V. La fiecare pas se alege arcul de cost minim dintre arcele tăiate și se generează o altă listă de candidati.

Vom prezenta două variante de implementare a acestui algoritm.

Prima variantă traduce fidel descrierea algoritmului folosind mulțimi, dar nu este foarte eficientă ca timp de execuție:

```
// algoritmul Prim cu rezultat in vectorii x si y
void prim ( Net g ) {          // g este o retea (cu costuri)
    Set s,r;
    int i,j;
    int cmin,imin,jmin;
    // initializare mulțimi de varfuri
    initS(s); initS(r);
    addS(s,1);                  // S={1}
    for (i=2;i<=g.n;i++)        // R={2,3,...}
        addS(r,i);
    // ciclul greedy
    while (! emptyS(s)) {
        cmin=MARE;
        //scaneaza toate muchiile
        for (i=1;i<=g.n;i++)
            for (j=1;j<=g.n;j++) {
                if (findS(s,i) && findS(s,j) || // daca i si j in aceeasi multime s
                    findS(r,i) && findS(r,j)) // sau in r
                    continue;                // atunci se ignora muchia (i-j)
                if (carc(g,i,j) < cmin) {      // determina muchia de cost minim
                    cmin=carc(g,i,j);
                    imin=i; jmin=j;           // muchia (imin,jmin) are cost minim
                }
            }
        printf ("%d-%d \n",imin,jmin);      // afisare extremitati muchie
        addS(s,imin); addS(s,jmin);         // adauga varfuri la s
        delS(r,imin); delS(r,jmin);         // elimina varfuri din r
    }
}
```

Programul următor, mai eficient, folosește doi vectori:

p [i] = numărul nodului din S cel mai apropiat de nodul i din R
c [i] = costul arcului dintre i și p[i]

La fiecare pas se caută în vectorul “c” pentru a găsi nodul k din R cel mai apropiat de nodul i din S. Pentru a nu mai folosi o mulțime S, se atribuie lui c[k] o valoare foarte mare astfel ca nodul k să nu mai fie luat în considerare în pașii următori.

Mulțimea S este deci implicit mulțimea nodurilor i cu c[i] foarte mare. Celelalte noduri formează mulțimea R.

```
# define M 20          // nr maxim de noduri
# define M1 10000      // un nr. foarte mare (cost arc absent)
# define M2 (M1+1)     // alt numar foarte mare (cost arc folosit)
// alg. Prim pentru arbore minim de acoperire
void prim (Net g, int x[ ], int y[ ]){
    int c[M], cmin;
    int p[M], i,j,k;
    int n=g.n;          // n = nr de varfuri
    for(i=2;i<=n;i++) {
        p[i]=1; c[i]=carc (g,1,i); // costuri initiale
    }
    for(i=2;i<=n;i++) {
```

```

- // cauta nodul k cel mai apropiat de un nod din mst
  cmin = c[2]; k=2;
  for(j=2;j<=n;j++)
    if ( c[j] < cmin) {
      cmin=c[j]; k=j;
    }
  x[i-1]=p[k]; y[i-1]= k; // retine muchie de cost minim in x si y
  c[k]=M2;
  // ajustare costuri in U
  for(j=2;j<=n;j++)
    if (carc(g,k,j) < c[j] && c[j] < M2) {
      c[j]= carc(g,k,j); p[j] =k;
    }
}
}

```

Evolutia vectorilor “c” si “p” pentru exemplul dat este următoarea:

c[2]	p[2]	c[3]	p[3]	c[4]	p[4]	c[5]	p[5]	c[6]	p[6]	k
6	1	1	1	5	1	M1	1	M1	1	3
5	3	M2	1	5	1	6	3	4	3	6
5	3	M2	1	2	6	6	3	M2	3	4
5	3	M2	1	M2	6	6	3	M2	3	2
M2	3	M2	1	M2	6	3	2	M2	3	5
M2	3	M2	1	M2	6	M2	2	M2	3	

Au fost necesare două constante mari: M1 arată că nu există un arc între două noduri, iar M2 arată că acel arc a fost inclus în MST si că va fi ignorat în continuare.

Vectorul “p” folosit în programul anterior corespunde reprezentării unui arbore printr-un singur vector, de predecesori.

Complexitatea algoritmului Prim cu vectori este $O(n*n)$, dar poate fi redusă la $O(m*\lg(n))$ prin folosirea unui heap pentru memorarea costurilor arcelor dintre U si V

Ideea algoritmului Kruskal este de a alege la fiecare pas arcul de cost minim dintre cele rămase (încă neselectate), dacă el nu formează ciclul cu arcele deja incluse în MST (selectate). Condiția ca un arc (x,y) să nu formeze ciclul cu celelalte arce selectate se poate exprima astfel: nodurile x si y trebuie să se afle în componente conexe diferite. Inițial fiecare nod formează o componentă conexă, iar apoi o componentă conexă conține toate nodurile acoperite cu arce din MST, iar nodurile neacoperite formează alte componente conexe.

Algoritmul Kruskal pentru găsirea unui arbore de acoperire de cost minim folosește două tipuri abstracte de date: o coadă cu priorități si o colecție de mulțimi disjuncte si poate fi descris astfel :

```

citire date si creare coada de arce
repetă {
  extrage arcul de cost minim din coada
  dacă arc acceptabil atunci {
    afisare arc
    actualizare componente conexe
  }
} până când toate nodurile conectate

```

Un arc care leagă două noduri dintr-o aceeași componentă conexă va forma un ciclu cu arcele selectate anterior si nu poate fi acceptat. Va fi acceptat numai un arc care leagă între ele noduri aflate în două componente conexe diferite.

Pentru rețeaua cu 6 noduri si 10 arce (1,2)=6; (1,3)=1; (1,4)=5; (2,3)=5; (2,5)=3; (3,4)=5; (3,5)=6; (3,6)=4; (4,6)=2; (5,6)=6

evolutia algoritmului Kruskal este următoarea :

Pas	Arc (Cost)	Acceptabil	Cost total	Afisare
1	1,3 (1)	da	1	1 - 3
2	4,6 (2)	da	3	4 - 6
3	2,5 (3)	da	6	2 - 5
4	3,6 (4)	da	10	3 - 6
5	1,4 (5)	nu	10	
6	3,4 (5)	nu	10	
7	2,3 (5)	da	15	2 - 3

Toate nodurile din graf trebuie să se afle în componentele conexe. Initial sunt atâtea componente (multimi) câte noduri există. Atunci când un arc este acceptat, se reunesc cele două multimi (componente) care contin extremitățile arcului în una singura; în felul acesta numărul de componente conexe se reduce treptat până când ajunge egal cu 1 (toate nodurile legate într-un graf conex care este chiar arborele de acoperire cu cost minim).

Evolutia componentelor conexe pentru exemplul anterior :

Pas	Componente conexe
1	{1}, {2}, {3}, {4}, {5}, {6}
2	{1,3}, {2}, {4}, {5}, {6}
3	{1,3}, {2,5}, {4,6}
4	{1,3,4,6}, {2,5}
7	{1,2,3,4,5,6}

In programul următor graful este un vector de arce, ordonat crescător după costuri înainte de a fi folosit. Exemplu:

```
typedef struct { int v,w,cost ;} Arc;
// compara arce dupa cost (ptr qsort)
int cmparc (const void * p, const void* q) {
    Arc * pp =(Arc*) p; Arc *qq=(Arc*) q;
    return pp->cost -qq->cost;
}
// algoritmul Kruskal
void main ( ) {
    DS ds; Arc arce[M], a;
    int x,y,n,na,mx,my,nm,k;
    printf ("nr.noduri în graf: "); scanf ("%d", &n);
    initDS (ds,n); // ds = colectie de multimi disjuncte
    printf ("Lista de arce cu costuri: \n");
    nm=0; // nr de muchii in graf
    while ( scanf ("%d%d%d",&a.v,&a.w,&a.cost) > 0)
        arce[nm++]=a;
    qsort (arce, nm, sizeof(Arc), cmparc); // ordonare lista arce
    k=0; // nr arc extras din coada
    for ( na=n-1; na > 0; na--) {
        a=arce[k++]; // următorul arc de cost minim
        x=a.v; y=a.w; // x, y = extremitati arc
        mx= findDS (ds,x); my=findDS (ds,y);
        if (mx !=my ) { // daca x si y in componente conexe diferite
            unifDS (ds,x,y); // atunci se reunesc cele doua componente
            printf ("%d - %d \n",x,y); // si se scrie arcul gasit ptr mst
        }
    }
}
```

Complexitatea algoritmului Kruskal depinde de modul de implementare al colecției de mulțimi disjuncte și este în cel mai bun caz $O(m \cdot \lg(n))$ pentru o implementare eficientă a tipului DS (este practic timpul de ordonare a listei de arce).

9.8 GRAFURI VIRTUALE

Un graf virtual este un model abstract pentru un algoritm, fără ca graful să existe efectiv în memorie. Fiecare nod din graf reprezintă o “stare” în care se află programul iar arcele modelează trecerea dintr-o stare în alta (nu orice tranziție între stări este posibilă și graful nu este complet). Vom menționa două categorii de algoritmi de acest tip: algoritmi ce modelează automate (mașini) cu număr finit de stări (“Finite State Machine”) și algoritmi de optimizare discretă (“backtracking” și alte metode).

Un algoritm de tip “automat finit” trece dintr-o stare în alta ca urmare a intrărilor furnizate algoritmului, deci prin citirea succesivă a unor date. Exemple sunt programe de recepție a unor mesaje conforme unui anumit protocol de comunicație, programe de prelucrare expresii regulate, interpretoare și compilatoare ale unor limbaje.

Un analizor sintactic (“parser”) poate avea drept stări: “într-un comentariu” și “în afara unui comentariu”, “într-o constantă sir” și “în afara unei constante sir”, “într-un bloc de instrucțiuni și declarații” sau “terminare bloc”, s.a.m.d. Un analizor pentru limbajul C, de exemplu, trebuie să deosebească caractere de comentariu care sunt în cadrul unui sir (încadrat de ghilimele) sau caractere ghilimele într-un comentariu, sau comentarii C++ într-un comentariu C, etc.

Ca exemplu vom prezenta un tabel cu tranzițiile între stările unui parser interesat de recunoașterea comentariilor C sau C++:

Stare curentă	Caracter citit	Starea următoare
între comentarii	/	posibil început de comentariu
posibil început de comentariu	/	în comentariu C++
posibil început de comentariu	*	în comentariu C
posibil început de comentariu	alte caractere	între comentarii
în comentariu C++	\n	între comentarii
în comentariu C++	alte caractere	în comentariu C++
în comentariu C	*	posibil sfârșit comentariu
posibil sfârșit comentariu	/	între comentarii
posibil sfârșit comentariu	alte caractere	în comentariu C

Stările pot fi codificate prin numere întregi iar programul conține un bloc *switch* cu câte un caz (*case*) pentru fiecare stare posibilă.

O problemă de optimizare discretă poate avea mai multe soluții vectoriale (sau matriciale) și fiecare soluție are un cost asociat; scopul este găsirea unei soluții optime pentru care funcția de cost este minimă sau maximă.

O serie de algoritmi de optimizare realizează o căutare într-un graf, numit și spațiu al stărilor. Acest graf este construit pe măsură ce algoritmul progresează și nu este memorat integral, având în general un număr foarte mare de noduri (stări). Graful este de obicei orientat și arcele au asociate costuri.

O soluție a problemei este o cale în graful stărilor iar costul soluției este suma costurilor arcelor ce compun calea respectivă.

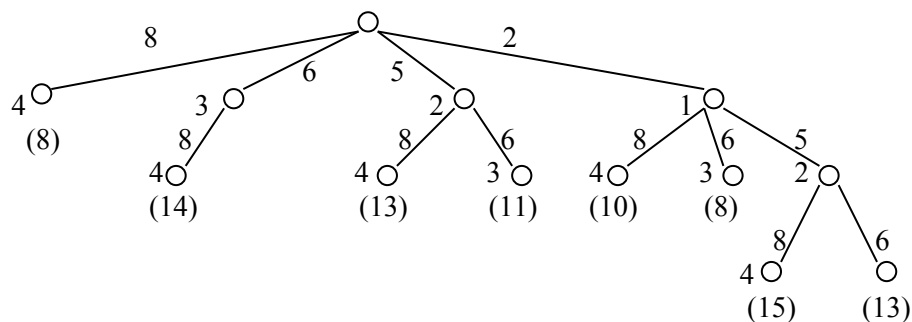
Vom considera două exemple clasice: problema rucsacului și ieșirea din labirint.

Problema rucsacului are ca date n obiecte de greutate $g[k]$ și valoare $v[k]$ fiecare și un sac de capacitate t , iar cerința este să selectăm acele obiecte cu greutate totală mai mică sau egală cu t pentru care valoarea obiectelor selectate este maximă. Soluția este fie un vector x cu valori 1 sau 0 după cum obiectul respectiv a fost sau nu selectat, fie un vector x cu numerele obiectelor din selecția optimă (din rucsac).

Fie cazul concret în care sacul are capacitatea $t=15$ și există 4 obiecte de greutate $g[] = \{8, 6, 5, 2\}$ și valori unitare egale. Soluția optimă (cu valoare maximă) este cea care folosește obiectele 1, 3 și 4.

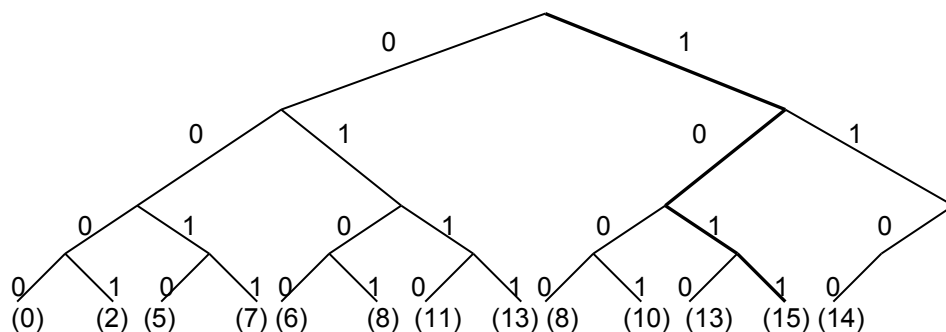
În varianta binară vectorul soluție este $x[] = \{1, 0, 1, 1\}$, iar în varianta cu numere de obiecte soluția este $x[] = \{1, 2, 4\}$.

Spațiul stărilor pentru varianta cu $x[k]$ egal cu numărul obiectului ales în pasul k și după eliminarea soluțiilor echivalente:



Arcele arborelui de stări au drept costuri greutatea (valorile) obiectelor, iar la capătul fiecărei ramuri este notată greutatea selecției respective (deci costul soluției). Soluțiile optime sunt două: una care folosește două obiecte cu greutatea 6 și 8 și alta care folosește 3 obiecte cu greutatea 2, 4 și 8.

Spațiul stărilor în varianta binară este un arbore binar a cărui înălțime este egală cu numărul de obiecte. Alternativele de pe nivelul k sunt includerea în soluție sau nu a obiectului k . La fiecare cale posibilă este trecut costul însumat al arcelor folosite (valorile obiectelor selectate).



Problema ieșirii din labirint ilustrează o problemă la care spațiul stărilor nu mai este un arbore ci un graf care poate conține și cicluri.

Un labirint este reprezentat printr-o matrice L de carouri cu m linii și n coloane cu convenția că $L[i][j]=1$ dacă caroul din linia i și coloana j este liber (poate fi folosit pentru deplasare) și $L[i][j]=0$ dacă caroul (i,j) este ocupat (de ziduri despărțitoare).

Pornind dintr-un carou dat (liber) se cere drumul minim de ieșire din labirint sau toate drumurile posibile de ieșire din labirint, cu condiția ca un drum să nu treacă de mai multe ori prin același carou (pentru a evita deplasarea în cerc închis, la infinit). Ieșirea din labirint poate însemna că se ajunge la orice margine sau la un carou dat.

Fie un exemplu de labirint cu 4 linii și 4 coloane și punctul de plecare $(2,2)$.

Graful de stări se poate reduce ca dimensiune dacă impunem și alte condiții în funcția “posibil”. Pentru probleme de minimizare putem compara costul soluției parțiale în curs de generare (costul unei căi incomplete) cu un cost minim de referință și să oprim expandarea căii dacă acest cost este mai mare decât costul minim stabilit până la acel moment. Ideea se poate folosi în problema ieșirii din labirint: dacă suntem pe o cale incompletă egală cu o cale completă anterioară nu are rost să mai continuăm pe calea respectivă.

Se poate spune că diferențele dintre diferiți algoritmi de optimizare discretă provin din modul de expandare a grafului de stări, pentru minimizarea acestuia.

Căutarea în adâncime în graful de stări implicit (metoda “backtracking”) se poate exprima recursiv sau iterativ, folosind o stivă.

Pentru concretizare vom folosi problema umplerii prin inundare a unei suprafețe delimitate de un contur oarecare, problemă care seamănă cu problema labirintului dar este ceva mai simplă. Problema permite vizualizarea diferenței dintre explorarea în adâncime și explorarea în lățime, prin afișarea suprafeței de colorat după fiecare pas.

Datele problemei se reprezintă printr-o matrice pătratică de caractere inițializată cu caracterul punct ‘.’, iar punctele colorate vor fi marcate prin caracterul ‘#’. Se dă un punct interior din care începe colorarea (umplerea) spre punctele vecine.

Evoluția unei matrice de 5x5 la explorare în lățime a spațiului stărilor, plecând din punctul (2,2), cu extindere în ordinea sus, dreapta, jos, stânga (primele imagini):

```
. . . . . . . . . . . . . . . . . . . . . # . . . . # . .
. . . . . . # . . . . . # . . . . # . . . . # . . . . # # .
. . # . . . . # . . . . # # . . . . # # . . . . # # # . .
. . . . . . . . . . . . . . . # . . . . # . . . . # . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . # . . . . # . . . . # . . . . # . . . . # # . . # # # .
. # # # . . # # # . . # # # . . # # # . . # # # . . # # # .
. # # # . . # # # . . # # # . . # # # . . # # # . . # # # .
. . # . . . . # . . . . # # . . . . # # . . . . # # # . .
. . . . . . . . . . . . . . . # . . . . # . . . . # . .
```

Evoluția matricei 5x5 la explorare în adâncime a spațiului stărilor, cu ordine de extindere inversă (punctele se scot din stivă în ordine inversă introducerii în stivă) :

```
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . .
. . # . . . . # # . . # # # . . # # # . . # # # . . # # # .
. . . . . . . . . . . . . . . # . . . . # . . . . # . . . .
. . . . . . . . . . . . . . . # . . . . # # . . . . # # # .

. . . . . . . . . . . . . . . . . . . . . . . . . . .
# # # . . . # # # . . # # # . . # # # . . # # # . . # # # .
# . . . . # . . . . # # . . # # # . # # # # # # # # # # #
# # # # . # # # # # # # # # # # # # # # # # # # # # #
```

În coadă sau în stivă vom pune adrese de puncte (adrese de structuri):

```
typedef struct { int x,y; } point;
// construiește o variabilă “point”
point * make (int i, int j) {
    point *p= new point;
    p->x=i;p->y=j;
    return p;
}
// Funcție de umplere prin explorare în lățime cu coadă :
void fillQ (int i, int j) { // colorarea porneste din punctul (i,j)
    int k,im,jm; Queue q; // q este o coada de pointeri void*
    point *p; // point este o pereche (x,y)
```

```

int dx[4] = {-1,0,1,0}; // directii de extindere pe orizontala
int dy[4] = {0,1,0,-1}; // directii de extindere pe verticala
initQ(q); // initializare coada
p=make(i,j); // adresa punct initial (i,j)
addQ(q,p); // adauga adresa la coada
while ( ! emptyQ(q) ) { // repeta cat timp e ceva in coada
    p= (point*)delQ(q); // scoate adresa punct curent din coada
    i=p->x; j=p->y; // coordonate celula curenta
    a[i][j]='#'; // coloreaza celula (i,j)
    for (k=0;k<4;k++) { // extindere in cele 4 directii
        im=i+dx[k]; jm= j+dy[k]; // coordonate punct vecin cu punctul curent
        if ( ! posibil(im,jm)) // daca punct exterior sau colorat
            continue; // nu se pune in coada
        p=make(im,jm); // adresa punct vecin cu (i,j)
        addQ (q,p); // adauga adresa punct vecin la coada q
    }
}
}
}

```

Funcția “posibil” verifică dacă punctul primit este interior conturului și nu este colorat (are culoarea inițială):

```

int posibil (int i,int j) {
    if ( i<0 || i>n-1 || j<0 || j>n-1 ) // daca punct exterior
        return 0;
    return a[i][j]!='.' ; // matricea a initializata cu caracterul '.'
}

```

Funcția de umplere cu exploare în adâncime este la fel cu cea anterioară, dar folosește o stivă de pointeri în locul cozii. Funcția anterioară conduce la o creștere rapidă a lungimii cozii (stivei), deoarece același punct necolorat este pus în listă de mai multe ori, ca vecin al punctelor adiacente cu el. Pentru a reduce lungimea listei se poate “colora” fiecare punct pus în listă cu o culoare intermediară (diferită de culoarea inițială dar și de cea finală).

Varianța recursivă de explorare în adâncime este:

```

void fill (int i,int j ) { // colorare din punctul (i,j)
    int k,im,jm;
    for (k=0;k<4;k++) { // pentru fiecare vecin posibil
        im=i+dx[k]; jm= j+dy[k]; // (im,jm) este un punct vecin cu (i,j)
        if (posibil(im,jm)) {
            a[im][jm]='#';
            fill (im,jm); // continua colorarea din punctul (im,jm)
        }
    }
}
}

```

Pentru cele mai multe probleme metoda “backtracking” face o căutare în adâncime într-un arbore (binar sau multicăi). Varianta cu arbore binar (și vector soluție binar) are câteva avantaje: programe mai simple, toate soluțiile au aceeași lungime și nu pot apărea soluții echivalente (care diferă numai prin ordinea valorilor).

Exemplu de funcție recursivă pentru algoritmul “backtracking” (soluții binare):

```

void bkt (int k) { // k este nivelul din arborele binar (maxim n)
    int i;
    if (k > n) { // daca s-a ajuns la un nod terminal
        print (); // scrie vector solutie x (cu n valori binare)
        return; // si revine la apelul anterior (continua cautarea)
    }
}

```

```
x[k]=1;           // incerca la stanga (cu valoarea 1 pe nivelul k)
if (posibil(k))    // daca e posibila o solutie cu x[k]=1
    bkt(k+1);      // cauta in subarborele stanga
x[k]=0;           // incerca la dreapta (cu valoarea 0 pe niv. k)
bkt(k+1);         // cauta in subarborele dreapta
}
```

Pentru problema rucsacului $x[k]=1$ semnifică prezenta obiectului k în sac (într-o soluție) iar $x[k]=0$ semnifică absenta obiectului k dintr-o soluție. Pentru valoarea $x[k]=0$ nu am verificat dacă soluția este acceptabilă, considerând că neîncluderea unor obiecte în selecția optimă este posibilă întotdeauna.

Ordinea explorării celor doi subarbori poate fi modificată, dar am preferat să obținem mai întâi soluții cu mai multe obiecte și valoare mai mare.

Funcția “print” fie afișează o soluție obținută (în problemele de enumerare a tuturor soluțiilor posibile), fie compară costul soluției obținute cu cel mai bun cost anterior (în probleme de optimizare, care cer o soluție de cost minim sau maxim).

Capitolul 10

STRUCTURI DE DATE EXTERNE

10.1 SPECIFICUL DATELOR PE SUPORT EXTERN

Principala diferență dintre memoria internă (RAM) și memoria externă (disc) este modul și timpul de acces la date:

- Pentru acces la disc timpul este mult mai mare decât timpul de acces la RAM (cu câteva ordine de mărime), dar printr-un singur acces se poate citi un număr mare de octeți (un multiplu al dimensiunii unui sector sau bloc disc);
- Datele memorate pe disc nu pot folosi pointeri, dar pot folosi adrese relative în fisier (numere întregi lungi). Totuși, nu se folosesc date dispersate într-un fisier din cauza timpului foarte mare de repositionare pe diferite sectoare din fisier.

În consecință apar următoarele recomandări:

- Utilizarea de zone tampon ("buffer") mari, care să corespundă unui număr oarecare de sectoare disc, pentru reducerea numărului de operații cu discul (citire sau scriere);
- Gruparea fizică pe disc a datelor între care există legături logice și care vor fi prelucrate foarte probabil împreună; vom numi aceste grupări "blocuri" de date ("cluster" sau "bucket").
- Amânarea modificărilor de articole, prin marcarea articolelor ca sterse și rescrierea periodică a întregului fisier (în loc de a șterge fizic fiecare articol) și colectarea articolelor care trebuie inserate, în loc de a insera imediat și individual fiecare articol.

Un exemplu de adaptare la specificul memoriei externe este chiar modul în care un fisier este creat sau extins pe mai multe sectoare neadiacente. În principiu se folosește ideea listelor înlănțuite, care cresc prin alocarea și adăugarea de noi elemente la listă. Practic, nu se folosesc pointeri pentru legarea sectoarelor disc neadiacente dar care fac parte din același fisier; fiecare nume de fisier are asociată o listă de sectoare disc care aparțin fisierului respectiv (ca un vector de pointeri către aceste sectoare). Detaliile sunt mai complicate și depind de sistemul de operare.

Un alt exemplu de adaptare la specificul memoriei externe îl constituie structurile arborescente: dacă un sector disc ar conține mai multe noduri oarecare dintr-un arbore binar, atunci ar fi necesar un număr mare de sectoare citite (și recitite) pentru a parcurge un arbore. Pentru exemplificare să considerăm un arbore binar de căutare cu 4 noduri pe sector, în care ordinea de adăugare a cheilor a condus la următorul conținut al sectoarelor disc (numerotate 1,2,3,4) :

50, 30, 40, 20 70, 80, 35, 85 60, 55, 35, 25 65, 75, -, -

Pentru căutarea valorii 68 în acest arbore ar fi necesară citirea următoarelor sectoare, în această ordine:

1 (rădăcina 50), 2 (compară cu 70), 3 (compară cu 60), 4 (compară cu 65)

Soluția găsită a fost o structură arborescentă mai potrivită pentru discuri, în care un sector (sau mai multe) conține un nod de arbore, iar arborele nu este binar pentru a reduce numărul de noduri și înălțimea arborelui; acești arbori se numesc arbori B.

Structurile de date din memoria RAM care folosesc pointeri vor fi salvate pe disc sub o altă formă, fără pointeri; operația se numește și "serializare". Serializarea datelor dintr-un arbore, de exemplu, se va face prin traversarea arborelui de la rădăcină către frunze (în preordine, de obicei), astfel ca să fie posibilă reconstituirea legăturilor dintre noduri la o încărcare ulterioară a datelor în memorie.

Serializarea datelor dintr-o foaie de calcul ("spreadsheet") se va face scriind pe disc coordonatele (linie, coloană) și conținutul celulelor, deși în memorie foaia de calcul se reprezintă ca o matrice de pointeri către conținutul celulelor.

De multe ori datele memorate permanent pe suport extern au un volum foarte mare ceea ce face imposibilă memorarea lor simultană în memoria RAM. Acest fapt are consecințe asupra algoritmilor de sortare externă și asupra modalităților de căutare rapidă în date pe suport extern.

Sistemele de operare actuale (MS-Windows si Linux) folosesc o memorie virtuală, mai mare decât memoria fizică RAM, dar totusi limitată. Memoria virtuală înseamnă extinderea automată a memoriei RAM pe disc (un fisier sau o partitie de “swap”), dar timpul de acces la memoria extinsă este mult mai mare decât timpul de acces la memoria fizică si poate conduce la degradarea performantelor unor aplicatii cu structuri de date voluminoase, aparent păstrate în memoria RAM.

10.2 SORTARE EXTERNĂ

Sortarea externă, adică sortarea unor fisiere mari care nu încap în memoria RAM sau în memoria virtuală pusă la dispozitie de către sistemul gazdă, este o sortare prin interclasare: se sortează intern secvente de articole din fisier si se interclasează succesiv aceste secvente ordonate de articole.

Există variante multiple ale sortării externe prin interclasare care diferă prin numărul, continutul si modul de folosire al fisierelor, prin numărul fisierelor create.

O secvență ordonată de articole se numeste si “monotonie” (“run”). Faza inițială a procesului de sortare externă este crearea de monotonii. Un fisier poate contine o singură monotonie sau mai multe monotonii (pentru a reduce numărul fisierelor temporare). Interclasarea poate folosi numai două monotonii (“2-way merge”) sau un număr mai mare de monotonii (“multiway merge”). După fiecare pas se reduce numărul de monotonii, dar creste lungimea fiecărei monotonii (fisierul ordonat contine o singură monotonie).

Crearea monotoniilor se poate face prin citirea unui număr de articole succesive din fisierul initial, sortarea lor (prin metoda quicksort) si scrierea secventei ordonate ca o monotonie în fisierul de iesire. Pentru exemplificare să considerăm un fisier ce contine articole cu următoarele chei (în această ordine):

7, 6, 4, 8, 3, 5

Să considerăm că se foloseste un buffer de două articole (în practică sunt zeci, sute sau mii de articole într-o zonă tampon). Procesul de creare a monotoniilor:

Input	Buffer	Output
7,6,4,8,3,5	7,6	6,7
4,8,3,5	4,8	6,7 4,8
3,5	3,5	6,7 4,8 3,5

Crearea unor monotonii mai lungi (cu aceeasi zonă tampon) se poate face prin metoda selectiei cu înlocuire (“replacement selection”) astfel:

- Se alege din buffer articolul cu cea mai mică cheie care este mai mare decât cheia ultimului articol scris în fisier.
- Dacă nu mai există o astfel de cheie atunci se termină o monotonie si începe o alta cu cheia minimă din buffer.
- Se scrie în fisierul de iesire articolul cu cheia minimă si se citeste următorul articol din fisierul de intrare.

Pentru exemplul anterior, metoda va crea două monotonii mai lungi:

Input	Buffer	Output
7,6,4,8,3,5	7,6	6
4,8,3,5	7,4	6,7
8,3,5	4,8	6,7,8
3,5	4,3	6,7,8 3
5	4,5	6,7,8 3,4
-	5	6,7,8 3,4,5

Pentru reducerea timpului de sortare se folosesc zone buffer cât mai mari, atât la citire cât si pentru scriere în fisiere. In loc să se citească câte un articol din fiecare monotonie, se va citi câte un grup de articole din fiecare monotonie, ceea ce va reduce numărul operatiilor de citire de pe disc (din fisiere diferite, deci cu deplasarea capetelor de acces între piste disc).

Detaliile acestui proces pot fi destul de complexe, pentru a obtine performante cât mai bune.

10.3 INDEXAREA DATELOR

Datele ce trebuie memorate permanent se păstrează în fișiere și baze de date. O bază de date reunește mai multe fișiere necesare unei aplicații, împreună cu metadate ce descriu formatul datelor (tipul și lungimea fiecărui câmp) și cu fișiere index folosite pentru accesul rapid la datele aplicației, după diferite chei.

Modelul principal utilizat pentru baze de date este modelul relational, care grupează datele în tabele, legăturile dintre tabele fiind realizate printr-o coloană comună și nu prin adrese disc. Relațiile dintre tabele pot fi de forma 1 la 1, 1 la n sau m la n (“one-to-one”, “one-to-many”, “many-to-many”). În cadrul modelului relational există o diversitate de soluții de organizare fizică a datelor, care să asigure un timp bun de interogare (de regăsire) după diverse criterii, dar și modificarea datelor, fără degradarea performanțelor la căutare.

Cea mai simplă organizare fizică a unei baze de date (în dBASE și FoxPro) face din fiecare tabel este un fișier secvențial, cu articole de lungime fixă, iar pentru acces rapid se folosesc fișiere index. Metadatele ce descriu fiecare tabel (numele, tipul, lungimea și alte atribute ale coloanelor din tabel) se află chiar la începutul fișierului care conține și datele din tabel. Printre aceste metadate se poate afla și numele fișierului index asociat fiecărei coloane din tabel (dacă a fost creat un fișier index pentru acea coloană).

Organizarea datelor pe disc pentru reducerea timpului de căutare este și mai importantă decât pentru colecții de date din memoria internă, datorită timpului mare de acces la discuri (față de memoria RAM) și a volumului mare de date. În principiu există două metode de acces rapid după o cheie (după conținut):

- Calculul unei adrese în funcție de cheie, ca la un tabel “hash”;
- Crearea și menținerea unui tabel index, care reunește cheile și adresele articolelor din fișierul de date indexat.

Prima metodă poate asigura cel mai bun timp de regăsire, dar numai pentru o singură cheie și fără a menține ordinea cheilor (la fel ca la tabele “hash”).

Atunci când este necesară căutarea după chei diferite (câmpuri de articole) și când se cere o imagine ordonată după o anumită cheie a fișierului principal se folosesc tabele index, câte unul pentru fiecare câmp cheie (cheie de căutare și/sau de ordonare). Aceste tabele index sunt realizate de obicei ca fișiere separate de fișierul principal, ordonate după chei.

Un index conține perechi cheie-adresă, unde “adresă” este adresa relativă în fișierul de date a articolului ce conține cheia. Ordinea cheilor din index este în general alta decât ordinea articolelor din fișierul indexat; în fișierul principal ordinea este cea în care au fost adăugate articolele la fișier (mereu la sfârșit de fișier), iar în index este ordinea valorilor cheilor.

Id	Adr	Id	Nume	Marca	Pret	...
20		50	aaaaa	AAA	100	
30		20	dddd	DDD	450	
50		90	vvvv	VVV	130	
70		30	cccc	CCC	200	
90		70	bbbb	BBB	330	

Fișierul index este întotdeauna mai mic decât fișierul indexat, deoarece conține doar un singur câmp din fiecare articol al fișierului principal. Timpul de căutare va fi deci mai mic în fișierul index decât în fișierul principal, chiar dacă indexul nu este ordonat sau este organizat secvențial. De obicei fișierul index este ordonat și este organizat astfel ca să permită reducerea timpului de căutare, dar și a timpului necesar actualizării indexului, la modificări în fișierul principal.

Indexul dat ca exemplu este un index “dens”, care conține câte un articol pentru fiecare articol din fișierul indexat. Un index “rar”, cu mai puține articole decât în fișierul indexat, poate fi folosit atunci când fișierul principal este ordonat după cheia conținută de index (situația când fișierul principal este relativ stabil, cu puține și rare modificări de articole).

Orice acces la fisierul principal se face prin intermediul fisierului index "activ" la un moment dat si permite o imagine ordonată a fisierului principal (de exemplu, afisarea articolelor fisierului principal în ordinea din fisierul index).

Mai mult, fisierele index permit selectarea rapidă de coloane din fisierul principal si "imagini" ("views") diferite asupra unor fisiere fizice; de exemplu, putem grupa coloane din fisiere diferite si în orice ordine, folosind fisierele index. Astfel se creează aparenta unor noi fisiere, derivate din cele existente, fără crearea lor efectivă ca fisiere fizice.

Un index dens este de fapt un dictionar în care valorile asociate cheilor sunt adresele articolelor cu cheile respective în fisierul indexat, dar un dictionar memorat pe un suport extern. De aceea, solutiile de implementare eficiente a dictionarelor ordonate au fost adaptate pentru fisiere index: arbori binari de căutare echilibrati (în diferite variante, inclusiv "treap") si liste skip.

Adaptarea la suport extern înseamnă în principal că un nod din arbore (sau din listă) nu contine o singură cheie ci un grup de chei. Mai exact, fiecare nod contine un vector de chei de capacitate fixă, care poate fi completat mai mult sau mai puțin. La depășirea capacității unui nod se creează un nou nod.

Cea mai folosită solutie pentru fisiere index o constituie arborii B, în diferite variante (B+, B*).

10.4 ARBORI B

Un arbore B este un arbore de căutare multicăi echilibrat, adaptat memoriilor externe cu acces direct. Un arbore B de ordinul n are următoarele proprietăți:

- Rădăcina fie nu are succesori, fie are cel puțin doi succesori.
- Fiecare nod interior (altele decât rădăcina si frunzele) au între $n/2$ si n succesori.
- Toate căile de la rădăcină la frunze au aceeași lungime.

Fiecare nod ocupă un articol disc (preferabil un multiplu de sectoare disc) si este citit integral în memorie. Sunt posibile două variante pentru nodurile unui arbore B:

- Nodurile interne contin doar chei si pointeri la alte noduri, iar datele asociate fiecărei chei sunt memorate în frunze.
- Toate nodurile au aceeași structură, continând atât chei cât si date asociate cheilor.

Fiecare nod (intern) contine o secvență de chei si adrese ale fiilor de forma următoare:

$$p[0], k[1], p[1], k[2], p[2], \dots, k[m], p[m] \quad (n/2 \leq m \leq n)$$

unde $k[1] < k[2] < \dots < k[m]$ sunt chei, iar $p[0], p[1], \dots, p[m]$ sunt legături către nodurile fii (pseudo-pointeri, pentru că sunt adrese de octet în cadrul unui fisier disc).

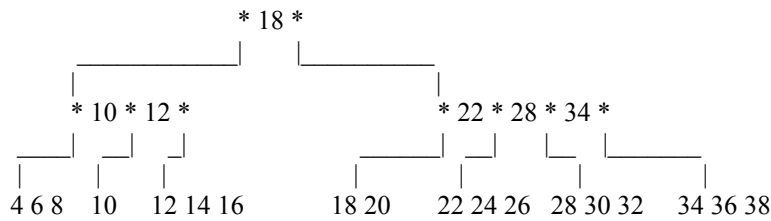
În practică, un nod contine zeci sau sute de chei si adrese, iar înălțimea arborelui este foarte mică (rareori peste 3). Pentru un arbore cu un milion de chei si maxim 100 de chei pe nod sunt necesare numai 3 operatii de citire de pe disc pentru localizarea unei chei, dacă toate nodurile contin numărul maxim de chei: radacina are 100 de fii pe nivelul 1, iar fiecare nod de pe nivelul 1 are 100 de fii pe nivelul 2, care contin câte 100 de chei fiecare.

Toate cheile din subarborele cu adresa $p[0]$ sunt mai mici decât $k[1]$, toate cheile din subarborele cu adresa $p[m]$ sunt mai mari decât $k[m]$, iar pentru orice $1 \leq i < n$ cheile din subarborele cu adresa $p[i]$ sunt mai mari sau egale cu $k[i]$ si mai mici decât $k[i+1]$.

Arborii B pot fi priviti ca o generalizare a arborilor 2-3-4 si cunosc mai multe variante de implementare:

- Date si în nodurile interioare (B) sau date numai în noduri de pe ultimul nivel (B*).
- Numărul minim de chei pe nod si strategia de spargere a nodurilor (respectiv de contopire noduri cu acelasi părinte): $n/2$ sau $2n/3$ sau $3n/4$.

Exemplu de arbore B de ordinul 4 (asteriscurile reprezintă adrese de blocuri disc):



În desenul anterior nu apar și datele asociate cheilor.

Arborii B au două utilizări principale:

- Pentru dicționare cu număr foarte mare de chei, care nu pot fi păstrate integral în memoria RAM;
- Pentru fișiere index asociate unor fișiere foarte mari (din baze de date), caz în care datele asociate cheilor sunt adrese disc din fișierul mare indexat.

Cu cât ordinul unui arbore B (numărul maxim de succesori la fiecare nod) este mai mare, cu atât este mai mică înălțimea arborelui și deci timpul mediu de căutare.

Căutarea unei chei date într-un arbore B seamănă cu căutarea într-un arbore binar de căutare BST, dar arborele este multicăi și are nodurile pe disc și nu în memorie.

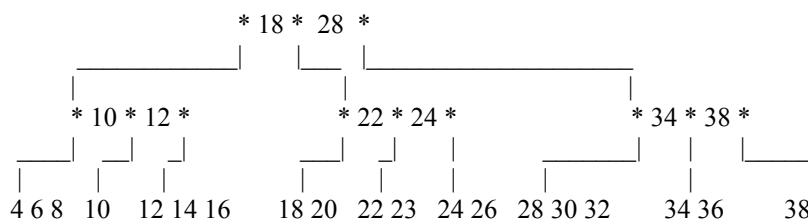
Nodul rădăcină nu este neapărat primul articol din fișierul arbore B din cel puțin două motive:

- Primul bloc (sau primele blocuri, funcție de dimensiunea lor) conține informații despre structura fișierului (metadate): dimensiune bloc, adresa bloc rădăcină, numărul ultimului bloc folosit din fișier, dimensiune chei, număr maxim de chei pe bloc, ș.a.
- Blocul rădăcină se poate modifica în urma creșterii înălțimii arborelui, consecință a unui număr mai mare de articole adăugate la fișier.

Fiecare bloc disc trebuie să conțină la început informații cum ar fi numărul de chei pe bloc și (eventual) dacă este un nod interior sau un nod frunză.

Insertia unei chei într-un arbore B începe prin căutarea blocului de care aparține noua cheie și pot apărea două situații:

- mai este loc în blocul respectiv, cheia se adaugă și nu se fac alte modificări;
- nu mai este loc în bloc, se sparge blocul în două, mutând jumătate din chei în noul bloc alocat și se introduce o nouă cheie în nodul părinte (dacă mai este loc). Acest proces de apariție a unor noi noduri se poate propaga în sus până la rădăcină, cu creșterea înălțimii arborelui B. Exemplu de adăugare a cheii 23 la arborele anterior:

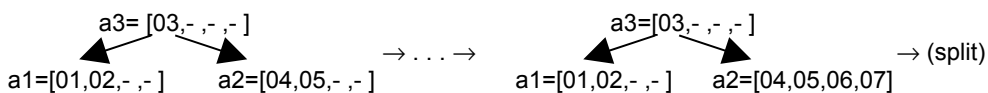


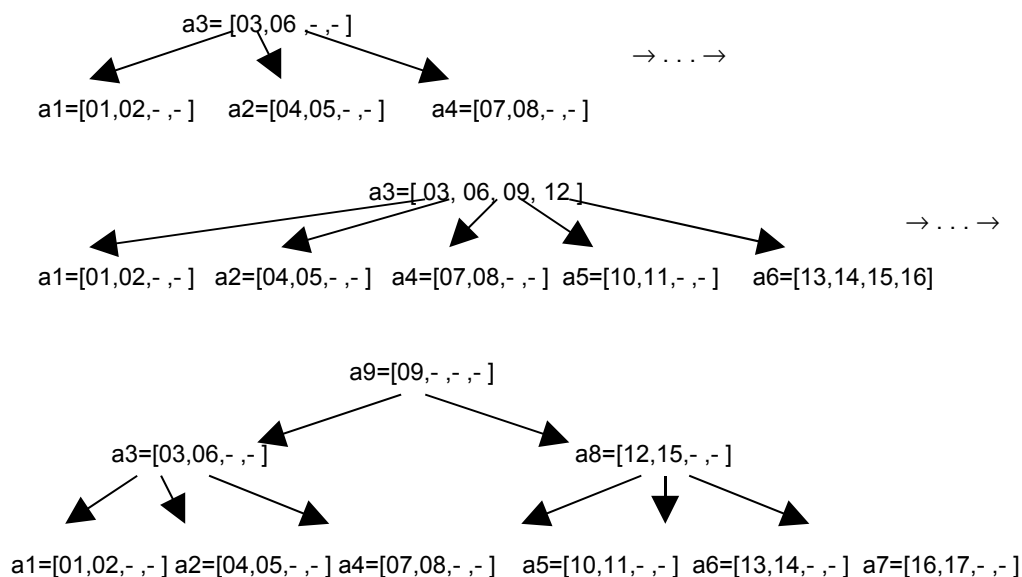
Propagarea în sus pe arbore a unor modificări de blocuri înseamnă recitirea unor blocuri disc, deci revenirea la noduri examinate anterior. O soluție mai bună este anticiparea umplerii unor blocuri: la adăugarea unei chei într-un bloc se verifică dacă blocul este plin și se “sparge” în alte două blocuri.

Eliminarea unei chei dintr-un arbore B poate antrena comasări de noduri dacă rămân prea puține chei într-un nod (mai puțin de jumătate din capacitatea nodului).

Vom ilustra evoluția unui arbore B cu maxim 4 chei și 5 legături pe nod (un arbore 2-3-4 dar pe suport extern) la adăugarea unor chei din două caractere cu valori succesive: 01, 02, ..., 09, 10, 11, ..., 19 prin câteva cadre din filmul acestei evoluții. Toate nodurile conțin chei și date, iar articolul 0 din fișier conține metadate; primul articol cu date este 1 (notat “a1”), care inițial este și rădăcina arborelui.

a1=[01,-,-,-] → [01,02,-,-] → [01,02,03,-] → [01,02,03,04] → (split)





Secvența de chei ordonate este cea mai rea situație pentru un arbore B, deoarece rămân articole numai pe jumătate completate (cu câte 2 chei), dar am ales-o pentru că ea conduce repede la noduri pline și care trebuie sparte (“split”), deci la crearea de noi noduri. Crearea unui nod nou se face mereu la sfârșitul fișierului (în primul loc liber) pentru a nu muta date dintr-un nod în altul (pentru minimizarea operațiilor cu discul).

Vom prezenta câteva funcții pentru operații cu un arbore B și structurile de date folosite de aceste funcții:

```

// elemente memorate în nodurile arborelui B
typedef struct {           // o pereche cheie- date asociate
    char key[KMax];        // cheie (un sir de caractere)
    char data[DMax];       // date (un sir de car de lungime max. DMax)
} Item;

// structura unui nod de arbore B (un articol din fișier, inclusiv primul articol)
typedef struct {
    int count;              // Numar de chei dintr-un nod
    Item keys[MaxKeys];     // Chei și date dintr-un nod
    int link[MaxKeys+1];    // Legături la noduri fii (int sau long)
} BTreeNode;

// zona de lucru comună funcțiilor
typedef struct {
    FILE * file;           // Fișierul ce conține arborele B
    char fmode;             // Mod de utilizare fișier ('r' sau 'w')
    int size;               // Numar de octeți pe nod
    int items;              // Numar total de chei în arbore
    int root;              // Numar bloc cu radacina arborelui
    int nodes;             // Numar de noduri din arborele B
    BTreeNode node;        // aici se memorează un nod (nodul curent)
} Btree;
  
```

În această variantă într-un nod nu alternează chei și legături; există un vector de chei (“keys”) și un vector de legături (“link”). link[i] este adresa nodului ce conține chei cu valori mai mici decât keys[i], iar link[i+1] este adresa nodului cu chei mai mari decât keys[i]. Pentru n chei sunt n+1 legături la succesori. Prin “adresă nod” se înțelege aici poziția în fișier a unui articol (relativ la începutul fișierului).

Înainte oricărei operații este necesară deschiderea fișierului ce conține arborele B, iar la închidere se rescrie antetul (modificat, dacă s-au făcut adăugări sau stergeri).

Căutarea unei chei date în arborele B se face cu funcția următoare:

```
// cauta cheia "key" si pune elementul care o contine in "item"
int retrieve(Btree & bt, char* key, Item & item) {
    int rec=bt.root;          // adresa articol cu nod radacina (extras din antet)
    int idx;                  // indice cheie
    int found=0;              // 1 daca cheie gasita in arbore
    while ((rec != NilPtr) && (! found)) {
        fseek( bt.file, rec*bt.size, 0); // pozitionare pe articolul cu numarul "rec"
        fread( & bt.node, bt.size,1,bt.file); // citire articol in campul "node" din "bt"
        if (search( bt, key, idx)) {        // daca "key" este in nodul curent
            found = 1;
            item = bt.node.keys[idx];        // cheie+date transmise prin arg. "item"
        }
        else                          // daca nu este in nodul curent
            rec = bt.node.link[idx + 1];    // cauta in subarborele cu rad. "rec"
    }
    return found;
}
```

Funcția de căutare a unei chei într-un nod :

```
// cauta cheia "key" in nodul curent si pune in "idx" indicele din nod
// unde s-a gasit (rezultat 1) sau unde poate fi cautata (rezultat 0)
// "idx" este -1 daca "key" este mai mica decat prima cheie din bloc
int search( Btree & bt, KeyT key, int & idx) {
    int found=0;
    if (strcmp(key, bt.node.keys[0].key) < 0)
        idx = -1;                // chei mai mici decat prima cheie din nod
    else { // cautare secventiala in vectorul de chei
        idx = bt.node.count - 1; // incepe cu ultima cheie din nod (maxima)
        while ((strcmp(key, bt.node.keys[idx].key) < 0) && (idx > 0))
            idx--;                // se opreste la prima cheie >= key
        if (strcmp(key, bt.node.keys[idx].key) == 0)
            found = true;
    }
    return found; // cheie negasita, dar mai mare ca keys[idx].key
}
```

Adăugarea unui nou element la un arbore B este realizată de câteva funcții:

- "addItem" adaugă un element dat la nodul curent (știind că este loc)
- "find" caută nodul unde trebuie adăugat un element, vede dacă nodul este plin, creează un nod nou și raportează dacă trebuie creat nod nou pe nivelul superior
- "split" sparge un nod prin crearea unui nod nou și repartizarea cheilor în mod egal între cele două noduri
- "insert" folosește pe "find" și, dacă e nevoie, creează un alt nod rădăcină.

```
void insert(Btree & bt, Item item) {
    int moveUp;                // initializat de "find"
    int newRight;              // initializat de "find"
    Item newItem;              // initializat de "find"
    // cauta nodul ce trebuie sa contine "item"
    find(bt, item, bt.root, moveUp, newItem, newRight);
    if (moveUp) { // daca e nevoie se creeaza un alt nod radacina
        bt.node.count = 1;      // cu o singura cheie
        bt.node.keys[0] = newItem; // cheie si date asociate
        bt.node.link[0] = bt.root; // la stanga are vechiul nod radacina
        bt.node.link[1] = newRight; // la dreapta nodul creat de "find"
        bt.nodes++;             // primul nod liber (articol)
    }
}
```

```

    bt.root = bt.nodes;          // devine nod radacina
    fseek(bt.file, bt.nodes*bt.size, 0);    // si se scrie in fisier
    fwrite(&bt.node, bt.size, 1, bt.file);
}
bt.items++;                      // creste numărul de elemente din arbore
}

// determina nodul unde trebuie plasat "item": "moveUp" este 1 daca
// "newItem" trebuie plasat in nodul parinte (datorita spargerii unui nod)
// "moveUp" este 0 daca este loc in nodul gasit in subarb cu rad. "croot"
void find( Btree & bt, Item item, int croot, int & moveUp, Item & newItem,
          int & newRight) {
    int idx;
    if (croot == NilPtr) { // daca arbore vid se creeaza alt nod
        moveUp = true;
        newItem = item;
        newRight = NilPtr;
    }
    else { // continua cautarea
        fseek(bt.file, croot * bt.size, 0); // citire nod radacina
        fread(&bt.node, bt.size, 1, bt.file);
        if (search(bt, item.key, idx))
            error("Error: exista deja o cheie cu aceasta valoare");
        // cauta in nodul fiu
        find(bt, item, bt.node.link[idx + 1], moveUp, newItem, newRight);
        // daca nod plin, plaseaza newItem mai sus in arbore
        if (moveUp) {
            fseek (bt.file, croot * bt.size, 0);
            fread(&bt.node, bt.size, 1, bt.file);
            if ( bt.node.count < MaxKeys) {
                moveUp = 0;
                addItem (newItem, newRight, bt.node, idx + 1);
                fseek (bt.file, croot * bt.size, 0);
                fwrite(&bt.node, bt.size, 1, bt.file);
            }
            else {
                moveUp = 1;
                split(bt, newItem, newRight, croot, idx, newItem, newRight);
            }
        }
    }
}

// sparge blocul curent (din memorie) in alte 2 blocuri cu adrese in
// croot si *newRight; "item" a produs umplerea nodului, "newItem"
// se muta in nodul parinte
void split(Btree & bt, Item item, int right, int croot, int idx, Item & newItem,
          int & newRight) {
    int j, median;
    BTreeNode rNode;          // nod nou, creat la dreapta nodului croot
    if (idx < MinKeys)
        median = MinKeys;
    else
        median = MinKeys + 1;
    fseek(bt.file, croot * bt.size, 0);
    fread( &bt.node, bt.size, 1, bt.file);
    for (j = median; j < MaxKeys; j++) {
        // muta jumătate din elemente in rNode
        rNode.keys[j - median] = bt.node.keys[j];
    }
}

```

```

    rNode.link[j - median + 1] = bt.node.link[j + 1];
}
rNode.count = MaxKeys - median;
bt.node.count = median;      // is then incremented by addItem
// put CurrentItem in place
if (idx < MinKeys)
    addItem(item, right, bt.node, idx + 1);
else
    addItem(item, right, rNode, idx - median + 1);
newItem = bt.node.keys[bt.node.count - 1];
rNode.link[0] = bt.node.link[bt.node.count];
bt.node.count--;
fseek(bt.file, croot*bt.size, 0);
fwrite(&bt.node, bt.size, 1, bt.file);
bt.nodes++;
newRight = bt.nodes;
fseek(bt.file, newRight * bt.size, 0 );
fwrite( &rNode, bt.size, 1, bt.file);
}

// adauga "item" la nodul curent "node" in pozitia "idx"
// prin deplasarea la dreapta a elementelor existente
void addItem(Item item, int newRight, BTreeNode & node, int idx) {
    int j;
    for (j = node.count; j > idx; j--) {
        node.keys[j] = node.keys[j - 1];
        node.link[j + 1] = node.link[j];
    }
    node.keys[idx] = item;
    node.link[idx + 1] = newRight;
    node.count++;
}

```


Capitolul 11

PROGRAMAREA STRUCTURILOR DE DATE IN C++

11.1 AVANTAJELE LIMBAJULUI C++

Un limbaj cu clase permite un nivel de generalizare si de abstractizare care nu poate fi atins într-un limbaj fără clase. In cazul structurilor de date generalizare înseamnă genericitate, adică posibilitatea de a avea ca elemente componente ale structurilor de date (colectiilor) date de orice tip, inclusiv alte structuri de date.

Clasele abstracte permit implementarea conceptului de tip abstract de date, iar derivarea permite evidentierea legăturilor dintre diferite structuri de date. Astfel, un arbore binar de căutare devine o clasă derivată din clasa arbore binar, cu care folosește în comun o serie de metode (operatii care nu depind de ordinea valorilor din noduri, cum ar fi afisarea arborelui), dar față de care posedă metode proprii (operatii specifice, cum ar fi adăugarea de noi noduri sau căutarea unei valori date).

Din punct de vedere pragmatic, metodele C++ au mai putine argumente decât functiile C pentru aceleasi operatii, iar aceste argumente nu sunt de obicei modificate în functii. Totusi, principalul avantaj al unui limbaj cu clase este posibilitatea utilizării

unor biblioteci de clase pentru colectii generice, ceea ce simplifică programarea anumitor aplicatii si înlocuirea unei implementări cu o altă implementare pentru acelasi tip abstract de date.

Structurile de date se pretează foarte bine la definirea de clase, deoarece reunesc variabile de diverse tipuri si operatii (functii) asupra acestor variabile (structuri). Nu este întâmplător că singura bibliotecă de clase acceptată de standardul C++ contine practic numai clase pentru structuri de date (STL = Standard Template Library).

Programul următor creează si afisează un dictionar ordonat pentru problema frecventei cuvintelor, folosind clasele “map”, “iterator” si “string” din biblioteca STL

```
#include <iostream>           // definitii clase de intrare-iesire
#include <string>              // definitia clasei "string"
#include <map>                 // definitia clasei "map"
#include <iterator>            // definitia clasei "iterator"
using namespace std;         // spatiu de nume ptr clasele standard

int main () {
    map<string,int> dic;       // dictionar cu chei "string" si valori "int"
    string cuv;               // aici se citeste un cuvânt
    map<string,int>::iterator it; // un iterator pe dictionar
    while (cin >> cuv)        // citeste cuvânt de la tastatura
        dic[cuv]++;           // actualizare dictionar
    for (it=dic.begin(); it !=dic.end(); it++) // parcurge dictionar cu iterator
        cout << (*it).first <<":" << (*it).second << endl; // si afisare elemente
}
```

Programul anterior este compact, usor de citit (după o familiarizare cu clasele STL) si eficient, pentru că dictionarul este implementat ca un arbore binar cu autoechilibrare, de înăltime minimă.

Faptul că se folosesc aceleasi clase standard în toate programele C++ asigură acestora un aspect uniform, ceea ce facilitează si mai mult înțelegerea lor rapidă si modificarea lor fără introducerea de erori. Programele C sunt mult mai diverse datorită multiplelor posibilități de codificare si de asigurare a genericității, precum si absentei unor biblioteci standard de functii pentru operatii cu structuri de date uzuale.

Pe de altă parte, biblioteca STL (ca si biblioteca de clase colectie Java) nu contine toate structurile de date, ci numai pe cele mai importante. De aceea, programatorii trebuie să poată defini noi clase sablon în care să folosească facilitățile oferite de STL

Clasele STL sunt definite independent unele de altele, fără a pune în evidență relațiile existente între ele. Tehnici specifice programării orientate pe obiecte, cum sunt derivarea și mostenirea nu sunt folosite în definirea acestor clase. În schimb, apar funcții polimorfice, cu aceeași formă dar cu implementare diferită în clase diferite (metode comune claselor container STL).

Clase pentru structuri de date generice se pot realiza în C++ (și în Java) și altfel decât prin clase sablon ("template"), soluție folosită în multe cărți de structuri de date care folosesc limbajul C++ pentru că permite definirea unor familii de clase înrudite, folosind derivarea și mostenirea. Există și biblioteci de clase pentru structuri de date bazate pe această soluție (CLASSLIB din Borland C 3.1), dar nici una care să se bucure de o recunoaștere atât de largă ca biblioteca STL.

Această soluție de definire a unor clase C++ pentru colecții generice seamănă cu utilizarea de pointeri generici (void*) în limbajul C.

Ideea este ca toate clasele colecție să contină ca elemente pointeri la obiecte de un tip abstract, foarte general ("object"), iar clasele care generează obiecte membre în astfel de colecții să fie derivate din clasa "object". Deci toate clasele folosite în astfel de aplicații ar trebui să fie derivate direct sau indirect dintr-o clasă de bază "object", aflată la rădăcina arborelui de clase.

Un pointer la tipul "object" poate fi înlocuit cu un pointer către orice alt subtip al tipului "object", deci cu un pointer la o clasă derivată din "object" (derivarea creează subtipuri ale tipului de bază). Pentru a memora date de un tip primitiv (numere, de exemplu) într-o astfel de colecție, va trebui să dispunem (sau să definim) clase cu astfel de date și care sunt derivate din clasa "object".

În containerele STL se pot introduce atât valori cât și pointeri, ceea ce permite în final și containere cu obiecte de diferite tipuri înrudite (derivate unele din altele).

11.2 CLASE ȘI OBIECTE ÎN C++

Clasele C++ reprezintă o extindere a tipurilor structură, prin includerea de funcții ca membri ai clase, alături de variabilele membre ale clasei. Funcțiile, numite și metode ale clasei, realizează operații asupra datelor clasei, utile în aplicații.

O clasă ce corespunde unei structuri de date grupează împreună variabilele ce definesc colecția și operațiile asociate, specifice fiecărui tip de colecție. De exemplu, o clasă "Stivă" poate avea ca date un vector și un întreg (indice spre vârful stivei), iar ca metode funcții pentru punerea unei valori pe stivă ("push"), scoaterea valorii din vârful stivei ("pop") și altele.

De obicei datele unei clase nu sunt direct accesibile pentru funcții din afara clasei având atributul "private" (implicit), ele fiind accesibile numai prin intermediul metodelor publice ale clasei. Aceste metode formează "interfața" clasei cu exteriorul. Exemplu de definire a unei clase pentru stivă vector de numere întregi:

```
class Stiva {
private:
    int s[100];          // vector cu dimensiune fixa ca stiva
    int sp;              // prima adresa libera din stiva
public:                // urmeaza metodele clasei
    Stiva() { sp=0; }    // un constructor pentru obiectele clasei
    void push ( int x ) { s[sp++] =x; } // pune x in aceasta stiva
    int pop () { return s[--sp]; }    // scoate valoarea din varful stivei
    int empty() { return sp==0; }    // daca stiva este goala
};                      // aici se termina definitia clasei
```

Orice clasă are unul sau mai mulți constructori pentru obiectele clasei, sub forma unor funcții fără nici un tip și cu numele clasei. Constructorul alocă memorie (dacă sunt date alocate dinamic în clasă) și initializează variabilele clasei. Exemplu de clasă pentru o stivă cu vector alocat dinamic și cu doi constructori:

```
class Stiva {
private:
    int *s;              // vector alocat dinamic ca stiva
```

```

    int sp;           // prima adresa libera din stiva
public:              // urmeaza metodele clasei
    Stiva(int n) {    // un constructor pentru obiectele clasei
        s= new int[n]; // alocare memorie ptr n intregi
        sp=0;         // initializare varf stiva
    }
    Stiva () {
        s= new int[100];
        sp=0;
    }
    . . . // metodele clasei
};

```

O functie constructor este apelată automat în două situații:

- La declararea unei variabile de un tip clasă, însoțită de paranteze cu argumente pentru funcția constructor. Exemplu:

```
Stiva a(20); // a este o stiva cu maxim 20 de elemente
```

- La alocarea de memorie pentru un obiect cu operatorul *new* (dar nu și cu funcția “alloc”), care poate fi urmat de paranteze și de argumente pentru constructor. Exemple:

```
Stiva * ps = new Stiva(20);
Stiva a = * new Stiva(20);
```

Ambele moduri de initializare sunt posibile în C++ și pentru variabile de un tip primitiv (tipuri ale limbajului C). Exemple:

```
int x = *new int(7);
int x(7); // echivalent cu int x=7
```

În C++ funcțiile pot avea și argumente cu valori implicite (ultimele argumente); dacă la apel lipsește argumentul efectiv corespunzător, atunci se folosește implicit valoarea declarată cu argumentul formal corespunzător. Această practică se folosește și pentru constructori (și pentru metode). Exemplu:

```

class Stiva {
private:
    int *s, sp; // datele clasei
public:        // metodele clasei
    Stiva(int n=100) { // constructor cu argument cu valoare implicita
        s = new [n]; // alocare memorie
        sp=0;
    }
    . . .
};

```

O clasă este un tip de date, iar numele clasei poate fi folosit pentru a declara variabile, pointeri și funcții cu rezultat de tip clasă (la fel ca și la tipuri structurale). O variabilă de un tip clasă se numește și obiect. Sintaxa apelării metodelor unei clase C++ diferă de sintaxa apelării funcțiilor C și exprimă acțiuni de forma “apelează metoda push pentru obiectul stivă cu numele a”. Exemplu de utilizare:

```

#include "stiva.h"
void main () {
    Stiva a; int x; // a este un obiect de tip Stiva
    while (cin >> x) // citeste in x numere de la stdin
        a.push(x); // si le pune in stiva a
    cout << "\n Continut stiva: \n" ;
    while ( ! a.empty()) // cat timp stiva a nu este goala

```

```

    cout << a.pop() << " "; // afiseaza pe ecran numarul scos din stiva
}

```

Un alt exemplu este clasa următoare, pentru un graf reprezentat printr-o matrice de adiacente, cu câteva operații strict necesare în aplicațiile cu grafuri:

```

class graf {
char **a;          // adresa matrice de caractere (alocata dinamic)
int n;             // numar de noduri
public:
    graf(int n) {          // un constructor
        this->n=n;         // this->n este variabila clasei, n este argumentul
        a = new char*[n+1]; // aloca memorie pentru vectorul principal
        for (int i=0;i<=n;i++) { // si pentru fiecare linie din matrice
            a[i]= new char[n+1]; // pozitia 0 nu este folosita (nodurile sunt 1,2,...)
            memset (a[i],0,n+1); // initializare linie cu zerouri
        }
    }
    int size() { return n;} // dimensiune graf (numar de noduri)
    int arc (int v, int w) { // daca exista arc de la v la w
        return a[v][w];
    }
    void addarc (int v, int w) { // adauga arc de la v la w
        a[v][w]=1;
    }
    void print() {          // afisare graf sub forma de matrice
        for (int i=1;i<=n;i++) {
            for (int j=1; j <=n;j++)
                cout << (int)a[i][j] << " ";
            cout << "\n";
        }
    }
};
// utilizarea unui obiect de tip "graf"
int main (){
    int n, x, y;
    cout << "nr noduri: "; cin >> n;
    graf g(n);          // apelare constructor
    while (cin >> x >> y ) // citeste perechi de noduri (arce)
        g.addarc(x,y);   // adauga arc la graf
    g.print();
}

```

Alte operații cu grafuri pot fi incluse ca metode în clasa “graf”, sau pot fi metode ale unei clase derivate din “graf” sau pot fi funcții separate, cu un argument “graf”. Exemplu de funcție independentă pentru vizitarea unui graf în adâncime și crearea unui vector de noduri, în ordinea DFS:

```

void dfs (graf g, int v, int t[]) { // vizitare din v cu rezultat in t
    static int k=0;                // k indica ordinea de vizitare
    int n=g.size();
    t[v]=++k;                      // nodul v vizitat in pasul k
    for (int w=1;w<=n;w++)
        if ( g.arc(v,w) && t[w]==0 ) // daca w este succesor nevizitat
            dfs (g,w,t);           // continua vizitare din w
}

```

Definirea metodelor unei clase se poate face și în afara clasei, dar ele trebuie declarate la definirea clasei. De fapt, anumite metode (care contin cicluri, de ex.) chiar trebuie definite în afara clasei pentru

a nu primi mesaje de la compilator. In general definiția clasei este introdusă într-un fișier antet (de tip H), iar definițiile metodelor sunt continute în fișiere de tip CPP sau sunt deja compilate și introduse în biblioteci statice sau dinamice (LIB sau DLL în sisteme MS-Windows).

Exemplu de clasă pentru obiecte folosite în extragerea cuvintelor dintr-un sir, ca o soluție mai bună decât funcția standard “strtok”:

```
// fișier tokenizer.h
#include <string.h>
#include "stddef.h"
class tokenizer {
    char* str;    // sir analizat
    char* sep;    // separatori de cuvinte
    char *p;      // pozitie curenta in str
    char token[256]; // aici se depune cuvantul extras din str
public:          // metode publice ale clasei
    tokenizer ( char* st, char* delim=" ") {    // un constructor
        str=st; sep=delim; p=str;
    }
    char* next();    // urmatorul cuvant din sirul analizat
    bool hasNext () ; // verifica daca mai exista cuvinte
};
```

```
// fișier tokenizer.cpp
// daca mai sunt cuvinte in sirul analizat
bool tokenizer::hasNext(){
    return *p != 0;
}
// urmatorul cuvant
char* tokenizer::next() {
    char * q = token;
    while ( *p && strchr(sep,*p) !=0 )
        p++; // ignora separatori dintre cuvinte
    while ( *p && strchr(sep,*p)== 0 )
        *q++=*p++;
    *q=0;
    return token;
}
```

Numele bibliotecii de clase sau numele fișierului OBJ cu metodele clasei trebuie să apară în același proiect cu numele fișierelor din aplicația care le folosește; excepție face biblioteca standard STL, în care editorul de legături caută implicit.

Exemplu de utilizare a clasei “tokenizer” :

```
#include <stdio.h>
void main () {
    char line[256];
    gets(line);
    tokenizer tok (line);
    while ( tok.hasNext())
        puts (tok.next());
}
```

Clasa “tokenizer” este utilă în aplicații, dar suportă și alte variante de definire:

- În locul tipului “char*” sau pe lângă acest tip se poate folosi și tipul “string” definit în biblioteca STL și recomandat pentru sirurile introduse în containere STL.
- În locul unui constructor cu argument sirul analizat se poate defini un constructor fără argumente și o metodă care să preia sirul analizat; în felul acesta nu am mai fi obligați să declarăm variabila de tip

“tokenizer” după citirea liniei analizate. Aceeasi situatie apare si la clasa “vector” din STL, pentru care capacitatea vectorului se poate specifica în constructor sau se transmite prin metoda “reserve”.

Specifică limbajului C++ este supradefinirea operatorilor, care permite extinderea utilizării limbajului C si pentru operatii asupra obiectelor unei clase (în loc de a folosi functii pentru aceste operatii). Astfel, operatorul “>>” aplicat obiectului predefinit “cin” are efectul unei citiri din fisierul standard de intrare (“console input”), iar operatorul “<<” aplicat obiectului “cout” are efectul unei scrieri în fisierul standard de iesire (“console output”).

Un operator supradefinit în C++ este tot o functie dar cu un nume mai special, format din cuvântul cheie *operator* si unul sau două caractere speciale, folosite în limbajul C ca operator (unar sau binar). Un operator poate fi definit în câteva moduri:

- ca metodă a unei clase (operator declarat în cadrul clasei);
- ca functie externă clasei (dacă datele clasei sunt publice sau public accesibile);
- ca functie externă declarată “prieten” (“friend”) în clasă, pentru ca functia externă să aibă acces la datele “private” ale clasei.

Exemplu de supradefinire a operatorului de incrementare prefixat printr-o metodă:

```
class counter {
    int m;                // un contor intreg
public:
    counter() { m=0;}      // constructor fara argumente
    void operator++ () { m++;} // metoda cu numele “operator++”
    int val() { return m;}  // obtine valoare contor
};
// utilizare
void main () {
    counter c;             // se apeleaza implicit constructor fara argumente
    ++c;                   // echivalent cu: c.operator++()
    cout << c.val() << endl; // afisare valoare contor
}
```

Exemplu de supradefinire a operatorului << ca functie prieten a clasei “counter”:

```
class counter {
    int m;
public:
    counter() { m=0;}
    void operator++ () { m++;}
    int& val() { return m;}
    friend ostream & operator << (ostream & os, counter & c);
};
ostream & operator << (ostream & os, counter & c) {
    os << c.m;           // referire la date private ale clasei counter
    return os;
}
```

Exemplu de supradefinire a operatorului << ca functie externă clasei si care nu trebuie declarată în clasă:

```
ostream & operator << (ostream & os, counter & c) {
    return (os << c.val());
}
```

Exemplu de supradefinire a operatorului de indexare într-o clasă vector:

```
class Vector {
    int * vec;           // adresa vector (alocat dinamic)
```

```

int n, max;      // max=capacitate vector, n= nr de elemente in vector
public:
Vector (int m) { vec = new int[max=m]; n=0; }
int get (int i) { return vec[i]; }           // functie de acces la elemente
int& operator [ ] (int i) { return vec[i]; } // operator de indexare []
void add (int x) { vec[n++]= x; }           // adaugare la sfarsit de vector
};

```

Elementul din pozitia k a unui obiect "Vector" poate fi obtinut fie prin metoda "get", fie cu operatorul de indexare, dar operatorul permite si modificarea valorii din pozitia k deoarece are rezultat referinta . Exemplu:

```

void main () {
    Vector a(10);
    a.add(1); cout << a.get(0) << endl;    // scrie 1
    a[0]= 5; cout << a[0] << endl;        // scrie 5
}

```

Clasele ale căror obiecte se introduc în containere STL trebuie să aibă definiți operatorii de comparatie si de afisare, folositi în cadrul claselor container. Exemplu:

```

// clasa ptr arce de graf
class arc {
public:
    int v,w; // extremitati arc
    int cost; // cost arc
    arc(int x=0, int y=0, int c=0) {v=x;w=y;cost=c;}
};

bool operator== (const arc &x,const arc &y) {
    return x.cost == y.cost;
}
bool operator< (const arc &x,const arc &y) {
    return x.cost < y.cost;
}
bool operator> (const arc &x,const arc &y) {
    return x.cost > y.cost;
}
ostream& operator<< (ostream &s, arc a) {
    return s << "(" << a.v << "-" << a.w << " = " << a.cost << ")";
}
istream& operator >> (istream &s, arc &a) {
    return s >> a.v >> a.w >> a.cost ;
}

```

O clasă A poate contine ca membri obiecte, deci variabile de un tip clasă B. În acest fel metodele clasei A pot (re)folosi metode ale clasei B. De exemplu, o clasă stivă poate contine un obiect de tip vector si refoloseste metode ale clasei vector, cum ar fi extinderea automată a capacității la umplerea vectorului.

Problema cu aceste clase agregat (compuse) este că anumite date primite de constructorul clasei agregat A trebuie transmise constructorului obiectului continut, care le foloseste la initializarea unor variabile ale clasei B. De exemplu, capacitatea initială a vectorului este primită de constructorul obiectelor stivă, pentru că este si capacitatea initială a stivei. Un constructor nu poate fi apelat ca o metodă (ca o functie) obisnuită. Din acest motiv s-a introdus o sintaxă specială pentru transmiterea datelor de la un constructor A la un constructor B al obiectului b :

```

A ( tip x): b(x) { ... // alte operatii din constructorul A }

```

Exemplu:

```
class Stiva {
private:
    Vector s;          // vector ca obiect din clasa Vector
    int sp;            // prima adresa libera din stiva
public:               // urmeaza metodele clasei
    Stiva(int n): s(n) { sp=0;} // un constructor pentru obiectele clasei
    ...                // alte metode ale clasei Stiva
};
```

Această sintaxă este extinsă și pentru variabile ale clasei A care nu sunt obiecte (variabile de tipuri primitive). Exemplu de constructor pentru clasa anterioară:

```
Stiva(int n): s(n),sp(0) { } // echivalent cu sp=0
```

O noutate a limbajului C++ față de limbajul C o constituie și spațiile de nume pentru funcții (și clase), în sensul că pot exista funcții (și clase) cu nume diferite în spații de nume diferite. Toate clasele și funcțiile STL sunt definite în spațiul de nume “std”, care trebuie specificat la începutul programelor ce folosesc biblioteca STL:

```
using namespace std;
```

Am menționat aici acele aspecte ale definirii de clase C++ care sunt folosite și în clasele STL, pentru a facilita înțelegerea exemplelor cu clase STL.

În realitate, definițiile de clase sunt mult mai complexe decât exemplele anterioare și folosesc facilități ale limbajului C++ care nu au fost prezentate.

11.3 CLASE SABLON (“TEMPLATE”)

În C++ se pot defini funcții și clase sablon având ca parametri tipurile de date folosite în funcția sau în clasa respectivă.

Exemplu de funcție sablon pentru determinarea valorii minime dintre două variabile de orice tip T, tip neprecizat la definirea funcției:

```
// definire funcție sablon cu un parametru "class"
template <class T> T min (T a, T b) { // T poate fi orice tip definit anterior
    return a < b? a: b;
}
```

Cuvântul “class” arată că T este un parametru ce desemnează un tip de date și nu o valoare, dar nu este obligatoriu ca funcția “min” să folosească un parametru efectiv de un tip clasă. În funcția “min” tipul care va înlocui tipul neprecizat T trebuie să cunoască operatorul ‘<’ cu rol de comparație la mai mic. Exemplul următor arată cum se poate folosi funcția sablon “min” cu câteva tipuri de date primitive:

```
// utilizari ale funcției sablon
void main () {
    double x=2.5, y=2.35 ;
    cout << min (x,y); // min (double,double)
    cout << min (3,2); // min (int,int)
    cout << min (&x,&y); // min (double*,double*)
}
```


O clasă sablon este o clasă în a cărei definiție se folosesc tipuri de date neprecizate pentru variabile și/sau pentru funcții membre ale clasei. Toate tipurile neprecizate trebuie declarate într-un preambul al definiției clasei, care începe prin cuvântul cheie "template" și este urmat, între paranteze ascuțite, de o listă de nume de tipuri precedate fiecare de cuvântul "class". Exemplu:

```
// o clasa stiva cu componente de orice tip T
template <class T> class Stiva {
    T* s;           // adresa vector folosit ca stiva
    int sp;         // indice la varful stivei
public:
    Stiva (int n=100) { s= new T [n]; sp=0;}
    void push (T x) { s[sp++]=x; }
    T pop () { return s[--sp]; }
    int empty() { return sp==0;}
    void print ();      // definita in afara clasei
};
```

La declararea unei variabile de tip clasă sablon trebuie precizat numele tipului efectiv utilizat, între paranteze ascuțite, după numele clasei. Exemplu:

```
// utilizare clasa sablon
void main () {
    Stiva <int> a (20);    // a este o stiva de intregi
    for (int k=0;k<10;k++)
        a.push (k);
}
```

Pentru metodele definite în afara clasei trebuie folosit cuvântul "template". Exemplu:

```
template <class T> void Stiva<T> :: print () {
    for (int i=0;i<sp;i++)
        cout << s[i] << ' ';    // daca se poate folosi operatorul << pentru tipul T
}
```

Pe baza definiției clasei sablon și a tipului parametrilor efectivi de la instantierea clasei, compilatorul înlocuiește tipul T prin tipul parametrilor efectivi. Pentru fiecare tip de parametru efectiv se generează o altă clasă, așa cum se face expandarea unei macroinstrucțiuni (definită prin "define"). Definiția clasei este folosită de compilator ca un "sablon" (tipar, model) pentru a genera definiții de clase "normale".

Între parantezele unghiulare pot fi mai multe tipuri, dacă în clasă se folosesc două sau mai multe tipuri neprecizate. Exemplu:

```
#include "stldef.h"
// dictionar cu chei unice din 2 vectori
template <class KT, class VT> class mmap {
private:
    KT * keys;           // vector de chei
    VT * values;         // vector de valori asociate
    int n;               // n = numar de chei si de valori
// functie interna clasei (private)
    int find (KT k) {    // cauta cheia k in dictionar
        for (int i=0;i<n;i++)
            if (k==keys[i])    // cheie gasita in pozitia i
                return i;
        return -1;          // cheie negasita
    }
public:
```

```

mmap (int m=100) { // constructor cu argum dimensiune vectori
    keys = new KT[m];
    values = new VT[m];
    n=0;
}
void put (KT k, VT v) { // pune cheia k si valoarea v in dictionar
    int j= find(k);
    if ( j >=0 )
        values[j]=v; // modifica valoarea asociata cheii k
    else { // cheia exista, nu se modifica dictionarul
        keys[n]=k; values[n]=v;
        n++;
    }
}
VT get (KT k) { // gaseste valoarea asociata unei chei date
    int j=find(k);
    return values[j];
}
void print () { // afisare continut dictionar
    for (int i=0;i<n;i++)
        cout << '['<<keys[i]<<']='<<values[i]<<"] ";
    cout << endl;
}
bool hasKey (KT k) { // daca exista cheia k in dictionar
    return find(k)>=0;
}
};
// utilizare dictionar ptr frecventa cuvintelor
int main () {
    mmap <string,int> dic(20);
    string cuv; int nr;
    while (cin >> cuv)
        if ( dic.hasKey(cuv)) {
            nr=dic.get(cuv);
            dic.put(cuv,nr+1);
        }
        else
            dic.put (cuv,1);
    dic.print();
}

```

Pentru cuvinte am folosit tipul “string” (clasă STL) si nu “char*” pentru că are (supra)definiti operatorii de comparatie, utilizati la căutarea unei chei în dictionar. In general, clasele ale căror obiecte se introduc în containere STL trebuie să aibă definiti acesti operatori (==, !=, <, >, <=, >=), pentru ca să putem exprima la fel comparatiile, indiferent de tipul datelor care se compară.

O altă problemă, rezolvată prin exceptii, este “ce rezultat ar trebui să aibă metoda get atunci când cheia dată nu se află în dictionar”, deoarece nu stim nimic despre tipul VT al functiei “get” si deci ce valori speciale de acest tip am putea folosi.

Exemplul următor schitează o definiție a clasei “stack” din STL, clasă care poate contine orice tip de container secvential STL:

```

template <class E, class Container > class mstack {
    Container c; // aici se memoreaza date de tip E
public:
    bool empty () { return c.empty(); } // daca stiva goala
    int size () { return c.size(); } // dimensiune stiva
    E top () { return c.back(); } // valoare din varful stivei
}

```

```

void push ( E & x) { c.push_back(x);} // pune x pe stiva
void pop () { c.pop_back(); }        // elimina valoare din varful stivei
};
// utilizare clasa mstack
int main () {
    mstack <int,vector<int> > a;      // stiva realizata ca vector
    mstack <int,list<int> > b;       // stiva realizata ca lista inlantuita
    for (int i=1;i<6;i++) {
        a.push(i); b.push(i);       // pune numere in stiva
    }
    while ( !a.empty()) {            // scoate din stiva si afiseaza
        cout << a.top()<< endl;
        a.pop();
    }
}

```

Clasa “mstack” refoloseste metodele clasei container (back, push_back, pop_back, size, s.a.) altfel decât prin mostenire, deoarece “mstack” nu este subclasă derivată din clasa container. De aceea si operatorii de comparatie trebuie definiti în clasa mstack, chiar dacă definirea lor se reduce la compararea obiectelor container continute de obiectele stivă.

Clasa mstack se numeste si clasă adaptor deoarece nu face decât să modifice interfata clasei container (alte nume pentru metodele din container), fără a modifica si comportamentul clasei (nu există metode noi sau modificate ca efect).

11.4 CLASE CONTAINER DIN BIBLIOTECA STL

Biblioteca de clase STL contine în principal clase “container” generice pentru principalele structuri de date, dar si alte clase si functii sablon utile în aplicatii:

- Clase iterator pentru clasele container si pentru clase de I/E
- Clase adaptor, pentru modificarea interfetei unor clase container
- Functii sablon pentru algoritmi generici (nu fac parte din clase)

Clasele container se împart în două grupe:

- Secvente liniare: clasele *vector*, *list*, *deque*
- Containere asociative: *set*, *multiset*, *multimap*

Fiecare clasă STL este definită într-un fisier antet separat, dar fără extensia H. Exemplu de utilizare a clasei *string* din biblioteca STL:

```

#include <string>
#include <iostream>
using namespace std;
void main () {
    string a, b("Am citit: ");
    cout << "Astept un sir:"; cin >> a;
    cout << b+a << "\n";
    cout << "primul caracter citit este: " << a[0] << "\n";
}

```

Trecerea de la siruri C la obiecte *string* se face prin constructorul clasei, iar trecerea de la tipul *string* la siruri C se face prin metoda *c_str()* cu rezultat “char*”

Pentru simplificare vom considera că avem definit un alt fisier “defstl.h” care include toate fisierele antet pentru clasele STL si declaratia “using namespace std”.

Toate clasele container au câteva metode comune, dintre care mentionăm:

```

int size();        // numar de elemente din container
bool empty();      // daca container gol (fara elemente în el)
void clear();       // golire container (eliminarea tuturor elementelor)

```

```
iterator begin(); // pozitia primului element din container
iterator end();   // pozitia urmatoare ultimului element din container
```

Operatorii `==` si `<` pot fi folositi pentru compararea a două obiecte container de acelasi tip.

Containerele secventiale se folosesc pentru memorarea temporară a unor date; ele nu sunt ordonate automat la adăugarea de noi elemente, dar pot fi ordonate cu functia STL “sort”. Metode mai importante comune tuturor secventelor:

```
void push_bak() (T & x);      // adaugare x la sfarsitul secventei
void pop_back();             // eliminare element de la sfarsitul secventei
T& front();                  // valoarea primului element din secventa
T& back();                   // valoarea ultimului element din secventa
void erase (iterator p);     // eliminare element din pozitia p
void insert (iterator p, T& x); // insertie x in pozitia p
```

Clasa *vector* corespunde unui vector extensibil, clasa *list* corespunde unei liste dublu înlăntuite. Toate clasele secventă permit adăugarea de elemente la sfârșitul secventei (metoda “push_back”) într-un timp constant (care nu depinde de mărimea secventei). Exemplu de folosire a clasei *vector*:

```
int main () {
    vector<string> vs (10);      // clasa “string” este predefinita in STL
    char cb[30];                // aici se citeste un sir
    while (cin >> cb) {
        string str= *new string(cb); // se creeaza siruri distincte
        vs.push_back (str);          // adauga la sfarsitul vectorului
    }
    // afisare vector
    for (int i=0;i<vs.size();i++) // “size” este metoda a clasei “vector”
        cout << vs[i] << ' ';    // operatorul [ ] supradefinit in clasa “vector”
    cout << endl;                 // cout << “\n”
}
```

Clasele *vector* si *deque* au redefinit operatorul de indexare `[]` pentru acces la elementul dintr-o pozitie dată a unei secvente.

Clasele *list* si *deque* permit si adăugarea de elemente la începutul secventei în timp $O(1)$ (metoda “push_front”). Exemplu de utilizare a clasei *list* ca o stivă:

```
int main () {
    list<int> a;
    for (int i=1;i<6;i++)
        a.push_back(i);          // adauga la sfarsitul listei
    while ( !a.empty()) {
        cout << a.back()<< endl; // elementul de la sfarsitul listei
        a.pop_back();            // elimina ultimul element
    }
}
```

Clasele adaptor *stack*, *queue* si *priority_queue* (coada cu priorități) sunt construite pe baza claselor secventă de bază, fiind liste particulare. Clasa *stack*, de exemplu, adaugă metodele “push” si “pop” unei secvente (implicit se foloseste un container de tip *deque*, dar se poate specifica explicit un container *list* sau *vector*). Exemplu de utilizare a unei stive:

```
int main () {
    stack<int> a;
    for (int i=1;i<6;i++)
        a.push(i);
    while ( !a.empty()) {
```

```

        cout << a.top() << endl;
        a.pop();
    }
}

```

Metodele “pop”, “pop_back”, “pop_front” sunt de tip *void* si se folosesc de obicei împreună (după) metodele “top”, “back”, “front”, din motive legate de limbajul C++.

Orice container de tip “cont” poate avea mai multe obiecte iterator asociate de tipul `cont::iterator` si folosite la fel ca variabilele pointer pentru a parcurge elementele colectiei. Exemplu:

```

// afisare vector, cu iterator
vector<string>::iterator it;          // declarare obiect iterator ptr vector de siruri
for (it=vs.begin(); it!=vs.end(); it++) // “begin”, “end” metode ale clasei vector
    cout << *it << ' ';              // afisare obiect de la adresa continută în it
cout << endl;

```

Un iterator STL este o generalizare a notiunii de pointer si permite accesul secvential la elementele unui container, în acelasi fel, indiferent de structura de date folosită de container. Pentru clasele iterator sunt supradefiniti operatorii de indirectare (*), comparatie la egalitate (==) si inegalitate (!=), incrementare (++), dar operatorul de decrementare (--) există numai pentru iteratori bidirectionali.

Prin indirectare se obtine valoarea elementului din pozitia indicată de iterator. Exemplu de functie sablon pentru căutarea unei valori date *x* între două pozitii date *p1* si *p2* dintr-un container si care nu depinde de tipul containerului în care se caută:

```

template <class iterator, class T>
iterator find (iterator p1, iterator p2, T x) {
    while ( p1 != p2 && *p1 != x)
        ++p1;
    return p1;      // p1=p2 daca x negasit
}

```

Folosind obiecte iterator se poate specifica o subsecventă dintr-o secventă, iar mai multe functii STL (algoritmi generici) au două argumente de tip iterator. De exemplu, ordonarea crescătoare a unui vector *v* se poate face astfel:

```

sort (v.begin(), v.end());

```

Pentru obiectele de tip *vector* sau *deque* nu se recomandă inserări si stergeri frecvente (metodele “insert” si “erase”); dacă aplicatia foloseste secvente cu continut dinamic, atunci este mai eficientă secventa *list* (timp de ordinul $O(1)$ pentru orice pozitie din listă).

Pentru liste nu este posibil accesul prin indice (pozitional) si nici ordonarea prin metoda “sort” din biblioteca STL.

În STL multimile sunt considerate drept cazuri particulare de dictionare, la care lipsesc valorile asociate cheilor, sau un dictionar este privit ca generalizare a unei multimi cu elemente compuse dintr-o cheie si o valoare. De aceea, se foloseste denumirea de container asociativ pentru dictionare si multimi, iar metodele lor sunt practic aceleasi.

Clasele STL *set* si *map* sunt implementate ca arbori binari echilibrati (RBT) si respectă restrictia de chei distincte (unice). Clasele *multiset* si *multimap* permit si memorarea de chei identice (multimi si dictionare cu valori multiple).

Desi nu fac parte din standard, bibliotecile STL contin de obicei si clasele *hash_set*, *hash_map*, *hash_multiset* si *hash_multimap* pentru asocieri realizate ca tabele de dispersie.

Operatiile principale cu multimi, realizate ca metode ale claselor sunt:

```

iterator insert (T& x);          // adauga x la o multime multiset (modifica multimea)
pair<iterator,bool> insert (T & x); // adauga x la o multime, daca nu exista deja

```

```

iterator insert (iterator p, T& x);    // insertie x in pozitia p din multime
void erase (iterator p);              // elimina elementul din pozitia p
iterator find (T& x);                 // cauta pe x in multime

```

Metoda “insert” pentru multimii cu chei multiple are ca rezultat pozitia în care a fost inserat noul element, iar dimensiunea multimii crește după fiecare apel. Metoda “insert” pentru multimii cu chei unice are ca rezultat o pereche cu primul membru iterator (pozitia în multime) și cu al doilea membru un indicator care arată dacă s-a modificat conținutul (și dimensiunea) multimii (valoarea *true*), sau dacă exista deja un element cu valoarea x și nu s-a modificat multimea (valoarea *false*).

Pentru operații cu două multimii (incluere, reuniune, intersecție și diferență) nu există metode în clasele multime, dar există funcții externe cu argumente iterator. În felul acesta se pot realiza operații cu submultimii și nu numai cu multimii complete.

Parcurgerea elementelor unei multimii sau unui dicționar (pentru afișarea lor, de exemplu) se poate face numai folosind iteratori, pentru că nu este posibil accesul prin indici (pozițional) la elementele acestor containere.

Elementele unui dicționar sunt obiecte de tip “pair” (pereche de obiecte), clasă STL definită astfel:

```

template <class T1,class T2> class pair {
public:
    T1 first; T2 second;          // date publice ale clasei
    pair (T1 & x, T2 & y): first(x),second(y) { }    // constructor de obiecte pair
};

```

Pentru dicționare, primul membru al perechii (“first”) este cheia, iar al doilea membru (“second”) este valoarea asociată cheii.

Metoda “insert” adaugă o pereche cheie-valoare la dicționar, metoda “find” determină poziția acelei perechi care conține o cheie dată, iar obținerea valorii asociate unei chei nu se face printr-o metodă ci prin operatorul de selecție []. Ideea ar fi că accesul direct la o valoare prin cheie este similar accesului direct la elementul unui vector printr-un indice întreg, iar cheia este o generalizare a unui indice (cheia poate fi de orice tip, dar indicele nu poate fi decât întreg).

Exemplu de creare și afișare a unui dicționar cu frecvența de apariție a cuvintelor într-un fișier text:

```

#include "defstl.h"          // include fisiere antet ale tuturor claselor STL
int main () {
    map<string,int> dic;
    string cuv;
    map<string,int>::iterator it;
    ifstream in ("words.txt");    // un fișier text
    while (in >> cuv) {          // citește cuvânt din fișier
        it=dic.find(cuv);        // caută cuvânt în dicționar
        if (it != dic.end())      // dacă există acel cuvânt
            (*it).second++;       // se mărește numărul de apariții
        else                      // dacă nu există anterior acel cuvânt
            dic.insert ( pair<string,int>(cuv,1));    // atunci se introduce în dicționar
    }

    // afișare conținut dicționar
    for (it=dic.begin(); it !=dic.end(); it++)
        cout << (*it).first <<":" << (*it).second << endl;
}

```

Folosind operatorul de selecție [], ciclul principal din programul anterior se poate rescrie astfel:

```

while (in >> cuv)
    dic[cuv]++;

```

Operatorul de selecție `[]` caută cheia dată ca argument iar dacă nu o găsește introduce automat în dicționar un obiect de tip “pair”, folosind constructorul implicit al acestei clase (care, la rândul lui, apelează constructorul implicit pentru tipul “int” și inițializează cu zero contorul de apariții).

11.5 UTILIZAREA DE CLASE STL ÎN APLICATII

Aplicațiile care sunt mai ușor de scris și de citit cu clase STL sunt cele care folosesc colecții de colecții (vector de liste, de exemplu) și cele în care datele memorate într-o colecție au mai multe componente (cum ar fi un arbore Huffman în care fiecare nod conține un caracter, o frecvență și eventual codul asociat).

Înainte de a relua anumite aplicații folosind clase STL trebuie spus că în aceste exemple nu vom folosi toate facilitățile oferite de STL pentru a face exemplele mai ușor de înțeles. În aplicațiile STL se folosește frecvent funcția “copy”, inclusiv pentru afisarea conținutului unui container pe ecran (în fișierul standard de ieșire). Exemplu:

```
string ts[]={“unu”, “doi”, “trei”}; // un vector de siruri C
vector<string> vs(10); // un obiect vector de siruri C++
copy (ts, ts+3, vs.begin()); // copiere din ts in vs
copy (vs.begin(), vs.end(), ostream_iterator<string> (cout, “\n”)); // afisare
```

Aceeași operație o vom scrie cu un ciclu explicit folosind un obiect iterator sau operatorul de selecție (numai pentru vectori):

```
vector<string>::iterator it;
for (it=vs.begin(); it!=vs.end(); it++)
    cout << *it << ‘;’ ;
cout << endl;
```

Nu vom utiliza nici alte funcții STL (*for_each*, *count*, *equal*, s.a) care pot face programele mai compacte, dar care necesită explicații în plus și măresc diferența dintre programele C și programele C++ pentru aceleași aplicații. Vom menționa numai funcțiile *find* și *erase* deoarece sunt folosite într-un exemplu și arată care stilul de lucru specific STL.

Funcția *erase* elimină o valoare dintr-un container și necesită ca parametru poziția din container a valorii eliminate, deci un iterator. Obținerea poziției se poate face prin funcția *find*, deci prin căutarea unei valori într-un container sau, mai exact, între două poziții date dintr-un container (într-o subcolecție). Exemplu de funcție șablon care elimină o valoare dintr-un container de orice fel:

```
// elimina pe x din colectia c
template <class T1, class T2> void del (T1 & c, T2 x) {
    T1::iterator p;
    p= find(c.begin(), c.end(), x); // cauta pe x in toata colectia c
    if ( p != c.end()) // daca s-a gasit x in pozitia p
        c.erase(p); // atunci elimina element din pozitia p din colectia c
}
```

Clasele container existente pot fi utilizate ca atare sau în definirea unor alte clase container.

Clasa *vector* o vom folosi atunci când nu putem aprecia dimensiunea maximă a unui vector, deci vectorul trebuie să se extindă dinamic. În plus, utilizarea unui obiect *vector* în locul unui vector C ca argument în funcții are avantajul reducerii numărului de argumente (nu mai trebuie dată dimensiunea vectorului ca argument).

De observat că metoda “size” are ca rezultat numărul de elemente introduse în vector, iar metoda “capacity” are ca rezultat capacitatea vectorului (care se poate modifica în urma adăugărilor de elemente la vector). Capacitatea inițială poate fi sau specificată în constructorul obiectului *vector* sau în metoda “reserve”.

Programele cu clase STL pot fi simplificate folosind nume de tipuri introduse prin declaratia “typedef” sau prin directiva “define”.

Exemplul următor este o sortare topologică care folosește un vector de liste de conditionări, unde elementele sortate sunt numere întregi, dar programul se poate modifica ușor pentru alte tipuri de date (siruri de exemplu):

```
// tipul "clist" este o colectie de liste de intregi
typedef vector<list<int> > clist;
// creare liste de conditionari

void readdata ( clist & cond) {
    int n,x,y;
    cout << "Numar de valori: ";
    cin >> n;
    cond.reserve(n+1);           // aloca memorie ptr vector
    for (y=0;y<=n;y++)           // initializare vector de liste
        cond.push_back(* new list<int>());
    cout << "Perechi x y (x conditioneaza pe y) \n";
    while (cin >> x >> y)         // citeste o pereche (x,y)
        cond[y].push_back(x);     // adauga pe x la lista de conditii a lui y
}

// afisare liste de conditionari
void writedata (clist cond) {
    for (int y=1;y< cond.size();y++) { // pentru fiecare element y
        cout << y << ": ";           // scrie y
        // scrie lista de conditionari a lui y
        for (list<int>::iterator p=cond[y].begin(); p !=cond[y].end();p++)
            cout << *p << " ";
        cout << endl;
    }
}

// elimina x din toate listele colectiei clist
void eraseall (clist & cond, int x) {
    for (int y=1;y< cond.size();y++)
        cond[y].erase(find(cond[y].begin(),cond[y].end(),x));
}

// sortare topologica cu rezultat in vectorul t
bool topsort (clist cond, vector<int>& t) {
    int n =cond.size()-1;           // n = numar de elemente sortate
    vector<bool> sortat (n);        // ptr marcare elemente sortate
    bool gasit; int ns=0;           // ns = numar de elemente sortate
    do {
        gasit=false;
        for (int i=1; i<=n;i++)      // cauta un element nesortat, fara conditii
            if ( !sortat[i] && cond[i].empty() ) {
                gasit= sortat[i]=true; // s-a gasit un element
                t.push_back(i); ns++;   // se adauga la vector si se numara
                eraseall(cond,i);       // elimina elementul i din toate listele
            }
    } while (gasit);
    return ns < n? false: true;      // false daca sortare imposibila
}

void main () {
    clist c; vector<int> t;
    bool ok;
    readdata(c);                     // citire date si creare colectie de liste
    ok=topsort (c,t);                // sortare in vectorul t
}
```



```

if ( ! ok)
    cout << "sortare imposibila";
else
    // afisare vector (ordine topologica)
    for (int i=0;i< t.size();i++)
        cout << t[i] <<" ";
}

```

În stilul specific programării orientate pe obiecte ar trebui definite și utilizate noi clase în loc să scriem funcții ca în C. Programul anterior se poate rescrie cu o clasă pentru obiecte “sortator topologic”, având ca metodă principală pe “topsort”.

11.6 DEFINIREA DE NOI CLASE CONTAINER

Biblioteca STL conține un număr redus de clase container, considerate esențiale, și care pot fi folosite în definirea altor clase.

Un exemplu este definirea unei clase pentru colecții de mulțimi disjuncte ca vector de mulțimi:

```

// fisier DS.H
#include "defstl.h"
class sets {
    vector <set <int> > a;    // vector de mulțimi de întregi
public:
    sets(int n);            // constructor
    int find (int x);        // caută multimea ce conține pe x
    void unif (int x, int y); // reunire mulțimi care conțin pe x și pe y
    void print();           // afisare colecție de mulțimi
};

// fisier DS.CPP
#include "ds.h"
sets::sets(int n): a(n+1) { // constructor clasa sets
    for (int i=1;i<=n;i++) {
        a[i]= set<int>();    // construiește o mulțime a[i]
        a[i].insert(i);      // fiecare a[i] conține inițial pe i
    }
}

int sets::find (int x) {
    for (unsigned int i=1;i<a.size();i++) {
        set<int> s = a[i];
        if ( std::find(s.begin(),s.end(),x) != s.end()) // dacă x în a[i]
            return i ;                                // atunci întoarce i
    }
    return -1;    // dacă x negăsit în colecție
}

void sets::unif (int x, int y) {
    int ix,iy;
    set<int> r;
    set<int>::iterator p;
    ix= find(x); iy=find(y); // ix=nr mulțime ce conține pe x
    for ( p=a[iy].begin(); p!=a[iy].end(); p++)
        a[ix].insert ( *p);    // adaugă elementele din a[iy] la a[ix]
    a[iy].clear();             // și golește mulțimea a[iy]
}

void sets::print() {
    unsigned int v;
    for (v=1;v<a.size();v++) {
        set<int> s= a[v];
        if ( s.empty()) continue;
    }
}

```

```

    cout << v << ": ";
    copy (s.begin(),s.end(),ostream_iterator <int> (cout," ") );
    cout << "\n";
}
}

```

Un program pentru algoritmul Kruskal, care foloseste clase STL si clasa “arc” definită anterior, va arăta astfel:

```

#include "arc.h"
#include "ds.h"
#include "defstl.h"
int main () {
    deque<arc> graf;           // graf ca lista de arce
    vector<arc> mst;          // arbore ca vector de arce
    arc a;
    // citire lista de arce si costuri
    int n=0;                  // nr noduri
    while (cin >> a) {
        graf.push_back ( a);
        n=max<int> (n,max<int>(a.v,a.w));
    }
    sets ds(n);
    sort (graf.begin(), graf.end());    // ordonare arce dupa costuri
    // algoritmul greedy
    while ( !graf.empty()) {
        a=graf.front(); graf.pop_front();
        int x=ds.find(a.v);
        int y=ds.find(a.w);
        if ( x != y) {
            mst.push_back(a);
            ds.unif(a.v,a.w);
        }
    }
    // afisare arce din arbore de acoperire de cost minim
    cout << "\n";
    for ( vector<arc>::iterator p = mst.begin(); p != mst.end(); p++)
        cout << *p;           // operatorul << supradefinit in clasa arc
}

```

Definirea unor noi clase container, pentru arbori de exemplu, ar trebui să se facă în spiritul claselor STL existente, deci folosind iteratori si operatori supradefiniti pentru operatii cu date din container. Pentru simplificare, vom folosi aici metode pentru enumerarea datelor dintr-un container.

În exemplul următor se definește o clasă minimală pentru arbori binari de căutare, din care se poate vedea că pentru metode ce corespund unor algoritmi recursivi trebuie definite functii auxiliare recursive (metodele nu pot fi recursive deoarece nu au ca argument rădăcina (sub)arborelui prelucrat):

```

#include "stldef.h"
// un nod de arbore binar
template <class T> class tnode {
public:
    T val;           //date din nodul respectiv
    tnode<T> * left, *right;    // pointeri la succesori
    // constructor de nod frunza
    tnode<T> ( T x) : val(x) {
        left=right=NULL;
    }
};

```

```

// un arbore binar de cautare
template <class T > class bst {
    tnode<T>* root;
    // functie nepublica recursiva de afisare
    void iprint (tnode<T>* r, int sp){
        if (r != NULL) {
            for (int i=0;i<sp;i++) cout<<' ';    // indentare cu sp spatii
            cout << r->val << endl;              // valoare din nodul curent
            iprint(r->left,sp+2);                 // afisare subarbore stanga
            iprint(r->right,sp+2);                // afisare subarbore dreapta
        }
    }
    // functie nepublica recursiva de adaugare
    void iadd (tnode<T>*& r, T x) {
        if (r==NULL)                            // daca arbore vid
            r=new tnode<T>(x);                  // creare nod nou
        else {
            if (x==r->val)                        // daca x exista
                return;                          // atunci nu se mai adauga
            if (x < r->val)
                iadd (r->left,x);                 // adauga la subarbore stanga
            else
                iadd (r->right,x);                // adauga la subarbore dreapta
        }
    }
public:
    bst ( ) { root=NULL; }                     // un constructor
    void print() { iprint(root,0);}             // afisare infixata arbore
    void add ( T x ) {iadd (root,x); }          // adaugare la arbore
};

// utilizare clasa bst
void main () {
    bst<int> t;
    for ( int x=1;x<10;x++)
        t.add(rand());
    t.print();
}
    
```

PROBLEME PROPUSE

1. Să se scrie funcțiile init, print, get, put pentru operații cu un dicționar realizat ca un tabel de dispersie (hash table) în două variante:

- vector de pointeri la liste înlănțuite de sinonime.
- un singur vector, cu plasare sinonime în prima adresă liberă.

Să se utilizeze acest dicționar ca tabel de simboluri într-un program interpretor de instrucțiuni de atribuire simple de forma $v=c|v$ (v =nume de variabilă, c =constanta numerică). Tabelul de simboluri conține numele și valoarea fiecărei variabile și va fi afișat la finalul programului.

2. Să se rescrie funcțiile pentru operații cu un dicționar pentru un dicționar realizat ca o listă înlănțuită de perechi cheie-valoare și să se folosească în problema anterioară (interpretarea unor atribuiri simple).

3. Să se scrie funcții pentru operații cu o mulțime cu valori multiple ("multiset"):

```
int add ( Mset m, T val, int n); // adauga n elemente cu valoarea val la multimea m
int count (Mset m, T val);      // numarul de valori egale cu val din multimea m
int equals (Mset m1, Mset m2); // compara la egalitate multimele m1 si m2
```

Tipul "Mset" se va implementa prin doi vectori sau printr-un vector de structuri cu 2 câmpuri.

4. Problema lui Josephus: se porneste cu o listă de n numere situate într-un cerc și se elimină elementul găsit la distanța m față de poziția curentă (deplasând la stânga elementele următoare pentru a păstra lista compactă), repetând eliminarea până când rămâne un singur element în listă. Fie lista inițială:

a b c d e f g h

și $m=2$. Dacă se numără de la primul element atunci evoluția listei va fi în acest caz:

Lista	poziție curentă	Element eliminat
a b c d e f g h	3	c
a b d e f g	5	f
a b d e g	2	b
a d e g	4	g
a d e	3	e
a d	1	a
d		

Să se scrie două programe pentru afișarea și eliminarea succesivă dintr-o listă circulară a valorilor aflate la distanța m , folosind un vector și respectiv o listă cu pointeri pentru implementarea listei.

5. Să se scrie și să se verifice funcții pentru operații cu o listă "deque" de pointeri (cu acces la ambele capete), realizată ca o listă dublu-înlănțuită.

6. Să se scrie funcții pentru operații cu o listă "unrolled" ordonată (cu vectori de m valori în fiecare nod al listei înlănțuite), după modelul arborilor B: la umplerea unui nod se creează un nod nou și se împart cele $m+1$ valori în mod egal între cele două noduri. La eliminarea unei valori se pot elimina noduri goale (fără nici o valoare).

7. Să se scrie și să se verifice o funcție pentru interclasarea a două liste înlănțuite ordonate de numere întregi într-o singură listă ordonată. Să se scrie și să se verifice o funcție pentru adunarea a două polinoame reprezentate prin liste înlănțuite de termeni. Un element din listă memorează coeficientul și puterea lui x din termenul respectiv. Listele sunt ordonate descrescător după puterile lui x . Exemplu:

$P(x) = 5x^3 - 7x + 9$ (lista conține perechile [5,3], [-7,1] și [9,0])

$Q(x) = x^4 + 2x^3 + 4x^2 - 6$ (lista contine [1,4],[2,3],[4,2],[-6,0])
 $R(x)=P(x)+Q(x)$ va fi în lista [1,4],[7,3],[4,2],[-7,1],[3,0]

8. Să se scrie si să se verifice functii pentru următoarele operatii cu multimi de numere întregi (neordonate) realizate ca liste înlătuite:

- initializare multime vida
- test de apartenenta a unui numar dat la o multime
- adaugarea unui numar dat la o multime
- eliminarea unui numar dat dintr-o multime
- afisarea elementelor unei multimi
- determinarea dimensiunii unei multimi (numărul de elemente)
- reuniunea a doua multimi
- diferenta a doua multimi
- intersectia a doua multimi

Sa se folosească aceste functii în problema acoperirii unei multimi scop S cu un număr minim de multimi candidat, folosind algoritmul următor:

```

repetă cat timp multimea scop S nu e vida {
    repetă pentru fiecare multime candidat k {
        intersectie multime candidat k si multime scop cu rezultat multime M
        reține dimensiunea maximă a unei multimi M si număr candidat kmax
    }
    diferența între multimea scop si multimea candidat k
    scrie k si golește sau elimină multime candidat k
}
    
```

9. Să se scrie functii pentru operatii de acces pozitional (prin indice întreg) la liste înlătuite de numere întregi:

```

int first (List lst, int x);    // determina pozitia primei aparitii a lui x in lista lst
int last (List lst, int x);    // determina pozitia ultimei aparitii a lui x in lista lst
int getP (List lst, int k);    // valoarea elementului din pozitia k din lista lst
void delP (List & lst, int k); // elimina elementul din pozitia k a listei lst
void insP (List & lst, int k, int x); // insertie x inainte de pozitia k in lista lst
void setP (List & lst, int k, int x); // inlocuieste valoarea elementului din pozitia k prin x
    
```

Să se scrie o functie “swap” (interschimb), folosită pentru “amestecarea” elementelor unei liste astfel ca să nu existe nici o ordine în listă (operatie inversă sortării):

```

void shuffle (List lst) { // creare dezordine in lista lst
    int i,j; int n=sizeL(lst);    // n= numar de elemente din lista lst
    for (i=n-1;i>0;i--) {
        j=rand()%n;    // j este un numar aleator intre 0 si n-1
        swap(lst,i,j); // schimba intre ele elementele din pozitiile i si j din lst
    }
}
    
```

10. Să se scrie si să se verifice functii pentru operatii cu o listă “skip” de numere, în care fiecare nod de listă este definit astfel:

```

typedef struct Node {
    int val;    // valoare din nod
    int level;  // numar de pointeri pe nod
    struct Node *leg[1]; // vector de pointeri extensibil dinamic
} Node;
    
```

Operatii : initializare, insertie în listă, eliminare din listă, afisare listă, căutare.

11. Să se scrie si să se verifice functii pentru operatii cu o listă "skip" de numere, în care fiecare nod de listă contine o listă de pointeri si este definit astfel:

```
typedef struct Node {
    int val;
    struct Node *right;    // legatura la elementul urmator de pe acelasi nivel
    struct Node *down;    // legatura la un nod de pe nivelul urmator
} Node;
```

Operatii cu lista skip: initializare, insertie în listă, afisare listă.

12. Să se definească tipul "Stack" si functii pentru operatii cu o stivă de pointeri (de tip void*) realizată ca o listă înlănțuită:

```
void initSt ( Stack &);
void push ( Stack &, void *);
void* pop (Stack&);
int emptySt (Stack);
void* top (Stack);
```

Să se folosească functiile anterioare într-un program care verifică dacă un fisier XML este corect formal ("well-formed") folosind o stivă de pointeri la siruri de caractere ("tag"-uri XML). O structură corectă de marcaje este ca si o structură de paranteze: fiecare pereche de marcaje trebuie să fie inclusă complet într-o altă pereche. Erorile formale ce trebuie detectate de program sunt:

- absenta unui marcaj de început sau de sfârșit
 - perechi de marcaje care se "intersectează":
- ```
<nume> cdrw <tip> NEC </nume> </tip>
```

13. Să se scrie o functie pentru evaluarea recursivă a unei expresii aritmetice prefixate si o functie pentru evaluarea nerecursivă, folosind o stivă a expresiei prefixate (prin analogie cu evaluarea expresiilor postfixate).

14. Să se scrie functii care pun pe stivă si scot din stivă câte doi întregi si să se folosească în functii nerecursive pentru sortare rapidă si respectiv pentru umplere prin inundare:

```
// umplere prin inundare (nerecursiv, cu stiva)
void fill (int x, int y) {
 stiva st;
 sinit(st); // initializare stiva
 push (st,x,y); // pune x si y pe stiva
 while (! empty (st)) {
 pop (st,x,y); // scoate din stiva in x si y
 if (a[x][y] ==0)
 a[x][y]=1;
 if (a[x-1][y]==0)
 push(st,x-1,y);
 if (a[x+1][y]==0)
 push(st,x+1,y);
 if (a[x][y-1]==0)
 push(st,x,y-1);
 if (a[x][y+1]==0)
 push (st,x,y+1);
 }
}
```

15. Să se scrie o funcție nerecursivă pentru umplere prin inundare a unei suprafețe (pentru modificarea valorii unor elemente dintr-o matrice care reprezintă puncte din figură) folosind o stivă matrice și apoi o stivă de structuri pentru salvarea argumentelor ce reprezintă coordonatele punctului curent. Se pleacă de la funcția recursivă următoare:

```
int px[4] = {-1,0,1,0}; // directii de extindere pe x
int py[4] = {0,1,0,-1}; // directii de extindere pe y

int posibil (int i,int j) { // daca se poate extinde in punctul (i,j)
 if (i<1 || i>n || j<1 || j>n)
 return 0;
 return a[i][j]==0 ; // a[i,j]=0 daca necolorat si a[i,j] >0 daca colorat
}

// umplere incepand din punctul (i,j)
void fill (int i, int j) {
 int k, x, y;
 for (k=0;k<4;k++) { // 4 posibilitati de extindere
 x=i+px[k]; y= j+py[k]; // din (i,j) se merge in (x,y)
 if (posibil(x,y)) { // daca e posibil
 a[x][y]=8; // pune culoarea 8 in a[x,y]
 fill (x,y); // continua extindere din (x,y)
 }
 }
}
```

16. Clasa “LinkedHashSet” din Java implementează un tabel de dispersie completat cu o listă înlăntuită care menține ordinea de adăugare a elementelor la multime. Tabelul de dispersie este un vector de pointeri la liste de sinonime. Fiecare element al multimii are doi pointeri: adresa următorului element din lista ce menține ordinea de adăugare și adresa următorului element din lista de sinonime. Să se definească o structură pentru o astfel de multime și funcții pentru următoarele operații cu multimii de întregi: initializare, adăugare element la multime, eliminare element, dimensiune multime, verificare de apartenență a unui element dat la multime.

17. Să se scrie funcții pentru :

- determinarea înălțimii unui arbore-vector heap cu număr cunoscut de valori (noduri) (un vector cu un element are înălțimea 1)
  - afisarea unui vector heap în lățime (nivel cu nivel, fiecare nivel pe o linie separată)
  - afisarea unui vector heap în adâncime (prefixat), cu indentare: fiecare valoare apare pe o linie separată precedată de 3m spații
- m (=0,1,...) este nivelul din arbore pe care se află o valoare.

Pentru verificare se va folosi un vector de constante initializat în funcția “main”. Ex:

```
int h[]={0,16,14,10,8,7,3,9,6,4,5,2,1}; int n=12; // n= număr de noduri
```

18. Funcții pentru determinarea succesorului și predecesorului unei valori date dintr-un arbore binar de căutare (succesor=valoarea imediat următoare, predecesor=valoarea imediat inferioară). Funcția poate primi adresa nodului cu valoare cunoscută. Funcții pentru eliminarea nodului cu valoare minimă (maximă) dintr-un arbore binar de căutare.

19. Să se scrie o funcție care să transforme un arbore de căutare dat într-un arbore de căutare perfect echilibrat folosind următoarele funcții:

- Funcție pentru crearea unui vector ordonat cu valorile din nodurile unui arbore binar de căutare, prin explorare infixată;
- Funcție pentru crearea unui arbore binar ordonat perfect echilibrat pe baza unui vector ordonat cu valorile din nodurile arborelui (se adaugă recursiv valorilor mediane din acest vector, și din

subvectorii săi, la arbore ).

20. Să se scrie și să se verifice o funcție pentru crearea unui arbore binar dintr-o expresie aritmetică în forma postfixată (prefixată) în două variante:

- folosind o stivă
- recursiv

21. Funcție pentru eliberarea memoriei ocupate de un arbore binar (și respectiv de un arbore general).

22. Funcție pentru transformarea unui arbore binar cu pointeri într-un vector de structuri cu câte 3 câmpuri: valoare nod interior, valoare fiu stanga, valoare fiu dreapta (pentru serializarea unui arbore cu pointeri). Absența unui fiu este marcată prin valoarea zero. Exemplu: arborele binar de căutare construit cu valorile 5,3,7,6,4,2,8,9 va fi : 5 3 7 / 3 2 4 / 7 6 8 / 8 0 9

23. Funcții pentru operații cu arbori binari de căutare reprezentați printr-o structură cu 3 vectori și dimensiunea lor; vectorii conțin valorile din noduri, indici fii stânga și indici fii dreapta: initializare arbore, adăugarea unei valori la arbore, afisare prefixată cu indentare. Exemplu pentru un arbore cu valori întregi:

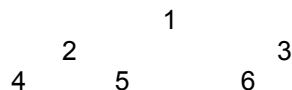
```
struct tree { int n; int val[]; int st[]; int dr[];};
```

24. Funcții pentru operații cu arbori binari reprezentați printr-un singur vector, de valori, după modelul unui vector "heap". Vectorul conține în prima poziție rădăcina, după care valorile de pe fiecare nivel, cu -1 pentru valorile (nodurile) care lipsesc din arbore. Se vor scrie funcții pentru adăugarea unei valori și pentru afisarea valorilor din arbore, nivel cu nivel în lățime) și prefixat (în adâncime).

25. Să se scrie o funcție pentru determinarea adresei nodului părinte al unui nod dat, în două variante:

- se dă valoarea unui nod;
- se dă adresa unui nod

26. Funcție pentru adăugarea de numere întregi la un arbore binar neordonat prin completare nivel cu nivel, începând de la rădăcină. Se va utiliza o coadă cu adresele ultimelor noduri adăugate la arbore. Exemplu de arbore creat cu valorile 1,2,3,4,5,6 :



27. Funcții pentru operații cu un arbore binar reprezentat printr-o listă înlănțuită de valori și un vector de biți care arată prezența sau absența unui nod dintr-un arbore complet. Operațiile sunt: adăugare valoare la arbore și afisare arbore (prefixat și cu indentare).

28. Funcții nerecursive pentru afisarea prefixată, cu indentare, a valorilor dintr-un arbore binar și respectiv dintr-un arbore general.

29. Funcție nerecursivă pentru căutarea unei valori date într-un arbore general în fiecare din cele două variante de implementare: cu vector de fii și cu fiu-frate.

30. Să se scrie o funcție pentru afisarea unui arbore multicăi printr-o expresie cu paranteze. Funcția primește adresa nodului rădăcină. Exemplu:

```
1 (2(21), 3(31(311,312), 32), 4, 5(51, 52(521, 522), 53(531))))
```

Variante:



- Arbore cu vector de pointeri la fii
- Arbore cu două adrese: fiu stânga și frate dreapta

31. Program pentru crearea și afisarea unui arbore multicăi pe baza unor linii ce conțin fiecare valoarea unui nod și valorile fiilor săi. Aceste linii pot apărea în orice ordine. Exemplu de date:

2 21 22 23 / 31 311 312 / 3 31 32 / 1 2 3 / 22 221

Arborele afisat prefixat:

```
1
 2
 21
 22
 221
 23
3
 31
 311
 312
 32
```

32. Program pentru crearea și afisarea unui arbore multicăi pe baza unor linii ce conțin fiecare câte o pereche de valori părinte-fiu. Exemplu de date:

1 2 / 1 3 / 2 21 / 2 22 / 2 23 / 3 31 / 3 32 / 22 221 / 31 311 / 31 312

Arborele este același de la problema anterioară.

33. Program pentru crearea și afisarea unui arbore multicăi pe baza unor linii ce conțin secvența de valori de pe calea de la rădăcină la fiecare nod frunză. Exemplu de date:

1 2 21 / 1 2 22 221 / 1 2 23 / 1 3 31 311 / 1 3 31 312

Arborele este același de la problema anterioară.

34. Program pentru crearea și afisarea arborelui DOM cu structura unui fisier XML; nodurile arborelui conțin marcaje de început și date încadrate de marcaje, cu relațiile de subordonare indicate de includerea perechilor de marcaje. Se va defini și folosi o funcție care extrage următorul element dintr-un fisier XML (un element este un marcaj sau un text dintre marcaje).

35. Funcție pentru crearea unui fisier XML pe baza arborelui creat de un parser XML compatibil DOM (serializare arbore DOM).

36. Program pentru crearea unui arbore cu numele fișierelor dintr-un director și din toate subdirectoarele sale. Se va verifica prin afisarea arborelui, cu indentare.

Exemplu de funcție pentru afisarea numelor fișierelor dintr-un director (de pe o cale) și din subdirectoarele sale (fără crearea unui arbore), în Dev-Cpp :

```
void dir (char * path, int sp) { // sp spatii inainte de nume fisier
 struct _finddata_t fb; // tip definit in <io.h>
 int done=0; int i;
 char tmp[256]; char* p; // ptr creare cale la fisiere din subdirector
 long first; // folosit intre findfirst si findnext
 first = _findfirst(path,&fb);
 while (done==0) {
 if (fb.name[0] != '.') // nu se afiseaza numele "." si ".."
 printf ("%*c %-12s \n",sp,' ', fb.name); // scrie nume fisier (subdirector)
 if (fb.attrib == _A_SUBDIR && fb.name[0] != '.') { // daca este subdirector
 i = strrchr(path,'/') - path; // pozitie '/' in calea "path"
 strncpy(tmp,path,i+1);
 tmp[i+1]=0;
 strcat(tmp,fb.name); strcat(tmp,"/*.*"); // creare sir de forma cale/*.* in "tmp"
 dir (tmp,sp+3); // apel recursiv ptr subdirector
 }
 done = !fb.name[0];
 }
}
```

```

 }
 done=_findnext (first,&fb); // urmatorul fisier de pe calea "path"
 }
}

```

37. Functie care verifică dacă un digraf dat este aciclic prin modificarea functiei de vizitare DFS. Se verifică la fiecare nod vizitat dacă el a fost vizitat anterior (dacă s-a ajuns la el din nou printr-un arc înapoi).

38. Functie care verifică dacă un graf dat este conex prin vizitare DFS (dacă s-au vizitat toate vârfurile plecând dintr-un vârf oarecare).

39. Program de sortare topologică a unui graf orientat cu vizitare DFS si utilizarea unei stive.

40. Program pentru afisarea tipurilor de arce dintr-un graf orientat prin vizitare în adâncime (arce de arbore, de înaintare, de revenire si de traversare).

41. Program pentru determinarea "diametrului" unui graf neorientat fără costuri; diametrul este cea mai mare distantă dintre oricare două vârfuri din graf. Distanțele dintre vârfuri se obtin prin explorarea grafului în lărgime (BFS).

42. Functie pentru adăugarea unei valori la un arbore binar de căutare cu rotatia nodului rădăcină în caz că înălțimile celor doi subarbori diferă cu mai mult de 1, astfel ca să se reducă această diferență (numai pentru rădăcină).

43. Program pentru adăugarea de valori crescătoare (descrescătoare) la un arbore AVL, cu afisarea arborelui si înălțimii sale după fiecare adăugare.

44. Program pentru adăugarea de valori crescătoare (descrescătoare) la un arbore Treap, cu afisarea arborelui si înălțimii sale după fiecare adăugare.

45. Program pentru adăugarea de valori crescătoare (descrescătoare) la un arbore RB, cu afisarea arborelui si înălțimii sale după fiecare adăugare.