CUPRINS

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 necesarã încã de la începuturile programãrii calculatoarelor. Prima structurã de date folositã a fost structura de vector (tabel), utilizatã în operatiile de sortare (de ordonare) a colectiilor si prezentã în primele limbaje de programare pentru aplicatii numerice (Fortran si Basic).

Un vector este o colectie de date de acelasi tip, în care elementele colectiei sunt identificate prin indici ce reprezintã pozitia relativã a fiecãrui element în vector.

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

Introducerea tipurilor pointer si alocãrii dinamice de memorie în limbajele Pascal si C a permis utilizarea de vectori cu dimensiuni stabilite si/sau modificate în cursul executiei programelor.

Gruparea mai multor date, de tipuri diferite, într-o singurã entitate, numitã “articol” (“record”) în Pascal sau “structurã” în C a permis definirea unor noi tipuri de date de cãtre programatori si utilizarea unor date dispersate în memorie, dar legate prin pointeri : liste înlãntuite, arbori si altele. Astfel de colectii se pot extinde dinamic pe mãsura necesitãtilor si permit un timp mai scurt pentru anumite operatii, cum ar fi operatia de eliminare a unei valori dintr-o colectie.

Limbajul C asigurã structurile de date fundamentale (vectori, pointeri, structuri) si posibilitatea combinãrii acestora în noi tipuri de date, care pot primi si nume sugestive prin declaratia *typedef* .

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

1.2 CLASIFICÃRI ALE STRUCTURILOR DE DATE

O structurã de date este caracterizatã prin relatiile dintre elementele colectiei si prin operatiile posibile cu aceastã colectie. Literatura de specialitate actualã identificã mai multe feluri de colectii (structuri de date), care pot fi clasificate dupã câteva criterii.

Un criteriu de clasificare foloseste relatiile dintre elementele colectiei:
- Colectii liniare (secvente, liste), în care fiecare element are un singur succesor si un singur predecesor;
- Colectii arborescente (ierarhice), în care un element poate avea mai multi succesori (fii), dar un singur predecesor (pãrinte);
- Colectii neliniare generale, în care relatiile dintre elemente au forma unui graf general (un element poate avea mai multi succesori si mai multi predecesori).

Un alt criteriu grupeazã diferitele colectii dupã rolul pe care îl au în aplicatii si dupã operatiile asociate colectiei, indiferent de reprezentarea în memorie, folosind notiunea de tip abstract de date:
- Structuri de cãutare (multimi si dictionare abstracte);
- Structuri de pãstrare temporarã a datelor (containere, liste, stive, cozi s.a.)

Un alt criteriu poate fi modul de reprezentare a relatiilor dintre elementele colectiei:
- Implicit, prin dispunerea lor în memorie (vectori de valori, vectori de biti, heap);
- Explicit, prin adrese de legãturã (pointeri).

Dupã numãrul de aplicatii în care se folosesc putem distinge între:
- Structuri de date de uz general ;
- Structuri de date specializate pentru anumite aplicatii (geometrice, cu imagini).

Organizarea datelor pe suport extern (a fisierelor si bazelor de date) prezintã asemãnãri dar si diferente fatã de organizarea datelor în memoria internã, datoritã particularitãtilor de acces la discuri fatã de accesul la memoria internã.

Un fisier secvential corespunde oarecum unui vector, un fisier de proprietãti este în fond un dictionar si astfel de paralele pot continua. Pe suport extern nu se folosesc pointeri, dar se pot folosi adrese relative în fisiere (ca numãr de octeti fatã de începutul fisierului), ca în cazul fisierelor index.

Ideea unor date dispersate dar legate prin pointeri, folositã la liste si arbori, se foloseste mai rar pentru fisiere disc pentru cã ar necesita acces la articole neadiacente (dispersate în fisier), operatii care consumã timp pe suport extern. Totusi, anumite structuri arborescente se folosesc si pe disc, dar ele tin seama de specificul suportului: arborii B sunt arbori echilibrati cu un numãr mic de noduri si cu numãr mare de date în fiecare nod, astfel ca sã se facã cât mai putine citiri de pe disc.

Salvarea unor structuri de date interne cu pointeri într-un fisier disc se numeste serializare, pentru cã în fisier se scriu numai date (într-o ordine prestabilitã), nu si pointeri (care au valabilitate numai pe durata executiei unui program). La încãrcarea în memorie a datelor din fisier se poate reconstrui o structurã cu pointeri (în general alti pointeri la o altã executie a unui program ce foloseste aceste date).

Tot pe suport extern se practicã si memorarea unor colectii de date fãrã o structurã internã (date nestructurate), cum ar fi unele fisiere multimedia, mesaje transmise prin e-mail, documente, rapoarte s.a. Astfel de fisiere se citesc integral si secvential, fãrã a necesita operatii de cãutare în fisier.

1.3 TIPURI ABSTRACTE DE DATE

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

Pentru programele nebanale este utilã o abordare în (cel putin) douã etape:
- o etapã de conceptie (de proiectare), care include alegerea tipurilor abstracte de date si algoritmilor necesari;
- o etapã de implementare (de codificare), care include alegerea structurilor concrete de date, scrierea de cod si folosirea unor functii de bibliotecã.

In faza de proiectare nu trebuie stabilite structuri fizice de date, iar aplicatia trebuie gânditã în tipuri abstracte de date. Putem decide cã avem nevoie de un dictionar si nu de un tabel de dispersie, putem alege o coadã cu prioritãti abstractã si nu un vector heap sau un arbore ordonat, s.a.m.d.

In faza de implementare putem decide ce implementãri alegem pentru tipurile abstracte decise în faza de proiectare. Ideea este de a separa interfata (modul de utilizare) de implementarea unui anumit tip de colectie. In felul acesta se reduc dependentele dintre diferite pãrti ale unui program si se faciliteazã modificãrile care devin necesare dupã intrarea aplicatiei în exploatare.

Conceptul de tip abstract de date are un corespondent direct în limbajele orientate pe obiecte, si anume o clasã abstractã sau o interfatã. In limbajul C putem folosi acelasi nume pentru tipul abstract si aceleasi nume de functii; înlocuirea unei implementãri cu alta poate însemna un alt fisier antet (cu definirea tipului) si o altã bibliotecã de functii, dar fãrã modificarea aplicatiei care foloseste tipul abstract. Un tip de date abstract poate fi implementat prin mai multe structuri fizice de date.

Trebuie spus cã nu existã un set de operatii unanim acceptate pentru fiecare tip abstract de date, iar aceste diferente sunt uneori mari, ca în cazul tipului abstract "listã" (asa cum se pot vedea comparând bibliotecile de clase din C++ si din Java).

Ca exemplu de abordare a unei probleme în termeni de tipuri abstracte de date vom considera verificarea formalã a unui fisier XML în sensul utilizãrii corecte a marcajelor ("tags"). Exemplele care urmeazã ilustreazã o utilizare corectã si apoi o utilizare incorectã 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 marcajele singulare, de forma <tag/>.

Algoritmul de verificare a unui fisier XML dacã este corect format foloseste tipul abstract “stivã” (“stack”) astfel: pune într-o stivã fiecare marcaj de început (<stud>, <nume>,...), iar la citirea unui marcaj de sfârsit (</stud>, </nume>,...) verificã dacã în vârful stivei este marcajul de început pereche si îl scoate din stivã :

```
initializare stiva
  repetã pânã la sfârsit de fisier
     extrage urmãtorul marcaj
     daca marcaj de început
        pune marcaj în stivã
     dacã marcaj de sfârsit
        dacã în varful stivei este perechea lui
           scoate marcajul din vârful stivei
        altfel
           eroare de utilizare marcaje
  daca stiva nu e goalã
     eroare de utilizare marcaje
```

In aceastã fazã nu ne intereseazã dacã stiva este realizatã ca vector sau ca listã înlãntuitã, dacã ea contine pointeri generici sau de un tip particular.

Un alt exemplu este tipul abstract “multime”, definit ca o colectie de valori distincte si având ca operatie specificã verificarea apartenentei unei valori la o multime (deci o cãutare în multime dupã valoare). In plus, existã operatii generale cu orice colectie : initializare, adãugare element la o colectie, eliminare element din colectie, afisare sau parcurgere colectie, s.a. Multimile se pot implementa prin vectori de valori, vectori de biti, liste înlãntuite, arbori binari si tabele de dispersie (“hash”).

In prezent sunt recunoscute câteva tipuri abstracte de date, definite prin operatiile specifice si modul de utilizare: multimi, colectii de multimi disjuncte, liste generale, liste particulare (stive,cozi), cozi ordonate (cu prioritãti), dictionare. Diferitele variante de arbori si de grafuri sunt uneori si ele considerate ca tipuri abstracte.

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

Conceperea unui program cu tipuri abstracte de date permite modificarea implementãrii colectiei abstracte (din motive de performantã, de obicei), fãrã modificarea restului aplicatiei.

Ca exemplu de utilizare a tipului abstract dictionar vom considera problema determinãrii frecventei de aparitie a cuvintelor într-un text. Un dictionar este o colectie de perechi cheie-valoare, în care cheile sunt unice (distincte). In exemplul nostru cheile sunt siruri (cuvinte), iar valorile asociate sunt numere întregi ce aratã de câte ori apare fiecare cuvânt în fisier.

Aplicatia poate fi descrisã astfel:

```
initializare dictionar
repetã pânã la sfârsit de fisier
   extrage urmãtorul cuvant
   dacã cuvantul existã în dictionar
      aduna 1 la numãrul de aparitii
   altfel
      pune in dictionar cuvant cu numãr de aparitii 1
afisare dictionar
```

Implementarea dictionarului de cuvinte se poate face printr-un tabel hash dacã fisierele sunt foarte mari si sunt necesare multe cãutãri, sau printr-un arbore binar de cãutare echilibrat dacã se cere afisarea sa numai în ordinea cheilor, sau printr-un vector (sau doi vectori) dacã se cere afisarea sa ordonatã si dupã valori (dupã numãrul de aparitii al fiecãrui cuvânt).

Existenta unor biblioteci de clase predefinite pentru colectii de date reduce problema implementãrii structurilor de date la alegerea claselor celor mai adecvate pentru aplicatia respectivã si conduce la programe compacte si fiabile. "Adecvare" se referã aici la performantele cerute si la particularitãtile aplicatiei: dacã se cere mentinerea colectiei în ordine, dacã se fac multe cãutari, dacã este o colectie staticã sau volatilã, etc.

1.4 EFICIENTA STRUCTURILOR DE DATE

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

Un bun exemplu este cel al structurilor de date folosite atunci când sunt necesare cãutãri frecvente într-o colectie de date dupã continut (dupã chei de cãutare); cãutarea într-un vector sau într-o listã înlãntuitã este ineficientã pentru un volum mare de date si astfel au apãrut tabele de dispersie (“hash table”), arbori de cãutare echilibrati, arbori B si alte structuri optimizate pentru operatii de cãutare.

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

Influenta alegerii structurii de date asupra timpului de executie a unui program stã si la baza introducerii tipurilor abstracte de date: un program care foloseste tipuri abstracte poate fi mai usor modificat prin alegerea unei alte implementãri a tipului abstract folosit, pentru îmbunãtãtirea performantelor.

Problema alegerii unei structuri de date eficiente pentru un tip abstract nu are o solutie unicã, desi existã anumite recomandãri generale în acest sens. Sunt mai multi factori care pot influenta aceastã alegere si care depind de aplicatia concretã.

Astfel, o structurã de cãutare poate sau nu sã pãstreze si o anumitã ordine între elementele colectiei, ordine cronologicã sau ordine determinatã de valorile memorate. Dacã nu conteazã ordinea atunci un tabel de dispersie (“hash”) este alegerea potrivitã, dacã ordinea valoricã este importantã atunci un arbore binar cu autoechilibrare este o alegere mai bunã, iar dacã trebuie pãstratã ordinea de introducere în colectie, atunci un tabel hash completat cu o listã coadã este mai bun.

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

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

In fine, dimensiunea colectiei poate influenta alegerea structurii adecvate: o structurã cu pointeri (liste de adiacente pentru grafuri, de exemplu) este bunã pentru o colectie cu numãr relativ mic de elemente si care se modificã frecvent, iar o structurã cu adrese succesive (o matrice de adiacente, de exemplu) poate fi preferabilã pentru un numãr mare de elemente.

Eficienta unei anumite structuri este determinatã de doi factori: memoria ocupatã si timpul necesar pentru operatiile frecvente. Mai des se foloseste termenul de “complexitate”, cu variantele “complexitate temporalã” si “complexitate spatialã”.

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

Complexitatea temporalã a unui algoritm se estimeazã 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 crescãtoare, de exemplu, cazul cel mai defavorabil este ca datele sã fie ordonate descrescãtor (sau crescãtor pentru metoda "quicksort"). Cazul mediu este al unui vector de numere generate aleator, iar cazul minim al unui vector deja ordonat.

In general, un algoritm care se comportã mai bine în cazul cel mai nefavorabil se comportã mai bine si în cazul mediu, dar existã si exceptii de la aceastã regulã cum este algoritmul de sortare rapidã QuickSort, care este cel mai bun pentru cazul mediu (ordine oarecare în lista initialã), dar se poate comporta slab pentru cazul cel mai nefavorabil (functie si de modul de alegere a elementului pivot).

Pentru a simplifica compararea eficientei algoritmilor se apreciazã volumul datelor de intrare printr-un singur numãr întreg N, desi nu orice problemã poate fi complet caracterizatã de un singur numãr. De exemplu, în problemele cu grafuri conteazã atât numãrul de noduri din graf cât si numãrul de arce din graf, dar uneori se considerã doar numãrul arcelor ca dimensiune a grafului (pentru cele mai multe aplicatii reale numãrul de arce este mai mare ca numãrul nodurilor).

O altã simplificare folositã în estimarea complexitãtii unui algoritm considerã cã toate operatiile de prelucrare au aceeasi duratã si cã putem numãra operatii necesare pentru obtinerea rezultatului fãrã sã ne intereseze natura acelor operatii. Parte din aceastã simplificare este si aceea cã toate datele prelucrate se aflã în memoria internã si cã necesitã acelasi timp de acces.

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

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

$T(n) = O(f(n))$ dacã $T(n) <= k * f(n)$ pentru orice $n > n0$

unde k este o constantã a cãrei importantã scade pe mãsurã ce n creste.

Relatia anterioarã spune cã rata de crestere a timpului de executie a unui algoritm T(n) în raport cu dimensiunea n a problemei este inferioarã ratei de crestere a functiei f(n). De exemplu, un algoritm de complexitate O(n) este un algoritm al cãrui timp de executie creste liniar (direct proportional) cu valoarea lui n.

Majoritatea algoritmilor utilizati au complexitate polinomialã, deci $f(n) = n^k$. Un algoritm liniar are complexitate O(n), un algoritm pãtratic 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 complexitãti este cu atât mai mare cu cât n este mai mare. Tabelul urmãtor aratã cum creste 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 echivalentã 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.

Dacã durata unei operatii nu depinde de dimensiunea colectiei, atunci se spune cã acea operatie are complexitatea O(1); exemple sunt operatiile de introducere sau de scoatere din stivã, care opereazã la vârful stivei si nu depind de adâncimea stivei. Un timp constant are si operatia de apartenentã a unui element la o multime realizatã ca un vector de biti, deoarece se face un calcul pentru determinarea pozitiei elementului cãutat si o citire a unui bit (nu este o cãutare prin comparatii repetate).

Operatiile de cãutare secventialã într-un vector neordonat sau într-o listã înlãntuitã au o duratã proportionalã cu lungimea listei, deci complexitate O(n) sau liniarã.

Cãutarea binarã într-un vector ordonat si cãutarea într-un arbore binar ordonat au o complexitate logaritmicã de ordinul $O(log_2n)$, deoarece la fiecare pas reduce numãrul de elemente cercetate la jumãtate. Operatiile cu vectori heap si cu liste skip au si ele complexitate logaritmicã (logaritm de 2). Cu cât dimensiunea colectiei n este mai mare, cu atât este mai mare câstigul obtinut prin cãutare logaritmicã în raport cu cãutarea liniarã.

Cãutarea într-un arbore ordonat are o duratã proportionalã cu înãltimea arborelui, iar înãltimea este minimã în cazul unui arbore echilibrat si are valoarea log_2n , unde 'n' este numãrul de noduri din arbore. Deci complexitatea operatiei de cãutare într-un arbore binar ordonat si echilibrat este logaritmicã în raport cu numãrul de noduri (cu dimensiunea colectiei).

Anumite structuri de date au ca specific existenta unor operatii de duratã mare dar care se executã relativ rar: extinderea unui vector, restructurarea unui arbore, s.a. Dacã am lua durata acestor operatii drept cazul defavorabil si am însuma pe toate operatiile am obtine rezultate gresite pentru complexitatea algoritmilor de adãugare elemente la colectie. Pentru astfel de cazuri devine importantã analiza amortizatã a complexitãtii unor secvente de operatii, care nu este neapãrat egalã cu suma complexitãtilor operatiilor din secventã. Un exemplu simplu de analizã amortizatã este costul adãugãrii unui nou element la sfârsitul unui vector care se extinde dinamic.

Fie C capacitatea momentanã a unui vector dinamic. Dacã numãrul de elemente din vector N este mai mic ca C atunci operatia de adãugare nu depinde de N si are complexitatea O(1). Dacã N este egal cu C atunci devine necesarã extinderea vectorului prin copierea celor C elemente la noua adresã obtinutã. In caz cã se face o extindere cu un singur element, la fiecare adãugare este necesarã copierea elementelor existente în vector, deci costul unei operatii de adãugare este O(N).

Dacã extinderea vectorului se va face prin dublarea capacitãtii sale atunci copierea celor C elemente se va face numai dupã încã C/2 adãugãri la vectorul de capacitate C/2. Deci durata medie a C/2 operatii de adãugare este de ordinul 3C/2, adicã O(C). In acest caz, când timpul total a O(N) operatii este de ordinul O(N) vom spune cã timpul amortizat al unei singure operatii este O(1). Altfel spus, durata totalã a unei secvente de N operatii este proportionalã cu N si deci fiecare operatie este O(1).

Aceastã metodã de analizã amortizatã se numeste metoda "agregat" pentru cã se calculeazã un cost "agregat" pe o secventã de operatii si se raporteazã la numãrul de operatii din secventã.

Prin extensie se vorbeste chiar de structuri de date amortizate, pentru care costul mare al unor operatii cu frecventã micã se "amortizeazã" pe durata celorlalte operatii. Este vorba de structuri care se reorganizeazã 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 rãdãcinã), arbori 2-3 si arbori B (reorganizati când un nod este plin dar mai trebuie adãugatã o valoare la acel nod), s.a.

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

- Costul mediu se calculeazã ca medie pe diferite intrãri (date) si presupune cã durata unei operatii (de adãugare de exemplu) nu depinde de operatiile anterioare;
- Costul amortizat se calculeazã ca medie pe o secventã 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 particularitãti ale aplicatiei. Din aceastã categorie fac parte vectori, matrice, stive, cozi, liste înlãntuite simple si chiar arbori binari fãrã reechilibrare.

Pentru alte structuri operatiile asociate pot fi destul de complexe, astfel cã este preferabil sã gãsim o bibliotecã sau surse care pot fi adaptate rapid la specificul aplicatiei. Din aceastã 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 existã numai pentru limbajele orientate pe obiecte (C++, C#, Java). Pot fi gãsite însã 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 importantã este între limbajele procedurale (Pascal si C) si limbajele orientate pe obiecte (C++ si Java).

Limbajul folosit în acest text este C dar unele exemple folosesc parametri de tip referintã în functii (declarati cu "tip &"), care sunt un împrumut din limbajul C++.

Uilizarea tipului referintã permite simplificarea definirii si utilizãrii functiilor care modificã continutul unei structuri de date, definite printr-un tip structurã. In C, o functie nu poate modifica valoarea unui argument de tip structurã decât dacã primeste adresa variabilei ce se modificã, printr-un argument de un tip pointer. Exemplul urmãtor foloseste o structurã care reuneste un vector si dimensiunea sa, iar functiile utilizeazã 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 uniformã a functiilor si pentru eficientã am putea folosi argumente pointer si pentru functiile care nu modificã vectorul (de ex. “printV”).

In C++ si în unele variante de C se pot folosi parametri de tip referintã, care simplificã mult definirea si utilizarea de functii cu parametri modificabili. Un parametru formal referintã se declarã folosind caracterul ‘&’ între tipul si numele parametrului. In interiorul functiei parametrul referintã se foloseste la fel ca un parametru de acelasi tip (transmis prin valoare). Parametrul efectiv care va înlocui un parametru formal referintã poate fi orice nume de variabilã (de un tip identic sau compatibil). Exemple de functii din programul anterior cu parametri referintã:

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

In continuare vom folosi parametri de tip referintã pentru functiile care trebuie sã modifice valorile acestor parametri. In felul acesta utilizarea functiilor este uniformã, indiferent dacã ele modificã sau nu variabila colectie primitã ca argument.

In cazul vectorilor sunt posibile si alte solutii care sã evite functii cu argumente modificabile (de ex. memorarea lungimii la începutul unui vector de numere), dar vom prefera solutiile general aplicabile oricãrei colectii de date.

O altã alegere trebuie fãcutã pentru functiile care au ca rezultat un element dintr-o colectie: functia poate avea ca rezultat valoarea elementului sau poate fi de tip *void* iar valoarea sã fie transmisã în afarã printr-un argument de tip referintã sau pointer. Pentru o functie care furnizeazã elementul (de un tip T) dintr-o pozitie datã a unui vector, avem de ales între urmãtoarele variante:

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

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

Alegerea între prima si ultima variantã este oarecum subiectivã si influentatã de limbajul utilizat.

O alternativã la functiile cu parametri modificabili este utilizarea de variabile externe (globale) pentru colectiile de date si scoaterea acestor colectii din lista de argumente a subprogramelor care opereazã cu colectia. Solutia este posibilã deseori deoarece multe aplicatii folosesc o singurã colectie de un anumit tip (o singurã stivã, un singur graf) si ea se întâlneste în textele mai simple despre structuri de date. Astfel de functii nu pot fi reutilizate în aplicatii diferite si nu pot fi introduse în biblioteci de subprograme, dar variabilele externe simplificã programarea si fac mai eficiente functiile recursive (cu mai putini parametri de pus pe stivã la fiecare apel).

Exemplu de utilizare a unui vector ca variabilã externã:

```
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
}
```

Functiile de mai sus pot fi folosite numai într-un program care lucreazã cu un singur vector, declarat ca variabilã externã cu numele "a". Dacã programul foloseste mai multi vectori, functiile anterioare nu mai pot fi folosite. In general se recomandã ca toate datele necesare unui subprogram si toate rezultatele sã fie transmise prin argumente sau prin numele functiei.

Majoritatea subprogramelor care realizeazã operatii cu o structurã de date se pot termina anormal, fie din cauza unor argumente cu valori incorecte, fie din cauza stãrii colectiei; de exemplu, încercarea de adãugare a unui nou element la un vector plin. In absenta unui mecanism de tratare a exceptiilor program (cum sunt cele din Java si C++), solutiile de raportare a acestei conditii de cãtre un subprogram sunt :

- Terminarea întregului program dupã afisarea unui mesaj, cu sau fãrã utilizarea lui "assert" (pentru erori grave dar putin 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 functii de tip boolean (întreg în C), cu rezultat 1 (sau altã valoare pozitivã) pentru terminare normalã si rezultat 0 sau negativ pentru terminare anormalã. 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 colectie poate contine valori numerice de diferite tipuri si lungimi sau siruri de caractere sau alte tipuri agregat (structuri), sau pointeri (adrese). Se doreste ca operatiile cu un anumit tip de colectie sã poatã fi scrise ca functii generale, adaptabile pentru fiecare tip de date ce poate face parte din colectie.

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

Realizarea unei colectii generice în limbajul C se poate face în douã moduri:

1) Prin utilizarea de tipuri generice (neprecizate) pentru elementele colectiei în subprogramele ce realizeazã operatii cu colectia. Pentru a folosi astfel de functii ele trebuie adaptate la tipul de date cerut de o aplicatie. Adaptarea se face partial de cãtre compilator (prin macro-substitutie) si partial de cãtre programator (care trebuie sã dispunã de forma sursã 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) sã se facã la executie. Utilizarea unor astfel de functii este mai dificilã, dar utilizatorul nu trebuie sã intervinã în textul sursã al subprogramelor si are mai multã flexibilitate în adaptarea colectiilor la diverse tipuri de date.

Primul exemplu aratã cum se defineste un vector cu componente de un tip T neprecizat în functii, dar precizat în programul care foloseste 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 dacã tipul T este un tip numeric pentru cã 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 douã posibilitãti:

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 urmãtoarele modificãri în secventele 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 functiilor de comparare, atribuire, s.a ca argumente la functiile care le folosesc (fãrã a impune anumite nume acestor functii). 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
}
```

In cazul structurilor de date cu elemente legate prin pointeri (liste si arbori) mai existã o solutie de scriere a functiilor care realizeazã operatii cu acele structuri astfel ca ele sã nu depindã de tipul datelor memorate: crearea nodurilor de listã sau de arbore se face în afara functiilor generale (în programul de aplicatie), iar functiile de insertie si de stergere primesc un pointer la nodul de adãugat sau de sters si nu valoarea ce trebuie adãugatã sau eliminatã. Aceastã solutie nu este adecvatã structurilor folosite pentru cãutarea dupã valoare (multimi, dictionare).

Uneori tipul datelor folosite de o aplicatie este un tip agregat (o structurã C): o datã calendaristicã ce grupeazã numere pentru zi, lunã, an , descrierea unui arc dintr-un graf pentru care se memoreazã numerele nodurilor si costul arcului, s.a. Problema care se pune este dacã tipul T este chiar tipul structurã sau este un tip pointer la acea structurã. Ca si în cazul sirurilor de caractere este preferabil sã se manipuleze în programe pointeri (adrese de structuri) si nu structuri. In plus, atribuirea între pointeri se face la fel ca si atribuirea între numere (cu operatorul '=').

In concluzie, tipul neprecizat T al elementelor unei colectii este de obicei fie un tip numeric, fie un tip pointer (inclusiv de tip “void *”). Avantajul principal al acestei solutii este simplitatea programelor, dar ea nu se poate aplica pentru colectii de colectii (un vector de liste, de exemplu) si nici pentru colectii neomogene.

2.3 UTILIZAREA DE POINTERI GENERICI

O a doua solutie pentru o colectie genericã este o colectie de pointeri la orice tip (void *), care vor fi înlocuiti cu pointeri la datele folosite în fiecare aplicatie. Si în acest caz functia de comparare trebuie transmisã ca argument functiilor de insertie sau de cãutare în colectie. Exemplu de vector generic cu pointeri:

```
#define M 100                          // dimens 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 urmãtoare aratã 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 bibliotecã si nu este necesar codul sursã.
- Se pot crea colectii cu elemente de tipuri diferite, pentru cã în colectie se memoreazã 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 alocatã memorie dinamic pentru fiecare numãr, ca sã obtinem câte o adresã distinctã pentru a fi memoratã în colectie. 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 adresã distincta
          addV(vecini,p);               // adauga la vector de vecini
       } while (j>=0);
       addV(graf,vecini);               // adauga vector de vecini la graf
    }
}
```

Pentru colectii ordonate (liste ordonate, arbori partial ordonati, arbori de cãutare) trebuie comparate datele memorate în colectie (nu numai la egalitate) iar comparatia depinde de tipul acestor date. Solutia este de a transmite adresa functie de comparatie la functiile de cautare, adaugare, eliminare s.a. Deoarece comparatia este necesarã în mai multe functii este preferabil ca adresa functiei de comparatie sã fie transmisã la initializarea colectiei si sã fie memoratã alãturi de alte variabile ce definesc colectia de date. Exemplu de operatii cu un vector ordonat:

```
typedef int (* fcmp) (Ptr,Ptr);  // tip functie de comparare
typedef struct  {
   Ptr *v;                     // adresa vector de pointeri alocat dinamic
   int dim;                    // dimensiune vector
   fcmp comp;                  // adresa functiei de comparare date
} Vector;
  // operatii cu tipul Vector
void initV (Vector & a, fcmp cmp) {   // initializare
  a.n = 0;
  a.comp=cmp;                         // retine adresa functiei de comparatie
}
int findV ( Vector a, Ptr p) {        // cautare in 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 si usurinta de întelegere; de aceea exemplele care urmeazã sacrificã generalitatea în favoarea simplitãtii, pentru cã scopul lor principal este acela de a ilustra algoritmi. Din acelasi motiv multe manuale folosesc un pseudo-cod pentru descrierea algoritmilor si nu un limbaj de programare.

2.4 STRUCTURI DE DATE SI FUNCTII RECURSIVE

Un subprogram recursiv este un subprogram care se apeleazã pe el însusi, o datã sau de mai multe ori. Orice subprogram recursiv poate fi rescris si nerecursiv, iterativ, prin repetarea explicitã a operatiilor executate la fiecare apel recursiv. O functie recursivã realizeazã repetarea unor operatii fãrã a folosi instructiuni de ciclare.

In anumite situatii exprimarea recursivã este mai naturalã si mai compactã decât forma nerecursivã. Este cazul operatiilor cu arbori binari si al altor algoritmi de tip “divide et impera” (de împãrtire în subprobleme).

In alte cazuri, exprimarea iterativã este mai naturalã si mai eficientã ca timp si ca memorie folositã, fiind aproape exclusiv folositã: calcule de sume sau de produse, operatii de cãutare, operatii cu liste înlãntuite, etc. In plus, functiile recursive cu mai multi parametri pot fi inutilizabile pentru un numãr mare de apeluri recursive, acolo unde mãrimea stivei implicite (folositã de compilator) este limitatã.

Cele mai simple functii recursive corespund unor relatii de recurentã de forma f(n)= rec(f(n-1)) unde n este un parametru al functiei recursive. La fiecare nou apel valoarea parametrului n se diminueazã, pânã când n ajunge 0 (sau 1), iar valoarea f(0) se calculeazã direct si simplu.

Un alt mod de a interpreta relatia de recurentã anterioarã este acela cã se reduce (succesiv) rezolvarea unei probleme de dimensiune n la rezolvarea unei probleme de dimensiune n-1, pânã când reducerea dimensiunii problemei nu mai este posibilã.

Functiile recursive au cel putin un argument, a cãrui valoare se modificã de la un apel la altul si care este verificat pentru oprirea procesului recursiv.

Orice subprogram recursiv trebuie sã continã o instructiune "if" (de obicei la început), care sã verifice conditia de oprire a procesului recursiv. In caz contrar se ajunge la un proces recursiv ce tinde la infinit si se opreste numai prin umplerea stivei.

Structurile de date liniare si arborescente se pot defini recursiv astfel:
- O listã de N elemente este formatã dintr-un element si o (sub)listã de N-1 elemente;
- Un arbore binar este format dintr-un nod rãdãcinã si cel mult doi (sub)arbori binari;
- Un arbore multicãi este format dintr-un nod rãdãcinã si mai multi (sub)arbori multicãi.

Aceste definitii recursive conduc la functii recursive care reduc o anumitã operatie cu o listã sau cu un arbore la una sau mai multe operatii cu sublista sau subarborii din componenta sa, ca în exemplele urmãtoare pentru numãrarea elementelor dintr-o listã sau dintr-un arbore:

```
int count ( struct nod * list) {          // numara elementele unei liste
  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
}
```

In cazul structurilor liniare functiile recursive nu aduc nici un avantaj fatã de variantele iterative ale acelorasi functii, dar pentru arbori functiile recursive sunt mai compacte si chiar mai usor de înteles decât variantele iterative (mai ales atunci când este necesarã o stivã pentru eliminarea recursivitãtii). Totusi, ideea folositã în cazul structurilor liniare se aplicã si în alte cazuri de functii recursive (calcule de sume si produse, de exemplu): se reduce rezolvarea unei probleme de dimensiune N la rezolvarea unei probleme similare de dimensiune N-1, în mod repetat, pânã când se ajunge la o problemã de dimensiune 0 sau 1, care are o solutie evidentã.

Exemplele urmãtoare aratã cã este important locul unde se face apelul recursiv:

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

Ideia reducerii la douã subprobleme de acelasi tip, de la functiile recursive cu arbori, poate fi folositã si pentru anumite operatii cu vectori sau cu liste liniare. In exemplele urmãtoare se determinã valoarea maximã dintr-un vector de întregi, cu unul si respectiv cu douã apeluri recursive:

```
  // maxim dintre doua numere (functie auxiliarã)
int max2 (int a, int b) {
  return a>b? a:b;
}
  // maxim din vector - recursiv bazat pe recurenta
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 injumatatire
int maxim1 (int a[], int i, int j) {
  int m;
  if ( i==j )
      return a[i];
  m= (i+j)/2;
  return max2 (maxim1(a,i,m), maxim1(a,m+1,j));
}
```

Exemple de cãutare secventialã 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 împãrtirii în douã subprobleme de dimensiuni apropiate (numitã "divide et impera") aplicatã unor operatii cu vectori necesitã douã argumente (indici initial si final care definesc fiecare din subvectori) si nu doar dimensiunea vectorului. Situatia functiei "max" din exemplul anterior se mai întâlneste la cãutarea binarã într-un vector ordonat si la ordonarea unui vector prin metodele "quicksort" si "mergesort". Diferenta dintre functia recursivã care foloseste metoda "divide et impera" si functia nerecursivã poate fi eliminatã printr-o functie auxiliarã:

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

Cãutarea binarã într-un vector ordonat împarte succesiv vectorul în douã pãrti egale, comparã valoarea cãutatã cu valoarea medianã, stabileste care din cei doi subvectori poate contine valoarea cãutatã si deci va fi împãrtit în continuare. Timpul unei cãutãri binare într-un vector ordonat de n elemente este de ordinul $\log_2(n)$ fatã de O(n) pentru cãutarea secventialã (singura posibilã într-un vector neordonat). Exemplu de functie recursivã pentru cãutare binarã:

```
  // cãutare binarã, recursivã a lui b între a[i] si a[j]
int caut(int b, int a[], int i, int j) {
  int m;
  if ( i > j)
    return -1;              // b negãsit în a
  m=(i+j)/2;                // m= indice median intre i si 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 intre i si m-1
    else                               // daca b in a doua jumatate
       return caut (b,a,m+1,j);        // cauta intre m+1 si
}
```

Varianta iterativã a cãutãrii binare foloseste un ciclu de înjumãtãtire repetatã:

```
int caut (int b, int a[], int i, int j) {
 int m;
 while (i < j ) {        // repeta cat timp  i<j
  m=(i+j)/2;
  if (a[m]==b)
    return m;
  else
    if (a[m] < b)
       i=m+1;
```

```
    else
      j=m-1;
  }
  return  -1;      // -1 daca b negasit
}
```

Sortarea rapidã ("quicksort") împarte repetat vectorul în douã partitii, una cu valori mai mici si alta cu valori mai mari ca un element pivot, pânã când fiecare partitie se reduce la un singur element. Indicii i si j delimiteazã 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 cã nu se comparã elemente vecine din vector (ca în alte metode), ci se comparã 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 sfârsitul 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 recursivitãtii din algoritmul quicksort nu mai este la fel de simplã ca eliminarea recursivitãtii din algoritmul de cãutare binarã, deoarece sunt douã apeluri recursive succesive.

In general, metoda de eliminare a recursivitãtii depinde de numãrul si de pozitia apelurilor recursive astfel:
- O functie recursivã cu un singur apel ca ultimã instructiune se poate rescrie simplu iterativ prin înlocuirea instructiunii "if" cu o instructiune "while" (de observat cã metoda de cãutare binarã are un singur apel recursiv desi sunt scrise douã instructiuni; la executie se alege doar una din ele);
- O functie recursivã cu un apel care nu este ultima instructiune sau cu douã apeluri se poate rescrie nerecursiv folosind o stivã pentru memorarea argumentelor si variabilelor locale.
- Anumite functii recursive cu douã apeluri, care genereazã apeluri repetate cu aceiasi parametri, se pot rescrie nerecursiv folosind o matrice (sau un vector) cu rezultate ale apelurilor anterioare, prin metoda programãrii dinamice.

Orice compilator de functii recursive foloseste o stivã pentru argumente formale, variabile locale si adrese de revenire din fiecare apel. In cazul unui mare de argumente si de apeluri se poate ajunge ca stiva folositã implicit sã depãseascã capacitatea rezervatã (functie de memoria RAM disponibilã) si deci ca probleme de dimensiune mare sã nu poatã fi rezolvate recursiv. Acesta este unul din motivele eliminãrii recursivitãtii, iar al doilea motiv este timpul mare de executie necesar unor probleme (cum ar fi cele care pot fi rezolvate si prin programare dinamicã).

Eliminarea recursivitãtii din functia "qsort" se poate face în douã etape:

- Se reduc cele douã apeluri la un singur apel recursiv;
- Se foloseste o stivã pentru a elimina apelul recursiv neterminal.

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

Cea mai simplã structurã de stivã este un vector cu adãugare si extragere numai de la sfârsit (vârful stivei este ultimul element din vector). In stivã se vor pune argumentele functiei care se modificã 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 si 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];
   }
 }
}
```

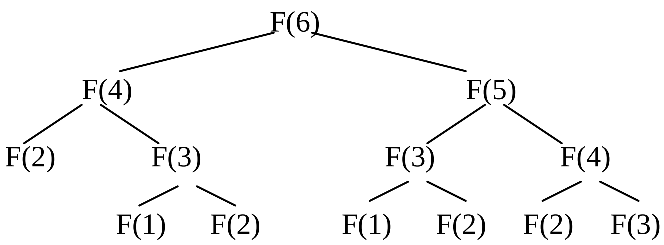
O functie cu douã apeluri recursive genereazã un arbore binar de apeluri. Un exemplu este calculul numãrului n din sirul Fibonacci F(n) pe baza relatiei de recurentã:

F(n) = F(n-2)+F(n-1)   si primele 2 numere din sir F(0)=F(1)=1;

Relatia de recurentã poate fi transpusã imediat într-o functie recursivã:

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

Utilizarea acestei functii este foarte ineficientã, iar timpul de calcul creste exponential în raport cu n. Explicatia este aceea cã se repetã rezolvarea unor subprobleme, iar numãrul de apeluri al functiei creste rapid pe mãsurã ce n creste. Arborele de apeluri pentru F(6) va fi:



Desi n este mic si nu am mai figurat unele apeluri (cu argumente 1 si 0), se observã cum se repetã apeluri ale functiei recursive pentru a recalcula aceleasi valori în mod repetat.

Ideea programãrii dinamice este de a înlocui functia recursivã cu o functie care sã completeze vectorul cu rezultate ale subproblemelor mai mici (în acest caz, numere din sirul Fibonaci):

```
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];
}
```

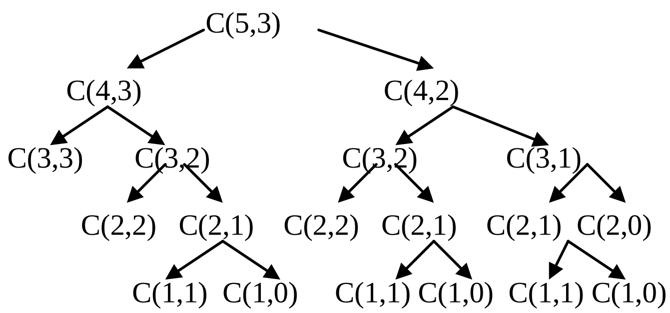
Un alt exemplu de trecere de la o functie recursivã la completarea unui tabel este problema calculului combinãrilor de n numere luate câte k:

C(n,k) = C(n-1,k) + C(n-1,k-1)   pentru 0 < k < n
C(n,k) = 1   pentru k=0 sau k=n

Aceastã relatie de recurentã exprimã descompunerea problemei C(n,k) în douã subprobleme mai mici (cu valori mai mici pentru n si k). Traducem direct relatia într-o functie recursivã:

```
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 abordãri rezultã din numãrul mare de apeluri recursive, dintre care o parte nu fac decât sã recalculeze aceleasi valori (functia "comb" se apeleazã de mai multe ori cu aceleasi valori pentru parametri n si k). Arborele acestor apeluri pentru n=5 si k=3 este :



Dintre cele 19 apeluri numai 11 sunt diferite.

Metoda programãrii dinamice construieste o matrice c[i][j] a cãrei completare începe cu prima coloanã c[i][0]=1, continuã cu coloana a doua c[i][1], coloana a treia s.a.m.d. Elementele matricei se calculeazã unele din altele folosind tot relatia de recurentã anterioarã. Exemplu de functie care completeazã aceastã 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
}
```

Urmeazã douã probleme clasice, ale cãror solutii sunt cunoscute aproape exclusiv sub forma recursivã, deoarece variantele nerecursive sunt mai lungi si mai greu de înteles: afisarea numelor fisierelor dintr-un director dat si din subdirectoarele sale si problema turnurilor din Hanoi.

Structura de directoare si subdirectoare este o structurã arborescentã si astfel se explicã natura recursivã a operatiilor cu asrfel de structuri. Pentru obtinerea numelor fisierelor dintr-un director se folosesc functiile de bibliotecã "findfirst" si "findnext" (parte a unui iterator). La fiecare nume de fisier se verificã dacã este la rândul lui un subdirector, pentru examinarea continutului sãu. Procesul se repetã pânã când nu mai existã fisiere director, ci numai fisiere normale.

Functia care urmeazã mai include unele detalii, cum ar fi:
- afisare cu indentare diferitã la fiecare nivel de adâncime (argumentul "sp");
- evitarea unei recursivitãti infinite pentru numele de directoare "." si "..";
- construirea sirului de forma "path/*.*" cerut de functia "_findfirst"

```
void fileList ( char * path, int sp) {    // listare fisiere identificate prin "path"
  struct _finddata_t fb;                  // structura predefinita folosita de findfirst (atribute fisier)
  int  done=0; int i;                     // done devine 1 cand nu mai sunt fisiere
  char tmp[256];                          // ptr creare cale la subdirectoarele unui director
  long first;                             // transmis de la findfirst la findnext
  first = _findfirst(path,&fb);           // cauta primul dintre fisiere si pune atribute in fb
  while (done==0) {                       // repeta cat mai sunt fisiere pe calea "path"
    if (fb.name[0] !='.')                 // daca numele de director nu incepe cu '.'
      printf ("%*c %-12s \n",sp,' ', fb.name);      // afisare nume fisier
     // daca subdirector cu nume diferit de "." si ".."
    if ( fb.attrib ==_A_SUBDIR &&  fb.name[0] !='.' ) {
      i= strrchr(path,'/') -path;                  //  extrage nume director
      strncpy(tmp,path,i+1);                       // copiaza calea in tmp
      tmp[i+1]=0;                                  // ca sir terminat cu zero
      strcat(tmp,fb.name); strcat(tmp,"/*.*");  // adauga la cale nume subdirector si /*.*
      fileList (tmp,sp+3);                         // listeaza continut subdirector, decalat cu 3
    }
    done=_findnext (first,&fb);                    // pune numele urmatorului fisier in "fb"
  }
}
```

Problema turnurilor din Hanoi este un joc cu 3 tije verticale si mai multe discuri de diametre diferite. Initial discurile sunt stivuite pe prima tijã astfel cã fiecare disc stã pe un disc mai mare. Problema cere ca toate discurile sã ajungã pe ultima tijã, ca în configuratia initialã, folosind pentru mutãri si tija din mijloc. Mutãrile de discuri trebuie sã satisfacã urmãtoarele conditii:
- Se mutã numai un singur disc de pe o tijã 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 mutãri care respectã aceste conditii.

Functia urmãtoare afiseazã discul mutat, sursa (de unde) si destinatia (unde) unei mutãri:

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

```
}
```

Functia recursivã care urmeazã rezolvã 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 proprietãti ale stãrilor prin care se trece; de exemplu, s-a arãtat cã se poate repeta de (2^n–1) ori functia "mutadisc" recalculând la fiecare pas numãrul tijei sursã si al tijei destinatie (numãrul discului nu conteazã deoarece nu se poate muta decât discul din vârful stivei de pe tija sursã):

```
int main(){
  int n=4, i;                    // n = numar de discuri
  for (i=1; i < (1 << n); i++)   // numar de mutari= 1<<n  = 2^n
    printf("Muta de pe  %d  pe  %d \n", (i&i-1)%3, ((i|i-1)+1)%3 );
}
```

O a treia categorie de probleme cu solutii recursive sunt probleme care contin un apel recursiv într-un ciclu, deci un numãr variabil (de obicei mare) de apeluri recursive. Din aceastã categorie fac parte algoritmii de tip "backtracking", printre care si problemele de combinatoricã. Exemplul urmãtor genereazã si afiseazã toate permutãrile posibile ale primelor n numere naturale:

```
void perm (int k, int a[], int n) {   // vectorul a contine o permutare de n numere
  int i;
  for (i=1;i<=n;i++){
      a[k]=i;                         // pune i in a[k]
      if (k<n) perm(k+1,a,n);         // apel recursiv ptr completare a[k+1]
      else printv(a,n);               // afisare vector a de n intregi
  }
}
```

In acest caz arborele de apeluri recursive nu mai este binar si fiecare apel genereazã n alte apeluri. De observat cã, pentru un n dat (de ex. n=3) putem scrie n cicluri unul în altul (ca sã generãm permutãri), dar când n este necunoscut (poate avea orice valoare) nu mai putem scrie aceste cicluri. Si pentru functiile cu un apel recursiv într-un ciclu existã o variantã nerecursivã care foloseste o stivã, iar aceastã stivã este chiar vectorul ce contine o solutie (vectorul "a" în functia "perm").

Capitolul 3

VECTORI

3.1 VECTORI

Structura de vector ("array") este foarte folositã datoritã avantajelor sale:
- Nu trebuie memorate decât datele necesare aplicatiei (nu si adrese de legãturã);
- Este posibil accesul direct (aleator) la orice element dintr-un vector prin indici;
- Programarea operatiilor cu vectori este foarte simplã.
- Cãutarea într-un vector ordonat este foarte eficientã, prin cãutare binarã.

Dezavantajul unui vector cu dimensiune constantã rezultã din necesitatea unei estimãri a dimensiunii sale la scrierea programului. Pentru un vector alocat si realocat dinamic poate apare o fragmentare a memoriei dinamice rezultate din realocãri repetate pentru extinderea vectorului. De asemenea, eliminarea de elemente dintr-un vector compact poate necesita deplasarea elementelor din vector.

Prin vectori se reprezintã si anumite cazuri particulare de liste înlãntuite sau de arbori pentru reducerea memoriei ocupate si timpului de prelucrare.

Ca tipuri de vectori putem mentiona:
- Vectori cu dimensiune fixã (constantã);
- Vectori extensibili (realocabili dinamic);
- Vectori de biti (la care un element ocupã un bit);
- Vectori "heap" (care reprezintã compact un arbore binar particular);
- Vectori ca tabele de dispersie.

De obicei un vector este completat în ordinea crescãtoare a indicilor, fie prin adãugare la sfârsit a noilor elemente, fie prin insertie între alte elemente existente, pentru a mentine ordinea în vector.

Existã si exceptii de la cazul uzual al vectorilor cu elemente consecutive : vectori cu interval ("buffer gap") si tabele de dispersie ("hash tables").

Un "buffer gap" este folosit în procesoarele de texte; textul din memorie este împãrtit în douã siruri pãstrate într-un vector ("buffer" cu text) dar separate între ele printr-un interval plasat în pozitia curentã de editare a textului. In felul acesta se evitã mutarea unor siruri lungi de caractere în memorie la modificarea textului; insertia de noi caractere în pozitia curentã mãreste secventa de la începutul vectorului si reduce intervalul, iar stergerea de caractere din pozitia curentã mãreste intervalul dintre caracterele aflate înainte si respectiv dupã pozitia curentã.

Mutarea cursorului necesitã mutarea unor caractere dintr-un sir în celãlalt, dar numai ca urmare a unei operatii de modificare în noua pozitie a cursorului.

Caracterele sterse sau inserate sunt de fapt memorate într-un alt vector, pentru a se putea reconstitui un text modificat din gresealã (operatia "undo" de anulare a unor operatii si de revenire la o stare anterioarã).

Vectorii cu dimensiune constantã, fixatã la scrierea programului, se folosesc în unele situatii particulare când limita colectiei este cunoscutã si relativ micã sau când se doreste simplificarea programelor, pentru a facilita întelegerea lor. Alte situatii pot fi cea a unui vector de constante sau de cuvinte cheie, cu numãr cunoscut de valori.

Vectori cu dimensiune fixã se folosesc si ca zone tampon la citirea sau scrierea în fisiere text sau în alte fluxuri de date.

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

Un fisier binar cu articole de lungime fixã poate fi privit ca un vector, deoarece are aceleasi avantaje si dezavantaje, iar operatiile sunt similare: adãugare la sfârsit de fisier, cãutare secventialã în fisier, acces direct la un articol prin indice (pozitie relativã în fisier), sortare fisier atunci când este nevoie, s.a. La fel ca într-un vector, operatiile de insertie si de stergere de articole consumã timp si trebuie evitate sau amânate pe cât posibil.

3.2 VECTORI ORDONATI

Un vector ordonat reduce timpul anumitor operatii, cum ar fi: cãutarea unei valori date, verificarea unicitãtii elementelor, gãsirea perechii celei mai apropiate, calculul frecventei de aparitie a fiecãrei valori distincte s.a.

Un vector ordonat poate fi folosit si drept coadã cu prioritãti, dacã nu se mai fac adãugãri de elemente la coadã, pentru cã valoarea minimã (sau maximã) se aflã la una din extremitãtile vectorului, de unde se poate scoate fãrã alte operatii auxiliare.

Mentinerea unui vector în ordine dupã fiecare operatie de adãugare sau de stergere nu este eficientã si nici nu este necesarã de multe ori; atunci când avem nevoie de o colectie dinamicã permanent ordonatã vom folosi un arbore binar sau o listã înlãntuitã ordonatã. Ordonarea vectorilor se face atunci când este necesar, de exemplu pentru afisarea elementelor sortate dupã o anumitã cheie.

Pe de altã parte, operatia de sortare este eficientã numai pe vectori; nu se sorteazã liste înlãntuite sau arbori neordonati sau tabele de dispersie.

Sunt cunoscuti mai multi algoritmi de sortare, care diferã atât prin modul de lucru cât si prin performantele lor. Cei mai simpli si ineficienti algoritmi de sortare au o complexitate de ordinul O(n*n), iar cei mai buni algoritmi de sortare necesitã pentru cazul mediu un timp de ordinul $O(n*\log_2 n)$, unde “n” este dimensiunea vectorului.

Uneori ne intereseazã un algoritm de sortare “stabilã”, care pãtreazã ordinea initialã a valorilor egale din vectorul sortat. Mai multi algoritmi nu sunt “stabili”.

De obicei ne intereseazã algoritmii de sortare “pe loc”, care nu necesitã memorie suplimentarã, desi existã câtiva algoritmi foarte buni care nu sunt de acest tip: sortare prin interclasare si sortare prin distributie pe compartimente.

Algoritmii de sortare “pe loc” a unui vector se bazeazã pe compararea de elemente din vector, urmatã eventual de schimbarea între ele a elementelor comparate pentru a respecta conditia ca orice element sã fie mai mare ca cele precedente si mai mic ca cele care-i urmeazã.

Vom nota cu T tipul elementelor din vector, tip care suportã comparatia prin operatori ai limbajului (deci un tip numeric). In cazul altor tipuri (structuri, siruri) se vor înlocui operatorii de comparatie (si de atribuire) cu functii pentru aceste operatii.

Vom defini mai întâi o functie care schimbã între ele elementele din douã pozitii 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 câtiva algoritmi usor de programat, chiar dacã nu au cele mai bune performante.

Sortarea prin metoda bulelor (“Bubble Sort”) comparã mereu elemente vecine; dupã ce se comparã toate perechile vecine (de la prima cãtre ultima) se coboarã valoarea maximã la sfârsitul vectorului. La urmãtoarele parcurgeri se reduce treptat dimensiunea vectorului, prin eliminarea valorilor finale (deja sortate). Dacã se comparã perechile de elemente vecine de la ultima cãtre prima, atunci se aduce în prima pozitie valoarea minimã, si apoi se modificã indicele de început. Una din variantele posibile de implementare a acestei metode este functia urmãtoare:

```
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 pãtratul dimensiunii vectorului (complexitatea algoritmului este de ordinul n*n).

Sortarea prin selectie determinã în mod repetat elementul minim dintre toate care urmează unui element a[i] si îl aduce în pozitia i, dupã care creste 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 si ea complexitatea O(n*n), dar în medie este mai rapidã decât sortarea prin metoda bulelor (constanta care înmulteste pe n*n este mai micã).

Sortarea prin insertie considerã vectorul format dintr-o partitie sortatã (la început de exemplu) si o partitie nesortatã; la fiecare pas se alege un element din partitia nesortatã si se inserează în locul corespunzãtor din partitia sortatã, dupã 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++) {       // partitia nesortata este intre pozitiile i si n
    x=a[i];                 //  x este un element
    j=i-1;                  // cauta pozitia j unde trebuie inserat x
    while (x<a[j] && j >=0) {
      a[j+1]=a[j];          // deplasare in jos din pozitia j
      j--;
    }
    a[j+1]=x;               // muta pe x in pozitia j+1
  }
}
```

Nici sortarea prin insertie nu este mai bunã de O(n*n) pentru cazul mediu si cel mai nefavorabil, dar poate fi îmbunãtãtitã prin modificarea distantei dintre elementele comparate. Metoda cu increment variabil (ShellSort) se bazeazã pe ideea (folositã si în sortarea rapidã QuickSort) cã sunt preferabile schimbãri între elemente aflate la distantã mai mare în loc de schimbãri între elemente vecine; în felul acesta valori mari aflate initial la începutul vectorului ajung mai repede în pozitiile finale, de la sfârsitul vectorului.

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

In functia urmãtoare se folosesc rezultatele unor studii pentru determinarea valorii initiale a pasului h, care scade apoi prin împãrtire succesivã la 3. De exemplu, pentru n > 100 pasii folositi vor fi 13,4 si 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 douã situatii de alocare dinamicã pentru vectori:
- Dimensiunea vectorului este cunoscutã de program înaintea valorilor ce trebuie memorate în vector si nu se mai modificã pe durata executiei programului; în acest caz este suficientã o alocare initialã de memorie pentru vector (“malloc”).
- Dimensiunea vectorului nu este cunoscutã de la început sau numãrul de elemente poate creste pe mãsurã ce programul evolueazã; în acest caz este necesarã extinderea dinamicã a tabloului (se apeleazã repetat "realloc").

In limbajul C utilizarea unui vector alocat dinamic este similarã utilizãrii unui vector cu dimensiune constantã, cu diferenta cã ultimul nu poate fi realocat dinamic. Functia "realloc" simplificã extinderea (realocarea) unui vector dinamic cu pãstrarea 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
}
```

In aplicatiile care prelucreazã cuvintele distincte dintr-un text, numãrul acestor cuvinte nu este cunoscut si 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 cuvant
      pc =strdup(cuv);                         // aloca memorie ptr cuvant
      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 cuvant
      }
  }
}
```

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

Utilizarea functiei "realloc" necesitã memorarea următoarelor informatii despre vectorul ce va fi extins: adresã vector, dimensiunea alocatã (maximã) si dimensiunea efectivã. Când dimensiunea efectivã ajunge egalã cu dimensiunea maximã, atunci devine necesarã extinderea vectorului. Extinderea se poate face cu o valoare constantã sau prin dublarea dimensiunii curente sau dupã altã metodã.

Exemplul următor aratã cum se pot încapsula în câteva functii operatiile cu un vector alocat si apoi extins dinamic, fãrã ca alocarea si realocarea sã 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 genereazã si afiseazã 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 cãutarea într-un vector neordonat este de ordinul O(n), deci proportional cu dimensiunea vectorului. Intr-un vector ordonat timpul de cãutare este de ordinul O(lg n). Adãugarea la sfârsitul unui vector este imediatã (are ordinul O(1)) iar eliminarea dintr-un vector compact necesitã mutarea în medie a n/2 elemente, deci este de ordinul O(n).

3.4 APLICATIE : COMPONENTE CONEXE

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

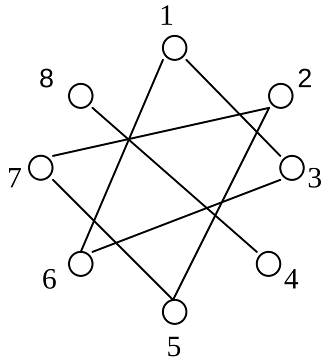
Fiind datã o multime de valori (de orice tip) si o serie de relatii de echivalentã între perechi de valori din multime, se cere sã se afiseze clasele de echivalentã formate cu ele. Dacã sunt n valori, atunci numãrul claselor de echivalentã poate fi între 1 si n, inclusiv.

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 altã formulare a problemei cere afisarea componentelor conexe dintr-un graf neorientat definit printr-o listã de muchii. Fiecare muchie corespunde unei relatii de echivalentã între vârfurile unite de muchie, iar o componentã conexã este un subgraf (o clasã de noduri) în care existã o cale între oricare douã 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 aflã vârful k.

In cazul mai general al claselor de echivalentã ce pot contine elemente de orice tip (numere oarecare sau siruri ce reprezintã nume), mai este necesar si un vector cu valorile elementelor. Pentru exemplul anterior cei doi vectori pot arãta 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 cã fiecare clasã de echivalentã 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 douã posibilitãti:

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

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

S-au stabilit urmãtoarele operatii specifice tipului abstract "Disjoint Sets":
- Initializare colectie (initDS)
- Gãsirea multimii (clasei) care contine o valoare datã (findDS)
- Reunire multimi (clase) ce contin douã valori date (unifDS)
- Afisare colectie de multimi (printDS)

La citirea unei perechi de valori (unei relatii de echivalentã) se stabileste pentru cele douã valori echivalente acelasi numãr de clasã, egal cu cel mai mic dintre cele douã (pentru a mentine ordinea în fiecare clasã).

Dacã valorile sunt chiar numerele 1,2,3...8 atunci evolutia vectorului de clase dupã fiecare pereche de valori cititã 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 modificã nimic)

Urmeazã un exemplu de implementare cu un singur vector a tipului "Colectie de multimi disjuncte" si utilizarea sa în problema afisãrii 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
}
```

In această implementare operatia "find" are un timp constant O(1), dar operatia de reuniune este de ordinul O(n). Vom arăta ulterior (la discutia despre multimi) o altă implementare, mai performantă, dar tot prin vectori a colectiei de multimi 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 contine elementele matricei, fie în ordinea liniilor, fie în ordinea coloanelor.

Matricele alocate dinamic sunt vectori de pointeri la liniile matricei.

Pentru comparatie vom folosi o functie care ordonează un vector de nume (de siruri) si functii de citire si afisare a numelor memorate si ordonate.

Prima formă (vector de vectori) este cea clasică, posibilă în toate limbajele de programare, si are avantajul simplitătii si claritătii operatiilor de prelucrare.

De remarcat că numărul de coloane al matricei transmise ca argument trebuie să fie o constantă, aceeasi pentru toate functiile 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 alocatã 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;
       }
}
```

In exemplul anterior am presupus cã vectorul de pointeri are o dimensiune fixã si este alocat în functia “main”.

Dacã se cunoaste de la început numãrul de linii si de coloane, atunci putem folosi o functie care alocã 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 dificilã programarea unor operatii (selectie elemente, sortarea liniilor, s.a.).

3.6 VECTORI DE BITI

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

In continuare vom ilustra câteva operatii pentru multimi realizate ca vectori de 32 de biti (variabile de tipul "long" pentru fiecare multime în parte). Operatiile cu multimi de biti se realizeazã simplu si rapid prin operatori la nivel de bit.

```
typedef long Set;     // multimi cu max 32 de elemente cu valori intre 0 si 31
void initS ( Set & a) {               // initializare multime
 a=0;
}
void addS (Set & a, int x) {        // adauga element la multime
  a= a | (1L<<x);
}
void delS ( Set& a, int x) {           // elimina element din multime
  a=a & ~(1L<<x);
}
void retainAll ( Set& a1, Set a2) {   // intersectie multimi
  a1= a1 & a2;
 }
void addAll ( Set& a1, Set a2) {      // reuniune de multimi
  a1= a1 | a2;
}
void removeAll (Set& a1, Set a2) {  // diferenta de multimi
  a1 = a1 & ~a2;
}
int findS (Set a,int x) {                  // test apartenenta la o multime
  long b= a & (1L<<x);
  return (b==0) ? 0 : 1;
}
int containsAll (Set a1, Set a2) {     // test includere multimi
   retainAll (a1,a2);
   if (a1==a2) return 1;
   return 0;
}
int sizeS ( Set a) {          // dimensiune (cardinal) multime
  int i, k=0;
   for (i=0;i< 32;i++)
     if ( findS (a,i))
       k++;
   return k;
 }
void printS (Set a) {        // afisare multime
  int i;
  printf("{ ");
  for (i=0;i<32;i++)
     if( findS(a,i))
       printf("%d,",i);
  printf("\b }\n");
}
```

De observat cã operatiile de cãutare (findS) si cu douã multimi (addAll s.a.) nu contin cicluri si au complexitatea O(1). Multimi ca vectori de biti existã în Pascal (tipul "Set") si în limbaje cu clase (clasa "BitSet" în Java).

Intr-o matrice de adiacente a unui graf elementul a[i][j] aratã prezenta (1) sau absenta (0) unei muchii între vârfurile i si j.

In exemplul urmãtor matricea de adiacentã este un vector de biti, obtinut prin memorarea succesivã a liniilor din matrice. Functia "getbit" aratã prezenta sau absenta unui arc de la nodul i la nodul j (graful este orientat). Functia "setbit" permite adãugarea 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 marcãrii prin biti a prezentei sau absentei unor elemente într-o colectie este folositã si pentru arbori binari (parcursi nivel cu nivel, pornind de la rãdãcinã), fiind generalizatã pentru asa-numite structuri de date succinte (compacte), în care relatiile dintre elemente sunt implicite (prin pozitia lor în colectie) si nu folosesc pointeri, a cãror dimensiune contribuie mult la memoria ocupatã de structurile cu pointeri.

Capitolul 4

LISTE CU LEGÃTURI

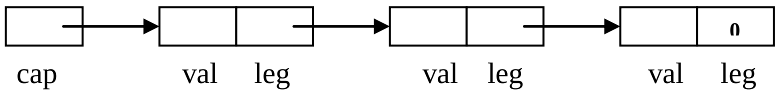
4.1 LISTE ÎNLÃNTUITE

O listã înlãntuitã ("Linked List") este o colectie de elemente, alocate dinamic, dispersate în memorie, dar legate între ele prin pointeri, ca într-un lant. O listã înlãntuitã este o structurã dinamicã, flexibilã, care se poate extinde continuu, fãrã ca utilizatorul sã fie preocupat de posibilitatea depãsirii unei dimensiuni estimate initial (singura limitã este mãrimea zonei "heap" din care se solicitã memorie).

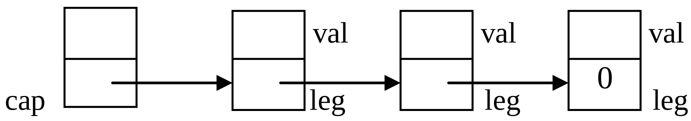
Vom folosi aici cuvântul "listã" pentru o listã liniarã, în care fiecare element are un singur succesor si un singur predecesor.

Intr-o listã înlãntuitã simplã fiecare element al listei contine adresa elementului urmãtor din listã. Ultimul element poate contine ca adresã de legãturã fie constanta NULL (un pointer cãtre nicãieri), fie adresa primului element din listã (dacã este o listã circularã), fie adresa unui element terminator cu o valoare specialã.

Adresa primului element din listã este memoratã într-o variabilã pointer cu nume (alocatã la compilare) si numitã cap de listã ("list head").



Este posibil ca variabila cap de listã sã fie tot o structurã si nu un pointer:



Un element din listã (numit si nod de listã) este de un tip structurã si are (cel putin) douã câmpuri: un câmp de date (sau mai multe) si un câmp de legãturã. Exemplu:

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

Continutul si tipul câmpului de date depind de informatiile memorate în listã si deci de aplicatia care o foloseste. Toate functiile care urmeazã sunt direct aplicabile dacã tipul de date nedefinit T este un tip numeric (aritmetic).

Tipul “List” poate fi definit ca un tip pointer sau ca un tip structurã:

```
typedef Nod* List;              // listã ca pointer
typedef Nod  List;              // listã ca structurã
```

O listã înlãntuitã este complet caracterizatã de variabila "cap de listã", care contine adresa primului nod (sau a ultimului nod, într-o listã circularã). Variabila care defineste o listã este de obicei o variabilã pointer, dar poate fi si o variabilã structurã.

Operatiile uzuale cu o listã înlãntuitã sunt :
- Initializare listã (a variabilei cap de listã):   initL (List &)
- Adãugarea unui nou element la o listã: addL (List&, T)

- Eliminarea unui element dintr-o listã: delL (List&, T)
- Cãutarea unei valori date într-o listã: findL (List, T)
- Test de listã vidã: emptyL(List)
- Determinarea dimensiunii listei: sizeL (List)
- Parcurgerea tuturor nodurilor din listã (traversare listã).

Accesul la elementele unei liste cu legãturi este strict secvential, pornind de la primul element si trecând prin toate nodurile precedente celui cãutat, sau pornind din elementul "curent" al listei, dacã se memoreazã si adresa elementului curent al listei.

Pentru parcurgere se foloseste o variabilã cursor, de tip pointer cãtre nod, care se initializeazã cu adresa cap de listã; pentru a avansa la urmãtorul element din listã se foloseste adresa din câmpul de legãturã al nodului curent:

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

Exemplu de afisare a unei liste înlãntuite 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
  }
}
```

Cãutarea secventialã a unei valori date într-o listã este asemãnãtoare operatiei de afisare, dar are ca rezultat adresa nodului ce contine valoarea cãutatã .

```
   // cãutare într-o listã neordonatã
  Nod* findL (Nod* lst, T x) {
    while (lst!=NULL &&  x != lst→val)
       lst = lst→leg;
    return lst;                  // NULL dacã x negãsit
  }
}
```

Functiile de adãugare, stergere si initializare a listei modificã adresa primului element (nod) din listã; dacã lista este definitã printr-un pointer atunci functiile primesc un pointer si modificã (uneori) acest pointer. Daca lista este definitã printr-o variabilã structurã atunci functiile modificã structura, ca si în cazul stivei vector.

In varianta listelor cu element santinelã nu se mai modificã variabila cap de listã deoarece contine mereu adresa elementului santinelã, creat la initializare.

Operatia de initializare a unei liste stabileste adresa de început a listei, fie ca NULL pentru liste fãrã santinelã, fie ca adresã a elementului santinelã.

Crearea unui nou element de listã necesitã alocarea de memorie, prin functia “malloc” în C sau prin operatorul *new* în C++. Verificarea rezultatului cererii de alocare (NULL, dacã alocare imposibilã) se poate face printr-o instructiune “if” sau prin functia “assert”, dar va fi omisã în continuare. Exemplu de alocare:

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

Adãugarea unui element la o listã înlãntuitã se poate face:
- Mereu la începutul listei;

- Mereu la sfârsitul listei;
- Intr-o pozitie determinatã de valoarea noului element.

Dacã ordinea datelor din listã este indiferentã pentru aplicatie, atunci cel mai simplu este ca adãugarea sã se facã numai la începutul listei. In acest caz lista este de fapt o stivã iar afisarea valorilor din listã se face în ordine inversã introducerii în listã.
Exemplu de creare si afisare a unei liste înlãntuite, cu adãugare la început de listã:

```
typedef Nod* List;                       // ptr a permite redefinirea tipului "List"
void 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;
  }
}
```

Operatiile elementare cu liste se scriu ca functii, pentru a fi reutilizate în diferite aplicatii. Pentru comparatie vom ilustra trei dintre posibilitãtile de programare a acestor functii pentru liste stivã, cu adãugare si eliminare de la început.

Prima variantã este pentru o listã definitã printr-o variabilã structurã, de tip “Nod”:

```
void initS ( Nod * s) {        // initializare stiva (s=var. cap de lista)
  s→leg = NULL;
}
    // pune in stiva un element
void push (Nod * s, int x) {
  Nod * nou = (Nod*)malloc(sizeof(Nod));
  nou→val = x; nou→leg = s→leg;
  s→leg = nou;
}
    // scoate din stiva un element
int pop (Nod * s) {
  Nod * p; int rez;
  p = s→leg;   // adresa primului element
  rez = p→val;      // valoare din varful stivei
  s→leg = p→leg; // adresa element urmator
  free (p) ;
  return rez;
}
  // utilizare
int main () {
  Nod st; int x;
  initS(&st);
  for (x=1;x<11;x++)
    push(&st,x);
  while (! emptyS(&st))
    printf ( "%d ", pop(&st));
}
```

A doua variantã foloseste un pointer ca variabilã cap de listã si nu foloseste argumente de tip referintã (limbaj C standard):

```
void initS ( Nod ** sp) {
  *sp = NULL;
}
     // pune in stiva un element
void push (Nod ** sp, int x) {
  Nod * nou = (Nod*)malloc(sizeof(Nod));
  nou→val = x; nou→leg = *sp;
  *sp = nou;
}
     // scoate din stiva un element
int pop (Nod ** sp) {
  Nod * p; int rez;
  rez = (*sp) →val;
  p = (*sp) →leg;
  free (*sp) ;
  *sp = p;
  return rez;
}
  // utilizare
int main () {
  Nod* st; int x;
  initS(&st);
  for (x=1;x<11;x++)
    push(&st,x);
  while (! emptyS(st))
    printf ( "%d ", pop(&st));
}
```

A treia variantã va fi cea preferatã în continuare si foloseste argumente de tip referintã pentru o listã definitã printr-un pointer (numai în C++):

```
void initS ( Nod* & s) {
  s = NULL;
}
     // pune in stiva un element
void push (Nod* & s, int x) {
  Nod * nou = (Nod*)malloc(sizeof(Nod));
  nou→val = x; nou→leg = s;
  s = nou;
}
     // scoate din stiva un element
int pop (Nod* & s) {
  Nod * p; int rez;
  rez = s→val;    // valoare din primul nod
  p = s→leg;      // adresa nod urmator
  free (s) ;
  s = p;          // adresa varf stiva
  return rez;
}
  // utilizare
int main () {
  Nod* st; int x;
  initS(st);
  for (x=1;x<11;x++)
    push(st,x);
  while (! emptyS(st))
    printf ( "%d ", pop(st));
}
```

Structura de listã înlãntuitã poate fi definitã ca o structurã recursivã: o listã este formatã dintr-un element urmat de o altã listã, eventual vidã. Acest punct de vedere poate conduce la functii recursive pentru operatii cu liste, dar fãrã nici un avantaj fatã de functiile iterative. Exemplu de afisare recursivã a unei liste:

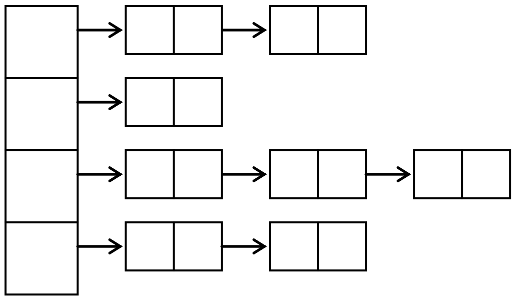
```
void printL ( Nod* lst) {
  if (lst != NULL) {              // daca (sub)lista nu e vidã
    printf ("%d ",lst→val);       // afisarea primului element
    printL (lst→leg);             // afisare sublistã de dupã primul element
  }
}
```

4.2 COLECTII DE LISTE

Listele sunt preferate vectorilor atunci când aplicatia foloseste mai multe liste de lungimi foarte variabile si impredictibile, deoarece asigurã o utilizare mai bunã 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 listã înlãntuitã de pointeri sau printr-un arbore ce contine în noduri pointeri la liste.

Mentionãm 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 listã;
- Un graf reprezentat prin liste de adiacente (liste cu vecinii fiecãrui nod);
- Un dictionar cu valori multiple, în care fiecare cheie are asociatã o listã de valori;
- Un tabel de dispersie ("Hashtable") realizat ca vector de liste de coliziuni;

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



In continuare se prezintã succint sortarea dupã ranguri (pe compartimente), metodã care împarte valorile de sortat în mai multe compartmente, cãrora le corespund tot atâtea liste înlãntuite.

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) dupã valoarea cifrei din pozitia k (k=1 pentru cifra din dreapta), si apoi se reunesc listele în vectorul de n numere (în care ordinea se modificã dupã fiecare trecere).

Algoritmul poate fi descris astfel:

```
repetã pentru k de la 1 la d          // pentru fiecare rang
  initializare vector de liste t
  repetã pentru i de la 1 la n        // distribuie elem. din x in 10 liste
    extrage in c cifra din pozitia k a lui x[i]
    adaugã  x[i] la lista  t[c]
  repetã pentru j de la 0 la 9        // reunire liste in vectorul x
    adaugã toatã 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	239
472	2	472,432	254	2		239	2	239,254	254
534	3		534	3	432,534,239	649	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 numãr y se obtine cu relatia: c = (y / pow(10,k-1)) % 10;

Adãugarea de elemente la o listã (în faza de distribuire) se face mereu la sfârsitul listei, dar extragerea din liste (în faza de colectare a listelor) se face mereu de la începutul listelor, ceea ce face ca fiecare listã sã se comporte ca o coadã.

Pentru ordonare de cuvinte formate din litere numãrul de compartimente va fi numãrul de litere distincte (26 dacã nu conteazã diferenta dintre litere mari si mici).

Functia urmãtoare implementeazã 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 dupã 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 urmãtor
   }
}
```

Tipul abstract “Colectie de multimi disjuncte” poate fi implementat si printr-o colectie de liste, cu câte o listã pentru fiecare multime. Adresele de început ale listelor din colectie sunt reunite într-un vector de pointeri. Numãrul de liste se modificã pe mãsurã ce se reunesc câte douã liste într-una singurã. Ordinea elementelor în fiecare listã nu este importantã astfel cã reunirea a douã liste se poate face legând la ultimul element dintr-o listã primul element din cealaltã listã.

Evolutia listelor multimi pentru 6 valori între care existã relatiile de echivalentã 2~4, 1~3, 6~3, 4~6 poate fi urmãtoarea:

```
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
```

In programul urmãtor se considerã cã ordinea în fiecare multime nu conteazã si reuniunea de multimi se face legând începutul unei liste la sfârsitul altei liste.

```
typedef struct sn {   // un element de lista
   int  nr;               // valoare element multime
   struct sn * leg;
} nod;
typedef struct {
   int n;                  // nr de multimi in colectie
   nod* m[M];          // vector de pointeri la liste
} ds;                     // tipul "disjoint sets"
   // initializare colectie c de n multimi
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 multimi 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 ÎNLÃNTUITE ORDONATE

Listele înlãntuite ordonate se folosesc în aplicatiile care fac multe operatii de adãugare si/sau stergere la/din listã si care necesitã mentinerea permanentã a ordinii în listã. Pentru liste adãugarea cu pãstrarea ordinii este mai eficientã decât pentru vectori, dar reordonarea unei liste înlãntuite este o operatie ineficientã.

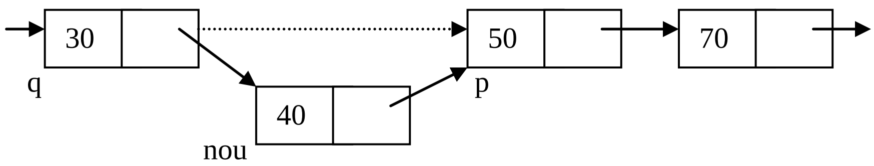
In comparatie cu adãugarea la un vector ordonat, adãugarea la o listã ordonatã este mai rapidã si mai simplã deoarece nu necesitã mutarea unor elemente în memorie. Pe de altã parte, cãutarea unei valori într-o listã înlãntuitã ordonatã nu poate fi la fel de eficientã ca si cãutarea într-un vector ordonat (cãutarea binarã nu se poate aplica si la liste). Crearea si afisarea unei liste înlãntuite ordonate poate fi consideratã si ca o metodã de ordonare a unei colectii de date.

Operatia de adãugare a unei valori la o lista ordonatã este precedatã de o cãutare a locului unde se face insertia, adicã de gãsirea nodului de care se va lega noul element. Mai exact, se cautã primul nod cu valoare mai mare decât valoarea care se adaugã. Cãutarea foloseste o functie de comparare care depinde de tipul datelor memorate si de criteriul de ordonare al elementelor.

Dupã cãutare pot exista 3 situatii:
- Noul element se introduce înaintea primului nod din listã;
- Noul element se adaugã dupã ultimul element din listã;
- Noul element se intercaleazã între douã noduri existente.

Prima situatie necesitã modificarea capului de lista si de aceea este tratatã separat.

Pentru inserarea valorii 40 într-o listã cu nodurile 30,50,70 se cautã prima valoare mai mare ca 40 si se insereazã 40 înaintea nodului cu 50. Operatia presupune modificarea adresei de legãturã a nodului precedent (cu valoarea 30), deci trebuie sã dispunem si de adresa lui. In exemplul urmãtor se foloseste o variabilã pointer q pentru a retine mereu adresa nodului anterior nodului p, unde p este nodul a cãrui valoare se comparã cu valoarea de adãugat (deci avem mereu q→leg == p).



Adãugarea unui nod la o listã ordonatã necesitã:
- crearea unui nod nou: alocare de memorie si completare câmp de date;
- cãutarea pozitiei din listã unde trebuie legat noul nod;
- legarea efectivã prin modificarea a doi pointeri: adresa de legãturã a nodului precedent q si legãtura noului nod (cu exceptia adãugãrii înaintea primului nod):

```
q→leg=nou;  nou→leg=p;
```

```
 // insertie in listã ordonatã, 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) {      //daca lista vida sau x mai mic ca primul elem
    nou→leg=lst; lst= nou;             // daca nou la început de listã
 } else {                              // altfel cauta 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 si p
    }
   nou→leg=p; q→leg=nou;               // nou se introduce între q si p
 }
}
```

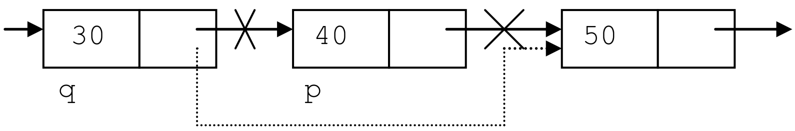
Functia următoare foloseste un singur pointer q: cãutarea se opreste pe nodul ‘q’, precedent celui cu valoare mai mare ca x (“nou” se leagã între q si q→leg):

```
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 altã solutie este ca noul element sã se adauge dupã cel cu valoare mai mare (cu adresa p) si apoi sã se schimbe între ele datele din nodurile p si nou; solutia permite utilizarea în functia de insertie a unei functii de cãutare a pozitiei unei valori date în listã, pentru a evita elemente identice în listã.

Stergerea unui element cu valoare datã dintr-o listã începe cu cãutarea elementului în listã, urmatã de modificarea adresei de legãturã a nodului precedent celui sters. Fie p adresa nodului ce trebuie eliminat si q adresa nodului precedent. Eliminarea unui nod p (diferit de primul) se realizează prin urmãtoarele operatii:

```
q→leg = p→leg;       // succesorul lui p devine succesorul lui q
free(p);
```



Dacã se sterge chiar primul nod, atunci trebuie modificatã si adresa de început a listei (primitã ca argument de functia respectivã).

Functia următoare eliminã 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 gãsit
    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
  }
}
```

Functia următoare de eliminare foloseste 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 listã ordonatã se pot exprima si recursiv:

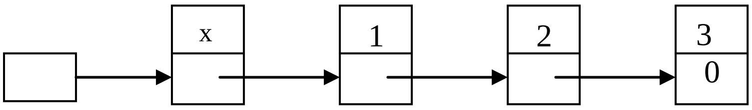
```
  // inserare recursiva in listã ordonatã
void insL (List & lst, T x) {
  Nod * aux;
  if ( lst !=NULL && x > lst→val)  // dacã x mai mare ca primul element
     insL ( lst→leg,x);              // se va introduce in sublista de dupa primul
  else {                             // lista vida sau x 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);
    }
}
```

Functiile pentru operatii cu liste ordonate pot fi simplificate folosind liste cu element santinelã si alte variante de liste înlãntuite.

4.4 VARIANTE DE LISTE ÎNLÃNTUITE

Variantele de liste întâlnite în literaturã si în aplicatii pot fi grupate în:
- Liste cu structurã diferitã fatã de o listã simplã deschisã: liste circulare, liste cu element santinelã, liste dublu înlãntuite, etc.
- Liste cu elemente comune: un acelasi element apartine la douã sau mai multe liste, având câte un pointer pentru fiecare din liste. In felul acesta elementele pot fi parcurse si folosite în ordinea din fiecare listã. Clasa LinkedHashSet din Java foloseste aceastã idee pentru a mentine ordinea de adãugare la multime a elementelor dispersate în mai mai multe liste de coliziuni (sinonime).
- Liste cu auto-organizare, în care fiecare element accesat este mutat la începutul listei ("Splay List"). In felul acesta elementele folosite cel mai frecvent se vor afla la începutul listei si vor avea un timp de regãsire mai mic.
- Liste cu acces mai rapid si/sau cu consum mai mic de memorie.

O listã cu santinelã contine cel putin un element (numit santinelã), creat la initializarea listei si care rãmâne la începutul listei indiferent de operatiile efectuate:



Deoarece lista nu este niciodatã vidã si adresa de început nu se mai modificã la adãugarea sau la stergerea de elemente, operatiile sunt mai simple (nu mai trebuie tratat separat cazul modificãrii primului element din listã). Exemple de functii:

```
   // initializare lista cu santinela
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 introdusã de elementul santinelã este importantã si de aceea se poate folosi la stive liste înlãntuite, la liste “skip” si alte variante de liste.

In elementul santinelã se poate memora dimensiunea listei (numãrul de elemente cu date), actualizat la adãugare si la eliminare de elemente. Consecinta este un timp O(1) în loc de O(n) pentru operatia de obtinere a dimensiunii listei (pentru cã nu mai trebuie numãrate elementele din listã). La compararea a douã multimi implementate ca liste neordonate pentru a constata egalitatea lor, se reduce timpul de comparare prin compararea dimensiunilor listelor, ca primã operatie.

In general se practicã memorarea dimensiunii unei colectii si actualizarea ei la operatiile de modificare a colectiei, dar într-o structurã (sau clasã) care defineste colectia respectivã, împreunã cu adresa de început a listei.

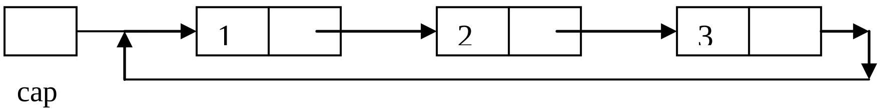
Prin simetrie cu un prim element (“head”) se foloseste uneori si un element terminator de listã (“tail”), care poate contine o valoare mai mare decât oricare valoare memoratã în listã. In acest fel se simplificã conditia de cãutare într-o listã ordonatã crescãtor. Elementul final este creat la initializarea 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 cãutare a unei valori date într-o listã ordonatã 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 listã pornind din pozitia curentã, fãrã a fi necesarã o parcurgere de la începutul listei. Intr-o listã circularã definitã prin adresa elementului curent, nici nu este important care este primul sau ultimul element din listã.



Definitia unui nod de listã circularã este aceeasi ca la o listã deschisã. Modificãri au loc la initializarea listei si la conditia de terminare a listei: se comparã adresa curentã cu adresa primului element în loc de comparatie cu constanta NULL.

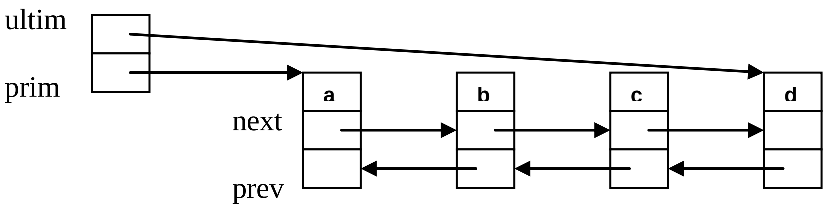
Exemplu de operatii cu o listã circularã cu element sentinelã:

```
   // 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 ÎNLÃNTUITE

Intr-o listã liniarã dublu înlãntuitã fiecare element contine douã adrese de legãturã: una cãtre elementul urmãtor si alta cãtre elementul precedent. Aceastã structurã permite accesul mai rapid la elementul precedent celui curent (necesar la eliminare din listã) si parcurgerea listei în ambele sensuri (inclusiv existenta unui iterator în sens invers pe listã).

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



Exemplu de definire nod de listã dublu înlãntuitã:

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

O altã variantã de listã dublu-înlãntuitã este o listã circularã cu element santinelã. La crearea listei se creeazã elementul santinelã. Exemplu de initializare:

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

In functiile care urmeazã nu se transmite adresa de început a listei la operatiile de inserare si de stergere, dar se specificã adresa elementului sters sau fatã de care se adaugã un nou element. Exemple de realizare a unor operatii cu liste dublu-înlãntuite:

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

Functiile anterioare folosesc un cursor extern listei si pot fi folosite pentru a realiza orice operatii cu o listã: insertie în orice pozitie, stergere din orice pozitie s.a.

Din cauza memoriei necesare unui pointer suplimentar la fiecare element trebuie cântãrit câstigul de timp obtinut cu liste dublu-înlãntuite. Pozitionarea pe un element din listã se face de multe ori prin cãutare secventialã, iar la cãutare se poate retine adresa elementului precedent celui gãsit:

```
    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 algoritmii de sortare, sau când este necesarã o regãsire rapidã pe baza pozitiei în listã sau pentru listele al cãror continut nu se mai modificã si trebuie mentinute în ordine (fiind posibilã si o cãutare binarã). Insertia si eliminarea de elemente în interiorul unui vector au însã complexitatea O(n), unde "n" este dimensiunea vectorului.

O listã înlãntuitã se recomandã atunci când dimensiunea listei este greu de estimat, fiind posibile multe adãugãri si/sau stergeri din listã, sau atunci când sunt necesare inserãri de elemente în interiorul listei. Desi este posibil accesul pozitional, printr-un indice întreg, la elementele unei liste înlãntuite, utilizarea sa frecventã afecteazã negativ performantele aplicatiei (complexitatea O(n)).

Dacã este necesarã o colectie ordonatã, atunci se va folosi o listã permanent ordonatã, prin procedura de adãugare la listã si nu se face o reordonare a unei liste înlãntuite, asa cum se face ca în cazul vectorilor.

Vectorii au proprietatea de localizare a referintelor, ceea ce permite un acces secvential mai rapid prin utilizarea unei memorii "cache" (asa cum au procesoarele moderne); memoria "cache" nu ajutã în aceeasi mãsurã si la reducerea timpului de prelucrare succesivã a elementelor unei liste înlãntuite (mai dispersate în memorie). Din acelasi motiv structura de listã înlãntuitã nu se foloseste 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 rapidã (de capacitate mai micã), în speranta cã programele fac un acces secvential la date (foloseste datele în ordinea în care sunt memorate).

Memorarea explicitã de pointeri conduce la un consum suplimentar de memorie, ajungându-se la situatii când memoria ocupatã de pointeri (si de metadatele asociate cu alocarea dinamicã de memorie) sã depãseascã cu mult memoria ocupatã de datele necesare aplicatiei. Prin "metadate" se înteleg informatiile folosite pentru gestiunea memoriei "heap" dar si faptul cã blocurile alocate din "heap" au o dimensiune multiplu de 8, indiferent de numãrul de octeti solicitat (poate fi un alt multiplu, dar în orice caz nu pot avea orice dimensiune). Blocurile de memorie alocate dinamic sunt legate împreunã într-o listã înlãntuitã, la fel ca si blocurile de memorie eliberate prin functia "free" si care nu sunt adiacente în memorie. Fiecare element din lista spatiului disponibil sau din lista blocurilor alocate este precedat de lungimea sa si de un pointer cãtre urmãtorul element din listã; acestea sunt "metadate" asociate alocãrii dinamice.

Aceste considerente fac ca de multe ori sã se prefere structura de vector în locul unei structuri cu pointeri, tendintã accentuatã odatã cu cresterea dimensiunii memoriei RAM si deci a variabilelor pointer (de la 2 octeti la 4 octeti si chiar la 8 octeti). Se vorbeste uneori de structuri de date "succinte" (compacte) atunci când se renuntã la structuri de liste înlãntuite sau de arbori cu pointeri în favoarea vectorilor.

La o analizã atentã putem deosebi douã modalitãti de eliminare a pointerilor din structurile de date:

- Se pãstreazã 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 legãturi. Aceastã solutie are si avantajul cã face descrierea unor algoritmi independentã de sintaxa utilizãrii de pointeri (sau de existenta tipurilor pointer într-un limbaj de programare) si de aceea este folositã în unele manuale (cea mai cunoscutã fiind cartea "Introduction to Algorithms" de T.Cormen, C.Leiserson, R.Rivest, C.Stein , tradusã si în limba românã). Ideea se foloseste mai ales pentru arbori binari, cum ar fi arbori Huffman sau alti arbori cu numãr limitat de noduri.
- Se renuntã la folosirea unor legãturi explicite între date, pozitia datelor în vector va determina si legãturile dintre ele. Cel mai bun exemplu în acest sens este structura de vector "heap" (vector partial ordonat) care memoreazã un arbore binar într-un vector fãrã a folosi legãturi: fiii unui nod aflat în pozitia k se aflã în pozitiile 2*k si 2*k+1, iar pãrintele unui nod din pozitia j se aflã în pozitia j/2. Un alt exemplu este solutia cea mai eficientã pentru structura "multimi disjuncte" (componente conexe dintr-un graf): un vector care contine o pãdure de arbori, dar în care se memoreazã numai indici cãtre pãrintele fiecãrui nod din arbore (valoarea nodului este chiar indicele sãu în vector).

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

Consideratiile anterioare nu trebuie sã conducã la neglijarea studiului structurilor de date care folosesc pointeri (diverse liste înlãntuite si arbori) din câteva motive:

- Structuri cu pointeri sunt folosite în biblioteci de clase (Java, C# s.a.), chiar dacã pointerii sunt mascati sub formã de referinte;

- Listele înlãntuite si arborii cu pointeri pot constitui un “model” de structuri de date (reflectat în operatiile asupra acestor structuri), chiar si atunci când implementarea se face cu vectori. Un exemplu în acest sens îl constituie listele Lisp, care sunt vãzute de toatã lumea ca liste înlãntuite sau ca arbori, desi unele implementãri de Lisp folosesc vectori (numãrul de liste într-o aplicatie Lisp poate fi foarte mare, iar diferenta de memorie necesarã pentru vectori sau pointeri poate deveni foarte importantã).

Pentru a ilustra acest ultim aspect vom exemplifica operatiile cu o listã înlãntuitã ordonatã cu santinelã dar fãrã pointeri (cu indici în cadrul unui vector). Evolutia listei în cazul secventei de adãugare 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

In pozitia 0 se aflã mereu elementul santinelã, care contine în câmpul de legãturã indicele elementului cu valoare minimã din listã. Elementul cu valoare maximã este ultimul din listã si are zero ca legãturã.

Ca si la listele cu pointeri ordinea fizicã (5,3,7,1) diferã de ordinea logicã (1,3,5,7)

Lista se defineste fie prin doi vectori (vector de valori si vector de legãturi), fie printr-un vector de structuri (cu câte douã 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 indicatã de legãturi:

```
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 listã ordonatã foloseste 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 cãutare într-o listã se poate face dacã în loc sã memorãm un singur element de date într-un nod de listã vom memora un vector de elemente. Putem deosebi douã situatii:
- Vectorii din fiecare nod al listei au acelasi numãr de elemente ("unrolled lists"), numãr corelat cu dimensiunea memoriilor cache;
- Vectorii din nodurile listei au dimensiuni în progresie geometricã, pornind de la ultimul cãtre primul ("VLists").

Economia de memorie se obtine prin reducerea numãrului de pointeri care trebuie memorati. O listã de n date, grupate în vectori de câte m în fiecare nod necesitã numai n/m pointeri, în loc de n pointeri ca într-o listã înlãntuitã cu câte un element de date în fiecare nod. Numãrul de pointeri este chiar mai mic într-o listã "VList", unde sunt necesare numai $\log_2(n)$ noduri.

Câstigul de timp rezultã atât din accesul mai rapid dupã indice (pozitie), cât si din localizarea referintelor într-un vector (folositã de memorii "cache"). Din valoarea indicelui se poate calcula numãrul nodului în care se aflã elementul dorit si pozitia elementului în vectorul din acel nod.

La cãutarea într-o listã ordonatã cu vectori de m elemente în noduri numãrul 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 douã 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).

Numãrul de noduri dintr-o astfel de listã creste când se umple vectorul din nodul la care se adaugã, la adãugarea unui nou element la listã. Initial se porneste cu un singur nod, de dimensiune datã (la "UList") sau de dimensiune 1 (la "VList").

La astfel de liste ori nu se mai eliminã elemente ori se eliminã numai de la începutul listei, ca la o listã stivã. De aceea o listã Ulist cu noduri complete este recomandatã pentru stive.

Exemple de operatii cu o stivã listã de noduri cu vectori de aceeasi dimensiune si plini:

```
#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 stiva
void init (unod * & lst){
    lst = new(unod);  // creare nod initial
    lst->m=0;       // completat de la prima pozitie spre ultima
    lst->leg=NULL;
}
  // adaugare la inceput de lista
void push (unod * & lst, int x ) {
    unod* nou;
    // daca mai e loc in primul nod
    if ( lst->m < M )
         lst->val[lst->m++]=x;
    else {                    // daca primul nod e plin
      nou= new(unod);        // creare nod nou
      nou->leg=lst;          // nou va fi primul nod
      nou->m=0;              // completare de la inceput
      nou->val [nou->m++]=x;    // prima valoare din nod
      lst=nou;              // modifica inceput lista
    }
}
  // scoate de la inceput de lista
int pop (unod* & lst) {
```

```
    unod* next;
    lst->m--;
    int x = lst->val[lst->m];          // ultima valoare adaugata
    if(lst->m == 0) {                  // daca nod gol
      next=lst->leg;                   // scoate nod din lista
      delete lst;
      lst=next;                        // modifica inceputul listei
    }
    return x;
}
  // afisare continut lista
void printL (unod* lst) {
 int i;
 while (lst != NULL) {
    for (i=lst->m-1;i>=0;i--)
      printf ("%d ", lst->val[i]);
    printf("\n");
     lst=lst->leg;
 }
}
```

In cazul listelor ordonate noile elemente se pot adãuga în orice nod si de aceea se prevede loc pentru adãugãri (pentru reducerea numãrului de operatii). Adãugarea unui element la un nod plin duce la crearea unui nou nod si la repartizarea elementelor în mod egal între cele douã noduri vecine.

La eliminarea de elemente este posibil ca numãrul de noduri sã scadã prin reunirea a douã noduri vecine ocupate fiecare mai putin de jumãtate.

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

Exemplu de evolutie a unei liste ordonate cu maxim 3 valori pe nod dupã ce se adaugã diverse valori (bara '/' desparte noduri succesive din listã):

adaugã	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

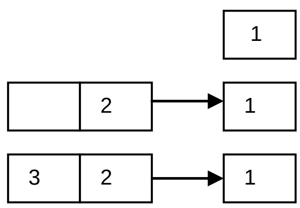
Algoritm de adãugare a unei valori x la o listã ordonatã cu maxim m valori pe nod:

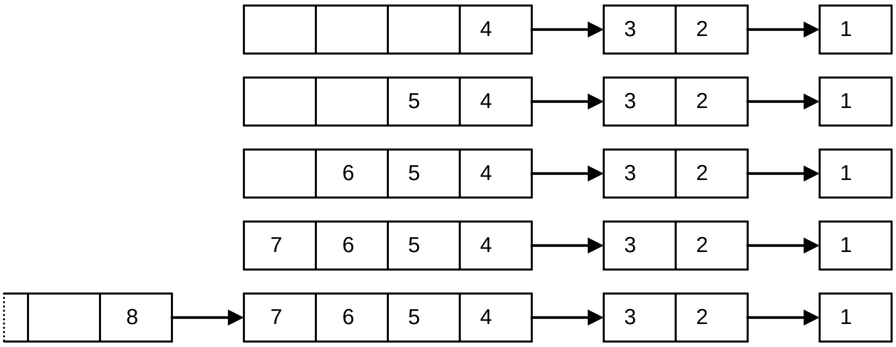
```
cauta nodul p in care va trebui introdus x  ( anterior nodului cu valori > x)
dacã mai este loc în nodul p atunci
   adaugã x în nodul p
dacã nodul p este plin atunci {
   creare nod nou si legare nou dupã nodul p
   copiazã ultimele m/2 valori din p în nou
   daca x trebuie pus in p atunci
       adauga x la nodul p
   altfel
       adauga x la nodul nou
}
```

Exemplu de functii pentru operatii cu o listã ordonatã în care fiecare nod contine între m/2 si 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
  return lst;
}
  // adauga x la nodul p in pozitia idx
void add (unod* p, int x, int idx) {
  int i;
   // deplasare dreapta intre idx si m
  for (i=p→m; i>idx; i--)
     p→val[i]=p→val[i-1];
  p→val[idx]=x;                  // pune x in pozitia idx
  p→m ++;                        // creste dimensiune vector
}
  // insertie x in lista lst
void insL (unod * lst, int x) {
  unod* nou, *p;
  int i,j,idx;
   // cauta locul din lista
  p= find(lst,x,idx);            // localizare x in lista
  if (p→m < M)                   // daca mai e loc in nodul lst
    add(p,x,idx);
  else {                         // daca nodul lst e plin
    nou=(unod*) malloc(sizeof(unod));
    nou→leg=p→leg;               // adauga nou dupa p
    p→leg=nou;
    for (i=0;i<p→m-M/2;i++)      // muta jumatate din valori din p in nou
      nou→val[i]=p→val[M/2+i];
    nou→m=p→m-M/2;
    p→m=M/2;
    if (idx < M/2)               // daca x trebuie pus in p
      add(p,x,idx);              // adauga x la nodul p
    else                     // daca x trebuie pus in nou
      add (nou,x,idx-M/2);
  }
}
```

O listã VList favorizeazã operatia de adãugare la început de listã. Exemplu de evolutie a unei liste VList la adãugarea succesivã a valorilor 1,2,3,4,5,6,7,8:





Fiecare nod dintr-o listã VList contine dimensiunea vectorului din nod (o putere a lui m, unde m este dimensiunea primului nod creat), adresa relativã în nod a ultimului element adãugat, vectorul de elemente si legãtura la nodul urmãtor. Numai primul nod (de dimensiune maximã) 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;
```

In cadrul unui nod elementele se adaugã de la sfârsitul vectorului cãtre începutul sãu, deci valoarea lui i scade de la max la 0. Eliminarea primului element dintr-o listã VList se reduce la incrementarea valorii lui i.

Pentru accesul la un element cu indice dat se comparã succesiv valoarea acestui indice cu dimensiunea fiecãrui nod, pentru a localiza nodul în care se aflã. Probabilitatea de a se afla în primul nod este cca ½ (functie de numãrul efectiv de elemente în primul nod), probabilitatea de a se afla în al doilea nod este ¼ , s.a.m.d.

4.8 TIPUL ABSTRACT LISTÃ (SECVENTÃ)

Vectorii si listele înlãntuite sunt cele mai importante implementãri ale tipului abstract "listã". In literatura de specialitate si în realizarea bibliotecilor de clase existã douã abordãri diferite, dar în esentã echivalente, ale tipului abstract "listã":

1) Tipul abstract "listã" este definit ca o colectie liniarã de elemente, cu acces secvential la elementul urmãtor (si eventual la elementul precedent), dupã modelul listelor înlãntuite. Se foloseste notiunea de element "curent" (pozitie curentã în listã) si operatii de avans la elementul urmãtor si respectiv la elementul precedent.

In aceastã abordare, operatiile specifice clasei abstracte "List" sunt: citire sau modificare valoare din pozitia curentã, inserare în pozitia curentã, avans la elementul urmãtor, pozitionare pe elementul precedent, pozitionare pe început/sfârsit de listã :

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

Pot fi necesare douã operatii de insertie în listã: una dupã pozitia curentã si alta înainte de pozitia curentã. Pozitia curentã se modificã dupã insertie.

Pentru detectarea sfârsitului de listã avem de ales între functii separate care verificã aceste conditii ("end") si modificarea functiei "next" pentru a raporta prin rezultatul ei situatia limitã (1 = modificare reusitã a pozitiei curente, 0 = nu se mai poate modifica pozitia curentã, pentru cã s-a ajuns la sfârsitul listei).

2) Tipul abstract listã este definit ca o colectie de elemente cu acces pozitional, printr-un indice întreg, la orice element din listã, dupã modelul vectorilor. Accesul prin indice este eficient numai pentru vectori, dar este posibil si pentru liste înlãntuite.

In acest caz, operatiile specifice tipului abstract “List” sunt: citire, modificare, inserare, stergere, toate într-o pozitie datã (deci acces pozitional):

```
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
```

Diferenta dintre utilizarea celor douã seturi de operatii este aceeasi cu diferenta dintre utilizarea unui cursor intern tipului listã si utilizarea unui cursor (indice) extern listei si gestionat de programator.

In plus, listele suportã operatii comune oricãrei colectii:

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

O caracteristicã a tipului abstract “Listã” este aceea cã într-o listã nu se fac cãutãri frecvente dupã valoarea (continutul) unui element, dar cãutarea dupã continut poate exista ca operatie pentru orice colectie. In general se prelucreazã secvential o parte sau toate elementele unei liste. In orice caz, lista nu este consideratã o structurã de cãutare ci doar o structurã pentru memorarea temporarã a unor date. Dintr-o listã se poate extrage o sublistã, definitã prin indicii de început si de sfârsit.

O listã poate fi folositã pentru a memora rezultatul parcurgerii unei colectii de orice tip, deci rezultatul enumerãrii elementelor unui arbore, unui tabel de dispersie. Asupra acestei liste se poate aplica ulterior un filtru, care sã selecteze numai acele elemente care satisfac o conditie. Elementele listei nu trebuie sã fie distincte.

Parcurgerea (vizitarea) elementelor unei colectii este o operatie frecventã, dar care depinde de modul de implementare al colectiei. De exemplu, trecerea la elementul urmãtor dintr-un vector se face prin incrementarea unui indice, dar avansul într-o listã se face prin modificarea unui pointer. Pentru a face operatia de avans la elementul urmãtor din colectie independentã de implementarea colectiei s-a introdus notiunea de iterator, ca mecanism de parcurgere a unei colectii.

Iteratorii se folosesc atât pentru colectii liniare (liste,vectori), cât si pentru structuri neliniare (tabel de dispersie, arbori binari si nebinari).

Conceptul abstract de iterator poate fi implementat prin câteva functii: initializare iterator (pozitionare pe primul sau pe ultimul element din colectie), obtinere element din pozitia curentã si avans la elementul urmãtor (sau precedent), verificare sfârsit de colectie. Cursorul folosit de functii pentru a memora pozitia curentã poate fi o variabilã internã sau o variabilã externã colectiei.

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

```
typedef struct {
  Nod* cap, * crt;              // cap lista si pozitia curenta
} List;

  // functii ale mecanismului iterator: first, next, hasNext
```

```
  // 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 oferã acces la elementul urmãtor dintr-o colectie si ascunde detaliile de parcurgere a colectiei, dar limiteazã operatiile asupra colectiei (de exemplu eliminarea elementului din pozitia curentã sau insertia unui nou element în pozitia curentã nu sunt permise de obicei prin iterator deoarece pot veni în conflict cu alte operatii de modificare a colectiei si afecteazã pozitia curentã).

O alternativã este programarea explicitã a vizitãrii elementelor colectiei cu apelarea unei functii de prelucrare la fiecare element vizitat; functia apelatã se numeste si "aplicator" pentru cã se aplicã fiecãrui element din colectie. Exemplu:

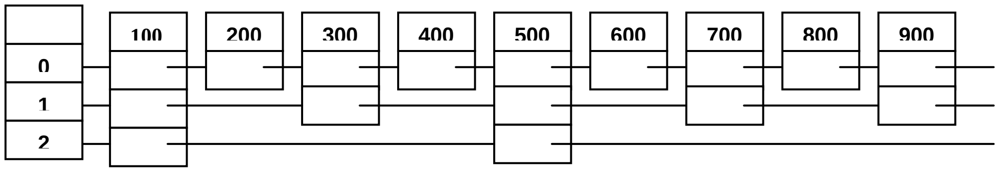
```
   // tip functie aplicator, cu argument adresa date
typedef void (*func)(void *) ;
   // functie de vizitare elemente lista si apel aplicator
void iter ( lista lst, func fp) {
  while ( lst != NULL) {     // repeta cat mai sunt elemente in lista
     (*fp) (lst→ptr);        // apel functie aplicator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

4.9 LISTE "SKIP"

Dezavantajul principal al listelor înlãntuite este timpul de cãutare a unei valori date, prin acces secvential; acest timp este proportional cu lungimea listei. De aceea s-a propus o solutie 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 cãutare comparabil cu alte structuri de cãutare (arbori binari si tabele de dispersie). Timpul mediu de cãutare este de ordinul O(lg n), dar cazul cel mai defavorabil este de ordinul O(n) (spre deosebire de arbori binari echilibrati unde este tot O(lg n).

Adresele de legãturã între elemente sunt situate pe câteva niveluri: pe nivelul 0 este legãtura la elementul imediat urmãtor din listã , pe nivelul 1 este o legãturã la un element aflat la o distantã d1, pe nivelul 2 este o legãturã la un element aflat la o distantã d2 > d1 s.a.m.d. Adresele de pe nivelurile 1,2,3 si urmãtoarele permit "salturi" în listã pentru a ajunge mai repede la elementul cãutat.

O listã skip poate fi privitã ca fiind formatã din mai multe liste paralele, cu anumite elemente comune.



Cãutarea începe pe nivelul maxim si se opreste la un element cu valoare mai micã decât cel cãutat, dupã care continuã pe nivelul imediat inferior s.a.m.d. Pentru exemplul din desen, cãutarea valorii 800 începe pe nivelul 2, “sare” direct si se opreste la elementul cu valoarea 500; se trece apoi pe nivelul 1 si se sare la elementul cu valoarea 700, dupã care se trece pe nivelul 0 si se cautã secvential între 700 si 900.

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

Pentru simplificarea operatiilor cu liste skip, ele au un element santinelã (care contine numãrul maxim de pointeri) si un element terminator cu o valoare superioarã tuturor valorilor din listã sau care este acelasi cu santinela (liste circulare).

Fiecare nod contine un vector de pointeri la elementele urmãtoare de pe câteva niveluri (dar nu si dimensiunea acestui vector) si 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 defineste un nod de lista
   int val;                      // date din fiecare nod
   struct Nod *leg[1];        // legãturi la nodurile urmatoare
} Nod;
```

De observat cã ordinea câmpurilor în structura Nod este importantã, pentru cã vectorul de pointeri poate avea o dimensiune variabilã si deci trebuie sã fie ultimul câmp din structurã.

Functiile urmãtoare lucreazã cu o listã circularã, în care ultimul nod de pe fiecare nivel contine adresa primului nod (santinelã).

```
  // 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 (numãr maxim de pointeri pe nod) poate creste la adãugarea de noduri si poate scãdea la eliminarea de noduri din listã. Pentru a stabili numãrul de pointeri la un nod nou (în functia de adãugare) se foloseste un generator de numere aleatoare în intervalul [0,1]: dacã iese 0 nu se adaugã alti pointeri la nod, dacã iese 1 atunci se adaugã un nou pointer si se repetã generarea unui nou numãr aleator, pânã când iese un 0. In plus, mai punem si conditia ca nivelul sã nu creascã cu mai mult de 1 la o adãugare de element.

Probabilitatea ca un nod sã aibã un pointer pe nivelul 1 este ½, probabilitatea sã aibã un pointer pe nivelul 2 este ¼ s.a.md.

Functia de insertie în listã a unei valori x va realiza urmãtoarele operatii:

```
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 legãturi la nodul urmãtor de pe fiecare nivel
```

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

Pentru a facilita întelegerea operatiei de insertie vom exemplifica cu o listã skip în care pot exista maxim douã niveluri, deci un nod poate contine 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;
}
  // insertie 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 maximã posibilã, simplificã codul operatiilor de insertie si de eliminare noduri într-o listã skip.

4.10 LISTE NELINIARE

Intr-o listã generalã (neliniarã) elementele listei pot fi de douã tipuri: elemente cu date (cu pointeri la date) si elemente cu pointeri la subliste. O listã care poate contine subliste, pe oricâte niveluri de adâncime, este o listã neliniarã.

Limbajul Lisp (“List Processor”) foloseste liste neliniare, care pot contine atât valori atomice (numere, siruri) cât si alte (sub)liste. Listele generale se reprezintã în limbajul Lisp prin expresii; o expresie Lisp contine un numãr oarecare de elemente (posibil zero), încadrate între paranteze si separate prin spatii. Un element poate fi un atom (o valoare numericã sau un sir) sau o expresie Lisp.

In aceste liste se pot memora expresii aritmetice, propozitii si fraze dintr-un limbaj natural sau chiar programe Lisp. Exemple de liste ce corespund unor expresii aritmetice în forma prefixatã (operatorul precede operanzii):

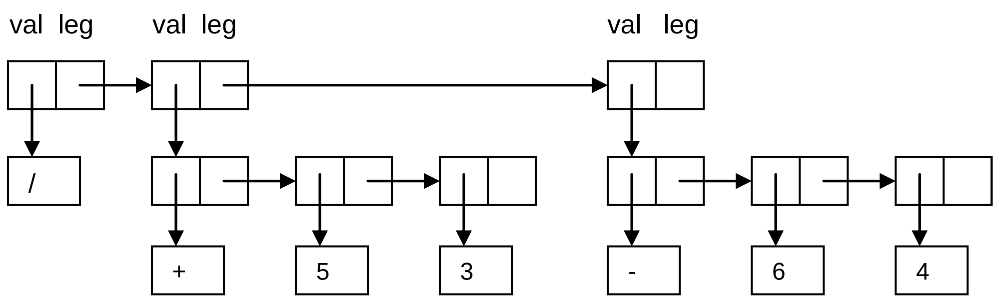
(- 5 3)	5-3	o expresie cu 3 atomi
(+ 1 2 3 4)	1+2+3+4	o expresie cu 5 atomi
(+ 1 (+ 2 (+ 3 4)))	1+2+3+4	o expresie cu 2 atomi si o subexpresie
(/ (+ 5 3) (- 6 4))	(5+3) / (6-4)	o expresie cu un atom si 2 subexpresii

Fiecare element al unei liste Lisp contine douã câmpuri, numite CAR (primul element din listã) si CDR (celelalte elemente sau restul listei). Primul element dintr-o listã este de obicei o functie sau un operator, iar celelalte elemente sunt operanzi.

Imaginea unei expresii Lisp ca listã neliniarã (aici cu douã subliste):

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

O implementare eficientã a unei liste Lisp foloseste douã tipuri de noduri: noduri cu date (cu pointeri la date) si noduri cu adresa unei subliste. Este posibilã si utilizarea unui singur tip de nod cu câte 3 pointeri: la date, la nodul din dreapta (din aceeasi listã) si la nodul de jos (sublista asociatã nodului). In figura urmãtoare am considerat cã elementele atomice memoreazã un pointer la date si nu chiar valoarea datelor, pentru a permite siruri de caractere ca valori.



Structura anterioarã corespunde expresiei fãrã paranteze exterioare /(+53)(-64) iar prezenta parantezelor pe toatã expresia (cum este de fapt în Lisp) necesitã adãugarea unui element initial de tip 1, cu NULL în câmpul “leg” si cu adresa elementului atomic ‘/’ în câmpul “val”.

Urmeazã o definire posibilã a unui nod de listã Lisp cu doi pointeri si un câmp care aratã cum trebuie interpretat primul pointer: ca adresã a unui atom sau ca adresã 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 anterioarã va ocupa 12 octeti. Folosind câmpuri de biti în structuri putem face ca un nod sã 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” si necesitã o conversie înainte de a fi utilizatã. Exemplu de functie care afiseazã o listã Lisp cu atomi 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 reprezintã expresii aritmetice sunt cazuri particulare ale unor expresii ce reprezintã apeluri de functii (în notatia prefixatã) : (f x y …). Mai întâi se evalueazã primul element (functia f), apoi argumentele functiei (x,y,..), si în final se aplicã functia valorilor argumentelor.

Exemplu de functie pentru evaluarea unei expresii aritmetice cu orice numãr de operanzi de o singurã cifrã:

```
  // 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;
}
```

Functia eval1 evalueazã un singur operand (o cifrã sau o listã 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 douã functii (eval si eval1) pot fi reunite într-una singurã.

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

```
  // 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 recursivã pentru crearea unei liste dintr-o expresie Lisp, cu rezultat adresa noului nod creat (care este si adresa listei care începe cu acel nod), dupã 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 listã Lisp se poate reprezenta si ca arbore binar având toate nodurile de acelasi tip: fiecare nod interior este o (sub)listã, fiul stânga este primul element din (sub)listã (o frunzã cu un atom), iar fiul dreapta este restul listei (un alt nod interior sau NIL). Exemplu:

```
      /  \
     +  /  \
        5  /  \
           3  NIL
```

Pentru expresii se foloseste o altã reprezentare prin arbori binari (descrisã 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. Fatã de alte colectii abstracte, multimea are drept caracteristicã definitorie cãutarea unui element dupã continut, cãutare care este o operatie frecventã si de aceea trebuie sã necesite un timp cât mai mic. Principalele operatii cu o multime sunt:

```
void initS ( Set & s );        // creare multime vidã (initializare )
int emptyS ( Set s ) ;         // test de multime vidã : 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 continutului unei multimi s
int sizeS( Set s);             // dimensiune multime
```

Pentru anumite aplicatii sunt necesare si operatii cu douã multimi:

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

Multimea nouã (reuniune, intersectie, diferentã) înlocuieste primul operand (multimea s1). Nu existã operatia de copiere a unei multimi într-o altã multime, dar ea se poate realiza prin initializare si reuniune multime vidã cu multimea sursã :

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

Nu existã comparatie de multimi la egalitate, dar se poate compara diferenta simetricã a douã multimi cu multimea vidã, sau se poate scrie o functie mai performantã pentru aceastã operatie.

Tipul “multime” poate fi implementat prin orice structurã de date: vector, listã cu legãturi sau multime de biti dacã sunt putine elemente în multime. Cea mai simplã implementare a tipului abstract multime este un vector neordonat cu adãugare la sfârsit. Realizarea tipului multime ca o listã înlãntuitã se recomandã pentru colectii de mai multe multimi, cu continut variabil.

Dacã sunt multe elemente atunci se folosesc acele implementãri care realizeazã un timp de cãutare minim: tabel de dispersie si arbore de cãutare echilibrat.

Anumite operatii se pot realiza mai eficient dacã multimile sunt ordonate: cãutare 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 memoreazã prezenta sau absenta unui element (potential) în multime. Bitul din pozitia k este 1 dacã valoarea k apartine multimii si este 0 dacã valoarea k nu apartine multimii. Aceastã reprezentare ocupã putinã memorie si permite cel mai bun timp pentru operatii cu multimi (nu se face o cãutare pentru verificarea apartenentei unei valori x la o multime, ci doar se testeazã bitul din pozitia x).

Pentru multimi realizate ca vectori sau ca liste înlãntuite, operatiile cu o singurã multime se reduc la operatii cu un vector sau cu o listã: initializare, cãutare, adãugare, eliminare, afisare colectie, dimensiune multime si/sau test de multime vidã.

O multime cu valori multiple (“Multiset”) poate contine elemente cu aceeasi valoare, dar nu este o listã (abstractã) pentru cã nu permite accesul direct la elemente. Justificarea existentei unei clase Multiset în limbaje ca Java si C++ este aceea cã 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 numãrului de aparitii a unui element cu valoare datã si eliminarea tuturor aparitiilor unui element dat.

Ideea este de a memora fiecare element distinct o singurã datã dar împreunã cu el se memoreazã si numãrul de aparitii în multime; acesta este un dictionar având drept chei elemente multimii si drept valori asociate numãrul de aparitii (un întreg pozitiv).

5.2 APLICATIE: ACOPERIRE OPTIMÃ CU MULTIMI

Problema acoperirii optime cu multimi ("set cover") este o problemã de optimizare si se formuleazã astfel: Se dã o multime scop S si o colectie C de n multimi candidat, astfel cã orice element din S apartine cel putin unei multimi candidat; se cere sã se determine numãrul minim de multimi candidat care acoperã 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 optimã este :
{ C[2], C[4] }

Algoritmul "greedy" pentru aceastã problemã selecteazã, la fiecare pas, acea multime C[k] care acoperã cele mai multe elemente neacoperite încã din S (intersectia lui C[k] cu S contine numãrul maxim de elemente). Dupã alegerea unei multimi C[k] se modificã multimea scop S, eliminând din S elementele acoperite de multimea C[k] (sau se reunesc candidatii selectati într-o multime auxiliarã A). Ciclul de selectie se opreste atunci când multimea S devine vidã (sau când A contine pe S).

Exemplu de date pentru care algoritmul "greedy" nu determinã solutia optimã :
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 optimã este { C[2], C[3] }

In programul urmãtor se alege, la fiecare pas, candidatul optim, adicã cel pentru care intersectia cu multimea scop are dimensiunea maximã. Dupã afisarea acelei multimi se eliminã din multimea scop elementele acoperite de multimea selectatã.

Colectia de multimi este la rândul ei o multime (sau o listã) 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 înlãntuite sau prin vectori, dar aceastã decizie poate fi amânatã dupã 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 dacã problema admite solutie astfel: se reunesc multimile candidat si se verificã dacã multimea scop este continutã în reuniunea candidatilor:

```
void main () {
  int i;
  getData();                          // citeste multimi scop si candidat
  initS (aux);                        // creare multime vida "aux"
  for (i=1;i<=n;i++)                  // reuniune multimi candidat
    addAll (aux,cand[i]);
  if (! containsAll(aux,scop))
    printf (" nu exista solutie \n");
  else
    setcover();
 }
```

5.3 TIPUL "COLECTIE DE MULTIMI DISJUNCTE"

Unele aplicatii necesitã gruparea elementelor unei multimi în mai multe submultimi disjuncte. Continutul si chiar numãrul multimilor din colectie se modificã de obicei pe parcursul executiei programului. Astfel de aplicatii sunt determinarea componentelor (subgrafurilor) conexe ale unui graf si determinarea claselor de echivalentã pe baza unor relatii de echivalentã.

O multime din colectie nu este identificatã printr-un nume sau un numãr, ci printr-un element care apartine multimii. De exemplu, o componentã conexã dintr-un graf este identificatã printr-un numãr de nod aflat în componenta respectivã.

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

Operatiile asociate tipului abstract "colectie de multimi disjuncte" sunt:
- Initializare colectie c de n multimi, fiecare multime k cu o valoare k: init (c,n)
- Gãsirea multimii dintr-o colectie c care contine o valoare datã x: find (c,x)
- Reuniunea multimilor din colectia c ce contin valorile x si y : union (c,x,y)

In aplicatia de componente conexe se creazã initial în colectie câte o multime pentru fiecare nod din graf, iar apoi se reduce treptat numãrul de multimi prin analiza muchiilor existente. Dupã epuizarea listei de muchii fiecare multime din colectie reprezintã un subgraf conex. Dacã graful dat este conex, atunci în final colectia va contine o singurã multime.

Cea mai bunã implementare pentru “Disjoint Sets” foloseste tot un singur vector de întregi, dar acest vector reprezintã o pãdure de arbori. Elementele vectorului sunt indici (adrese) din acelasi vector cu semnificatia de pointeri cãtre nodul pãrinte. Fiecare multime este un arbore în care fiecare nod (element) contine o legãturã la pãrintele sãu, dar nu si legãturi cãtre fii sãi. Rãdãcina fiecãrui arbore poate contine ca legãturã la pãrinte fie chiar adresa sa, fie o valoare nefolositã ca indice (-1).

Pentru datele folosite anterior (8 vârfuri în 3 componente conexe), starea finalã a vectorului ce reprezintã colectia si arborii corespunzãtori aratã astfel:

valoare	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
legãtura	-1	-1	1	-1	2	3	5	4

```
1        2        4
|        |        |
3        5        8
|        |
6        7
```

In functie de codul folosit sunt posibile si alte variante, dar tot cu trei arbori si cu aceleasi noduri (se modificã doar rãdãcina si structura arborilor).

Dacã se mai adaugã o muchie 3-7 atunci se reunesc arborii cu rãdãcinile în 1 si 2 într-un singur arbore, iar în vectorul ce reprezintã cei doi arbori rãmasi se modificã legãtura lui 2 (p[2]=1).

```
          1              4
         / \             |
        3   2            8
       /     \
      6       5
               \
                7
```

Gãsirea multimii care contine o valoare datã x se reduce la aflarea rãdãcinii arborelui în care se aflã x, mergând în sus de la x cãtre rãdãcinã. Reunirea arborilor ce contin un x si un y se face prin legarea rãdãcinii arborelui y ca fiu al rãdãcinii arborelui x (sau al arborelui lui x la arborele lui y).

Urmeazã functiile ce realizeazã operatiile specifice tipului "Disjoint Sets":

```
typedef struct {
  int p[M];          // legaturi la noduri parinte
  int n;             // dimensiune vector
} ds;
  // initializare colectie
void init (ds & c, int n) {
  int i;
  c.n=n;
  for (i=1;i<=n;i++)
    c.p[i]=-1;          // radacina contine legatura -1
}
  // determina multimea care contine 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 contin valorile x si 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;
}
```

In aceastã variantã operatia de cãutare are un timp proportional cu adâncimea arborelui, iar durata operatiei de reuniune este practic aceeasi cu durata lui "find". Pentru reducerea în continuare a duratei operatiei "find" s-au propus metode pentru reducerea adâncimii arborilor. Modificãrile au loc în algoritm, dar structura de date rãmâne practic neschimbatã (tot un vector de indici cãtre noduri pãrinte).

Prima idee este ca la reunirea a doi arbori în unul singur sã se adauge arborele mai mic (cu mai putine noduri) la arborele mai mare (cu mai multe noduri). O solutie simplã este ca numãrul de noduri dintr-un arbore sã se pãstreze în nodul rãdãcinã, ca numãr negativ. Functia de reuniune de multimi va arãta astfel:

```
void unif ( ds & c,int x,int y) {
  int cx,cy;
  cx=find(c,x); cy=find(c,y);        // indici noduri radacina
  if (cx ==cy) return;               // daca x si y in acelasi arbore
```

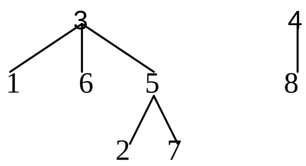
```
  if ( c.p[cx] <= c.p[cy]) {          // daca arborele cx este mai mic ca cy
     c.p[cx] += c.p[cy];              // actualizare nr de noduri din cx
     c.p[cy]=cx;                      // leaga radacina cy ca fiu al nodului cx
  } else {                            // daca arborele cy este mai 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 cãutãrii într-un arbore sã se modifice legãturile astfel ca toate nodurile din arbore sã fie legate direct la rãdãcina arborelui:

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

In cazul unui graf cu 8 vârfuri si muchiile 3-6, 5-7, 1-3, 2-5, 4-8, 1-6, 3-7 vor fi doi arbori cu 6 si respectiv 2 noduri, iar vectorul p de legãturi la pãrinti va arãta astfel:

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



Dacã se mai adaugã o muchie 2-4 atunci înãltimea arborelui rãmas va fi tot 2 iar nodul 4 va avea ca pãrinte rãdãcina 3.

Reuniunea dupã dimensiunea arborilor are drept efect proprietatea cã nici un arbore cu n noduri nu are înãltime mai mare ca log(n). Prin reunirea a doi arbori numãrul de noduri din arborele rezultat creste cel putin de douã ori (se dubleazã), dar înãltimea sa creste numai cu 1. Deci raportul dintre înãltimea unui arbore si numãrul sãu de noduri va fi mereu de ordinul $\log_2(n)$. Rezultã cã si timpul mediu de cãutare într-un arbore cu n noduri va creste doar logaritmic în raport cu dimensiunea sa.

Ca solutie alternativã se poate pãstra înãltimea fiecãrui arbore în locul numãrului de noduri, pentru a adãuga arborele cu înãltime mai micã la arborele cu înãltime mai mare.

5.4 TIPUL ABSTRACT "DICTIONAR"

Un dictionar (“map”), numit si tabel asociativ, este o colectie de perechi cheie - valoare, în care cheile sunt distincte si sunt folosite pentru regãsirea rapidã a valorilor asociate. Un dictionar este o structurã pentru cãutare rapidã (ca si multimea) având aceleasi implementãri: vector sau listã de înregistrãri dacã sunt putine chei si tabel de dispersie (“hash”) sau arbore binar echilibrat de cãutare dacã sunt multe chei si timpul de cãutare este important. Cheia poate fi de orice tip.

Un dictionar poate fi privit ca o multime de perechi cheie-valoare, iar o multime poate fi privitã ca un dictionar în care cheia si valoarea sunt egale. Din acest motiv si implementãrile principale ale celor douã tipuri abstracte sunt aceleasi.

Operatiile principale specifice unui dictionar, dupã modelul Java, sunt :

Introducerea unei perechi cheie-valoare într-un dictionar:

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

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

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

Eliminarea unei perechi cu cheie datã dintr-un dictionar:

```
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 (adevãrat) dacã cheia "key" este gãsitã sau 0 (fals) dacã cheia "key" nu este gãsitã. Functia "putD" modificã dictionarul, prin adãugarea unei noi perechi la dictionar sau prin modificarea valorii asociate unei chei existente.

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

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

Este importantã precizarea cã executia functiei "putD" cu o cheie existentã în dictionar nu adaugã un nou element (nu pot exista mai multe perechi cu aceeasi cheie) ci doar modificã valoarea asociatã cheii existente. De aici si numele functiei "put" (pune în dictionar) în loc de "add" (adãugare), ca la multimi. In functie de implementare, operatia se poate realiza prin înlocuirea valorii asociate cheii existente, sau prin eliminarea perechii cu aceeasi cheie, urmatã de adãugarea unei noi perechi.

Operatiile "getD" si "putD" necesitã o cãutare în dictionar a cheii primite ca argument, iar aceastã operatie poate fi realizatã ca o functie separatã.

Implementãrile cele mai bune pentru dictionare sunt:
- Tabel de dispersie ("Hash table")
- Arbori binari echilibrati de diferite tipuri
- Liste "skip"

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

Pentru un dictionar cu numãr mic de chei se poate folosi si o implementare simplã printr-o listã înlãntuitã, sau prin doi vectori (de chei si de valori) sau printr-un singur vector de structuri, care poate fi si ordonat dacã este nevoie.

De cele mai multe ori fiecare cheie are asociatã o singurã valoare, dar existã si situatii când o cheie are asociatã o listã de valori. Un exemplu este un index de termeni de la finalul unei cãrti tehnice, în care fiecare cuvânt important (termen tehnic) este trecut împreunã cu numerele paginilor unde apare acel cuvânt. Un alt exemplu este o listã de referinte încrucisate, cu fiecare identificator dintr-un program sursã însotit 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 aratã în ce linii dintr-un text sursã este folosit fiecare identificator. Exemplu de date initiale:

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

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

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

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

Putem privi aceastã listã 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 implementãrile unui dictionar simplu poate fi folositã si pentru un multidictionar, dacã se înlocuieste valoarea asociatã unei chei cu lista valorilor asociate acelei chei (un pointer la o listã înlãntuitã, în limbajul C).

O variantã de dictionar este dictionarul bidirectional (reversibil), numit "BiMap", în care si valorile sunt distincte putând fi folosite drept chei de cãutare într-un dictionar "invers". La încercarea de adãugare a unei perechi cheie-valoare ("putD") se poate elimina o pereche anterioarã cu aceeasi valoare si deci dimensiunea dictionarului BiMap poate creste, poate rãmâne neschimbatã (dacã existã o pereche cu aceeasi cheie dar cu valoare diferitã) sau poate sã scadã (dacã existã o pereche cu aceeasi cheie si o pereche cu aceeasi valoare). Structurile folosite de un BiMap nu diferã de cele pentru un dictionar simplu, dar diferã functia de adãugare la dictionar.

5.5 IMPLEMENTARE DICTIONAR PRIN TABEL DE DISPERSIE

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

Un tabel de dispersie ("hash table") este un vector pentru care pozitia unde trebuie introdus un nou element se calculeazã din valoarea elementului, iar aceste pozitii rezultã în general dispersate, fiind determinate de valorile elementelor si nu de ordinea în care ele au fost adãugate. O valoare nouã nu se adaugã în prima pozitie liberã ci într-o pozitie care sã permitã regãsirea rapidã a acestei valori (fãrã cãutare).

Ideea este de a calcula pozitia unui nou element în vector în functie de valoarea elementului. Acelasi calcul se face atât la adãugare cât si la regãsire :

- Se reduce cheia la o valoare numericã (dacã nu este deja un numãr întreg pozitiv);
- Se transformã numãrul obtinut (codul "hash") într-un indice corect pentru vectorul respectiv; de regulã acest indice este egal cu restul împãrtirii prin lungimea vectorului (care e bine sã fie un numãr prim). Se pot folosi si alte metode care sã producã numere aleatoare uniform distribuite pe multimea de indici în vector.

Procedura de calcul a indicelui din cheie se numeste si metodã de dispersie, deoarece trebuie sã asigure dispersia cât mai uniformã a cheilor pe vectorul alocat pentru memorarea lor.

Codul hash se calculeazã de obicei din valoarea cheii. De exemplu, pentru siruri de caractere codul hash se poate calcula dupã o relatie de forma:

(Σ s[k] * (k+1)) % m suma ptr k=0, strlen(s)

unde s[k] este caracterul k din sir, iar m este valoarea maximã pentru tipul întreg folosit la reprezentarea codului (int sau long). In esentã este o sumã ponderatã cu pozitia în sir a codului caracterelor din sir (sau a primelor n caractere din sir).

O variantã a metodei anterioare este o sumã modulo 2 a caracterelor din sir.

Orice metodã de dispersie conduce inevitabil la aparitia de "sinonime", adicã chei (obiecte) diferite pentru care rezultã aceeasi pozitie în vector. Sinonimele se numesc si "coliziuni" pentru cã mai multe obiecte îsi disputã o aceeasi adresã în vector.

Un tabel de dispersie se poate folosi la implementarea unei multimi sau a unui dictionar; diferentele apar la datele continute si la functia de punere în dictionar a unei perechi cheie-valoare (respectiv functia de adãugare la multime).

Pentru a exemplifica sã considerãm un tabel de dispersie de 5 elemente în care se introduc urmãtoarele chei: 2, 3, 4, 5, 7, 8, 10, 12. Resturile împãrtirii prin 5 ale acestor numere conduc la indicii: 2, 3, 4, 0, 2, 3, 0, 2. Dupã plasarea primelor 4 chei, în pozitiile 2,3,4,0 rãmâne liberã pozitia 1 si vor apãrea coliziunile 7 cu 2, 8 cu 3, 10 si 12 cu 5. Se observã cã este importantã si ordinea de introducere a cheilor într-un tabel de dispersie, pentru cã ea determinã continutul acestuia.

O altã dimensiune a vectorului (de exemplu 7 în loc de 5) ar conduce la o altã distributie a cheilor în vector si la alt numãr de coliziuni.

Metodele de redistribuire a sinonimelor care poat fi grupate în:

1) Metode care calculeazã o nouã adresã în acelasi vector pentru sinonimele ce gãsesc ocupatã pozitia rezultatã din calcul : fie se cautã prima pozitie liberã ("open-hash"), fie se aplicã o a doua metodã de dispersie pentru coliziuni ("rehash"), fie o altã solutie. Aceste metode folosesc mai bine memoria dar pot necesita multe comparatii. Pentru exemplul anterior, un tabel hash cu 10 pozitii ar putea arãta astfel:

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

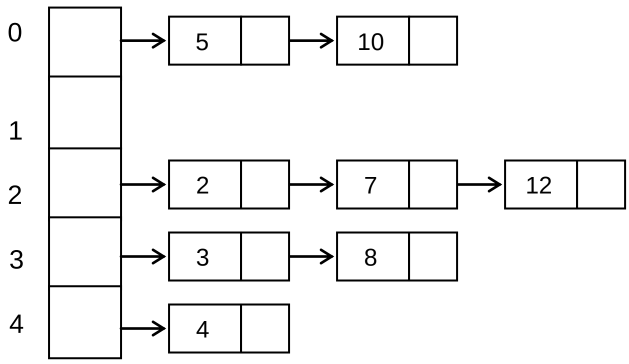
Pentru regãsirea sirului 12 se calculeazã adresa 2 (12%10) si apoi se mai fac 5 comparatii pentru a gãsi sirul în una din urmãtoarele pozitii. Numãrul de comparatii depinde de dimensiunea vectorului si poate fi foarte mare pentru anumite coliziuni.

O variantã este utilizarea a doi vectori: un vector în care se pun cheile care au gãsit liberã pozitia calculatã si un vector cu coliziuni (chei care au gãsit pozitia ocupatã):

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

2) Metode care plaseazã coliziunile în liste înlãntuite de sinonime care pleacã din pozitia rezultatã din calcul pentru fiecare grup de sinonime. Aceastã metodã asigurã un timp mai bun de regãsire, dar foloseste mai multã memorie pentru pointeri. Este metoda preferatã în clasele multime sau dictionar pentru cã nu necesitã o estimare a numãrului maxim de valori (chei si valori) ce vor introduse în multime sau dictionar.

In acest caz tabelul de dispersie este un vector de pointeri la liste de sinonime, iar câstigul de timp provine din faptul cã nu se cautã într-o listã a tuturor cheilor si se cautã numai în lista de sinonime care poate contine cheia cãutatã. Este de dorit ca listele de sinonime sã fie de dimensiuni cât mai apropiate. Dacã listele devin foarte lungi se va reorganiza tabelul prin extinderea vectorului si mãrirea numãrului de liste.



Avantajele structurii anterioare sunt timpul de cãutare foarte bun si posibilitatea de extindere nelimitatã (dar cu degradarea performantelor). Timpul de cãutare depinde de mai multi factori si este greu de calculat, dar o estimare a timpului mediu este O(1), iar cazul cel mai defavorabil este O(n).

Un dezavantaj al tabelelor de dispersie este acela cã datele nu sunt ordonate si cã se pierde ordinea de adãugare la tabel. O solutie este adãugarea unei liste cu toate elementele din tabel, în ordinea introducerii lor.

De observat cã în liste sau în vectori de structuri se poate face cãutare dupã diverse chei dar în tabele de dispersie si în arbori aceastã cãutare este posibilã numai dupã o singurã cheie, stabilitã la crearea structurii si care nu se mai poate modifica sau înlocui cu o altã cheie (cu un alt câmp).

Ideea înlocuirii unui sir de caractere printr-un numãr întreg (operatia de "hashing") are si alte aplicatii: un algoritm eficient de cãutare a unui sir de caractere într-un alt sir (algoritmul Rabin-Karp), în metode de criptare a mesajelor s.a.

In exemplul urmãtor se foloseste un dictionar tabel de dispersie în problema afisãrii numãrului de aparitii al fiecãrui cuvânt distinct dintr-un text; cheile sunt siruri de caractere iar valorile asociate sunt numere întregi (numãr de repetãri 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 structurã care sã includã vectorul de pointeri, dimensiunea lui (stabilitã la initializarea dictionarului) si functia folositã 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 folositã metodã de compresie a datelor deoarece nu necesitã informatii prealabile despre datele comprimate (este o metodã adaptivã) si este cu atât mai eficace cu cât fisierul initial este mai mare si contine mai multã redundantã.

Pentru texte scrise în englezã sau în românã rezultatele sunt foarte bune doarece ele folosesc în mod repetat anumite cuvinte uzuale, care vor fi înlocuite printr-un cod asociat fiecãrui cuvânt.

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

Metoda foloseste un dictionar prin care asociazã 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 datã când se gãseste o secventã de caractere care nu exista anterior în dictionar.

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

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

Din motive de eficientã pot exista diferente importante între descrierea principialã a metodei LZW si implementarea ei în practicã; astfel, sirurile de caractere se reprezintã 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 singurã trecere, fãrã revenire. La stânga pozitiei curente sunt subsiruri deja codificate, iar la dreapta cursorului se cautã cea mai lungã secventã care existã deja în dictionar. Odatã gãsitã aceastã secventã, ea este înlocuitã prin codul asociat deja si se adaugã la dictionar o secventã cu un caracter mai lungã.

Pentru exemplificare vom considera cã textul de codificat contine numai douã caractere ('a' si 'b') si aratã astfel (sub text sunt trecute codurile asociate secventelor respective):

```
a  b  b  a  a  b  b a  a b  a b b  a 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 urmãtorul 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 variantã putin modificatã se asociazã codul 0 cu sirul nul, dupã care toate secventele de unul sau mai multe caractere sunt codificate printr-un numãr întreg si un caracter:
1=0a / 2=0b / 3=1b (ab) / 4=2b (bb) / 5=2a (ba) / ...

Urmeazã o descriere posibilã pentru algoritmul de compresie LZW:

```
initializare dictionar cu n coduri de caractere individuale
w = NIL;  k=n                  // w este un cuvant (o secventa de caractere)
repeta cat timp mai exista caractere neprelucrate
    citeste un caracter c
    daca w+c exista in dictionar      // ‘+’ pentru concatenare de siruri
      w = w+c                         // prelungeste secventa w cu caracterul c
    altfel
      adauga wc la dictionar cu codul k=k+1
      scrie codul  lui w
      w = c
```

Este posibilã si urmãtoarea formulare a algoritmului de compresie LZW:

```
initializare dictionar cu toate secventele de lungime 1
repeta cat timp mai sunt caractere
   cauta cea mai lunga secventa de car. w care apare in dictionar
   scrie pozitia lui w in dictionar
   adauga w plus caracterul urmator la dictionar
```

Aplicarea acestui algoritm pe sirul “abbaabbaababbaaaabaabba” conduce la secventa de pasi rezumatã în tabelul urmãtor:

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)

In exemplul urmãtor codurile generate sunt afisate 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 dacã folosim drept chei ‘w’ întregi mici (“short”) obtinuti din codul k si caracterul ‘c’ adãugat la secventa cu codul k.

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

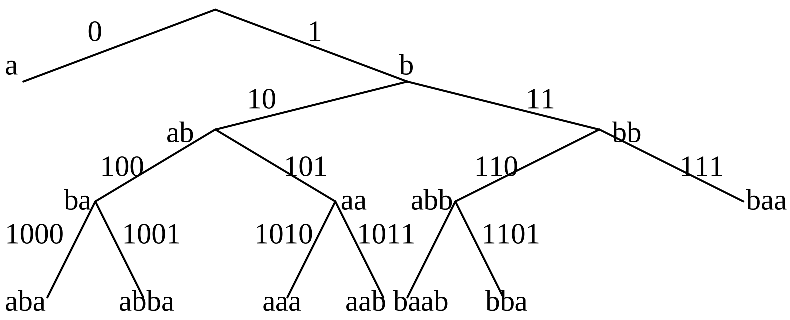
La decompresie se analizeazã un sir de coduri numerice, care pot reprezenta caractere individuale sau secvente de caractere. Cu ajutorul dictionarului se decodificã fiecare cod întâlnit. Urmeazã o descriere posibilã pentru algoritmul de decompresie LZW:

```
initializare dictionar cu codurile de caractere individuale
citeste primul cod k;
w = sirul din pozitia k a dictionarului;
repeta 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 numãr variabil de biti, iar la decompresie se poate determina numãrul de biti în functia de dimensiunea curentã 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, acelasi cu dimensiunea curentã a dictionarului, se observã cã numãrul de biti pe acest nivel este $\log_2(k) + 1$.

Capitolul 6

STIVE SI COZI

6.1 LISTE STIVÃ

O stivã este o listã cu acces la un singur capãt, 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 stivã) si "pop" (scoate din stivã) si nu mai specificã pozitia din listã, care este implicitã . O stivã mai este numitã si listã LIFO ('Last In First Out'), deoarece ultimul element pus este primul care va fi extras din stivã.

Operatiile asociate tipului abstract "stivã" sunt:
- initializare stivã vidã (initSt)
- test stivã vidã (emptySt)
- pune un obiect pe stivã (push)
- extrage obiectul din vârful stivei (pop)
- obtine valoare obiect din vârful stivei, fãrã scoatere din stivã (top)

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

In STL operatia de scoatere din stivã nu are ca rezultat valoarea scoasã din stivã, deci sunt separate operatiile de citire vârf stivã si de micsorare dimensiune stivã.

O solutie simplã este folosirea directã a unui vector, cu adãugare la sfârsit (în prima pozitie liberã) pentru "push" si extragerea ultimului element, pentru "pop".
Exemplu de afisare a unui numãr întreg fãrã semn în binar, prin memorarea în stivã a resturilor împãrtirii prin 2, urmatã 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 stivã sau ca pozitie a ultimului element pus în stivã. 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 si pentru alte colectii de date, vom prefera definirea unor functii pentru operatii asociate structurii de stivã. Vom exemplifica cu o stivã realizatã ca un vector, cu adãugare si stergere la sfârsitul vectorului.

```
#define M 100                  // dimens 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 stivã este dificilã în general, dar putem folosi un vector extensibil dinamic (alocat si realocat dinamic). Modificãrile apar numai initializarea stivei si la punere în stivã.

De asemenea, se poate folosi o listã înlãntuitã cu adãugare si extragere numai la începutul listei (mai rapid si 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;
}
```

Dacã sunt necesare stive cu continut diferit în acelasi program sau dacã în aceeasi stivã trebuie memorate date de tipuri diferite vom folosi o stivã de pointeri "void*".

Prima si cea mai importantã utilizare a unei stive a fost în traducerea apelurilor de functii, pentru revenirea corectã dintr-o secventã de apeluri de forma:

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

In stivã se pun succesiv adresele a,a1 si a2 pentru ca la iesirea din f3 sã se sarã la a2, la iesirea din f2 se sare la a1, si la iesirea din f1 se revine la adresa ‘a’.

Pentru executia corectã a functiilor recursive se vor pune în aceeasi stivã si valorile argumentelor formale si ale variabilelor locale.

Aplicatiile stivelor sunt cele în care datele memorate temporar în lista stivã se vor utiliza în ordine inversã punerii lor în stivã, cum ar fi în memorarea unor comenzi date sistemului de operare (ce pot fi readuse spre executie), memorarea unor modificãri asupra unui text (ce pot fi anulate ulterior prin operatii de tip “undo”), memorarea paginilor Web afisate (pentru a se putea reveni asupra lor) sau memorarea marcajelor initiale (“start tags”) dintr-un fisier XML, pentru verificarea utilizãrii lor corecte, împreunã cu marcajele finale (“end tags”).

Cealalatã categorie importantã de aplicatii sunt cele în care utilizarea stivei este solutia alternativã (iterativã) a unor functii recursive (direct sau indirect recursive).

6.2 APLICATIE : EVALUARE EXPRESII ARITMETICE

Evaluarea expresiilor aritmetice este necesarã într-un program interpretor BASIC, într-un program de calcul tabelar (pentru formulele care pot apare în celulele foii de calcul) si în alte programe care admit ca intrãri expresii (formule) si care trebuie sã furnizeze rezultatul acestor expresii.

Pentru simplificare vom considera numai expresii cu operanzi numerici întregi de o singurã cifrã, la care rezultatele intermediare si finale sunt tot întregi de o cifrã.

Problema evaluãrii expresiilor este aceea cã ordinea de aplicare a operatorilor din expresie (ordinea de calcul) este diferitã în general de ordinea aparitiei acestor operatori în expresie (într-o parcurgere de la stânga la dreapta). Exemplu:

```
( 5 – 6 / 2 ) * ( 1+ 3 )
```

Evaluarea acestei expresii necesitã calculele urmãtoare (în aceastã ordine):
6 / 2 = 3, 5 – 3 = 2, 1 + 3 = 4, 2 * 4 = 8

Ordinea de folosire a operatorilor este determinatã de importanta lor (înmultirea si împãrtirea sunt mai importante ca adunarea si scãderea) si de parantezele folosite.

Una din metodele de evaluare a expresiilor necesitã douã etape si fiecare din cele douã etape utilizeazã câte o stivã :

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

In forma postfixatã a unei expresii nu mai existã paranteze, iar un operator (binar) apare dupã cei doi operanzi folositi de operator. Exemple de expresii postfixate:

Expresie infixata	Expresie postfixata
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 acelasi tip de stivã sau stive diferite ca tip de date.

Comparând cele douã forme ale unei expresii se observã cã ordinea operanzilor se pãstreazã în sirul postfixat, dar operatorii sunt rearanjati în functie de importanta lor si de parantezele existente. Deci operanzii trec direct din sirul infixat în sirul postfixat, iar operatorii trec în sirul postfixat numai din stivã. Stiva memoreazã temporar operatorii pânã când se decide scoaterea lor în sirul postfixat.

Algoritmul de trecere la forma postfixatã cu stivã de operatori aratã astfel:

```
repetã pânã la terminarea sirului infixat
   extrage urmatorul caracter din sir in ch
   daca ch este operand atunci trece ch in sirul postfixat
   daca ch este '(' atunci se pune ch in stiva
   daca ch este ')' atunci
      repeta pana la '('
         extrage din stiva si trece in sirul postfixat
      scoate '(' din stiva
   daca ch este operator atunci
      repeta cat timp stiva nu e goala si prior(ch) <=  prior(operator din varful stivei)
        scoate operatorul din stiva in sirul postfixat
      pune ch in stiva
   scoate operatori din stiva in sirul postfixat
```

Functia urmãtoare foloseste o stivã de caractere:

```
void topostf (char * in, char * out) {
  Stack st;                  // stiva de operatori
  char ch,op;
  initSt (st);               // initializare stiva
  while (*in !=0) {              // repeta pana la sfarsit sir infixat
     while (*in==' ') ++in;     // ignora spatii dintre elementele expresiei
     ch=*in++;                  // urmatorul caracter din sirul infixat
     if (isdigit(ch))           // daca ch este operand
       *out++=ch;               // trece ch in sir postfixat
     if (ch=='(')
       push(st,ch);             // pune paranteze deschise in stiva
     if (ch==')')
        // scoate din stiva toti operatorii pana la o paranteza deschisa
        while (!emptySt(st) && ( op=pop(st)) != '(')
              *out++=op;            // si trece operatori in sirul postfixat
     else {                         // daca 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 rãmâne în stivã operatori, care trebuie scosi în sirul postfixat. O altã solutie este sã se punã de la început în stivã un caracter folosit si ca terminator de expresie (';'), cu prioritate minimã. Altã solutie adaugã paranteze în jurul expresiei primite si repetã ciclul principal pânã la golirea stivei (ultima parantezã din sirul de intrare va scoate din stivã toti operatorii rãmasi).

Evaluarea expresiei postfixate parcurge expresia de la stânga la dreapta, pune pe stivã toti operanzii întâlniti, iar la gãsirea unui operator aplicã acel operator asupra celor doi operanzi scosi din vârful stivei si pune în stivã rezultatul partial obtinut.

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 singurã cifrã:

```
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
}
```

Functia "calc" calculeazã 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 si printr-o functie recursivã, fãrã a recurge la o stivã. Ideea este aceea cã orice operator se aplicã asupra rezultatelor unor subexpresii, deci se poate aplica definitia recursivã urmãtoare:

<pf> ::= <val> | <pf> <pf> <op>

unde: <pf> este o expresie postfixatã, <val> este o valoare (un operand numeric) si <op> este un operator aritmetic.

Expresia postfixatã este analizatã 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));
}
```

Functia recursivã 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 postfixatã se face prin asa-numita analizã descendent recursivã, cu o recursivitate indirectã de forma:

A→B→A sau A→B→C→A

Regulile gramaticale folosite în analiza descendent recutsivã sunt urmãtoarele:

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

Functiile urmãtoare realizeazã analiza si interpretarea unor expresii aritmetice corecte sintactic. Fiecare functie primeste un pointer ce reprezintã pozitia curentã în expresia analizatã, modificã acest pointer si are ca rezultat valoarea (sub) expresiei. Functia "expr" este apelatã o singurã datã în programul principal si poate apela de mai multe ori functiile "term" si "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 apelatã
  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 apelatã
  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
}
```

Desi se bazeazã pe definitii recursive, functiile de analizã a subexpresiilor nu sunt direct recursive, folosind o rescriere iterativã a definitiilor dupã cum urmeazã:

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

6.3 ELIMINAREA RECURSIVITÃTII FOLOSIND O STIVÃ

Multe aplicatii cu stive pot fi privite si ca solutii alternative la functii recursive pentru aceleasi aplicatii.

Eliminarea recursivitãtii este justificatã atunci când dimensiunea maximã a stivei utilizate de compilator limiteazã dimensiunea problemei care trebuie rezolvatã prin algoritmul recursiv. In stiva implicitã se pun automat parametri formali, variabilele locale si adresa de revenire din functie.

Functiile cu un apel recursiv urmat de alte operatii sau cu mai multe apeluri recursive nu pot fi rescrise iterativ fãrã a utiliza o stivã.

In exemplul urmãtor se afiseazã un numãr întreg n în binar (în baza 2) dupã urmãtorul rationament: sirul de cifre pentru n este format din sirul de cifre pentru (n/2) urmat de o cifrã 0 sau 1 care se obtine ca n % 2. De exemplu, numãrul n = 22 se afiseazã 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 recursivã a functiei de afisare 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 urmãtor aratã cum se poate folosi o stivã pentru rescrierea iterativã a functiei recursive de afisare 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
  }
}
```

In cazul functiilor cu mai multe argumente se va folosi fie o stivã de structuri (sau de pointeri la structuri), fie o stivã matrice, în care fiecare linie din matrice este un element al stivei (dacã toate argumentele sunt de acelasi tip).

Vom exemplifica prin functii nerecursive de sortare rapidã ("quick sort"), în care se pun în stivã numai argumentele care se modificã între apeluri (nu si vectorul 'a').

Functia urmãtoare foloseste o stivã 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 (! sempty(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 cã argumentele trebuie scoase din stivã în ordine inversã introducerii lor, iar când sunt mai multe argumente se pot face erori.

In functia următoare se foloseste o stivã realizatã ca matrice cu douã coloane, iar punerea pe stivã înseamnã adãugarea 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 modificã între apeluri) sunt de tipuri diferite se va folosi o stivã 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 parametri 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 identicã cu utilizarea stivei matrice, adicã functiile "push" si "pop" au mai multe argumente, în aceeasi ordine pentru ambele functii.

6.4 LISTE COADÃ

O coadã ("Queue"), numitã si listã FIFO ("First In First Out") este o listã la care adãugarea se face pe la un capãt (de obicei la sfârsitul cozii), iar extragerea se face de la celalalt capãt (de la începutul cozii). Ordinea de extragere din coadã este aceeasi cu ordinea de introducere în coadã, ceea ce face utilã o coadã în aplicatiile unde ordinea de servire este aceeasi cu ordinea de sosire: procese de tip "vânzãtor - client" sau "producãtor - consumator". In astfel de situatii coada de asteptare este necesarã pentru a acoperi o diferentã temporarã între ritmul de servire si ritmul de sosire, deci pentru a memora temporar cereri de servire (mesaje) care nu pot fi încã prelucrate.

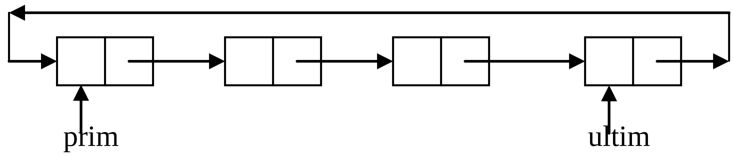
Operatiile cu tipul abstract "coadã" sunt:

- initializare coadã (initQ)
- test coadã goalã (emptyQ)
- adaugã un obiect la coadã (addQ, insQ, enqueue)
- scoate un obiect din coadã (delQ, dequeue)

In STL existã în plus operatia de citire din coadã, fãrã eliminare din coadã. Ca si alte liste abstracte, cozile pot fi realizate ca vectori sau ca liste înlãntuite, cu conditia suplimentarã ca durata operatiilor addQ si delQ sã fie minimã (O(1)).

O coadã înlãntuitã poate fi definitã prin :

- Adresa de început a cozii, iar pentru adãugare sã se parcurgã toatã coada (listã) pentru a gãsi ultimul element (durata operatiei addQ va fi O(n));
- Adresele primului si ultimului element, pentru a elimina timpul de parcurgere a listei la adãugare;
- Adresa ultimului element, care contine adresa primului element (coadã circularã).



Programul urmãtor foloseste o listã circularã definitã prin adresa ultimului element din coadã, fãrã element santinelã:

```
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;
    q=p;                  // 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 numãrul maxim de valori ce pot fi memorate temporar în coadã. Caracterul circular permite reutilizarea locatiilor eliberate prin extragerea unor valori din coadã.

Câmpul "ultim" contine indicele din vector unde se va adãuga un nou element, iar "prim" este indicele primului (celui mai vechi) element din coadã. Deoarece “prim” si “ultim” sunt egale si când coada e goalã si când coada e plinã, vom memora si numãrul de elemente din coadã. Exemplu de coadã realizatã 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 secventã de operatii cu o coadã 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 coadã poate prelua temporar un numãr variabil de elemente, care vor fi folosite în aceeasi ordine în care au fost introduse în coadã. In sistemele de operare apar cozi de procese aflate într-o anumitã stare (blocate în asteptarea unor evenimente sau gata de executie dar cu prioritate mai micã decât procesul în executie). Simularea unor procese de servire foloseste de asemenea cozi de clienti în asteptarea momentului când vor putea fi serviti (prelucrati).

Intr-un proces de servire existã 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 simulãrii este obtinerea unor date statistice, cum ar fi timpul mediu si maxim dintre sosire si plecare client, numãrul mediu si maxim de clienti în coada de asteptare la statie, în vederea îmbunãtãtirii procesului de servire (prin adãugarea altor statii de servire sau prin reducerea timpului de servire).

Vom considera cazul unei singure statii; clientii care sosesc când statia e ocupatã intrã într-o coadã de asteptare si sunt serviti în ordinea sosirii si/sau în functie de anumite prioritãti ale clientilor. Imediat dupã sosirea unui client se genereazã momentul de sosire al unui nou client, iar când începe servirea unui client se genereazã momentul plecãrii 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 timpi se calculeazã 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 mai multe moduri:

- Intervalul de simulare este împãrtit în intervale egale (secunde, minute); scurgerea timpului este simulatã printr-un ciclu, iar valorile variabilei contor reprezintã timpul curent din proces. Durata simulãrii este în acest caz proportionalã cu mãrimea intervalului de timp simulat. In fiecare pas se comparã timpul curent cu timpul de producere a unor evenimente generate anterior (sosire client si plecare client).
- Se foloseste o coadã ordonatã de evenimente (evenimente de sosire si de plecare clienti), din care evenimentele se scot în ordinea timpului de producere. Durata simulãrii depinde de numãrul de evenimente produse într-un interval dat si mai putin de mãrimea intervalului.

Algoritmul de simulare care foloseste o coadã ordonatã de evenimente poate fi descris astfel:

```
pune in coada de evenimente  un eveniment “sosire” cu ts=0
se face server liber
repeta cat timp coada de evenim nu e goala {
    scoate din coada un eveniment
    daca timpul depaseste durata simularii se termina
    daca este un eveniment “sosire” {
        daca 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
        }
    daca eveniment “plecare” {
        daca coada de clienti e goala
            se face server liber
        altfel {
            scoate client din coada de asteptare
            pune in coada un eveniment “plecare” cu tp
        }
    }
}
```

In cazul particular al unei singure cozi (o singurã statie de servire) este suficient sã alegem între urmãtoarea sosire (cerere) si urmãtoarea plecare (la terminare servire) ca eveniment de tratat :

```
  // calcul timp ptr urmatorul eveniment,  aleator distribuit intre limitele min si max
int calc (int min, int max) {
  return min + rand()% (max-min+1);
}
int main() {
  int ts,tp,wait;        // ts=timp de sosire, tp=timp de plecare
  int tmax=5000;     // timp maxim de simulare
  int s1=25,s2=45;   // timp minim si maxim de sosire
  int p1=23,p2=47;   // timp minim si maxim de servire
  Queue q;                  // coada de cereri (de sosiri)
  ts = 0 + calc(s1,s2);   // prima sosire a unui client
```

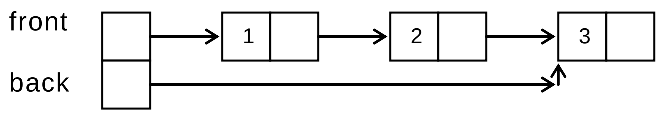
```
  tp = INT_MAX;         // prima plecare din coada
  initQ(q);
  while (ts <= tmax) {
    if (ts <= tp) {              // daca sosire
      if (empty(q))             // daca coada era goala
        tp= ts + calc(p1,p2);      // plecare= sosire+servire
      insQ(q,ts);                  // pune timp de sosire in coada
      ts=ts+ calc(s1,s2);         // urmatoarea sosire
    }
    else {                       // daca plecare
      wait = tp - delQ (q); // asteptare intre sosire si plecare
//      printf("wait = %d , queue size = %d\n", wait, size(q));
      if (empty(q))            // daca coada era goala
        tp = INT_MAX;          // nu exista o plecare planificata
      else                      // daca coada nu era goala
        tp = tp + calc(p1,p2);  // calculeaza urmatoarea plecare
    }
  }
  printf("coada in final=%d\n",size(q));  // coada poate contine si alte cereri neprelucrate
}
```

Calitatea procesului de servire este determinatã de lungimea cozii de clienti si deci de diferenta dintre momentul sosirii si momentul plecãrii unui client (compus din timpul de asteptare în coadã si timpul de servire efectivã).

Programul anterior poate fi modificat pentru alte distributii ale timpilor de sosire si de servire si pentru valori neîntregi ale acestor timpi.

Uneori se defineste o coadã cu posibilitãti de adãugare si de extragere de la ambele capete ale cozii, numitã "deque" ("double ended queue"), care are drept cazuri particulare stiva si coada, asa cum au fost definite aici. Operatiile caracteristice se numesc în biblioteca STL "pushfront", "pushback", "popfront", "popback".

O implementare adecvatã pentru structura "deque" este o listã înlãntuitã definitã printr-o pereche de pointeri: adresa primului si adresa ultimului element din listã:




```
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 pushfront (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 pushback (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* popfront (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;     // succesorul lui front devine primul nod
  free(t);
  return r;
}
  // scoate de la sfarsit de lista
void* popback (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 observã cã numai ultima operatie (pop_back) contine un ciclu si deci necesitã un timp ce depinde de lungimea listei O(n), dar ea poate fi evitatã. Utilizarea unei liste deque ca stivã foloseste operatiile pushfront, popfront iar utilizarea sa ca o coadã foloseste operatiile pushback, popfront.

6.5 TIPUL "COADÃ CU PRIORITÃTI"

O coadã cu prioritãti ("Priority Queue") este o colectie din care se extrage mereu elementul cu prioritate maximã (sau minimã). Prioritatea este datã de valoarea elementelor memorate sau de o cheie numericã asociatã elementelor memorate în coadã. Dacã existã mai multe elemente cu aceeasi prioritate, atunci ordinea de extragere este aceeasi cu ordinea de introducere .

Algoritmii de tip "greedy" folosesc în general o coadã cu prioritãti pentru listele de candidati; la fiecare pas se extrage candidatul optim din listã.

O coadã cu prioritãti este o structurã dinamicã, la care au loc alternativ introduceri si extrageri din coadã. Dacã nu se mai fac inserãri în coadã, atunci putem folosi un simplu vector, ordonat la început si apoi parcurs succesiv de la un cap la altul.

Operatiile specifice cozii cu prioritãti sunt:
- Adãugare element cu valoarea x la coada q : addPQ (q ,x)
- Extrage în x si sterge din coada q elementul cu cheia maximã (minimã): delPQ(q)
- Citire (fãrã extragere) valoare minimã sau maximã: minPQ(q)
- Initializare coadã: initPQ (q).
- Test coadã vidã: emptyPQ (q)

Sunt posibile diverse implementãri pentru o coadã cu prioritãti (vector ordonat, listã ordonatã, 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 partialã dupã fiecare extragere sau insertie.

O aplicatie simplã pentru o coadã cu prioritãti este un algoritm greedy pentru interclasarea mai multor vectori ordonati cu numãr minim de operatii (sau pentru reuniunea mai multor multimi cu numãr minim de operatii).

Interclasarea a doi vectori cu n1 si respectiv n2 elemente necesitã n1+n2 operatii. Fie vectorii 1,2,3,4,5,6 cu dimensiunile urmãtoare: 10,10,20,20,30,30.
Dacã ordinea de interclasare este ordinea crescãtoare a vectorilor, atunci numãrul de operatii la fiecare interclasare va fi:
10+10 =20, 20+20=40, 40+20=60, 60+30=90, 90+30=120.
Numãrul total de operatii va fi 20+40+60+90+120=330
Numãrul total de operatii depinde de ordinea de interclasare a vectorilor si are valoarea minimã 300.

Ordinea de interclasare poate fi reprezentatã printr-un arbore binar sau printr-o expresie cu paranteze. Modul de grupare care conduce la numãrul minim de operatii este (((1+2) +3) +6) + (4+5) deoarece la fiecare pas se executa operatiile:
10+10=20, 20+20=40, 40+30=70, 20+30=50, 70+50=120 (20+40+70+50+120=300)

Algoritmul de interclasare optimã poate fi descris astfel:

```
creare coadã ordonatã crescãtor cu lungimile vectorilor
repeta
    scoate valori minime din coada în n1 si n2
    n=n1+n2
    daca coada e goala
        scrie n  si stop
    altfel
        pune n în coada
```

Evolutia cozii cu prioritãti 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
```

Urmeazã aplicatia de interclasare vectori cu numãr minim de operatii, folosind o coadã 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 optimã a vectorilor si nici operatia de interclasare propriu-zisã, deoarece nu se memorează în coadã adresele vectorilor, dar se poate extinde cu memorarea numerelor (adreselor) vectorilor.

6.6 VECTORI HEAP (ARBORI PARTIAL ORDONATI)

Un "Heap" este un vector care reprezintã un arbore binar partial ordonat de înãltime minimã, completat de la stânga la dreapta pe fiecare nivel. Un max-heap are urmãtoarele proprietãti:
- Toate nivelurile sunt complete, cu posibila exceptie a ultimului nivel, completat de la stânga spre dreapta.
- Valoarea oricãrui nod este mai mare sau egalã cu valorile succesorilor sãi.

O definitie mai scurtã a unui (max)heap este: un arbore binar complet în care orice fiu este mai mic decât pãrintele sãu.

Rezultã de aici cã rãdãcina arborelui contine valoarea maximã dintre toate valorile din arbore (pentru un max-heap).

Vectorul contine valorile nodurilor, iar legãturile unui nod cu succesorii sãi sunt reprezentate implicit prin pozitiile lor în vector :
- Rãdãcina are indicele 1 (este primul element din vector).
- Pentru nodul din pozitia k nodurile vecine sunt:
    - Fiul stânga în pozitia 2*k
    - Fiul dreapta în pozitia 2*k + 1
    - Pãrintele în pozitia k/2

Exemplu de vector max-heap :

```
                        16
             ___________|___________
            |                       |
            14                      10
      ______|______           ______|______
     |             |         |             |
     8             7         9             3
  ___|___       ___|___
 |       |     |       |
 2       4     1
```

```
Indice    1    2    3    4   5   6   7   8   9   10
Valoare  16   14   10    8   7   9   3   2   4    1
```

De observat cã valorile din noduri depind de ordinea introducerii lor în heap, dar structura arborelui cu 10 valori este aceeasi (ca repartizare pe fiecare nivel). Altfel spus, cu aceleasi n valori se pot construi mai multi vectori max-heap (sau min-heap).

Intr-un min-heap prima pozitie (rãdãcinã) contine valoarea minimã, iar fiecare nod are o valoare mai micã decât valorile din cei doi fii ai sãi.

Vectorii heap au cel putin douã utilizãri importante:
- (Max-Heap) o metodã eficientã de sortare ("HeapSort");
- (Min-Heap) o implementare eficientã pentru tipul "Coadã cu prioritãti";

Operatiile de introducere si de eliminare dintr-un heap necesitã un timp de ordinul O(log n), dar citirea valorii minime (maxime) este O(1) si nu depinde de mãrimea sa.

Operatiile de bazã asupra unui heap sunt :
- Transformare heap dupã aparitia unui nod care nu este mai mare ca succesorii sãi, pentru mentinerea proprietãtii de heap (“heapify”,”percolate”);
- Crearea unui heap dintr-un vector oarecare;
- Extragere valoare maximã (minimã);
- Inserare valoare nouã în heap, în pozitia corespunzãtoare.
- Modificarea valorii dintr-o pozitie datã.

Primul exemplu este cu un max-heap de numere întregi, definit astfel:

```
typedef struct {
  int v[M]; int n;      // vector heap (cu maxim M numere) si dimensiune efectiva vector
} heap;
```

Operatia “heapify” reface un heap dintr-un arbore la care elementul k nu respectã conditia de heap, dar subarborii sãi respectã aceastã conditie; la aceastã situatie se ajunge dupã înlocuirea sau dupã modificarea valorii din rãdãcina unui arbore heap. Aplicatã asupra unui vector oarecare functia “heapify(k)” nu creeazã un heap, dar aduce în pozitia k cea mai mare dintre valorile subarborelui cu rãdãcina în k : se mutã succesiv în jos pe arbore valoarea v[k], dacã nu este mai mare decât fii sãi. Functia recursivã "heapify" din programul urmãtor face aceastã transformare propagând în jos pe arbore valoarea din nodul "i", astfel încât arborele cu rãdãcina în "i" sã fie un heap. In acest scop se determinã valoarea maximã dintre v[i], v[st] si v[dr] si se aduce în pozitia "i", pentru ca sã avem v[i] >= v[st] si v[i] >= v[dr], unde "st" si "dr" sunt adresele (indicii) succesorilor la stânga si la dreapta ai nodului din pozitia "i". Valoarea coborâtã din pozitia "i" în "st" sau "dr" va fi din nou comparatã cu succesorii sãi, la un nou apel al functiei "heapify".

```
void swap (heap h, int i, int j) {    // schimbã între ele valorile v[i] si v[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;                        // succesori 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
  }
}
```

Urmeazã o variantã iterativã pentru functia “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 rãdãcinã, 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 actiunea functiei "makeheap" pe exemplul urmãtor:

```
  operatie        vector                    arbore

 initializare     1  2  3  4  5  6               1
                                              2       3
                                            4   5   6

 heapify(3)       1  2  6  4  5  3               1
                                              2       6
                                            4   5   3

 heapify(2)       1  5  6  4  2  3               1
                                              5       6
                                            4   2   3

 heapify(1)       6  5  3  4  2  1               6
                                              5       3
                                            4   2   1
```

Programul de mai jos aratã cum se poate ordona un vector prin crearea unui heap si interschimb între valoarea maximã si 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];
}
```

In functia de sortare se repetã urmãtoarele operatii:

- schimbã valoarea maximã a[1] cu ultima valoare din vector a[n],
- se reduce dimensiunea vectorului
- se "ajusteazã" vectorul rãmas

Vom arãta actiunea procedurii "heapSort" pe urmãtorul exemplu:

dupã citire vector	4,2,6,1,5,3
dupã makeheap	6,5,4,1,2,3
dupã schimbare 6 cu 3	3,5,4,1,2,6
dupã heapify(1,5)	5,3,4,1,2,6
dupã schimbare 5 cu 2	2,3,4,1,5,6
dupã heapify(1,4)	4,3,2,1,5,6
dupã schimbare 4 cu 1	1,3,2,4,5,6
dupã heapify(1,3)	3,1,2,4,5,6
dupã schimbare 3 cu 2	2,1,3,4,5,6
dupã heapify(1,2)	2,1,3,4,5,6
dupã schimbare 2 cu 1	1,2,3,4,5,6

Extragerea valorii maxime dintr-un heap se face eliminând rãdãcina (primul element din vector), aducând în prima pozitie valoarea din ultima pozitie si aplicând functia "heapify" pentru mentinerea 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;
}
```

Adãugarea unei noi valori la un heap se poate face în prima pozitie liberã (de la sfârsitul vectorului), urmatã de deplasarea ei în sus cât este nevoie, pentru mentinerea proprietãtii 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 functiei insH este arãtat pe exemplul de adãugare 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 coadã cu prioritãti se memoreazã obiecte ce contin o cheie, care determinã prioritatea obiectului, plus alte date asociate acestei chei. De exemplu, în heap se memoreazã arce dintr-un graf cu costuri, iar ordonarea lor se face dupã 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 prioritãti apare uneori situatia când elementele din coadã au acelasi numãr, aceleasi date memorate dar prioritatea lor se modificã în timp. Un exemplu este algoritmul Dijkstra pentru determinarea drumurilor minime de la un nod sursã la toate celelalte noduri dintr-un graf; în coadã se pun distantele calculate de la nodul sursã la celelalte noduri, dar o parte din aceste distante se modificã la fiecare pas din algoritm (se modificã doar costul dar nu si numãrul nodului). Pentru astfel de cazuri este utilã operatia de modificare a prioritãtii, cu efect asupra pozitiei elementului respectiv în coadã (fãrã adãugãri sau eliminãri de elemente din coadã).

La implementarea cozii printr-un vector heap operatia de modificare a prioritãtii 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 simplã dacã se cunoaste pozitia elementului în heap pentru cã seamãnã cu adãugarea unui nou element la heap (se comparã repetat noua prioritate cu prioritatea nodului pãrinte si se mutã elementul dacã e necesar, pentru a mentine un heap).

In literaturã sunt descrise diferite variante de vectori heap care permit reunirea eficientã a doi vectori heap într-un singur heap (heap binomial, skew heap, s.a.).

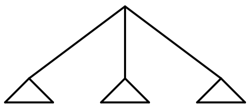
Capitolul 7

ARBORI

7.1 STRUCTURI ARBORESCENTE

Un arbore cu rãdãcinã ("rooted tree") este o structurã neliniarã, în care fiecare nod poate avea mai multi succesori, dar un singur predecesor, cu exceptia unui nod special, numit rãdãcinã si 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 (rãdãcina)
- un nod care are ca succesori un numãr finit de (sub)arbori.



Altfel spus, dacã se eliminã rãdãcina unui arbore rezultã mai multi arbori, care erau subarbori în arborele initial (dintre care unii pot fi arbori fãrã nici un nod).

Definitia recursivã este importantã pentru cã multe operatii cu arbori pot fi descompuse recursiv în câteva operatii componente:
- prelucrare nod rãdãcinã
- prelucrare subarbore 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 aceeasi parte, este de fapt o listã liniarã.

Structura de arbore este o structurã ierarhicã, cu noduri asezate pe diferite niveluri, cu relatii de tip pãrinte - fiu între noduri. Nodurile sunt de douã feluri:
- Nodurile terminale, fãrã succesori, se numesc si "frunze";
- Noduri interne (interioare), cu unul sau doi succesori.

Fiecare nod are douã proprietãti:
- Adâncimea (“depth”) este egalã cu numãrul de noduri de pe calea (unicã) de la rãdãcinã la acel nod;
- Inãltimea (“height”) este egalã cu numãrul de noduri de pe cea mai lungã cale de la nod la un descendent (calea de la nod la cel mai îndepãrtat descendent).

Inãltimea unui arbore este înãltimea rãdãcinii sale, deci de calea cea mai lungã de la rãdãcinã la o frunzã. Un arbore vid are înaltimea zero iar un arbore cu un singur nod (rãdãcinã) are înãltimea unu.

Un arbore este perfect echilibrat dacã înãltimile fiilor oricãrui nod diferã între ele cel mult cu 1. Un arbore este echilibrat dacã înãltimea sa este proportionalã cu log(N), ceea ce face ca durata operatiilor de cãutare, insertie, eliminare sã fie de ordinul O(log(N)), unde N este numãrul de noduri din arbore.

In fiecare nod dintr-un arbore se memoreazã valoarea nodului (sau un pointer cãtre informatii asociate nodului), pointeri cãtre fii sãi si eventual alte date: pointer la nodul pãrinte, adâncimea sa înãltimea nodului s.a. De observat cã adresa nodului pãrinte, înãltimea sau adâncimea nodului pot fi determinate prin apelarea unor functii (de obicei recursive), dacã nu sunt memorate explicit în fiecare nod.

Dupã numãrul maxim de fii ai unui nod arborii se împart în:
- Arbori multicãi (generali), în care un nod poate avea orice numãr de succesori;
- Arbori binari, în care un nod poate avea cel mult doi succesori.

In general construirea unui arbore începe cu rãdãcina, la care se adaugã noduri fii, la care se adaugã alti fii în mod recursiv, cu cresterea adâncimii (înãltimii) arborelui. Existã însã si câteva exceptii (arbori Huffman, arbori pentru expresii aritmetice), care se construiesc de la frunze cãtre rãdãcinã.

Cuvântul “arbore” se foloseste si pentru un caz particular de grafuri fãrã cicluri, la care orice vârf poate fi privit ca rãdãcinã. Diferenta dintre arborii cu rãdãcinã (din acest capitol) si arborii liberi (grafuri aciclice) este cã primii contin în noduri date importante pentru aplicatii, iar arborii grafuri nu contin date în noduri (dar arcele ce unesc aceste noduri pot avea asociate valori sau costuri).

Structurile arborescente se folosesc în programare deoarece:

- Reprezintã un model natural pentru o ierarhie de obiecte (entitãti, operatii etc).
- Sunt structuri de cãutare cu performante foarte bune, permitând si mentinerea în ordine a unei colectii de date dinamice (cu multe adãugãri si stergeri).

De cele mai multe ori legãturile unui nod cu succesorii sãi se reprezintã prin pointeri, dar sunt posibile si reprezentãri fãrã pointeri ale arborilor, prin vectori.

De obicei se întelege prin arbore o structurã cu pointeri, deoarece aceasta este mai eficientã pentru arbori multicãi si pentru arbori binari cu structurã imprevizibilã.

O reprezentare liniarã posibilã a unui arbore este o expresie cu paranteze complete, în care fiecare nod este urmat de o parantezã ce grupeazã succesorii sãi. Exemple:

1) a (b,c)

este un arbore binar cu 3 noduri: rãdãcina 'a', având la stânga pe 'b' si la dreapta pe 'c'

2) 5 (3 (1,), 7(,9))

este un arbore binar ordonat cu rãdãcina 5. Nodul 3 are un singur succesor, la stânga, iar nodul 7 are numai succesor la dreapta:

```
               5
        _______|_______
       3               7
    ___|___         ___|___
   1                       9
```

Afisarea arborilor binari sau multicãi se face de obicei prefixat si cu indentare diferitã la fiecare nivel (fiecare valoare pe o linie, iar valorile de pe acelasi nivel în aceeasi coloanã). Exemplu de afisare prefixatã, cu indentare, a arborelui de mai sus:

```
5
   3
      1
      -
   7
      -
      9
```

Uneori relatiile dintre nodurile unui arbore sunt impuse de semnificatia datelor memorate în noduri (ca în cazul arborilor ce reprezintã expresii aritmetice sau sisteme de fisiere), dar alteori distributia valorilor memorate în noduri nu este impusã, fiind determinatã de valorile memorate (ca în cazul arborilor de cãutare, unde structura depinde de ordinea de adãugare si poate fi modificatã prin reorganizarea arborelui).

7.2 ARBORI BINARI NEORDONATI

Un caz particular important de arbori îl constituie arborii binari, în care un nod poate avea cel mult doi succesori: un succesor la stânga si un succesor la dreapta.

Arborii binari pot avea mai multe reprezentãri:

a) 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

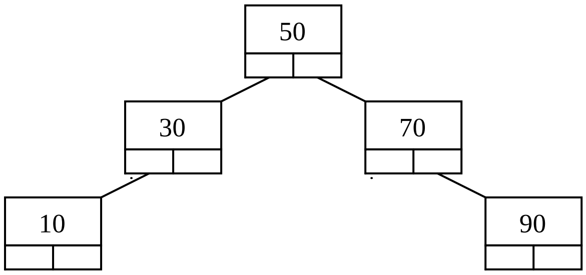
b) Reprezentare prin 3 vectori: valoare, valoare fiu stânga, valoare fiu dreapta (mai compact, fãrã frunze, dar necesitã cãutarea fiecãrui fiu). Exemplu:

```
 val   50  70  30
  st    30   -1  10
  dr   70  90    -1
```

c) Reprezentare printr-un singur vector, nivel cu nivel din arbore . Exemplu:

```
val   50  30  70  10  -1  -1 90
```

d) Noduri (structuri) cu pointeri pentru legãturi pãrinte-fii. Exemplu:

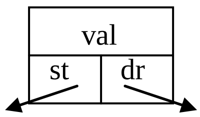


Un arbore relativ echilibrat poate fi reprezentat eficient printr-un singur vector, dupã ideea unui vector heap, chiar dacã nu este complet fiecare nivel din arbore (valorile lipsã fiind marcate printr-o valoare specialã); în acest caz relatiile dintre noduri pãrinte-fiu nu mai trebuie memorate explicit (prin indici sau valori noduri), ele rezultã implicit din pozitia fiecãrui element în vector (se pot calcula).

Aceastã reprezentarea devine ineficientã pentru arbori cu înãltime mare dar cu numãr de noduri relativ mic, deoarece numãrul de noduri într-un arbore complet creste exponential cu înãltimea sa. De aceea s-au propus solutii bazate pe vectori de biti: un vector de biti contine 1 pentru un nod prezent si 0 pentru un nod absent într-o liniarizare nivel cu nivel a arborelui, iar valorile din noduri sunt memorate separat dar în aceeasi ordine de parcurgere a nodurilor (în lãrgime). Pentru arborele folosit ca exemplu vectorul de biti va fi 1111001, iar vectorul de valori va fi 50,30,70,10,90.

Definitia unui nod dintr-un arbore binar, cu pointeri cãtre 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 memoreazã în fiecare nod si adresa nodului pãrinte, pentru a ajunge repede la pãrintele unui nod (pentru parcurgere de la frunze cãtre rãdãcinã sau pentru modificarea structurii unui arbore). Nodurile terminale pot contine valoarea NULL sau adresa unui nod sentinelã. Adresa cãtre nodul pãrinte si utilizarea unui nod unic sentinelã sunt utile pentru arborii echilibrati, care îsi modificã structura.

Un arbore este definit printr-o singurã variabilã pointer, care contine adresa nodului rãdãcinã; pornind de la rãdãcinã se poate ajunge la orice nod.

Operatiile cu arbori, considerati drept colectii de date, sunt:

- Initializare arbore (creare arbore vid);

- Adãugarea unui nod la un arbore (ca frunzã);
- Cãutarea unei valori date într-un arbore;
- Eliminarea (stergerea) unui nod cu valoare datã;
- Enumerarea tuturor nodurilor din arbore într-o anumitã ordine.

Alte operatii cu arbori, utile în anumite aplicatii :
- Determinarea valorii minime sau maxime dintr-un arbore
- Determinarea valorii imediat urmãtoare valorii dintr-un nod dat
- Determinarea rãdãcinii arborelui ce contine un nod dat
- Rotatii la stânga sau la dreapta noduri

Enumerarea (afisarea) nodurilor unui arbore cu N noduri necesitã O(N) operatii. Durata operatiilor de adãugare si de eliminare noduri depinde de înãltimea arborelui.

Initializarea unui arbore vid se poate reduce la atribuirea valorii NULL pentru variabila rãdãcinã, sau la crearea unui nod sentinelã, fãrã date. Poate fi luatã în considerare si o initializare a rãdãcinii cu prima valoare introdusã în arbore, astfel ca adãugãrile ulterioare sã nu mai modifice rãdãcina (dacã nu se face modificarea arborelui pentru reechilibrare, dupã adãugare sau stergere).

Functiile pentru operatii cu arbori binari sunt natural recursive, pentru cã orice operatie (afisare, cãutare etc) se reduce la operatii similare cu subarborii stânga si dreapta, plus operatia asupra rãdãcinii. Reducerea (sub)arborilor continuã pânã se ajunge la un (sub)arbore vid.

Adãugarea de noduri la un arbore binar oarecare poate folosi functii de felul urmãtor:

```
void addLeft (tnod* p, tnod* left);        // adauga lui p un fiu stanga
void addRight (tnod* p, tnod* right);    // adauga lui p un fiu dreapta
```

In exemplul urmãtor se considerã cã datele folosite la construirea arborelui se dau sub forma unor tripleti de valori: valoare nod pãrinte, valoare fiu stânga, valoare fiu dreapta. O valoare zero marcheazã absenta 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 TRAVERSAREA ABORILOR BINARI

Traversarea unui arbore înseamnã vizitarea tuturor nodurilor din arbore si poate fi privitã ca o liniarizare a arborelui, prin stabilirea unei secvente liniare de noduri. In functie de ordinea în care se iau în considerare rãdãcina, subarborele stânga si subarborele dreapta putem vizita în:
- Ordine prefixatã (preordine sau RSD) : rãdãcinã, stânga, dreapta
- Ordine infixatã (inordine sau SRD) : stânga, rãdãcinã, dreapta
- Ordine postfixatã (postordine sau SDR): stânga, dreapta, rãdãcinã

Fie arborele binar descris prin expresia cu paranteze:
5 (2 (1,4(3,)), 8 (6 (,7),9))
Traversarea prefixatã produce secventa de valori: 5 2 1 4 3 8 6 7 9
Traversarea infixatã produce secventa de valori: 1 2 3 4 5 6 7 8 9
Traversarea postfixatã produce secventa de valori: 1 3 4 2 7 6 9 8 5

Traversarea arborilor se codificã mai simplu prin functii recursive, dar uneori este preferabilã sau chiar necesarã o traversare nerecursivã (în cazul unui iterator pe arbore, de exemplu).

Exemplu de functie recursivã pentru afisare infixatã 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
}
```

Functia "infix" poate fi usor modificatã pentru o altã 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 minimalã de afisare se pot scrie si alte variante de afisare: ca o expresie cu paranteze sau cu evidentierea structurii de arbore:

```
  // afisare structura arbore (prefixat 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 spatii
  }
}
```

Majoritatea operatiilor cu arbori pot fi considerate drept cazuri de vizitare (parcurgere, traversare) a tuturor nodurilor din arbore; diferenta constã în operatia aplicatã nodului vizitat: afisare, comparare, adunare nod sau valoare la o sumã, verificarea unor conditii la fiecare nod, s.a.

Cãutarea unei valori date x într-un arbore binar se face prin compararea lui x cu valoarea din fiecare nod si se reduce la cãutarea succesivã 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;                   // rezultat adresa nod gasit
  else                          // daca nu s-a gasit in stanga
    return find (r→dr,x);       // rezultat cautare in subarbore dreapta
}
```

Explorarea unui arbore în lãrgime (pe niveluri succesive) necesitã memorarea succesorilor (fiilor) unui nod într-o coadã. Dupã vizitarea nodului de plecare (rãdãcina arborelui) se pun în coadã toti succesorii lui, care vor fi apoi extrasi în ordinea în care au fost pusi. Dupã ce se extrage un nod se adaugã la sfârsitul cozii succesorii lui. In felul acesta, fii unui nod sunt prelucrati dupã fratii nodului respectiv. Exemplu de evolutie a cozii de pointeri la noduri pentru arborele binar urmãtor:

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 coadã
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 prezentatã am considerat r !=NULL si nu s-au mai pus în coadã si pointerii egali cu NULL, dar este posibilã si varianta urmãtoare:

```
void bfs_bin ( tnod * r) {          // breadth first search
    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");
}
```

Traversarea nerecursivã a unui arbore binar în adâncime, prefixat, se poate face asemãnãtor, dar folosind o stivã în loc de coadã pentru memorarea adreselor nodurilor prin care s-a trecut dar fãrã prelucrarea lor, pentru o revenire ulterioarã.

```
void prefix (tnod * r) {            // traversare prefixata
 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 stivã a fiilor unui nod (fiu dreapta si apoi fiu stânga), pentru ca la scoatere din stivã si afisare sã se scrie în ordinea stânga-dreapta.

Evolutia stivei la afisarea infixatã 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

Dupã modelul afisãrii prefixate cu stivã se pot scrie nerecursiv si alte operatii; exemplu de cãutare iterativã a unei valori x în arborele cu rãdãcina r:

```
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
}
```

Traversarea nerecursivã infixatã si postfixatã nu se pot face doar prin modificarea traversãrii prefixate, la fel de simplu ca în cazul formelor recursive ale functiilor de traversare. Cea mai dificilã este traversarea postfixatã. Pentru afisarea infixatã nerecursivã existã o variantã relativ simplã:

```
void infix (tnod * r) {
 Stiva s;
 initS(s);
 push (s,NULL);                    // pune NULL (sau alta adresa) pe stiva
 while ( ! emptyS (s)) {           // cat timp stiva mai contine ceva
  if( r != NULL) {                 // mergi la stanga cat se poate
```

```
    push (s,r);                 // pune pe stiva adrese noduri vizitate dar neafisate
    r=r->st;                    // mereu la stanga
  }
  else {                        // daca nu se mai poate la stanga atunci retragere
    r=(tnod*)pop(s);            // scoate ultimul nod pus in stiva
    if (r==NULL) return;        // iesire daca era adresa pusa initial
    printf ("%d ",r->val);      // afisare valoare nod curent
    r=r->dr;                    // si continua la dreapta sa
  }
 }
}
```

Traversarea nerecursivã în adâncime se poate face si fãrã stivã dacã fiecare nod memoreazã si adresa nodului pãrinte, pentru cã stiva folosea la revenirea de la un nod la nodurile de deasupra sa. In aceastã variantã trebuie evitatã afisarea (vizitarea) repetatã a unui aceluiasi nod; evidenta nodurilor deja afisate se poate face fie printr-o multime cu adresele nodurilor vizitate (un vector , de exemplu) sau printr-un câmp suplimentar în fiecare nod care îsi schimbã valoarea dupã vizitare.

Exemplul urmãtor foloseste un tip "set" neprecizat si operatii tipice cu multimi:

```
void prefix (tnod* r) {
 set a;                        // multime noduri vizitate
 init(a);                      // initializare multime vida
 tnod* p = r;                  // nod initial
  while (p != NULL)
     if ( ! contains(a,p)) {   // daca p nevizitat
       printf("%d ",p->val);   // se scrie valoarea din p
       add(a,p);               // si se adauga p la multimea de noduri vizitate
     }
     else                      // daca p a fost vizitat
     if (p->st != 0 && ! contains(a,p->st) )      // daca exista fiu stanga nevizitat
         p = p->st;                               // el devine nod curent
     else if (p->dr != 0 && ! contains(a,p->dr) )  // daca exista fiu dreapta nevizitat
         p = p->dr;                               // fiul dreapta devine nod curent
     else                                         // daca p nu are succesori nevizitati
       p = p->sus;                                // se revine la parintele nodului curent
}
```

Aceastã solutie are avantajul cã poate fi modificatã relativ simplu pentru altã ordine de vizitare a nodurilor. Exemplu de afisare postfixatã nerecursivã si fãrã stivã:

```
void postfix (tnod* r) {
  set a;                       // multime noduri vizitate
  init(a);
  tnod* p = r;
  while (p != 0)
     if (p->st != 0 && ! contains(a,p->st))           // stanga
         p = p->st;
     else if (p->dr != 0 && !contains(a,p->dr))       // dreapta
         p = p->dr;
     else
     if ( ! contains(a,p)) {                          // radacina
       printf("%d ",p->val);
       add(a,p);
     }
     else
       p = p->sus;
}
```

Un iterator pe arbore contine minim douã functii: o functie de pozitionare pe primul nod (“first”) si o functie (“next”) care are ca rezultat adresa nodului urmãtor în ordine pre, post sau infixatã. Functia “next” are rezultat NULL dacã nu mai existã un nod urmãtor pentru cã au fost toate vizitate.

Iteratorul nu poate folosi o vizitare recursivã, iar traversãrile nerecursive prezentate folosesc o altã structurã de date (o stivã sau o multime) care ar fi folosite în comun de functiile “first” si “next”. De aceea vom da o solutie de iterator prefixat care foloseste un indicator de stare (vizitat/nevizitat) memorat în fiecare nod:

```
tnod * first (tnod* r) { return r;}       // pozitionare pe primul nod vizitat
tnod* next (tnod* p) {                    // urmatorul nod din arbore
     static int n=size(p);                // pentru a sti cand s-au vizitat toate nodurile
     if (n==0) return NULL;               // daca s-au vizitat toate nodurile
      if (! p->v){                        // daca s-a gasit un nod p nevizitat
          p->v=1; n--;                    // marcare p ca vizitat
          return p;                       // p este urmatorul nod vizitat
      }                                   // daca p vizitat
     if (p->st != 0 && !p->st->v )        // incearca cu fiul stanga
         p = p->st;
     else if (p->dr != 0 && ! p->dr->v )   // apoi cu fiul dreapta
         p = p->dr;
     else if ( p->sus)                    // daca are parinte
         p= p->sus;                       // incearca cu nodul parinte
     return next(p);                      // si cauta alt nod nevizitat
}
  // utilizare iterator
…
    p=first(r);      // prima valoare (radacina)
    while (p=next(p))
      printf("%d ", p->val);
```

7.4 ABORI 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 optimizãri la evaluarea expresiilor cu subexpresii comune. Un sir postfixat este de fapt o altã reprezentare, liniarã, a unui arbore binar.

Reprezentarea expresiilor prin arbori rezolvã problema ordinii efectuãrii operatiilor prin pozitia operatorilor în arbore, fãrã a folosi paranteze sau prioritãti relative între operatori: operatorii sunt aplicati începând de la frunze cãtre rãdãcinã, deci în ordine postfixatã.

Constructia arborelui este mai simplã dacã se porneste de la forma postfixatã sau prefixatã a expresiei deoarece nu existã problema prioritãtii operatorilor si a parantezelor; construirea progreseazã de la frunze spre rãdãcinã. Un algoritm recursiv este mai potrivit dacã se pleacã de la sirul prefixat, iar un algoritm cu stivã este mai potrivit dacã se pleacã de la sirul postfixat.

Pentru simplificarea codului vom considera aici numai expresii cu operanzi dintr-o singurã cifrã, cu operatorii aritmetici binari '+', '-', '*', '/' si fãrã spatii albe între operanzi si operatori. Eliminarea acestor restrictii nu modificã esenta problemei si nici solutia discutatã, dar complicã implementarea ei.

Pentru expresia 1+3*2 - 8/4 arborele echivalent aratã astfel:

```
                   _
       ____________|___________
      |                        |
      +                        /
   ___|_____             ______|_____
  |         |           |            |
  1         *           8            4
       _____|_____
      |           |
      3           2
```

Operanzii se aflã numai în noduri terminale iar operatorii numai în noduri interne. Evaluarea expresiei memorate într-un arbore binar este un caz particular de vizitare postfixatã a nodurilor arborelui si se poate face fie recursiv, fie folosind o stivã de pointeri la noduri. Nodurile sunt interpretate diferit (operanzi sau operatori), fie dupã continutul lor, fie dupã pozitia lor în arbore (terminale sau neterminale).

Evaluarea recursivã a unui arbore expresie se poate face cu functia urmãtoare.

```
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 postfixatã sau prefixatã seamãnã cu algoritmul de evaluare a unei expresii postfixate (prefixate). Functia urmãtoare foloseste o stivã de pointeri la noduri si creeazã (sub)arbori care se combinã 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 frunzã
     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 postfixatã 132*+84/- evolutia stivei dupã 5 pasi va fi urmãtoarea:

```
                        |__|                                       +
|__|      |__|          |__|→2    |__|        *                   / \
| _ |     |__|→3        |__|→3    |__|       / \         __      1   *
|__|→1    |__|→1        |__|→1    |__|→1    3   2        |__|       / \
                                                                   3   2
```

Functia urmãtoare creeazã un arbore binar pornind de la o expresie prefixatã:

```
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 infixatã, cu paranteze (forma uzualã) se poate face modificând functiile mutual recursive care permit evaluarea acestei expresii.

7.5 ARBORI HUFFMAN

Arborii Huffman sunt arbori binari folositi într-o metodã de compresie a datelor care atribuie fiecãrui caracter (octet) un cod binar a cãrui lungime depinde de frecventa octetului codificat; cu cât un caracter apare mai des într-un fisier cu atât se folosesc mai putini biti pentru codificarea lui. De exemplu, într-un fisier apar 6 caractere cu urmãtoarele frecvente:

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

Numãrul de biti necesari pentru un fisier de 1000 caractere va fi 3000 în cazul codificãrii 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 fiecãrui caracter functie de probabilitatea lui de aparitie astfel încât numãrul total de biti folositi în codificarea unui sir de caractere sã fie minim. Pentru generarea codurilor de lungime variabilã 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 :

```
        5(100)
     0  /          \ 1
   a(45)        4(55)
             0 /          \ 1
            2(25)        3(30)
          0 /     \ 1   0 /    \ 1
       c(12)  b(13) 1(14)  d(16)
                      0 /    \ 1
                      f(5)   e(9)
```

Se observã introducerea unor noduri intermediare notate cu cifre.

Pentru codificare se parcurge arborele începând de la rãdãcinã si se adaugã câte un bit 0 pentru un succesor la stânga si câte un bit 1 pentru un succesor la dreapta.

Construirea unui arbore Huffman seamãnã cu construirea arborelui echivalent unei expresii aritmetice: se construiesc treptat subarbori cu numãr tot mai mare de noduri pânã când rezultã un singur arbore. Diferenta este cã în cazul expresiilor se foloseste o stivã pentru memorarea rãdãcinilor subarborilor, iar în algorimul Huffman se foloseste o coadã cu prioritãti de subarbori binari care se combinã treptat.

Algoritmul genereazã arborele de codificare începând de jos în sus, folosind o coadã cu prioritãti, ordonatã crescãtor dupã frecventa de aparitie a caracterelor. La fiecare pas se extrag primele douã elemente din coadã (cu frecvente minime), se creeazã cu ele un subarbore si se introduce în coadã un element a cãrui frecventã este egalã cu suma frecventelor elementelor extrase.

Coada poate memora adrese de noduri de arbore sau valori din nodurile rãdãcinã (dar atunci mai este necesarã o cãutare în arbore pentru aflarea adresei nodului).

Evolutia 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 adãugate la coadã au fost numerotate în ordinea producerii lor.

La început se introduc în coadã toate caracterele, sau pointeri la noduri de arbore construite cu aceste caractere si frecventa lor. Apoi se repetã n-1 pasi (sau pânã când coada va contine un singur element) de forma urmãtoare:

- extrage si sterge din coadã primele douã elemente (cu frecventa minimã)
- construieste un nou nod cu suma frecventelor si având ca subarbori adresele scoase din coadã
- introduce în coadã adresa noului nod (rãdãcinã a unui subarbore)

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;
```

Functia urmãtoare construieste 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 frecventelor 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 înseamnã aflarea cãii de la rãdãcinã la nodul ce contine caracterul c, prin cãutare în arbore. Pentru simplificarea programãrii si verificãrii vom genera siruri de caractere '0' si '1' si nu configuratii binare (siruri de biti 0).

Functia urmãtoare produce codul Huffman al unui caracter dat ca sir de cifre binare (terminat cu zero), dar în ordine inversã (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 subarbore stanga
          return strcat(hc,"0");       // si adauga cifra 0 la codul hc
      if (encode (r→dr, ch, hc))       // cauta in subarborele dreapta
          return strcat(hc,"1");       // si adauga cifra 1 la codul hc
      else                             // daca ch negasit in arbore
        return NULL;
  }
```

Un program pentru decompresie Huffman trebuie sã primeascã atât fisierul codificat cât si arborele folosit la compresie (sau datele necesare pentru reconstruirea sa). Arborele Huffman (si orice arbore binar) poate fi serializat într-o formã fãrã pointeri, prin 3 vectori care sã continã valoarea (caracterul) din fiecare nod, valoarea fiului stânga si 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 rãdãcinã spre stânga pentru o cifrã zero si la dreapta pentru o cifrã 1; parcurgerea se reia de la rãdãcinã pentru fiecare secventã de biti arborele Huffman. Functia urmãtoare foloseste 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 necesarã trimiterea arborelui Huffman împreunã cu fisierul codificat, putem construi de la început un arbore fãrã pointeri, cu vectori de indici cãtre succesorii fiecãrui nod. Putem folosi un singur vector de structuri (o structurã corespunde unui nod) sau mai multi vectori reuniti într-o structurã. Exemplu de definire a tipurilor de date folosite într-un arbore Huffman fãrã pointeri, cu trei vectori: de date, de indici la fii stânga si 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 functia 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 asociazã fiecãrui caracter ASCII (cheia) un cod Huffman (valoarea asociatã cheii); la codificare se cautã dupã cheie iar la decodificare se cautã dupã valoare (este un dictionar bidirectional). Implementarea ca arbore permite cãutarea rapidã a codurilor de lungime diferitã, la decodificare.

Metoda de codificare descrisã este un algoritm Huffman static, care necesitã douã treceri prin fisierul initial: una pentru determinarea frecventei de aparitie a fiecarui octet si una pentru construirea arborelui de codificare (arbore static, nemodificabil).

Algoritmul Huffman dinamic (adaptiv) face o singurã trecere prin fisier, dar arborele de codificare este modificat dupã fiecare nou caracter citit. Pentru decodificare nu este necesarã transmiterea arborelui Huffman deoarece acesta este recreat la decodificare (ca si în algoritmul LZW). Acelasi caracter poate fi înlocuit cu diferite coduri binare Huffman, functie de momentul când a fost citit din fisier si de structura arborelui din acel moment. Arborele Huffman rezultat dupã citirea întregului fisier nu este identic cu arborele Huffman static, dar eficienta lor este comparabilã ca numãr de biti pe caracter.

Arborii Huffman au proprietatea de “frate” (“sibling property”): orice nod, în afarã de rãdãcinã, are un frate si este posibilã ordonarea crescãtoare a nodurilor astfel ca fiecare nod sã fie lângã fratele sãu, la vizitarea nivel cu nivel. Aceastã proprietate este mentinutã prin schimbãri de noduri între ele, la incrementarea ponderii unui caracter (care modificã pozitia nodului cu acel caracter în arbore).

7.6 ARBORI GENERALI (MULTICÃI)

Un arbore general (“Multiway Tree”) este un arbore în care fiecare nod poate avea orice numãr de succesori, uneori limitat (arbori B si arbori 2-3) dar de obicei nelimitat.

Arborii multicãi pot fi clasificati în douã grupe:
- Arbori de cãutare, echilibrati folositi pentru multimi si dictionare (arbori B);
- Arbori care exprimã relatiile dintre elementele unei colectii si a cãror structurã nu mai poate fi modificatã pentru reechilibrare (nu se pot schimba relatiile pãrinte-fiu).

Multe structuri arborescente “naturale” (care modeleazã situatii reale) nu sunt arbori binari, iar numãrul succesorilor unui nod nu este limitat. Exemplele cele mai cunoscute sunt: arborele de fisiere care reprezintã continutul unui volum disc si arborele ce reprezintã continutul unui fisier XML.

In arborele XML (numit si arbore DOM) nodurile interne corespund marcajelor de început (“start tag”), iar nodurile frunzã contin textele dintre marcaje pereche.

In arborele creat de un parser XML (DOM) pe baza unui document XML fiecare nod corespunde unui element XML. Exemplu de fisier XML:

```
<priceList>
  <computer>
    <name> CDC </name>
    <price> 540 </price>
  </ computer >
  <computer>
     <name> SDS </name>
     <price> 495 </price>
  </ computer >
</priceList>
```

Arborele DOM (Document Object Model) corespunzãtor acestui document XML:

```
                    priceList
                   /         \
          computer             computer
          /      \             /      \
      name       price      name      price
       |           |          |         |
      CDC         540        SDS       495
```

Sistemul de fisiere de pe un volum are o rãdãcinã cu nume constant, iar fiecare nod corespunde unui fisier; nodurile interne sunt subdirectoare, iar nodurile frunzã sunt fisiere “normale” (cu date). Exemplu din sistemul MS-Windows:

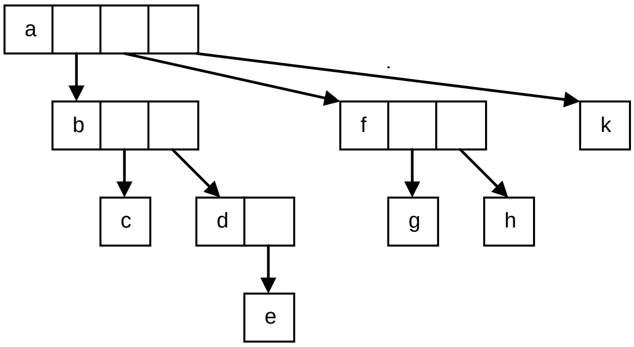
```
\
   Program Files
      Adobe
         Acrobat 7.0
            Reader
               . . .
      Internet Explorer
            . . .
            iexplorer.exe
      WinZip
            winzip.txt
            wz.com
            wz.pif
            . . .
```

Un arbore multicãi cu rãdãcinã se poate implementa în cel putin douã moduri:

a) - Fiecare nod contine 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)

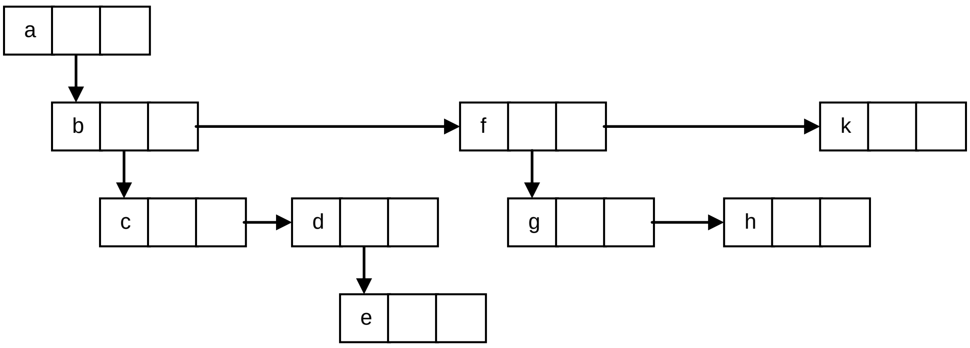
se va reprezenta prin vectori de pointeri la fii ca în figura următoare:



In realitate numărul de pointeri pe nod va fi mai mare decât cel strict necesar (din motive de eficientă vectorul de pointeri nu se extinde prin mărirea capacitătii cu 1 ci prin dublarea capacitătii sau prin adunarea unui increment constant):

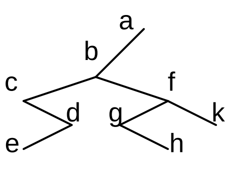
```
  // definitia unui nod de arbore cu vector extensibil de fii
typedef struct tnod {
  int val;                    // valoare (date) din nod
  int nc, ncm;                // nc=numar de fii ai acestui nod, ncm=numar maxim de fii
  struct tnod ** kids;        // vector cu adrese noduri fii
} tnod;
```

b) - Fiecare nod contine 2 pointeri: la primul fiu si la fratele următor (“left son, right sibling”). In acest fel un arbore multicăi este redus la un arbore binar. Putem considera si că un nod contine un pointer la lista de fii si un pointer la lista de frati. De exemplu, arborele a (b (c,d (e)), f (g, h), k) se va reprezenta prin legături la fiul stânga si 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 anterior fiu-frate ca arbore binar:



Succesorul din stânga al unui nod reprezintã primul fiu, iar succesorul din dreapta este primul frate. In reprezentarea fiu-frate un nod de arbore poate fi definit astfel:

```
typedef struct tnod {
  int val;
  struct tnod *fiu, *frate;
} tnod;
```

Exemple de functii pentru operatii cu arbori ce contin adrese cãtre fiu si frate :

```
  void addChild (tnod* crt, tnod* child) {      // adaugare fiu la un nod crt
    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→frate=child;                 // adauga child la sfarsitul listei
    }
  }
        // afisare arbore 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
      }
    }
  }
```

Pentru afisare, cãutare si alte operatii putem folosi functiile de la arbori binari, fatã de care adresa primului fiu corespunde subarborelui stânga iar adresa fratelui corespunde subarborelui dreapta.

Exemple de functii pentru arbori generali vãzuti 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+2);       // fiu pe nivelul urmator
          print (r→frate,ns);      // frate pe acelasi nivel
      }
 }
     // 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 multicai (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 cã vectorul de pointeri la fii se extinde cu 1 la fiecare adãugare a unui nou fiu, desi în acest fel se poate ajunge la o fragmentare excesivã 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 prefixatã (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 subarborele i
     if ( p != NULL)          // daca x gasit in subarborele i
       return p;
   }
   return NULL;             // x negasit in toti subarborii lui r
}
```

Pentru ambele reprezentãri de arbori multicãi adãugarea unui pointer cãtre pãrinte în fiecare nod permite afisarea rapidã a cãii de la rãdãcinã la un nod dat si simplificarea altor operatii (eliminare nod, de exemplu), fiind o practicã curentã.

In multe aplicatii relatiile dintre nodurile unui arbore multicãi nu pot fi modificate pentru a reduce înãltimea arborelui (ca în cazul arborilor binari de cãutare), deoarece aceste relatii sunt impuse de aplicatie si nu de valorile din noduri.

Crearea unui arbore nebinar se face prin adãugarea de noduri frunzã, folosind functiile “addChild” si “find”.

Nodul fiu este un nod nou creat cu o valoare datã (cititã sau extrasã dintr-un fisier sau obtinutã prin alte metode). Nodul pãrinte 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 existã un astfel de cursor care se deplaseazã de la un nod la altul.

Datele pe baza cãrora se construieste un arbore pot fi date în mai multe forme, care reprezintã 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 tatã-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 fiecãrui nod din arbore:
  1 1.1 1.2 1.3 ; 1.1 1.1.1 1.1.2 ; 1.2 1.2.1
- secvente de valori de pe o cale ce pleacã de la rãdãcina si se terminã la o frunzã:
  1/1.1/1.1.1 ; 1/1.1/1.1.2 ; 1/1.2 /1.2.1 ; 1/1.3

Ultima formã este un mod de identificare a unor noduri dintr-un arbore si se foloseste pentru calea completã la un fisiere si în XPath pentru noduri dintr-un arbore (dintr-o structurã) XML.

Algoritmul de construire a unui arbore cu fisierele dintr-un director si din subdirectoarele sale este recursiv: la fiecare apel primeste un nume de fisier; dacã acest fisier este un subdirector atunci creeazã noduri pentru fisierele din subdirector si repetã apelul pentru fiecare din aceste fisiere. Din fisierele normale se creeazã 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 curentã în arbore coboarã dupã fiecare nod creat pentru un subdirector si urcã dupã crearea unui nod frunzã (fisier normal).

Nodul rãdãcinã este construit separat, iar adresa sa este transmisã la primul apel.

Standardul DOM (Document Object Model), elaborat de consortiul W3C, stabileste tipurile de date si operatiile (functiile) necesare pentru crearea si prelucrarea arborilor ce reprezintã structura unui fisier XML. Standardul DOM urmãreste separarea programelor de aplicatii de modul de implementare a arborelui si unificarea accesului la arborii creati de programe parser XML de tip DOM .

DOM este un model de tip arbore general (multicãi) în care fiecare nod are un nume, o valoare si un tip. Numele si 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 si simbolice):

```
1 (ELEMENT_NODE)          nod ce contine un marcaj (tag)
3 (TEXT_NODE)             nod ce contine un text delimitat de marcaje
9 (DOCUMENT_NODE)         nod rãdãcinã al unui arbore document
```

Un nod element are drept nume marcajul corespunzãtor si ca valoare unicã pentru toate nodurile de tip 1 un pointer NULL. Toate nodurile text au acelasi nume ("#text"), dar valoarea este sirul dintre marcaje. Tipul "Node" (sau "DOMNode") desemneazã un nod de arbore DOM si este asociat cu operatii 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 oferã programatorilor de aplicatii operatii pentru crearea unui arbore DOM prin program sau pe baza analizei unui fisier XML, precum si pentru acces la nodurile arborelui în vederea extragerii informatiilor necesare în aplicatie. Programul parser face si o verificare a utilizãrii corecte a marcajelor de început si de sfârsit (de corectitudine formalã a fisierului XML analizat).

Construirea unui arbore XML se poate face fie printr-o functie recursivã, fie folosind o stivã de pointeri la noduri (ca si în cazul arborelui de fisiere), fie folosind legãtura la nodul pãrinte: în cazul unui marcaj de început (de forma <tag>) se coboarã un nivel, iar în cazul unui marcaj de sfârsit (de forma </tag>) se urcã un nivel în arbore. Acest ultim algoritm de creare a unui arbore DOM pe baza unui fisier XML poate fi descris astfel:

```
creare nod radacina r cu valoarea “Document”
crt=r                          // pozitie curenta in arbore
repeta cat timp nu e sfarsit de fisier xml {
    extrage urmatorul simbol din fisier in token
    daca token este marcaj de inceput atunci {
        creare nod “nou” avand ca nume marcaj
        adauga la crt pe nou
        crt=nou                    // coboara un nivel
    }
    daca token este marcaj de sfarsit atunci
          crt = parent(crt)        // urca un nivel, la nod parinte
    daca token este text atunci {
         creare nod “nou” cu valoare text
         adauga la crt pe nou      // si ramane pe acelasi nivel
    }
}
```

7.7 ALTE STRUCTURI DE ARBORE

Reprezentarea sirurilor de caractere prin vectori conduce la performante slabe pentru anumite operatii asupra unor siruri (texte) foarte lungi, asa cum este cazul editãrii unor documente mari. Este vorba de durata unor operatii cum ar fi intercalarea unui text într-un document mare, eliminarea sau înlocuirea unor portiuni de text, concatenarea de texte, s.a., dar si de memoria necesarã pentru operatii cu siruri nemodificabile (“immutable”), sau pentru pãstrarea unei istorii a operatiilor de modificare a textelor necesarã pentru anularea unor operatii anterioare (“undo”).

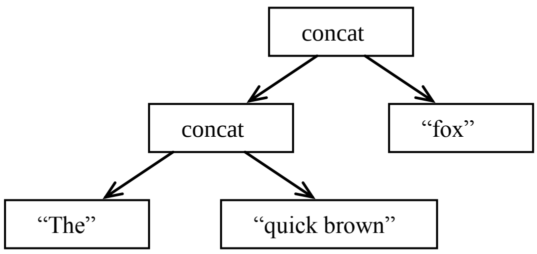
Structura de date numitã “rope” (ca variantã a cuvântului “string”, pentru a sugera o însiruire de caractere) a fost propusã si implementatã (în diferite variante) pentru a permite operatii eficiente cu texte foarte lungi (de exemplu clasa “rope” din STL).

Un “rope” este un arbore multicãi, realizat de obicei ca arbore binar, în care numai nodurile frunzã contin (sub)siruri de caractere (ca pointeri la vectori alocati dinamic).

Dacã vrem sã scriem continutul unui “rope” într-un fisier atunci se vor scrie succesiv sirurile din nodurile frunzã, de la stânga la dreapta.

Nodurile interne sunt doar puncte de reunire a unor subsiruri, prin concatenarea cãrora a rezultat textul reprezentat printr-un “rope”. Anumite operatii de modificare a textului dintr-un “rope” sunt realizate prin modificarea unor noduri din arbore, fãrã deplasarea în memorie a unor blocuri mari si fãrã copierea inutilã a unor siruri dintr-un loc în altul (pentru a pãstra intacte sirurile concatenate).

Figura urmãtoare este preluatã din articolul care a lansat ideea de “rope”:



Crearea de noduri intermediare de tip “concat” la fiecare adãugare de caractere la un text ar putea mãri înãltimea arborelui “rope”, si deci timpul de cãutare a unui caracter (sau subsir) într-un “rope”.

Din acest motiv concatenarea unor siruri scurte se face direct în nodurile frunzã, fãrã crearea de noduri noi. S-au mai propus si alte optimizãri pentru structura de "rope", inclusiv reechilibrarea automatã a arborelui, care poate deveni un arbore B sau AVL. Pentru a stabili momentul când devine necesarã reechilibrarea se poate impune o înãltime maximã si se poate memora în fiecare nod intern înãltimea (sau adâncimea) sa.

Determinarea pozitiei unui caracter (subsir) dat într-un text "rope" necesitã memorarea în fiecare nod a lungimii subsirului din fiecare subarbore. Algoritmul care urmeazã extrage un subsir de lungime "len" care începe în pozitia "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 subarborele stanga
 else
    left= substr(rope.left,start,len)       // subsirul se afla numai in subarborele stanga
      // partea dreapta din subsir
  if start <=length(rope.left) and start + len >= length(rope.left) + length(rope.right)
     right=rope.right                        // subsirul include subarborele 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 douã structuri: una pentru noduri interne si una pentru noduri frunzã.

Un arbore "Trie" (de la "retrieve" = regãsire) 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. In exemplul urmãtor este un trie construit cu sirurile: cana, cant, casa, dop, mic, minge.

```
              -
          /   |   \
         c    d    m
       /      |      \
      a       o       i
    /   \     |      /  \
   n     s    p     c    n
  / \    |               |
 a   t   a               g
                         |
                         e
```

Nodurile unui trie pot contine sau nu date, iar un sir este o cale de la rãdãcinã la un nod frunzã sau la un nod interior. Pentru siruri de biti arborele trie este binar, dar pentru siruri de caractere arborele trie nu mai este binar (numãrul de succesori ai unui nod este egal cu numãrul de caractere distincte din sirurile memorate).

Intr-un trie binar pozitia unui fiu (la stânga sau la dreapta) determinã implicit valoarea fiului respectiv (0 sau 1).

Avantajele unui arbore trie sunt:

- Regãsirea rapidã a unui sir dat sau verificarea apartenentei unui sir dat la dictionar; numãrul de comparatii este determinat numai de lungimea sirului cãutat, indiferent de numãrul de siruri memorate în dictionar (deci este un timp constant O(1) în raport cu dimensiunea colectiei). Acest timp poate fi important într-un program "spellchecker" care verificã dacã fiecare cuvânt dintr-un text apartine sau nu unui dictionar.
- Determinarea celui mai lung prefix al unui sir dat care se aflã în dictionar (operatie necesarã în algoritmul de compresie LZW).
- O anumitã reducere a spatiului de memorare, dacã 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

```
                 -
             /       \
           0/           \1                  bit 0
            -            -
       0/     \1       0/     \1            bit 1
        -      -        -      -
     0/  \1 0/  \1  0/  \1 0/  \1           bit 2
     -  4  2  6  -  5  3  7
      \1    \1      \1                      bit 3
      8     10      9
```

Este de remarcat cã structura unui arbore trie nu depinde de ordinea în care se adaugã valorile la arbore, iar arborele este în mod natural relativ echilibrat. Inãltimea unui arbore trie este determinatã de lungimea celui mai lung sir memorat si nu depinde de numărul de valori memorate.

Arborele Huffman de coduri binare este un exemplu de trie binar, în care codurile sunt cãi de la rãdãcinã la frunzele arborelui (nodurile interne nu sunt semnificative).

Pentru arbori trie este avantajoasã memorarea lor ca vectori (matrice) si nu ca arbori cu pointeri (un pointer ocupã uzual 32 biti, un indice de 16 biti este suficient pentru vectori de 64 k elemente). O solutie si mai compactã este un vector de biti, în care fiecare bit marcheazã prezenta sau absenta unui nod, la parcurgerea în lãtime.

Dictionarul folosit de algoritmul de compresie LZW poate fi memorat ca un "trie". Exemplul următor este arborele trie, reprezentat prin doi vectori "left" si "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	-	-	-	-	-	-	-

In acest arbore trie toate nodurile sunt semnificative, pentru cã reprezintã secvente codificate, iar codurile sunt chiar pozitiile în vectori (notate cu 'i'). In pozitia 0 se aflã nodul rãdãcinã, care are la stânga nodul 1 ('a') si la dreapta nodul 2 ('b'), s.a.m.d.

Cãutarea unui sir 'w' în acest arbore aratã 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[i];                  // continua la stanga
    else                          // daca este 'b'
      i=right[i];                 // continua la dreapta
    k++;                          // caracterul urmator din w
   }
  return j;    // ultimul nivel din trie care se potriveste
}
```

Adãugarea unui sir 'w' la arborele trie începe prin cãutarea pozitiei (nodului) unde se terminã cel mai lung prefix din 'w' aflat în trie si continuã cu adãugarea la trie a caracterelor următoare din 'w'.

Pentru reducerea spatiului de memorare în cazul unor cuvinte lungi, cu prea putine caractere comune cu alte cuvinte în prefix, este posibilã comasarea unei subcãi din arbore ce contine noduri cu un singur fiu într-un singur nod; acesti arbori trie comprimati se numesc si arbori Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric).

Intr-un arbore Patricia nu existã noduri cu un singur succesor si în fiecare nod se memoreazã 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 rapidã (într-un timp proportional cu lungimea lui q) a conditiei ca un sir dat q sã fie un suffix al unui sir dat s.

Arborii kD sunt un caz special de arbori binari de cãutare, iar arborii QuadTree (QT) sunt arbori multicãi, dar utilizarea lor este aceeasi: pentru descompunerea unui spatiu k-dimensional în regiuni dreptunghiulare (hiperdreptunghiulare pentru k >2). Fiecare regiune (celulã) contine un singur punct sau un numãr redus de puncte dintr-o portiune a spatiului k-dimensional. Impãrtirea spatiului se face prin (hiper)plane paralele cu axele.

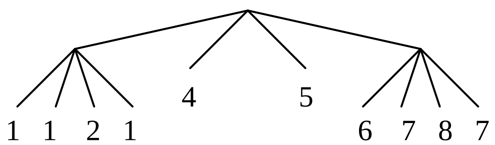
Vom exemplifica cu cazul unui spatiu bidimensional (k=2) deoarece arborii "QuadTree" (QT) reprezintã alternativa arborilor 2D. Intr-un arbore QT fiecare nod care nu e o frunzã are exact 4 succesori. Arborii QT sunt folositi pentru reprezentarea compactã a unor imagini fotografice care contin un numãr mare de puncte diferit colorate, dar în care existã regiuni cu puncte de aceeasi culoare. Fiecare regiune apare ca un nod frunzã în arborele QT.

Construirea unui arbore QT se face prin împãrtire succesivã a unui dreptunghi în 4 dreptunghiuri egale (stânga, dreapta, sus, jos) printr-o linie verticalã si una orizontalã. Cei 4 succesori ai unui nod corespund celor 4 dreptunghiuri (celule) componente. Operatia de divizare este aplicatã recursiv pânã când toate punctele dintr-un dreptunghi au aceeasi valoare.

O aplicatie pentru arbori QT este reprezentarea unei imagini colorate cu diferite culori, încadratã într-un dreptunghi ce corespunde rãdãcinii arborelui. Dacã una din celulele rezultate prin partitionare contine puncte de aceeasi culoare, atunci se adaugã un nod frunzã etichetat cu acea culoare. Dacã o celulã contine puncte de diferite culori atunci este împãrtitã în alte 4 celule mai mici, care corespund celor 4 noduri fii.

Exemplu de imagine si de arbore QT asociat acestei imagini.

1 1 4 4
2 1 4 4
5 5 6 7
5 5 8 7



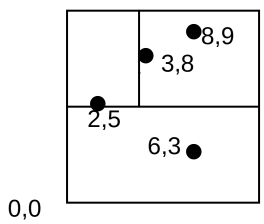
Nodurile unui arbore QT pot fi identificate prin numere întregi (indici) si/sau prin coordonatele celulei din imagine pe care o reprezintã în arbore.

Reprezentarea unui quadtree ca arbore cu pointeri necesitã multã memorie (în cazul unui numãr mare de noduri) si de aceea se folosesc si structuri liniare cu legãturi implicite (vector cu lista nodurilor din arbore), mai ales pentru arbori statici, care nu se modificã în timp.

Descompunerea spatiului 2D pentru un quadtree se face simultan pe ambele directii (printr-o linie orizontalã si una verticalã), iar în cazul unui arbore 2D se face succesiv pe fiecare din cele douã directii (sau pe cele k directii, pentru arbori kD).

Arborii kD se folosesc pentru memorarea coordonatelor unui numãr 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 planã dreptunghiularã delimitatã de punctele (0,0) si (10,10) va putea fi descompusã astfel:



Prima linie a fost orizontala la y=5 prin punctul (2,5), iar a doua linie a fost semi-dreapta verticalã la x=3, prin punctul (3,8). Arborele 2D corespunzãtor acestei împãrtiri a spatiului este urmãtorul:

```
          (2,5)
          /    \
     (6,3)   (3,8)
                \
               (8,9)
```

Punctul (6,3) se aflã în regiunea de sub (2,5) iar (3,8) în regiunea de deasupra lui (2,5); fatã 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).

Altã secventã de puncte sau de orizontale si verticale ar fi condus la un alt arbore, cu acelasi numãr de noduri dar cu altã înãltime si altã rãdacinã. Dacã toate punctele sunt cunoscute de la început atunci ordinea în care sunt folosite este importantã si ar fi de dorit un arbore cu înãltime cât mai micã .

In ceea ce priveste ordinea de "tãiere" a spatiului, este posibilã fie o alternantã de linii orizontale si verticale (preferatã), fie o secventã de linii orizontale, urmatã de o secventã de linii verticale, fie o altã secventã. Este posibilã si o variantã de împãrtire a spatiului în celule egale (ca la arborii QT) în care caz nodurile arborelui kD nu ar mai contine coordonatele unor puncte date.

Fiecare nod dintr-un arbore kD contine un numãr de k chei, iar decizia de continuare de pe un nivel pe nivelul inferior (la stânga sau la dreapta) este dictatã de o altã cheie (sau de o altã coordonatã). Dacã se folosesc mai întâi toate semidreptele ce trec printr-un punct si apoi se trece la punctul urmãtor, atunci nivelul urmãtor celui cu numãrul j va fi (j+1)% k unde k este numãrul de dimensiuni.

Pentru un arbore 2D fiecare nod contine ca date 2 întregi (x,y), iar ordinea de tãiere în ceea ce urmeazã va fi y1, x1, y2, x2, y3, x3, ... Cãutarea si inserarea într-un arbore kD seamãnã cu operatiile corespunzãtoare dintr-un arbore binar de cãutare BST, cu diferenta cã pe fiecare nivel se foloseste o altã 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] )            // dacã se continuã 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 operatii de cãutare într-o imagine sau într-o bazã de date (unde fiecare cheie de cãutare corespunde unei dimensiuni): localizarea celulei în care se aflã un anumit punct, cãutarea celui mai apropiat vecin, cãutare regiuni (ce puncte se aflã într-o anumitã regiune), cãutare cu informatii partiale (se cunosc valorile unor chei dar nu se stie nimic despre unul sau câteva atribute ale articolelor cãutate).

Exemplu cu determinarea punctelor care se aflã într-o regiune dreptunghiularã cu punctul de minim "low" si 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 );
}
```

Cãutarea celui mai apropiat vecin al unui punct dat folosind un arbore kD determinã o primã aproximatie ca fiind nodul frunzã care ar putea contine punctul dat. Exemplu de functie de cãutare a punctului în a cãrui regiune s-ar putea gãsi un punct dat.

```
  // cautare (nod) regiune care (poate) contine 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 in t nod curent inainte de avans in arbore
     if (r→x[0]==c[0] && r→x[1]==c[1])
       return 1;               // gasit
     else
       if (c[k] <= r→x[k])
         r=r→left;
       else
         r=r→right;
   }
   return 0;         // negasit cand 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) si nu (8,7), care este indicat de functia "find"; la fel (2,4) este mai apropiat de (2,5) desi este continut în regiunea definitã de punctul (6,3).

De aceea, dupã ce se gãseste nodul cu "find", se cautã în apropierea acestui nod (în regiunile vecine), pânã când se gãseste cel mai apropiat punct. Nu vom intra în detaliile acestui algoritm, dar este sigur cã timpul necesar va fi mult mai mic decât timpul de cãutare a celui mai apropiat vecin într-o multime de N puncte, fãrã 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)), adicã este cel mult egal cu înãltimea arborelui.

Reducerea înãltimii unui arbore kD se poate face alegând la fiecare pas tãierea pe dimensiunea maximã în locul unei alternante regulate de dimensiuni; în acest caz mai trebuie memorat în fiecare nod si indicele cheii (dimensiunii) folosite în acel nod.

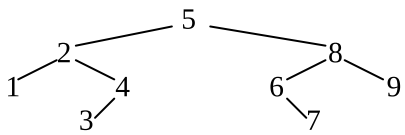
Capitolul 8

ARBORI DE CAUTARE

8.1 ARBORI BINARI DE CÃUTARE

Un arbore binar de cãutare (BST=Binary Search Tree), numit si arbore de sortare sau arbore ordonat, este un arbore binar cu proprietatea cã orice nod interior are valoarea mai mare decât orice nod din subarborele stânga si mai micã decât orice nod din subarborele dreapta. Exemplu de arbore binar de cãutare:

5 (2 (1,4(3,)), 8 (6 (,7),9))



Arborii BST permit mentinerea datelor în ordine si o cãutare rapidã a unei valori si de aceea se folosesc pentru implementarea de multimi si dictionare ordonate. Afisarea infixatã a unui arbore de cãutare produce un vector ordonat de valori.

Intr-un arbore ordonat, de cãutare, este importantã ordinea memorãrii succesorilor fiecãrui nod, deci este important care este fiul stânga si care este fiul dreapta.

Valoarea maximã dintr-un arbore binar de cãutare se aflã în nodul din extremitatea dreaptã, iar valoarea minimã în nodul din extremitatea stângã. Exemplu :

```
  // determina adresa nod cu valoare minima din arbore nevid
tnod* min ( tnod * r) {          // minim din arbore ordonat cu rãdãcina r
  while ( r→st != NULL)          // mergi la stanga cât se poate
      r=r→st;
   return r;                      // r poate fi chiar radacina (fara fiu stanga)
}
```

Functia anterioarã poate fi utilã în determinarea succesorului unui nod dat p, în ordinea valorilor din noduri; valoarea imediat urmãtoare este fie valoarea minimã din subarborele dreapta al lui p, fie se aflã mai sus de p, dacã p nu are fiu dreapta:

```
tnod* succ (tnod* r,tnod* p) {  // NULL ptr nod cu valoare maxima
  if (p→dr !=NULL)               // daca are fiu dreapta
    return min (p→dr);           // atunci e minim din subarborele dreapta
  tnod* pp = parent (r,p);       // parinte nod p
  while ( pp != NULL && pp→dr==p) {  // de la parinte urca sus la stanga
       p=pp; pp=parent(r,pp);
  }
  return pp;                     // ultimul nod cu fiu dreapta (sau NULL)
}
```

Functia "parent" determinã pãrintele unui nod dat si este fie o cãutare în arbore pornitã de la rãdãcinã, fie o singurã instructiune, dacã se memoreazã în fiecare nod si o legãturã la nodul pãrinte.

Cãutarea într-un arbore BST este comparabilã cu cãutarea binarã pentru vectori ordonati: dupã ce se comparã valoarea cãutatã cu valoarea din rãdãcinã se poate decide în care din cei doi subarbori se aflã (dacã existã) valoarea cãutatã. Fiecare nouã comparatie eliminã un subarbore din cãutare si reduce cu 1 înãltimea arborelui în care se cautã. Procesul de cãutare într-un arbore binar ordonat poate fi exprimat recursiv sau nerecursiv.

```
  //  cãutare recursivã 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
}
  //  cãutare nerecursivã 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 cãutare se realizeazã pentru un arbore BST echilibrat (cu înãltime minimã), la care înãltimile celor doi subarbori sunt egale sau diferã cu 1. Acest timp este de ordinul $\log_2 n$, unde n este numãrul total de noduri din arbore.

Determinarea pãrintelui unui nod p în arborele cu rãdãcina r ,prin cãutare, în varianta recursivã:

```
tnod* parent (tnod* r, tnod* p) {
  if (r==NULL || r==p) return NULL;          // daca p nu are parinte
  tnod* q =r;                           // q va fi parintele lui p
  if (p→val < q→val)                    // daca p in stanga lui q
    if (q→st == p) return q;            // q este parintele lui p
    else return parent (q→st,p);        // nu este q, mai cauta in stanga lui q
  if (p→val > q→val)                    // daca p in dreapta lui q
    if (q→dr == p) return q;            // q este parintele lui p
    else return parent (q→dr,p);        // nu este q, mai cauta in dreapta lui q
}
```

Adãugarea unui nod la un arbore BST seamãnã cu cãutarea, pentru cã se cautã nodul frunzã cu valoarea cea mai apropiatã de valoarea care se adaugã. Nodul nou se adaugã ca frunzã (arborele creste 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
   r =(tnod*) malloc (sizeof(tnod));       // creare radacina (sub)arbore
   r→val =x;  r→st = r→dr = NULL;
   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
}
```

Aceeasi functie de adãugare, fãrã argumente de tip referintã:

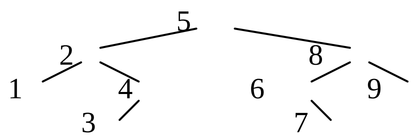
```
tnod * add (tnod * r, int x) {        // rezultat= noua radacina
 if (r==NULL) {
   r =(tnod*) malloc (sizeof(tnod));
   r→val =x;
   r→st = r→dr = NULL;
 }
 else
   if (x < r→val)
      r→st= add2 (r→st,x);
   else
      r→dr= add2 (r→dr,x);
 return r;
}
```

Eliminarea unui nod cu valoare datã dintr-un arbore BST trebuie sã considere urmãtoarele situatii:
- Nodul de sters nu are succesori (este o frunzã);
- Nodul de sters are un singur succesor;
- Nodul de sters are doi succesori.

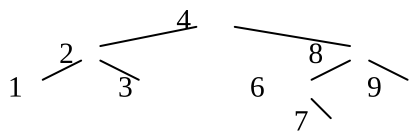
Eliminarea unui nod cu un succesor sau fãrã succesori se reduce la înlocuirea legãturii la nodul sters prin legãtura acestuia la succesorul sãu (care poate fi NULL).

Eliminarea unui nod cu 2 succesori se face prin înlocuirea sa cu un nod care are cea mai apropiatã valoare de cel sters; acesta poate fi nodul din extremitatea dreaptã a subarborelui stânga sau nodul din extremitatea stânga a subarborelui dreapta (este fie predecesorul, fie succesorul în ordine infixatã). Acest nod are cel mult un succesor

Fie arborele BST urmãtor



Eliminarea nodului 5 se face fie prin înlocuirea sa cu nodul 4, fie prin înlocuirea sa cu nodul 6. Acelasi arbore dupã înlocuirea nodului 5 prin nodul 4 :



Operatia de eliminare nod se poate exprima nerecursiv sau recursiv, iar functia se poate scrie ca functie de tip "void" cu parametru referintã sau ca functie cu rezultat pointer (adresa rãdãcinii arborelui se poate modifica în urma stergerii valorii din nodul rãdãcinã). Exemplu de functie nerecursivã 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;         // modifca parintele nodului eliminat
    else pp→dr=q;                    // prin inlocuirea fiului p cu nepotul q
  }
  free (p);
}
  // eliminare nod cu valoare x nod din bst (recursiv)
tnod*  del (tnod * r, int x) {
  tnod* tmp;
  if( r == NULL ) return r;       // x negasit
  if( x < r→val )                 // daca x mai mic
    r→st = del( r→st,x );         // elimina din subarb stanga
  else                            // daca x mai mare sau egal
   if( x > r→val )                // daca x mai mare
      r→dr = del ( r→dr,x );      // elimina din subarb dreapta
   else                           // daca x in nodul r
     if( r→st && r→dr ){          // daca r are doi fii
       tmp = min( r→dr );         // tmp= nod proxim lui r
       r→val = tmp→val;           // copiaza din tmp in r
       r→dr = del ( r→dr, tmp→val );     // si elimina nod proxim
     }
    else {                        // daca r are un singur fiu
      tmp = r;                    // pentru eliberare memorie
      r= r→st == 0 ? r→dr : r→st;      // inlocire r cu fiul sau
      free( tmp );
    }
    return r;                     // radacina, modificata sau nemodificata
}
```

8.2 ARBORI BINARI ECHILIBRATI

Cãutarea într-un arbore binar ordonat este eficientã dacã arborele este echilibrat. Timpul de cãutare într-un arbore este determinat de înãltimea arborelui, iar aceastã înãltime este cu atât mai micã cu cât arborele este mai echilibrat. Inãltimea minimã este O(lg n) si se realizeazã pentru un arbore echilibrat în înãltime.

Structura si înãltimea unui arbore binar de cãutare depinde de ordinea în care se adaugã valori în arbore, ordine impusã de aplicatie si care nu poate fi modificatã.

In functie de ordinea adãugãrilor de noi noduri (si eventual de stergeri) se poate ajunge la arbori foarte dezechilibrati; cazul cel mai defavorabil este un arbore cu toate nodurile pe aceeasi parte, cu un timp de cãutare de ordinul O(n).

Ideea generalã este ajustarea arborelui dupã operatii de adãugare sau de stergere, dacã aceste operatii stricã echilibrul existent. Structura arborelui se modificã prin rotatii de noduri, dar se mentin

relatiile dintre valorile continute în noduri. Este posibilã si modificarea anticipatã a unui arbore, înainte de adãugarea unei valori, pentru cã se poate afla subarborele la care se va face adãugarea.
Exemple de arbori binari de cãutare cu acelasi continut dar cu structuri si înãltimi diferite:

```
1                  2                  3                  4
 \                / \                / \                /
  2      →       1   3      →       2   4      →       3
   \  rot.st. 1       \  rot.st.2  /       rot.st.3   /
    3                  4          1                  2
     \                                              /
      4                                            1
```

De cele mai multe ori se verificã echilibrul si se modificã structura dupã fiecare operatie de adãugare sau de eliminare, dar în cazul arborilor Scapegoat modificãrile se fac numai din când în când (dupã un numãr oarecare de operatii asupra arborelui).

Criteriile de apreciere a echilibrului pot fi deterministe sau probabiliste.

Criteriile deterministe au totdeauna ca efect reducerea sau mentinerea înãltimii arborelui. Exemple:
- diferenta dintre înãltimile celor doi subarbori ai fiecãrui nod (arbore echilibrat în înãltime), criteriu folosit de arborii AVL;
- diferenta dintre cea mai lungã si cea mai scurtã cale de la rãdãcinã la frunze, criteriu folosit de arborii RB (Red-Black);

Criteriile probabiliste pornesc de la observarea efectului unei secvente de modificãri asupra reducerii înãltimii arborilor de cãutare, chiar dacã dupã anumite operatii înãltimea arborelui poate creste (arbori Treap, Splay sau Scapegoat) .

In cele mai multe variante de arbori echilibrati se memoreazã în fiecare nod si o informatie suplimentarã, folositã la reechilibrare (înãltime nod, culoare nod, s.a.).

Arborii "scapegoat" memoreazã în fiecare nod atât înãltimea cât si numãrul de noduri din subarborele cu rãdãcina în acel nod. Ideea este de a nu face restructurarea arborelui prea frecvent, ea se va face numai dupã un numãr de adãugãri sau de stergeri de noduri. Stergerea unui nod nu este efectivã ci este doar o marcare a nodurilor respective ca invalidate. Eliminarea efectivã si restructurarea se va face numai când în arbore sunt mai mult de jumãtate de noduri marcate ca sterse. La adãugarea unui nod se actualizeazã înãltimea si numãrul de noduri pentru nodurile de pe calea ce contine nodul nou si se verificã pornind de la nodul adãugat în sus, spre rãdãcinã dacã existã un arbore prea dezechilibrat, cu înãltime mai mare ca logaritmul numãrului de noduri: $h(v) > m + \log(|v|)$. Se va restructura numai acel subarbore gãsit vinovat de dezechilibrarea întregului arbore ("scapegoat"=tap ispãsitor).

Fie urmãtorul subarbore dintr-un arbore BST:

```
  15
 /  \
10   20
```

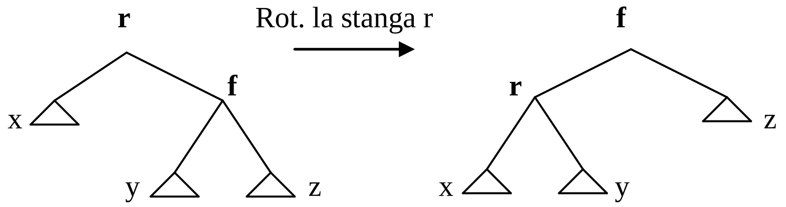
Dupã ce se adaugã valoarea 8 nu se face nici o modificare, desi subarborele devine "putin" dezechilibrat. Dacã se adaugã si valoarea 5, atunci subarborele devine "mult" dezechilibrat si se va restructura, fãrã a fi nevoie sã se propage în sus modificarea (pãrintele lui 15 era mai mare ca 15, deci va fi mai mare si ca 10). Exemplu:

```
       15                       10
      /  \                     /  \
    10    20                  8    15
    /                        /       \
   8                        5         20
  /
 5
```

Costul amortizat al operatiilor de insertie si stergere într-un arbore "scapegoat" este tot O(log(n)).

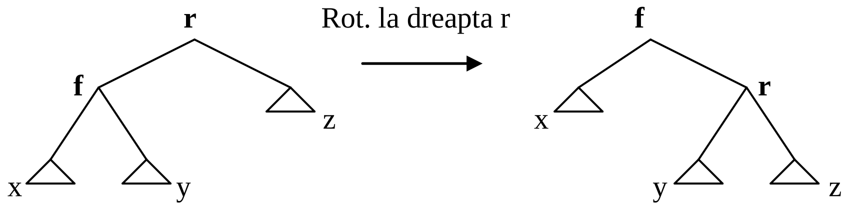
Restructurarea unui arbore binar de cãutare se face prin rotatii; o rotatie modificã structura unui (sub)arbore, dar mentine relatiile dintre valorile din noduri.

Rotatia la stânga în subarborele cu rãdãcina r coboarã nodul r la stânga si aduce în locul lui fiul sãu dreapta f, iar r devine fiu stânga al lui f (val(f) > val(r)).



Prin rotatii se mentin relatiile dintre valorile nodurilor:
x < r < f < z ; r < y < f ;

Rotatia la dreapta a nodului r coboarã pe r la dreapta si aduce în locul lui fiul sãu stânga f ; r devine fiu dreapta al lui f.



Se observã cã la rotatie se modificã o singurã legãturã, cea a subarborelui y în figurile anterioare.

Rotatiile au ca efect ridicarea (si coborârea) unor noduri în arbore si pot reduce înãltimea arborelui. Pentru a ridica un nod ('f' în figurile anterioare) se roteste pãrintele nodului care trebuie ridicat (notat cu 'r' aici), fie la dreapta, fie la stânga.

Exemplul urmãtor aratã cum se poate reduce înãltimea unui arbore printr-o rotatie (nodul 7 coboara la dreapta iar nodul 5 urcã în rãdãcinã):

```
     7                                5
    /                               /   \
   5        Rot. dreapta 7   ->    3     7
  /
 3
```

Codificarea rotatiilor depinde de utilizarea functiilor respective si poate avea o formã mai simplã sau mai complexã.

In forma simplã se considerã cã nodul rotit este rãdãcina unui (sub)arbore si nu are un nod pãrinte (sau cã pãrintele se modificã într-o altã functie):

```
  // Rotatie dreapta radacina prin inlocuire cu fiul din stanga
void rotR ( tnod* & r) {
  tnod* 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 primitã se modificã
}
  // Rotatie stanga radacina prin inlocuire cu fiul din dreapta
void rotL ( tnod* & r) {
  tnod* 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
}
```

Dacã nodul rotit p este un nod interior (cu pãrinte) atunci trebuie modificatã si legãtura de la pãrintele lui p cãtre nodul adus în locul lui p. Pãrintele nodului p se poate afla folosind un pointer pãstrat în fiecare nod, sau printr-o cãutare pornind din rãdãcina arborelui. Exemplu de functie pentru rotatie dreapta a unui nod interior p într-un arbore cu legãturi în sus (la noduri pãrinte):

```
void rotateR (tnod* & root, tnod *p) {
   tnod * 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
}
```

Rotatiile de noduri interne se aplicã dupã terminarea operatiei de adãugare si necesitã gãsirea nodului care trebuie rotit.

Rotatiile simple, care se aplicã numai rãdãcinii unui (sub)arbore, se folosesc în functii recursive de adãugare de noduri, unde adãugarea si rotatia se aplicã recursiv unui subarbore tot mai mic, identificat prin rãdãcina sa. Subarborii sunt afectati succesiv (de adãugare si rotatie), de la cel mai mic la cel mai mare (de jos în sus), astfel încât modificarea legãturilor dintre noduri se propagã treptat în sus.

8.3 ARBORI SPLAY SI TREAP

Arborii binari de cãutare numiti "Splay" si "Treap" nu au un criteriu determinist de mentinere a echilibrului, iar înãltimea lor este mentinutã în limite acceptabile.

Desi au utilizãri diferite, arborii Splay si Treap folosesc un algoritm asemãnãtor de ridicare în arbore a ultimului nod adãugat; acest nod este ridicat mereu în rãdãcinã (arbori Splay) sau pânã când este îndeplinitã o conditie (Treap).

In anumite aplicatii acelasi nod face obiectul unor operatii succesive de cãutare, insertie, stergere. Altfel spus, probabilitatea cãutãrii aceleasi valori dintr-o colectie este destul de mare, dupã un prim acces la acea valoare. Aceasta este si ideea care stã la baza memoriilor "cache". Pentru astfel de cazuri este utilã modificarea automatã a structurii dupã fiecare operatie de cãutare, de adãugare sau de stergere, astfel ca valorile cãutate cel mai recent sã fie cât mai aproape de rãdãcinã.

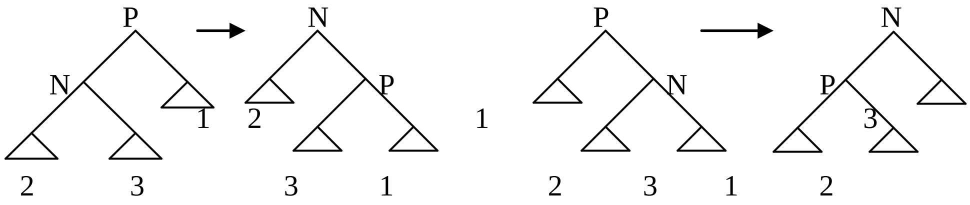
Un arbore "splay" este un arbore binar de cãutare, care se modificã automat pentru aducerea ultimei valori accesate în rãdãcina arborelui, prin rotatii, dupã cãutarea sau dupã adãugarea unui nou nod, ca frunzã. Pentru stergere, se aduce întâi nodul de eliminat în rãdãcinã si apoi se sterge.

Timpul necesar aducerii unui nod în rãdãcinã depinde de distanta acestuia fatã de rãdãcinã, dar în medie sunt necesare O(n*log(n) + m*log(n)) operatii pentru m adãugãri la un arbore cu n noduri, iar fiecare operatie de "splay" costã O(n*log(n)).

Operatia de ridicare a unui nod N se poate realiza în mai multe feluri:
- Prin ridicarea treptatã a nodului N, prin rotatii simple, repetate, functie de relatia dintre N si pãrintele sãu ("move-to-root");
- Prin ridicarea pãrintelui lui N, urmatã de ridicarea lui N ("splay").

Cea de a doua metodã are ca efect echilibrarea mai bunã a arborelui "splay", în anumite cazuri de arbori foarte dezechilibrati, dar este ceva mai complexã.

Dacã N are doar pãrinte P si nu are “bunic” (P este rãdãcina arborelui) atunci se face o singurã rotatie pentru a-l aduce pe N în rãdãcina arborelui (nu existã nici o diferentã între “move-to-root” si “splay”):



(N<P) rot. la dreapta (zig) (N>P) rot. la stanga (zag)

Dacã N are si un bunic B (pãrintele lui P) atunci se deosebesc 4 cazuri, functie de pozitia nodului (nou) accesat N fatã de pãrintele sãu P si a pãrintelui P fatã de “bunicul” B al lui N :

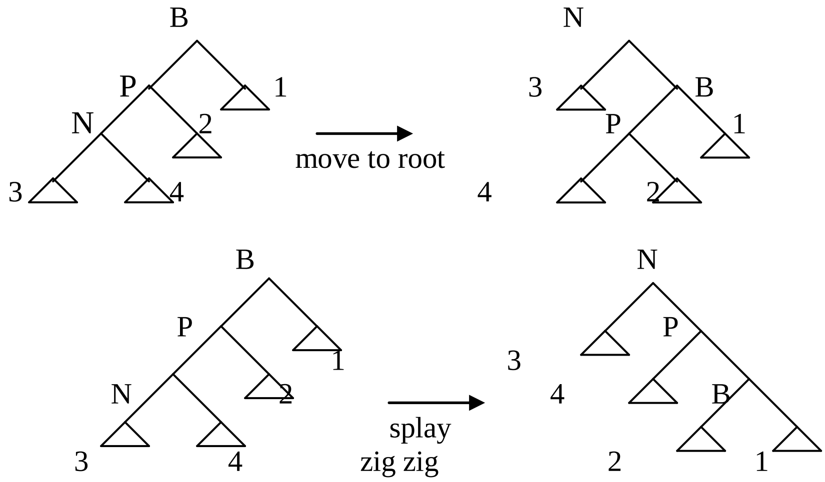
Cazul 1(zig zig): N < P < B (N si P fii stânga) - Se ridicã mai întâi P (rotatie dreapta B) si apoi se ridicã N (rotatie dreapta P)

Cazul 2(zag zag): N > P > B (N si P fii dreapta), simetric cu cazul 1 - Se ridicã P (rotatie stânga B) si apoi se ridicã N (rotatie stânga P)

Cazul 3(zig zag): P < N < B (N fiu dreapta, P fiu stânga) - Se ridicã N de douã 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 ridicã N de douã 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 functie pentru adãugarea unei valori la un arbore Splay:

```
void insertS (tnod* &t, int x){
    insert (t,x);          // adaugare ca la orice arbore binar de cãutare
    splayr (t,x);          // ridicare x in radacina arborelui
}
```

Urmeazã douã variante de functii “move-to-root” pentru ridicare în rãdãcinã:

```
  // movetoroot  recursiv
void splayr( tnod * & r, int  x ) {
   tnod* p;
   p=find(r,x);
   if (p==r) return;
   if (x > p→parent→val)
       rotateL (r,p→parent);
```

```
    else
      rotateR (r,p→parent);
   splayr(r,x);
}
 // movetoroot iterativ
void splay( tnod * & r, int  x ) {
   tnod * p;
   while ( x != r→val) {
     p=find(r,x);
     if (p==r) return;
     if (x > p→parent→val)
        rotateL (r,p→parent);
     else
        rotateR (r,p→parent);
   }
}
```

Functia "splay" este apelatã si dupã cãutarea unei valori x în arbore. Dacã valoarea cãutatã x nu existã în arbore, atunci se aduce în rãdãcinã nodul cu valoarea cea mai apropiatã de x, ultimul pe calea de cãutare a lui x. Dupã eliminarea unui nod cu valoarea x se aduce în rãdãcinã valoarea cea mai apropiatã de x.

In cazul arborilor Treap se memoreazã în fiecare nod si o prioritate (numãr întreg generat aleator), iar arborele de cãutare (ordonat dupã valorile din noduri) este obligat sã respecte si conditia de heap relativ la prioritãtile nodurilor. Un treap nu este un heap deoarece nu are toate nivelurile complete, dar în medie înãltimea sa nu depãseste dublul înãltimii minime (2*lg(n)).

Desi nu sunt dintre cei mai cunoscuti arbori echilibrati (înãltimea medie este mai mare ca pentru alti arbori), arborii Treap folosesc numai rotatii simple si prezintã analogii cu structura "Heap", ceea ce îi face mai usor de înteles.

S-a arãtat cã pentru o secventã de chei generate aleator si adãugate la un arbore binar de cãutare, arborele este relativ echilibrat; mai exact, calea de lungime minimã este 1.4 lg(n)-2 iar calea de lungime maximã este 4.3 lg(n).

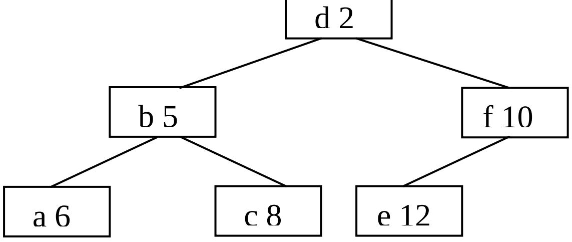
Numele "Treap" provine din "Tree Heap" si desemneazã o structurã care combinã caracteristicile unui arbore binar de cãutare cu caracteristicile unui Heap. Ideea este de a asocia fiecãrui nod o prioritate, generatã aleator si folositã la restructurare.

Fiecare nod din arbore contine o valoare (o cheie) si o prioritate. In raport cu cheia nodurile unui treap respectã conditia unui arbore de cãutare, iar în raport cu prioritatea este un min-heap. Prioritãtile 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
} tnod;
```

Exemplu de arbore treap construit cu urmãtoarele chei si prioritãti:

Cheie	a	b	c	d	e	f
Prior	6	5	8	2	12	10



In lipsa acestor prioritãti arborele ar fi avut înãltimea 6, deoarece cheile vin în ordinea valorilor. Echilibrarea se asigurã prin generarea aleatoare de prioritãti si rearanjarea arborelui binar de cãutare pentru a respecta si conditia de min-heap.

In principiu, adãugarea unei valori într-un treap se face într-o frunzã (ca la orice arbore binar de cãutare) dupã care se ridicã în sus nodul adãugat pentru a respecta conditia de heap pentru prioritate.

In detaliu, insertia si corectia se pot face în douã moduri:
- Corectia dupã insertie (care poate fi iterativã sau recursivã);
- Corectie si insertie, în mod recursiv (cu functii de rotatie scurte).

Varianta de adãugare cu corectie dupã ce se terminã adãugarea:

```
 // insertie nod  in Treap
void insertT( tnod *& r, int x,  int pri) {
  insert(r,x,pri);
  tnod * p= find(r,x);                    // adresa nod cu valoarea x
  fixup (r,p);                            // sau  fixupr(r,p);
}
 // corectie Treap functie de prioritate (recursiv)
void fixupr ( tnod * & r, tnod * t){
  tnod * 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
      rotateR (r,p);          // rotatie dreapta p
    else                          // daca t e fiu dreapta al lui p
      rotateL (r,p);          // rotatie stanga p
  fixupr(r,p);                    // continua recursiv in sus (p s-a modificat)
}
```

Functie iterativã de corectie dupã insertie, pentru mentinere ca heap dupã prioritate:

```
void fixup ( tnod * & r, tnod * t) {
  tnod * p;                          // nod parinte al lui t
  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
       rotateR (r,p);                // rotatie: se aduce t in locul lui p
     else                            // daca t e fiu dreapta al lui p
       rotateL (r,p);                // rotatie pentru inlocuire p cu t
     t=p;                            // muta comparatia mai sus un nivel
   }
}
```

Varianta cu adãugare recursivã si rotatie (cu functii scurte de rotatie):

```
void add ( 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 ) {
      add ( r→st, x,p );
      if( r→st→pri < r→pri )
        rotL ( r );
      }
    else if( x > r→val ) {
      add ( r→dr, x, p );
      if( r→dr→pri < r→pri )
        rotR ( r );
```

```
    }
   // else : x exista si nu se mai adauga
}
```

La adãugarea unui nod se pot efectua mai multe rotatii (dreapta si/sau stânga), dar numãrul lor nu poate depãsi înãltimea arborelui. Exemplul urmãtor aratã etapele prin care trece un treap cu rãdãcin E3 la adãugarea cheii G cu prioritatea 2:

```
      E3              E3             E3             E3              G2
     /  \            /  \           /  \           /  \            /  \
   B5    H7        B5    H7       B5    H7       B5    G2        E3    H7
   /    /  \       /    /  \      /    /  \      /    /  \      /  \     \
 A6   F9   K8    A6   F9   K8   A6   G2   K8   A6   F9   H7    B5   F9   K8
                        \             /                    \    /
                         G2          F9                    K8   A6

   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 complicatã decât eliminarea dintr-un arbore binar de cãutare; numai dupã eliminarea unui nod cu doi succesori se comparã prioritãtile fiilor nodului sters si se face o rotatie în jurul nodului cu prioritate mai mare (la stânga pentru fiul stânga si la dreapta pentru fiul dreapta).

O altã utilizare posibilã a unui treap este ca structurã de cãutare pentru chei cu probabilitãti diferite de cãutare; prioritatea este în acest caz determinatã de frecventa de cãutare a fiecãrei chei, iar rãdãcina are prioritatea maximã (este un max-heap).

8.4 ARBORI AVL

Arborii AVL (Adelson-Velski, Landis) sunt arbori binari de cãutare în care fiecare subarbore este echilibrat în înãltime. Pentru a recunoaste rapid o dezechilibrare a arborelui s-a introdus în fiecare nod un câmp suplimentar, care sã arate fie înãltimea nodului, fie diferenta dintre înãltimile celor doi subarbori pentru acel nod (–1, 0, 1 pentru noduri “echilibrate” si –2 sau +2 la producerea unui dezechilibru).

La adãugarea unui nou nod (ca frunzã) factorul de echilibru al unui nod interior se poate modifica la –2 (adãugare la subarborele stânga) sau la +2 (adãugare la subarborele dreapta), ceea ce va face necesarã modificarea structurii arborelui.

Reechilibrarea se face prin rotatii simple sau duble, însotite de recalcularea înãltimii fiecãrui nod întâlnit parcurgând arborele de jos în sus, spre rãdãcinã.

Fie arborele AVL urmãtor:

```
   c
  /
 b
```

Dupã adãugarea valorii ‘a’ arborele devine dezechilibrat spre stânga si se roteste nodul ‘c’ la dreapta pentru reechilibrare (rotatie simplã):

```
      c                    b
     /                    / \
    b        →           a   c
   /      rot. dreapta c
  a
```

Rotatia dublã este necesarã în cazul adãugãrii valorii ‘b’ la arborele AVL urmãtor:

```
a
 \
  c
```

Pentru reechilibrare se roteste c la dreapta si apoi a la stânga (rotatie dublã stânga):

```
a                    a                     b
 \                    \                   /  \
  c        →           b        →        a    c
  /   rot.dreapta c     \  rot.stanga a
 b                       c
```

Dacã cele 3 noduri formeazã o cale in zig-zag atunci se face o rotatie pentru a aduce cele 3 noduri in linie si apoi o rotatie pentru ridicarea nodului din mijloc.

Putem generaliza cazurile anterioare astfel:
- Insertia în subarborele dreapta al unui fiu dreapta necesitã o rotatie simplã la stânga
- Insertia în subarborele stânga al unui fiu stânga necesitã o rotatie simplã la dreapta
- Insertia în subarborele stânga al unui fiu dreapta necesitã o rotatie dublã la stânga
- Insertia în subarborele dreapta al unui fiu stânga necesitã o rotatie dublã la dreapta

Exemplu de arbore AVL (în paranteze înãltimea nodului):

```
              80 (3)
              /       \
       30 (2)       100 (1)
          /  \           /
    15(1)  40(0)  90(0)
        /  \
10(0)   20(0)
```

Adãugarea valorilor 120 sau 35 sau 50 nu necesitã nici o ajustare în arbore pentru cã factorii de echilibru rãmân în limitele [-1,+1].

Dupã adãugarea unui nod cu valoarea 5, arborele se va dezechilibra astfel:

```
              80 (4)
              /       \
       30 (3)       100 (1)
          /  \           /
    15(2)  40(0)  90(0)
        /  \
10(1)   20(0)
   /
 5(0)
```

Primul nod ,de jos în sus, dezechilibrat (spre stânga) este 30, iar solutia este o rotatie la dreapta a acestui nod, care rezultã în arborele urmãtor:

```
                80 (3)
              /         \
        15 (2)        100 (1)
        /     \          /
  10 (1)  30 (1)  90 (0)
    /          /   \
5 (0)  20(0)  40(0)
```

Exemplu de rotatie dublã (stânga,dreapta) pentru corectia dezechilibrului creat dupã adãugarea valorii 55 la arborele AVL urmãtor:

```
              80 (3)                                      80(4)
            /        \                                  /         \
        30(2)       100 (1)                         30(3)        100(1)
       /     \       /     \          →            /     \        /     \
   20(1)  50(1)  90(0) 120(0)                  20(1)  50(2)  90(0)  120(0)
     /        /   \                              /        /    \
 10(0)  40(0) 60(0)                            10(0)  40(0) 60(1)
                                                             /
                                                         55(0)
```

Primul nod dezechilibrat de deasupra celui adãugat este 80; de aceea se face întâi o rotatie la stânga a fiului sãu 30 si apoi o rotatie la dreapta a nodului 80 :

```
              80 (4)                                       50(3)
            /         \                                  /          \
        50(3)          100 (1)                       30(2)          80(2)
       /      \          /      \         →          /      \        /      \
   30(2)     60(1)   90(0) 120(0)              20(1)  40(0)  60(1)  100(1)
    /   \       /                                /                /        /    \
 20(1) 40(0) 55(0)                           10(0)           55(0)   90(0) 120(0)
   /
10(0)
```

Inãltimea maximã a unui arbore AVL este 1.44*log(n), deci în cazul cel mai rãu cãutarea într-un arbore AVL nu necesitã mai mult de 44% comparatii fatã de cele necesare într-un arbore perfect echilibrat. In medie, este necesarã o rotatie (simplã sau dublã) cam la 46,5% din adãugãri si este suficientã o singurã rotatie pentru refacere.

Implementarea care urmeazã memoreazã în fiecare nod din arbore înãltimea sa, adicã înãltimea subarborelui cu rãdãcina în acel nod. Un nod vid are înãltimea –1, iar un nod frunzã are înãltimea 0.

```
typedef struct tnod {
   int  val;                         // valoare din nod
   int h;                            // inaltime nod
   struct tnod  *st, *dr;        // adrese succesori
} tnod;
  // determina inaltime nod cu adresa p
int ht (tnod * p) {  return p==NULL? -1: p→h; }
```

Operatiile de rotatie simplã recalculeazã în plus si înãltimea:

```
    //  rotatie simpla la dreapta (radacina)
void rotR( tnod * & r ) {
    tnod *f = r→st;            // fiu stanga
    r→st = f→dr;
    f→dr = r;                    // r devine fiu dreapta al lui f
    r→h = max( ht(r→st), ht(r→dr))+1; // inaltime r dupa rotatie
    f→h=max( ht(f→st),r→h)+1;         // inaltime f dupa rotatie
    r = f;
}
    // rotatie simpla la stanga (radacina)
void rotL( tnod * & r ) {
    tnod *f = r→dr;         // fiu dreapta
    r→dr = f→st;
    f→st = r;                  // r devine fiu stanga al lui f
    r→h = max( ht(r→st), ht(r→dr))+1;  // inaltime r dupa rotatie
    f→h=max( ht(f→dr),r→h)+1;
    r = f;
}
```

Pentru arborii AVL sunt necesare si urmãtoarele rotatii duble:

```
  // rotatie dubla la stanga (RL)
void rotRL ( tnod * & p ){
   rotR ( p→dr );       // rotatie fiu dreapta la dreapta
   rotL ( p );          // si apoi rotatie p la stanga
}
  // rotatie dubla la dreapta (LR)
void rotLR ( tnod * & p ) {
   rotL ( p→st );       // rotatie fiu stanga la stanga
   rotR ( p );          // si apoi rotatie p la dreapta
}
```

Evolutia unui arbore AVL la adãugarea valorilor 1,2,3,4,5,6,7 este urmãtoarea:

```
 1   1     2     2       2          4           4
      \   / \   / \     / \        / \         / \
       2 1   3 1   3   1   4      2   5       2   6
                    \     / \    / \   \     / \ / \
                     4   3   5  1   3   6   1  3 5  7
```

Exemplu de functie recursivã pentru adãugarea unei noi valori la un arbore AVL :

```
    // adauga x la arbore AVL cu radacina r
void addFix ( tnod * & r, int x ) {
   if ( r == NULL ) {                          // daca arbore vid
     r =  (tnod*) malloc(sizeof(tnod));        // atunci se creeaza radacina r
     r→val=x; r→st = r→dr =NULL;
     r→h = 0;                       // inaltime nod unic
     return;
   }
   if (x==r→val)                    // daca x exista deja in arbore
      return;                       // atunci nu se mai adauga
   if( x < r→val ) {                // daca x este mai mic
      addFix ( r→st,x );            // atunci se adauga in subarbore stanga
      if( ht ( r→st ) – ht ( r→dr ) == 2 )       // daca subarbore dezechilibrat
          if( x < r→st→val )        // daca x mai mic ca fiul stanga
             rotR( r );             // rotatie dreapta r (radacina subarbore)
          else                      // daca x mai mare ca fiul stanga
             rotLR ( r );           // atunci dubla rotatie la dreapta r
      }
      else  {                       // daca x este mai mare
        addFix ( r→dr,x );          // atunci se adauga la subarborele dreapta
        if( ht ( r→dr ) – ht ( r→st ) == 2 )    // daca subarbore dezechilibrat
           if( x > r→dr→val )       // daca x mai mare ca fiul dreapta
              rotL ( r );           // atunci rotatie stanga r
           else                     // daca x mai mic ca fiul dreapta
              rotRL ( r );          // atunci dubla rotatie la stanga
      }
   r→h = max( ht ( r→st ), ht ( r→dr ) ) + 1;       // recalculeaza inaltime nod r
}
```

Spre deosebire de solutia recursivã, solutia iterativã necesitã accesul la nodul pãrinte, fie cu un pointer în plus la fiecare nod, fie folosind o functie "parent" care cautã pãrintele unui nod dat.

Pentru comparatie urmeazã o functie de corectie ("fixup") dupã adãugarea unui nod si dupã cãutarea primului nod dezechilibrat de deasupra celui adãugat ("toFix"):

```
   // cauta nodul cu dezechilibru 2 de deasupra lui “nou”
tnod* toFix (tnod* r, tnod* nou) {
  tnod* p= parent(r,nou);
  while ( p !=0 && abs(ht(p→st) - ht(p→dr))<2) // cat timp p este echilibrat
    p=parent(r,p);                              // urca la parintele lui p
  return p;                         // daca p are factor  2
}
  // rotatie stanga nod interior cu recalculare inaltimi noduri
void rotateL (tnod* & r, tnod* p) {
   tnod *f = p→dr;            // f=fiu dreapta al lui p
   if (f==NULL) return;
   p→dr = f→st;               // modifica fiu dreapta p
   f→st = p;
   p→h = max( ht(p→st), ht(p→dr))+1;        // inaltime p dupa rot
   f→h = max( p→h, ht(f→dr))+1;          // inaltime f dupa rot
   tnod* pp=parent(r,p);        // pp= parinte nod p
   if (pp==NULL)  r=f;          // daca p este radacina
   else {                       // daca p are un parinte pp
      if (f→val < pp→val)
        pp→st = f;              // f devine fiu stanga al lui pp
      else
        pp→dr = f;              // f devine fiu dreapta al lui pp
     while (pp != 0) {          // recalculare inaltimi deasupra lui p
       pp→h=max (ht(pp→st),ht(pp→dr)) + 1;
       pp=parent(r,pp);
     }
   }
}
  // reechilibrare prin rotatii dupa adaugare nod “nou”
void fixup (tnod* & r, tnod* nou ) {
   tnod *f, *p;
   p= toFix (r,nou);                 // p = nod dezechilibrat
   if (p==0) return ;                // daca nu s-a creat un dezechilibru
     // daca p are factor  2
   if ( ht(p→st) > ht(p→dr)) {      // daca p mai inalt la stanga
     f=p→st;                         // f = fiul stanga (mai inalt)
     if ( ht(f→st) > ht(f→dr))      // daca f mai inalt la stanga
       rotateR (r,p);                // cand p,f si f->st in linie
     else
       rotateLR (r,p);               // cand p, f si f->st in zig-zag
   }
   else {                            // daca p mai inalt la dreapta
     f=p→dr;                         // f= fiul dreapta (mai inalt)
     if (ht(f→dr) > ht(f→st))       // daca f mai inalt la dreapta
       rotateL (r,p);                // cand p,f si f->dr in linie
     else
       rotateRL (r,p);               // cand p, f si f->dr in zig-zag
   }
}
```

De observat cã înaltimile nodurilor se recalculeazã de jos în sus în arbore, într-un ciclu while, dar în varianta recursivã era o singurã instructiune, executatã la revenirea din fiecare apel recursiv (dupã adãugare si rotatii).

8.5 ARBORI RB SI AA

Arborii de cãutare cu noduri colorate ("Red Black Trees") realizeazã un bun compromis între gradul de dezechilibru al arborelui si numãrul de operatii necesare pentru mentinerea acestui grad. Un arbore RB are urmãtoarele proprietãti:

- Orice nod este colorat fie cu negru fie cu rosu.
- Fiii (inexistenti) ai nodurilor frunzã se considerã colorati în negru
- Un nod rosu nu poate avea decât fii negri
- Nodul rãdãcinã este negru
- Orice cale de la rãdãcinã la o frunzã are acelasi numãr de noduri negre.

Se considerã cã toate frunzele au ca fiu un nod sentinelã negru.

De observat cã nu este necesar ca pe fiecare cale sã alterneze noduri negre si rosii.

Consecinta acestor proprietãti este cã cea mai lungã cale din arbore este cel mult dublã fatã de cea mai scurtã cale din arbore; cea mai scurtã cale poate avea numai noduri negre, iar cea mai lungã are noduri negre si rosii care alterneazã.

O definitie posibilã a unui nod dintr-un arbore RB:

```
typedef struct tnod {
   int val;               //date din nod
   char color;              // culoare nod (‘N’ sau ‘R’)
   struct tnod *st, *dr;
} tnod;
tnod sentinel = { NIL, NIL, 0, ‘N’, 0};    // santinela este un nod negru
#define NIL &sentinel                       // adresa memorata in nodurile frunza
```

Orice nod nou primeste culoarea rosie si apoi se verificã culoarea nodului pãrinte si culoarea "unchiului" sãu (frate cu pãrintele sãu, pe acelasi nivel). La adãugarea unui nod (rosu) pot apãrea douã situatii care sã necesite modificarea arborelui:

a) Pãrinte rosu si unchi rosu:

```
      7(N)                       7(R)
     /    \          →          /    \
  5(R)    9(R)               5(N)    9(N)
   /                          /
 3(R)                       3(R)
```

Dupã ce se adaugã nodul rosu cu valoarea 3 se modificã culorile nodurilor cu valorile 5 (pãrinte) si 9 (unchi) din rosu în negru si culoarea nodului 7 din negru în rosu. Dacã 7 nu este rãdãcina atunci modificarea culorilor se propagã în sus.

b) Pãrinte rosu dar unchi negru (se adaugã nodul 3):

```
      7(N)                       5(N)
     /    \          →          /    \
  5(R)    9(N)               3(R)    7(R)
  /   \                              /    \
3(R)   6(N)                       6(N)  9(N)
```

In acest caz se roteste la dreapta nodul 7, dar modificarea nu se propagã în sus deoarece rãdãcina subarborelui are aceeasi culoare dinainte (negru).

Dacã noul nod se adaugã ca fiu dreapta (de ex. valoarea 6, dacã nu ar fi existat deja), atunci se face mai întâi o rotatie la stânga a nodului 5, astfel ca 6 sã ia locul lui 5, iar 5 sã devinã fiu stânga a lui 6.

Pentru a întelege modificãrile suferite de un arbore RB vom arãta evolutia sa la adãugarea valorilor 1,2,...8 (valori ordonate, cazul cel mai defavorabil):

```
1(N)        1(N)          2(N)            2(N)              2(N)
       →      \     →     /    \    →     /    \      →      /    \
              2(R)      1(R)   3(R)     1(N)   3(N)         1(N)  4(N)
                                                \                 /   \
                                               4(R)            3(R)  5(R)
```

```
    2(N)                2(N)                       4(N)
   /     \             /     \                    /          \
1(N)     4(R)       1(N)     4(R)              2(R)         6(R)
        /    \              /    \            /     \      /     \
     3(N)   5(N)        3(N)   6(N)        1(N)  3(N)  5(N) 7(N)
               \               /    \                          \
              6(R)           5(R)  7(R)                       8(R)
```

Si dupã operatia de eliminare a unui nod se apeleazã o functie de ajustare pentru mentinerea conditiilor de arbore RB.

Functiile de corectie dupa adãugare si stergere de noduri sunt relativ complicate deoarece sunt mai multe cazuri diferite care trebuie tratate.

O simplificare importantã a codului se obtine prin impunerea unei noi conditii la adãugarea de noduri (rosii): numai fiul dreapta al unui nod (negru) poate fi rosu. Dacã valoarea nodului nou este mai micã si el trebuie adãugat la stânga (dupã regula unui arbore de cãutare BST), atunci urmeazã o corectie prin rotatie.

Rezultatul acestei conditii suplimentare sunt arborii AA (Arne Andersson), la care culoarea nodului este însã înlocuitã cu un numãr de nivel (rang="rank"), care este altceva decât înãltimea nodului în arbore. Fiul rosu (fiu dreapta) are acelasi rang ca si pãrintele sãu, iar un fiu negru are rangul cu 1 mai mica ca pãrintele sãu. Orice nod frunzã are nivelul 1. Legãtura de la un nod la fiul dreapta (cu acelasi nivel) se mai numeste si legãturã orizontalã, iar legãtura la un fiu cu nivel mai mic se numeste si legãturã pe verticalã. Nu sunt permise: o legãturã orizontalã spre stânga (la un fiu stânga de acelasi nivel) si nici douã legãturi orizontale succesive la dreapta (fiu si nepot de culoare rosie).

Dupã adãugarea unui nou ca la orice arbore BST se fac (conditionat) douã operatii de corectie numite "skew" si "split", în aceastã ordine.

"skew" eliminã o legãturã la stânga pe orizontalã printr-o rotatie la dreapta; în exemplul urmãtor se adaugã nodului cu valoarea x un fiu cu valoarea y<x si apoi se roteste nodul cu valoarea x la dreapta:

```
     x (1)                          y(1)
    /        skew = rotR (x)          \
  y(1)                                 x(1)
```

"split" eliminã douã legãturi orizontale succesive pe dreapta, prin rotatie la stânga a nodului cu fiu si nepot pe dreapta; în exemplul urmãtor se adaugã un nod cu valoare z>y>x la un subarbore cu radacina x si fiul dreapta y si apoi se roteste x la stânga:

```
  x(1)                            y(2)
    \                            /    \
    y(1)     split = rotL (x)   x(1)    z(1)
      \
      z(1)
```

Functiile de corectie arbore AA sunt foarte simple:

```
void skew ( tnod * & r ) {
  if( r→st→niv == r→niv )   rotR(r);
}
void split( tnod * & r ) {
  if( r→dr→dr→niv == r→niv ) {
```

```
      rotL( r );   r→niv++;
  }
```

In functia urmãtoare corectia se face imediat dupã adãugarea la un subarbore si de aceea se pot folosi rotatiile scurte aplicabile numai unui nod rãdãcinã:

```
 // adaugare nod cu valoarea x la arbore AA cu radacina r
void add( tnod * & r , int x ) {
  if( r == NIL ) {               // daca (sub)arbore vid
     r = new tnod;               // aloca memorie pentru noul nod
     r→val=x;r→niv= 1;           // valoare si nivel nod nou
     r→st = r→dr= NIL;           // nodul nou este o frunza
     return;
  }
  if (x==r→val) return;          // daca x era deja in arbore, nimic
  if( x < r→val )                // daca x mai mic ca valoarea din r
     add( r→st, x );             // adauga x in subarborele stanga al lui r
  if( x > r→val )                // daca x este mai mare ca valoarea din r
     add( r→dr, x );             // adauga x la subarborele dreapta
  skew( r );                     // corectii dupa adaugare
  split( r );
}
```

Exemplu de evolutie arbore AA la adãugarea valorilor 9,8,7,6,5 (în paranteza nivelul nodului, cu 'R' si 'L' s-au notat rotatiile la dreapta si la stanga) :

```
9(1)     9 (1)          8 (1)       8(1)           7(1)             8(2)
        /        R(9)     \        /    \    R(8)    \      L(7)    /    \
      8 (1)               9(1)   7(1)  9(1)          8(1)         7(1)   9(1)
                                                       \
                                                       9(1)

   8(2)              8(2)        8(2)             8(2)           8(3)         6(2)
  /    \    R(7)    /    \      /    \    R(6)   /    \    L(5)  /   \   R(8)  /    \
 7(1)  9(1)       6(1)  9(1)  6(1)  9(1)        5(1)  9(1)     6(2)  9(1)     5(1)  8(2)
 /                  \          /  \              \             /  \               /   \
6(1)                7(1)    5(1) 7(1)            6(1)       5(1) 7(1)           7(1)   9(1)
                                                   \
                                                   7(1)
```

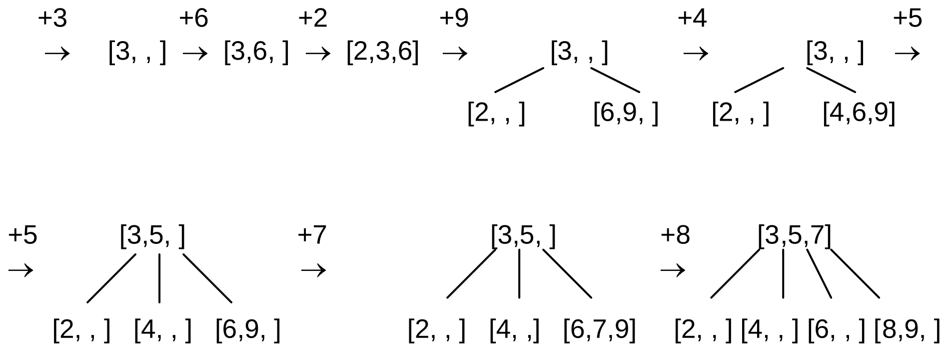
8.6 ARBORI 2-3-4

Arborii de cãutare multicãi, numiti si arbori B, sunt arbori ordonati si echilibrati cu urmãtoarele caracteristici:
- Un nod contine n valori si n+1 pointeri cãtre noduri fii (subarbori); n este cuprins între M/2 si M; numãrul maxim de pointeri pe nod M+1 determina ordinul arborelui B: arborii binari sunt arbori B de ordinul 2, arborii 2-3 sunt arbori B de ordinul 3, arborii 2-3-4 sunt arbori B de ordinul 4.
- Valorile dintr-un nod sunt ordonate crescãtor;
- Fiecare valoare dintr-un nod este mai mare decât valorile din subarborele stânga si mai micã decât valorile aflate în subarborele din dreapta sa.
- Valorile noi pot fi adãugate numai în noduri frunzã.
- Toate cãile au aceeasi lungime (toate frunzele se aflã pe acelasi nivel).
- Prin adãugarea unei noi valori la un nod plin, acesta este spart în alte douã noduri cu câte M/2 valori, iar valoarea medianã este trimisã pe nivelul superior;
- Arborele poate creste numai în sus, prin crearea unui nou nod rãdãcinã.

- La eliminarea unei valori dintr-un nod se pot contopi doua noduri vecine, de pe acelasi nivel, dacã suma valorilor din cele douã noduri este mai micã ca M.

Fie urmãtoarea secventã de valori adãugate la un arbore 2-3-4: 3, 6, 2, 9, 4, 8, 5, 7

Evolutia arborelui dupã fiecare valoare adãugatã este prezentatã mai jos:



La adãugarea unei noi valori într-un arbore B se cautã mai întâi nodul frunzã care ar trebui sã continã noua valoare, dupã care putem avea douã cazuri:
- dacã este loc în nodul gãsit, se adaugã noua valoare într-o pozitie eliberatã prin deplasarea altor valori la dreapta în nod, pentru mentinerea conditiei ca valorile dintr-un nod sã fie ordonate crescãtor.
- dacã nodul gãsit este plin atunci el este spart în douã: primele n/2 valori rãmân în nodul gãsit, ultimele n/2 valori se mutã într-un nod nou creat, iar valoarea medianã se ridicã în nodul pãrinte. La adãugarea în nodul pãrinte pot apãrea iar cele douã situatii si poate fi necesarã propagarea în sus a unor valori pânã la rãdãcinã; chiar si nodul rãdãcinã poate fi spart si atunci creste înãltimea arborelui.

Spargerea de noduri pline se poate face :
- de jos in sus (bottom-up), dupã gãsirea nodului frunzã plin;
- de sus în jos (top-down), pe mãsurã ce se cautã nodul frunzã care trebuie sã primeascã noua valoare: orice nod plin pe calea de cãutare este spart anticipat si astfel se evitã adãugarea la un nod plin.

Pentru exemplul anterior (cu valorile 3,5,7 in radacina) metoda de sus în jos constatã cã nodul rãdãcinã (de unde începe cãutarea) este plin si atunci îl sparge în trei: un nou nod rãdãcinã cu valoarea 5, un nou nod cu valoarea 7 (la dreapta) si vechiul nod cu valoarea 3 (la stanga).

Spargerea radacinii în acest caz nu era necesarã deoarece nici un nod frunzã nu este plin si ea nu s-ar fi produs dacã se revenea de jos în sus numai la gãsirea unui nod frunzã plin.

Arborii anteriori pot arãta diferit dupã cum se alege ca valoare medianã dintr-un numãr par 'n' de valori fie valoarea din pozitia n/2, fie din pozitia n/2+1 a secventei ordonate de valori .

Exemplu de definire a unui nod de arbore B (2-3-4) cu valori întregi:

```
#define M 3                    // nr maxim de valori in nod (arbore 2-3-4)
typedef struct bnod {
   int n;                      // Numar de chei dintr-un nod
   int val[M];                 // Valorile (cheile) din nod
   struct bnod* leg[M+1];      // Legaturi la noduri fii
} bnod;
```

Functie de afisare infixatã (în ordine crescãtoare) a valorilor dintr-un arbore B:

```
void infix (bnod* r) {
   if (r==0) return;
   for (int i=0;i<r→n;i++) {       // repeta ptr fiecare fiu
     infix (r→leg[i]);             // scrie valori mai mici ca r->val[i]
     printf("%d ",r→val[i]);   // scrie valoarea i
   }
   infix (r→leg[r→n]);         // scrie valori mai mari ca ultima din nodul r
}
```

Functie de afisare structurã arbore B (prefixat, cu indentare):

```
 void prefix (bnod *r, int ns) {
 if (r != 0) {
   printf("%*c",ns,' ');            // indentare cu ns spatii
   for (int i=0;i<r→n;i++)            // scrie toate valorile din nodul r
     printf("%d,",r→val[i]);
   printf("\n");
   for (int i=0;i<=r→n;i++)          // repeta pentru fiecare fiu al lui r
       prefix (r→leg[i], ns+3);  // cu deplasare spre dreapta a valorilor
 }
}
```

Spargerea unui nod p este mai simplã dacã se face top-down pentru cã nu trebuie sã tinã seama si de valoarea care urmeazã a fi adãugatã:

```
void split (bnod* p, int & med, bnod* & nou) {
  int m=M/2;                       // indice median nod plin
  med=p->val[m];                   // valoare care se duce in sus
  p->n=m;                          // in p raman m valori
  nou=make(M-m-1,&(p->val[m+1]),&(p->leg[m+1]));      // nod nou cu m+1,m+2,..M-1
  for (int i=m+1;i<M;i++) p->leg[i]=0;      // anulare legaturi din p
}
```

Dacã nodul frunzã gãsit p nu este plin atunci insertia unei noi valori x necesitã gãsirea pozitiei unde trebuie inserat x în vectorul de valori; în aceeasi pozitie din vectorul de adrese se va introduce legãtura de la x la subarborele cu valori mai mari ca x:

```
void ins (int x, bnod* legx, bnod * p) {      // legx= adresa subarbore cu valori mai mari ca x
  int i,j;
    // cauta pozitia i unde se introduce x
  i=0;
  while (i<p->n && x>p->val[i]) i++;
  for (j = p->n; j > i; j--) {          // deplasare dreapta intre i si n
       p->val[j] = p->val[j - 1];      // ptr a elibera pozitia i
       p->leg[j+1] = p->leg[j];
  }
  p->val[i] = x;                       // pune x in pozitia i
  p->leg[i+1] = legx;                  // adresa fiu cu valori mai mari ca v
  p->n++;                              // creste numarul de valori si fii din p
}
```

Cãutarea nodului frunzã care ar trebui sã continã o valoarea datã x se poate face iterativ sau recursiv, asemãnãtor cu cãutarea într-un arbore binar ordonar BST. Se va retine si adresa nodului pãrinte al nodului gãsit, necesarã la propagarea valorii mediane în sus. Exemplu de functie recursivã:

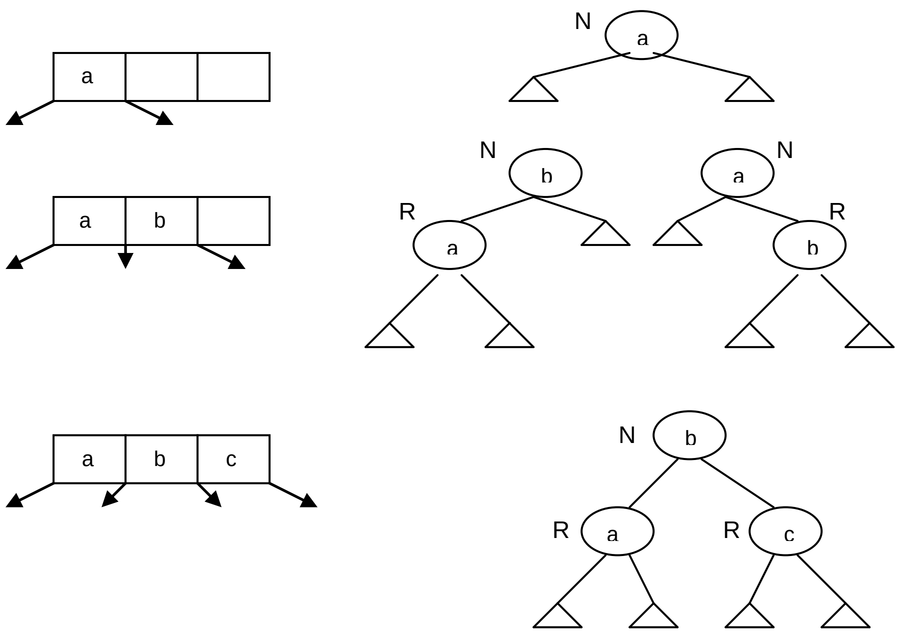
```
void findsplit (int x, bnod* & r, bnod* & pp) {
  bnod* p=r; bnod* nou, *rnou;
  int med;                     // val mediana dintr-un nod plin
  if (p->n==M) {               // daca nod plin
    split(p,med,nou);          // sparge nod cu creare nod nou
    if (pp!=0)                 // daca nu e nodul radacina
      ins(med,nou,pp);         // pune med in nodul parinte
    else {                     // daca p e nodul radacina
      rnou= new bnod;          // rnou va fi noua radacina
      rnou->val[0]=med;        // pune med in noua radacina
      rnou->leg[0]=r; rnou->leg[1]=nou;     // la stanga va fi r, la dreapta nou
```

```
      rnou->n=1;              // o singura valoare in noua radacina
      r=rnou; pp=rnou;        // modifica radacina r pentru noul arbore (mai inalt)
    }
    if (x > med) p=nou;       // p=nod curent, de unde continua cautarea
  }
     // cauta subarborele i al lui p care va contine pe x
    int i=0;
    while (i<p->n && x > p->val[i])   // determina pozitia lui x in p->val
      i++;
    if (x==p->val[i]) return ;          // daca x exista in p nu se mai adauga
    if (p->leg[0]==0 )                  // daca p e nod frunza
       ins (x,0,p);                     // atunci se introduce x in p si se iese
    else {                              // daca p nu e nod frunza
       pp=p; p=p->leg[i];               // cauta in fiul i al lui p
       findsplit(x,p,pp);               // apel recursiv ptr cautare in jos din p
    }
}
```

Pentru adãugarea unei valori x la un arbore B vom folosi o functie cu numai 2 argumente:

```
void add (int x, bnod* & p) {
  bnod* pp=0;                    // parinte nod radacina
  findsplit (x,p,pp);            // cauta, sparge si adauga x la nodul gasit
}
```

Se pot stabili echivalente între nodurile de arbori 2-4 si subarbori RB, respectiv între noduri 2-3 si subarbori AA. Echivalenta arbori 2-4 si arbori Red-Black:



Capitolul 9

STRUCTURI DE GRAF

9.1 GRAFURI CA STRUCTURI DE DATE

Operatiile cu grafuri pot fi considerate:
- Ca un capitol de matematicã (teoria grafurilor a fost dezvoltatã de matematicieni);
- Ca o sursã de algoritmi interesanti, care pot ilustra diferite clase de algoritmi, solutii alternative pentru o aceeasi problemã si metode de analizã a complexitãtii lor;
- Ca probleme de programare ce folosesc diverse structuri de date.

Aici ne intereseazã acest ultim aspect – probleme de grafuri ca studii de caz în folosirea unor structuri de date, cu implicatii asupra performantelor aplicatiilor, mai ales cã unele probleme practice cu grafuri au dimensiuni foarte mari.

Graful este un model abstract (matematic) pentru multe probleme reale, concrete, a cãror rezolvare necesitã folosirea unui calculator. In matematicã un graf este definit ca o pereche de douã multimi G = (V,M), unde V este multimea (nevidã) a vârfurilor (nodurilor), iar M este multimea muchiilor (arcelor). O muchie din M uneste o pereche de douã vârfuri din V si se noteazã cu (v,w).

De obicei nodurile unui graf se numeroteazã începând cu 1 si deci multimea V este o submultime a multimii numerelor naturale N.

Termenii "vârf" si "muchie" provin din analogia unui graf cu un poliedru si se folosesc mai ales pentru grafuri neorientate. termenii "nod" si "arc" se folosesc mai ales pentru grafuri orientate.

Intr-un graf orientat, numit si digraf, arcul (v,w) pleacã din nodul v si intrã în nodul w; el este diferit de arcul (w,v) care pleacã de la w la v. Intr-un graf neorientat poate exista o singurã muchie între douã vârfuri date, notatã (v,w) sau (w,v).

Deoarece în multimea M nu pot exista elemente identice înseamnã cã între douã noduri dintr-un graf orientat pot exista cel mult douã arce, iar între douã vârfuri ale un graf neorientat poate exista cel mult o muchie. Douã noduri între care existã un arc se numesc si noduri vecine sau adiacente. Intr-un graf orientat putem vorbi de succesorii si de predecesorii unui nod, respectiv de arce care ies si de arce care intrã într-un nod.

Un drum (o cale) într-un graf uneste o serie de noduri v[1], v[2],...v[n] printr-o secventã de arce (v[1],v[2]), (v[2],v[3]),...Intre douã noduri date poate sã nu existe un arc, dar sã existe o cale, ce trece prin alte noduri intermediare.

Un graf este conex dacã, pentru orice pereche de noduri (v,w) existã cel putin o cale de la v la w sau de la w la v.

Un digraf este tare conex (puternic conectat) dacã, pentru orice pereche de noduri (v,w) existã (cel putin) o cale de la v la w si (cel putin) o cale de la w la v. Un exemplu de graf tare conex este un graf care contine un ciclu care trece prin toate nodurile: (1,2), (2,3), (3,4), (4,1).

O componentã conexã a unui graf (V,M) este un subgraf conex (V',M') unde V' este o submultime a lui V, iar M' este o submultime a lui M. Impãrtirea unui graf neorientat în componente conexe este unicã, 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} si {3} sau {3,2,4} si {1}.

Un ciclu în graf (un circuit) este o cale care porneste si se terminã în acelasi nod. Un ciclu hamiltonian este un ciclu complet, care uneste toate nodurile dintr-un graf.

Un graf neorientat conex este ciclic dacã numãrul de muchii este mai mare sau egal cu numãrul de vârfuri.

Un arbore liber este un graf conex fãrã cicluri si poate fi neorientat sau orientat.

Putem deosebi trei categorii de grafuri:

a) Grafuri de relatie (simple), în care se modeleazã doar relatiile dintre entitãti, iar arcele nu au alte atribute.

b) Grafuri cu costuri (retele), în care fiecare arc are un cost asociat (o distantã geometricã, un timp de parcurgere, un cost exprimat în bani). Intre costurile arcelor nu existã nici o relatie.
c) Retele de transport, în care fluxul (debitul) prin fiecare arc (tronson de retea) este corelat cu fluxul prin arcele care vin sau pleacã din acelasi nod.

Anumite probleme reale sugereazã în mod natural modelarea lor prin grafuri: probleme asociate unor retele de comunicatie, unor retele de transport de persoane sau de mãrfuri, retele de alimentare cu apã, cu energie electricã sau termicã, s.a.

Alteori asocierea obiectelor din lumea realã cu nodurile si arcele unui graf este mai putin evidentã. Arcele pot corespund unor relatii dintre persoane (persoana x cunoaste persoana y) sau dintre obiecte (piesa x contine piesa y) sau unor relatii de conditionare (operatia x trebuie precedatã de operatia y).

Un graf poate fi privit si ca un tip de date abstract, care permite orice relatii între componentele structurii. Operatiile uzuale asociate tipului "graf" sunt:
- Initializare graf cu numãr dat de noduri: initG (Graph & g,int n);
- Adãugare muchie (arc) la un graf: addArc (Graph & g, int x, int y);
- Verificã existenta 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 multi algoritmi pe grafuri necesitã parcurgerea vecinilor (succesorilor) unui nod dat, care poate folosi functia "arc" într-un ciclu repetat pentru toti vecinii posibili (deci pentru toate nodurile din graf). Pentru grafuri reprezentate prin liste de vecini este suficientã parcurgerea listei de vecini a unui nod, mult mai micã decât numãrul de noduri din graf (egalã cu numãrul de arce asociate acelui nod).

De aceea se considerã uneori ca operatii elementare cu grafuri urmãtoarele:
- Pozitionare pe primul succesor al unui nod dat ("firstSucc");
- Pozitionare pe urmãtorul succesor al unui nod dat ("nextSucc").

Exemplu de afisare 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 si "retea") apar câteva mici diferente la functiile "arc" (costul unui arc) si "addArc" (mai are un argument care este costul arcului) :

```
typedef struct {                // tip retea (graf cu costuri)
  int n,m;        // nr de noduri si 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 directã a unui graf este printr-o matrice de adiacente (de vecinãtãti), pentru grafuri de relatie respectiv printr-o matrice de costuri, pentru retele. Avantajele reprezentãrii unui graf printr-o matrice sunt:
- Simplitatea si claritatea programelor.
- Aceeasi reprezentare pentru grafuri orientate si neorientate, cu sau fãrã 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 douã noduri date (nu necesitã cãutare, deci nu depinde de dimensiunea grafului).

Reprezentarea matricialã este preferatã în determinarea drumurilor dintre oricare douã vârfuri (tot sub formã de matrice), în determinarea drumurilor minime dintre oricare douã 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 naturalã pentru o colectie de puncte cu atribute diferite: un labirint (puncte accesibile si puncte inaccesibile), o suprafatã cu puncte de diferite înãltimi, o imagine formatã din puncte albe si negre (sau colorate diferit), s.a.

Dezavantajul matricei de adiacente apare atunci când numãrul de noduri din graf este mult mai mare ca numãrul de arce, iar matricea este rarã (cu peste jumãtate din elemente nule). In astfel de cazuri se preferã reprezentarea prin liste de adiacente.

Matricea de adiacente "a" este o matrice pãtraticã cu valori întregi , având numãrul de linii si de coloane egal cu numãrul de noduri din graf. Elementele a[i][j] sunt:

1 (true) dacã existã arc de la i la j   sau   0 (false) dacã nu existã arc de la i la j

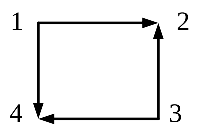
Exemplu de definire a unui tip graf printr-o matrice de adiacente alocatã dinamic:

```
  // cu matrice alocata dinamic
typedef struct {
    int n,m ;                  //  n=nr de noduri, m=nr de arce
    int ** a;                  // adresa matrice de adiacente
} Graf ;
```

In general numãrul de noduri dintr-un graf poate fi cunoscut de program încã de la început si matricea de adiacente poate fi alocatã 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 existã arce de la un nod la el însusi si deci a[i][i]=0.

Exemple de functii cu grafuri în cazul utilizãrii 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 sã modifice si dimensiunile matricei, dar vom elimina doar arcele ce pleacã si 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);
  }
```

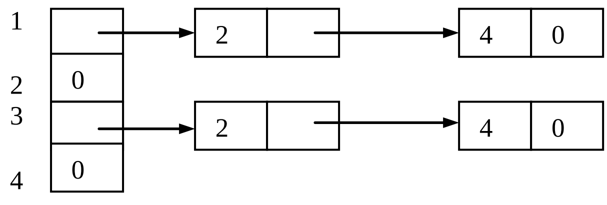
Pentru un graf cu costuri vom înlocui functia “arc” cu o functie “carc” care are ca rezultat costul unui arc, iar acolo unde nu existã arc vom pune o valoare foarte mare (mai mare ca orice cost din graf), care corespunde unui cost infinit.

```
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 micã decât jumãtate din cea mai mare valoare pentru tipul de date folosit la costul arcelor, deoarece altfel poate apare depãsire 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 împãrtitã în mai multe subliste, câte una pentru fiecare nod din graf. Listele de noduri vecine pot avea lungimi foarte diferite si de aceea se preferã implementarea lor prin liste înlãntuite. Reunirea listelor de succesori se poate face de obicei într-un vector, deoarece permite accesul direct la un nod pe baza numãrului sãu (fãrã cãutare). Figura urmãtoare aratã 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 listã de adiacente nu este importantã si de aceea putem adãuga mereu la începutul listei de noduri vecine.

Exemple de operatii 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 fiecãrui vârf asigurã cel mai bun timp de explorare a grafurilor (timp proprtional cu suma dintre numãrul de vârfuri si numãrul de muchii din graf), iar explorarea apare ca operatie în mai multi algoritmi pe grafuri.

Pentru un graf neorientat fiecare muchie (x,y) este memoratã de douã ori: y în lista de vecini a lui x si x în lista de vecini a lui y.

Pentru un graf orientat listele de adiacente sunt de obicei liste de succesori, dar pentru unele aplicatii intereseazã predecesorii unui nod (de ex. în sortarea topologicã). Lipsa de simetrie poate fi un dezavantaj al listelor de adiacente pentru reprezentarea grafurilor orientate.

Pe lângã reprezentãrile principale ale structurilor de graf (matrice si liste de adiacente) se mai folosesc uneori si alte reprezentãri:

- O listã de arce (de perechi de noduri) este utilã în anumiti algoritmi (cum este algoritmul lui Kruskal), dar mãreste timpul de cãutare: timpul de executie al functiei "arc" creste liniar cu numãrul de arce din graf.
- O matrice de biti este o reprezentare mai compactã a unor grafuri de relatie cu un numãr foarte mare de noduri.
- Un vector de pointeri la vectori (cu vectori în locul listelor de adiacente) necesitã mai putinã memorie si este potrivit pentru un graf static, care nu se mai modificã.
- Pentru grafuri planare care reprezintã puncte si distante pe o hartã poate fi preferabilã o reprezentare geometricã, 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 existã un nod special rãdãcinã. Intr-un arbore fiecare vârf are un singur pãrinte (predecesor), deci am putea reprezenta arborele printr-un vector de noduri pãrinte.

Rezultatul mai multor algoritmi este un arbore liber si 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 singurã listã - lista arcelor, iar numãrul de noduri este valoarea maximã 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 listã de structuri, sau doi vectori de noduri:

Figura urmãtoare aratã un arbore liber si lista lui de arce.

```
1 o        o 5
   \      /
3  o----o  4
   /      \
2 o        o 6
```

	1	2	3	4	5
x	1	2	3	4	4
y	3	3	4	5	6

Pentru arbori liberi aceastã reprezentare poate fi simplificatã si mai mult, dacã vom impune ca pozitia în vector sã fie egalã cu unul dintre noduri. Vom folosi deci un singur vector P, în care P[k] este perechea (predecesorul) nodului k. Este posibil întotdeauna sã notãm arcele din arbore astfel încât fiecare nod sã aibã un singur predecesor (sau un singur succesor).

Pentru arborele anterior vectorul P va fi:

	1	2	3	4	5	6
P		3	1	3	4	4

Lista arcelor (k, P[k]) este deci: (2,3),(3,1),(4,3),(5,4),(6,4).

Am considerat cã nodul 1 nu are nici un predecesor, dar putem sã considerãm cã nodul ultim nu are nici un predecesor:

	1	2	3	4	5
P	3	3	4	6	4

Un astfel de vector este chiar vectorul solutie într-o abordare backtracking a unor probleme de grafuri.

9.3 METODE DE EXPLORARE A GRAFURILOR

Explorarea unui graf înseamnã vizitarea sistematicã a tuturor nodurilor din graf, folosind arcele existente, astfel încât sã se treacã o singurã datã prin fiecare nod.

Rezultatul explorãrii unui graf este o colectie de arbori de explorare , numitã si "pãdure" de acoperire. Dacã se pot atinge toate nodurile unui graf pornind dintr-un singur nod, atunci rezultã un singur arbore de acoperire. Explorarea unui graf neorientat conex conduce la un singur arbore, indiferent care este nodul de pornire.

Rezultatul explorãrii 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 functie care primeste un nod de start si încearcã sã atingã cât mai multe noduri din graf. Aceastã functie poate fi apelatã în mod repetat, pentru fiecare nod din graf considerat ca nod de start. Astfel se asigurã vizitarea tuturor nodurilor pentru orice graf. Fiecare apel genereazã un arbore de acoperire a unei submultimi de noduri.

Explorarea unui graf poate fi vãzutã si ca o metodã de enumerare a tuturor nodurilor unui graf, sau ca o metodã de cãutare a unui drum cãtre un nod dat din graf.

Transformarea unui graf (structurã bidimensionalã) într-un vector (structurã liniarã) se poate face în multe feluri, deoarece fiecare nod are mai multi succesori si trebuie sã alegem numai unul singur pentru continuarea explorãrii.

Algoritmii de explorare dintr-un nod dat pot folosi douã metode:
- Explorare în adâncime (DFS = Depth First Search)
- Explorare în lãrgime (BFS = Breadth First Search)

Explorarea în adâncime foloseste, la fiecare nod, un singur arc (cãtre nodul cu numãr minim) si astfel se pãtrunde cât mai repede în adâncimea grafului. Dacã rãmân noduri nevizitate, se revine treptat la nodurile deja vizitate pentru a lua în considerare si alte arce, ignorate în prima fazã. Explorarea DFS din nodul 3 a grafului anterior produce secventa de noduri 3, 2, 1, 4 iar arborele de acoperire este format din arcele 3-2, 2-1 si 1-4.

Vizitarea DFS a unui graf aciclic corespunde vizitãrii prefixate de la arbori binari.

Explorarea în lãrgime foloseste, la fiecare nod, toate arcele care pleacã din nodul respectiv si dupã aceea trece la alte noduri (la succesorii nodurilor vizitate). In felul acesta se exploreazã mai întâi nodurile adiacente, din "lãtimea" grafului si apoi se coboarã mai adânc în graf. Explorarea BF din nodul 3 a grafului anterior conduce la secventa de noduri 3,2,4,1 si la arborele de acoperire 3-2, 3-4, 2-1 , dacã se folosesc succesorii în ordinea crescãtoare a numerelor lor.

Este posibil ca pentru grafuri diferite sã rezulte o aceeasi secventã de noduri, dar lista de arce este unicã pentru fiecare graf (dacã se aplicã acelasi algoritm). De asemenea este posibil ca pentru anumite grafuri sã rezulte acelasi arbore de acoperire atât la explorarea DF cât si 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 stivã de noduri. Ambele variante trebuie sã tinã evidenta nodurilor vizitate pânã la un moment dat, pentru a evita vizitarea repetatã a unor noduri. Cea mai simplã implementare a multimii de noduri vizitate este un vector "vãzut", initializat cu zerouri si actualizat dupã vizitarea fiecãrui nod x (vazut[x]=1).

Exemplul urmãtor contine o functie recursivã de explorare DF dintr-un nod dat v si o functie 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 modificã putin functia, dar ea nu va afisa nimic dacã nu se poate atinge nici un alt nod din nodul de plecare.

Un algoritm DFS nerecursiv trebuie sã foloseascã o stivã pentru a memora succesorii (vecinii) neprelucrati ai fiecãrui nod vizitat, astfel ca sã putem reveni ulterior la ei:

```
       pune nodul de plecare în stivã
       repetã cât timp stiva nu e goalã
           scoate din stivã în x
           afisare si marcare  x
           pune în stivã orice succesor nevizitat y al lui x
```

Pentru ca functia DFS nerecursivã sã producã aceleasi rezultate ca si functia DFS recursivã, succesorii unui nod sunt pusi în stivã în ordinea descrescãtoare a numerelor lor (extragerea lor din stivã si afisarea lor se va face în ordine inversã).

```
void dfs (Graf g,int v, int vazut[]) {
  int  x,y;   Stack s;            // s este o stiva de intregi
  initSt (s);                     // initializare stivã
  push (s,v);                     // pune nodul v pe stiva
  while (!emptySt(s)) {
     x=pop (s);                   // scoate din stivã 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 stivã
        }
   }
}
```

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 lãtime afiseazã si memoreazã pe rând succesorii fiecãrui nod. Ordinea de prelucrare a nodurilor memorate este aceeasi cu ordinea de introducere în listã, deci lista este de tip "coadã". Algoritmul BFS este asemãnãtor algoritmului DFS nerecursiv, diferenta apare numai la tipul listei folosite pentru memorarea temporarã a succesorilor fiecãrui nod: stivã la DFS si coadã la BFS

```
  // explorare în lãtime 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 coadã
 while (! emptyQ(q)) {
   x=delQ (q);                          // scoate din coadã 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 douã vârfuri este drumul care foloseste cel mai mic numãr de muchii. Drumurile minime de la un vârf v la toate celelalte noduri pot fi gãsite prin explorare în lãrgime din nodul v, cu actualizare distante fatã de v, la fiecare coborâre cu un nivel în graf. Vom folosi un vector d cu d[y]=distanta vârfului y fatã de "rãdãcina" v si un vector p, cu p[y]=numãr 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 coadã în x
   for (y=1;y <=g.n;y++)
     if ( arc(g,x,y) && vazut[y]==0) {   // test dacă arc între x si 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 → p[x] → p[p[x]] →… → v

9.4 SORTARE TOPOLOGICÃ

Problema sortãrii topologice poate fi formulatã astfel: între elementele unei multimi A existã relatii de conditionare (de precedentã) de forma a[i] << a[j], exprimate în cuvinte astfel: a[i] precede (conditioneazã) pe a[j], sau a[j] este conditionat de a[i]. Se mai spune cã a[i] este un predecesor al lui a[j] sau cã a[j] este un succesor al lui a[i]. Un element poate avea oricâti succesori si predecesori. Multimea A supusã unor relatii de precedentã poate fi vazutã ca un graf orientat, având ca noduri elementele a[i] ; un arc de la a[i] la a[j] aratã cã a[i] precede pe a[j].

Exemplu : A = { 1,2,3,4,5 }

2 << 1   1 << 3   2 << 3   2 << 4   4 << 3   3 << 5   4 << 5

Scopul sortãrii topologice este ordonarea (afisarea) elementelor multimii A într-o succesiune liniarã astfel încât fiecare element sã fie precedat în aceastã succesiune de elementele care îl conditioneazã.

Elementele multimii A pot fi privite ca noduri dintr-un graf orientat, iar relatiile de conditionare ca arce în acest graf. Sortarea topologicã a nodurilor unui graf orientat nu este posibilã dacã graful contine cel putin un ciclu. Dacã nu existã nici un element fãrã conditionãri atunci sortarea nici nu poate începe. Uneori este posibilã numai o sortare topologicã partialã, pentru o parte din noduri.

Pentru exemplul dat existã douã secvente posibile care satisfac conditiile de precedentã :

2, 1, 4, 3, 5 si 2, 4, 1, 3, 5

Determinarea unei solutii de ordonare topologicã se poate face în câteva moduri:

a) Incepând cu elementele fãrã predecesori (neconditionate) si continuând cu elementele care depind de acestea (nodul 2 este un astfel de element în exemplul dat);

b) Incepând cu elementele fãrã succesori (finale) si mergând cãtre predecesori, din aproape în aproape (nodul 5 în exemplu).

c) Algoritmul de explorare în adâncime a unui graf orientat, completat cu afisarea nodului din care începe explorarea, dupã ce s-au explorat toate celelalte noduri.

Aceste metode pot folosi diferite structuri de date pentru reprezentarea relatiilor dintre elemente; în cazul (a) trebuie sã putem gãsi usor predecesorii unui element, iar în cazul (b) trebuie sã putem gãsi usor succesorii unui element,

Algoritmul de sortare topologicã cu liste de predecesori este:

```
repetã
  cautã un nod nemarcat si fãrã predecesori
  dacã s-a gãsit atunci
    afiseazã nod si marcheazã nod
    sterge nod marcat din graf
pânã când nu mai sunt noduri fãrã predecesori
dacã rãmân noduri nemarcate atunci
   nu este posibilã sortarea topologicã
```

Pentru exemplul dat evolutia listelor de predecesori este urmãtoarea:

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 urmãtor ilustreazã 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 topologicã cu liste de succesori este:

```
repetã
   cautã un nod fãrã succesori
   pune nod gãsit în stivã si marcheazã ca sortat
   eliminã nod marcat din graf
pânã când nu mai existã noduri fãrã succesori
```

```
dacã nu mai sunt noduri nemarcate atunci
  repetã
     scoate nod din stivã si afisare nod
  pânã când stiva goalã
```

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 stivã se afiseazã: 2, 4, 1, 3, 5

9.5 Aplicatii ale explorãrii în adâncime

Explorarea în adâncime stã la baza altor algoritmi cu grafuri, cum ar fi: determinarea existentei ciclurilor într-un graf, gãsirea componentelor puternic conectate dintr-un graf, sortare topologicã, determinare puncte de articulare s.a.

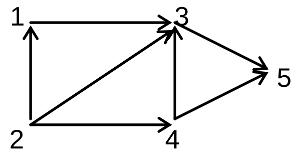
Determinarea componentelor conexe ale unui graf se poate face prin repetarea explorãrii DF din fiecare nod nevizitat în explorãrile anterioare. Un apel al functiei "dfs" afiseazã o componentã conexã. Pentru grafuri neorientate existã un algoritm mai performant de aflare a componentelor conexe, care foloseste tipul abstract de date "colectie de multimi disjuncte".

Algoritmul de sortare topologicã derivat din explorarea DF se bazeazã pe faptul cã explorarea în adâncime viziteazã toti succesorii unui nod. Explorarea DF va fi repetatã pânã când se viziteazã toate nodurile din graf. Functia "ts" este derivatã din functia "dfs", în care s-a înlocuit afisarea cu punerea într-o stivã a nodului cu care a început explorarea, dupã ce s-au memorat în stivã succesorii sãi. In final se scoate din stivã si se afiseazã tot ce a pus functia "ts".

Programul urmãtor realizeazã sortarea topologicã ca o variantã de explorare în adâncime a unui graf g si foloseste o stivã s pentru memorarea nodurilor.

```
Stack s;          // stiva folosita in doua functii
    // sortare topologica dintr-un nod v
void ts (Graf g,int v) {
  vazut[v]=1;
  for (int w=1;w<=g.n;w++)
    if ( arc (g,v,w) && ! vazut[w])
      ts(g,w);
  push (s,v);
}
  // sortare topologica graf
int 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);
  while( ! emptySt (s)) {     // scoate din stiva si afiseaza
     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:

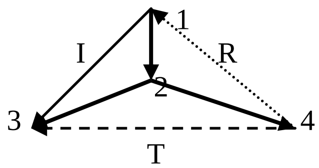
- Arce de arbore, componente ale arborilor de explorare în adâncime (de la un nod în curs de vizitare la un nod nevizitat încã).
- Arce de înaintare, la un succesor (la un nod cu numãr de vizitare mai mare).
- Arce de revenire, la un predecesor (la un nod cu numãr de vizitare mai mic).
- Arce de traversare, 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).

Dacã existã cel putin un arc de revenire (înapoi) la explorarea DF a unui graf orientat atunci graful contine cel putin un ciclu, iar un graf orientat fãrã arce de revenire este aciclic.

Pentru a diferentia arcele de revenire de arcele de traversare se memoreazã într-un vector P nodurile din arborele de explorare DF; un arc de revenire merge cãtre un nod din P, dar un arc de traversare nu are ca destinatie un nod din P.

```
    // clasificare arce la explorare în adâncime dintr-un nod dat v
void dfs (int v, int t[ ]) {         // t[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
     t[k]='A'; p[w]=v;       // atunci v-w este arc de arbore
     dfs (w,t);                // continua explorarea din w
   }
   else                         // daca w este deja vizitat
     if ( nv[v] < nv[w])      // daca w vizitat dupa v
       t[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
        t[k]='R';              // atunci v-w este arc inapoi (de revenire)
      else                     //  daca w nu precede pe v in arborele DFS
        t[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;
}
```

Functia de explorare DF poate fi completatã cu numerotarea nodurilor atât la primul contact cu nodul, cât si la ultimul contact (la iesirea din functia dfs). Functia dfs care urmeazã foloseste variabila externã ‘t’, initializatã cu zero în programul principal si incrementatã la fiecare intrare în functia "dfs". Vectorul t1 este actualizat la intrarea în functia dfs, iar vectorul t2 la iesirea 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 si cu arcele (1,3), (1,4), (2,1), (2,3), (3,4), (4,2) arborele dfs este secventa 1→3→4→2 , iar vectorii t1 si t2 vor contine urmãtoarele valori dupã 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)

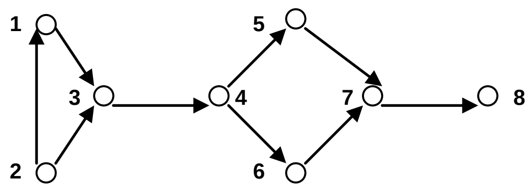
Se observã cã ultimul nod vizitat (2) este si primul pãrãsit, dupã care este pãrãsit nodul vizitat anterior;numerele t1(k) si t2(k) pot fi privite ca paranteze în jurul nodului k, iar structura de paranteze a grafului la vizitare dfs este :

(1 (3 (4 (2))))

O componentã puternic conectatã (tare conexã) dintr-un digraf este o submultime maximalã a nodurilor astfel încât existã o cale între oricare douã noduri din cpc.

Pentru determinarea componentelor puternic conectate (cpc) dintr-un graf orientat vom folosi urmãtorul graf orientat ca exemplu:

1→3, 3→2, 2→1, 3→4, 4→5, 5→7, 7→6, 6→4, 7→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 obtinem o altã multime de noduri, din care se poate ajunge în v. Intersectia celor douã multimi reprezintã componenta tare conexã care contine nodul v. Dupã eliminarea nodurilor acestei componente din graf se repetã operatia pentru nodurile rãmase, pânã când nu mai sunt noduri în graf.

Pentru graful anterior vizitarea DFS din 1 produce multimea {1,3,2,4,5,7,6,8} iar vizitarea grafului inversat din 1 produce multimea {1,2,3}. Intersectia celor douã multimi {1,2,3} reprezintã componenta tare conexã care contine nodul 1. Dupã eliminarea nodurilor 1,2 si 3, vizitarea grafului rãmas din nodul 4 produce multimea {4,5,7,6,8}, iar vizitarea din 4 a grafului cu arce inversate produce multimea {4,6,7,5}, deci componenta tare conexã care-l contine pe 4 este {4,5,6,7}. Ultima componentã cpc contine doar nodul 8.

Este posibilã îmbunãtãtirea acestui algoritm pe baza observatiei cã s-ar putea determina toate componentele cpc la o singurã vizitare a grafului inversat, folosind ca puncte de plecare nodurile în ordine inversã vizitãrii DFS a grafului initial. Algoritmul foloseste vectorul t2 cu timpii de pãrãsire ai fiecãrui nod si repetã vizitarea grafului inversat din nodurile considerate în ordinea inversã a numerelor t2.

Pentru graful anterior vectorii t1 si 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

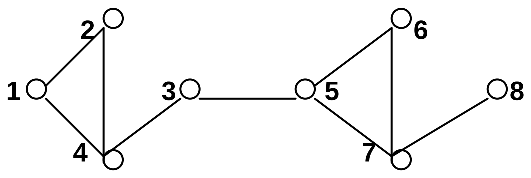
Vizitarea grafului inversat se va face din 1 (t2=16) cu rezultat {1,2,3}, apoi din 4 (t2=14) cu rezultat {4,6,7,5} si din 8 (singurul nod rãmas) cu rezultat {8}.

Graful cu arce inversate se obtine prin transpunerea matricei initiale.

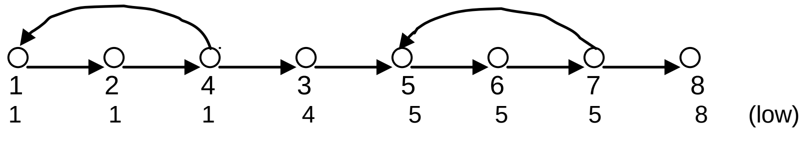
Pentru grafuri neorientate ce reprezintã retele de comunicatii sunt importante problemele de conectivitate. Un punct de articulare (un punct critic) dintr-un graf conex este un vârf a cãrui eliminare (împreunã cu muchiile asciate) face ca graful sã nu mai fie conex. O "punte" (o muchie criticã) este o muchie a cãrei eliminare face ca graful rãmas sã nu mai fie conex. O componentã biconexã este o submultime maximalã de muchii astfel cã oricare douã muchii se aflã 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
Muchii critice: 3-4, 3-5
Componente biconexe: (1,2,4), (5,6,7)



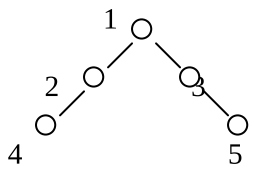
Un algoritm eficient pentru determinarea punctelor critice dintr-un graf conex foloseste vizitarea în adâncime dintr-un vârf rãdãcinã oarecare. Arborele de vizitare al unui graf neorientat contine numai douã tipuri de arce: de explorare (de arbore) si arce de revenire (înapoi). Pentru graful anterior arborele produs de vizitarea DFS din vârful 1 contine arcele 1→2, 2→4, 4→3, 3→5, 5→6, 6→7, 7→8, iar arcele înapoi sunt 4→1 si 7→5.



Un vârf terminal (o frunzã) 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. Rãdãcina arborelui DFS poate fi punct de articulare numai dacã are cel putin doi fii în arborele DFS, cãci eliminarea ei ar întrerupe legãtura dintre fiii sãi. Deci 1 nu este punct de articulare. Un vârf interior v din arborele DFS nu este punct de articulare dacã existã în graf o muchie înapoi de la un vârf u urmãtor lui v în arbore la un vârf w anterior lui v în arbore, pentru cã 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 t1 mare la un vârf cu t1 mic. Un nod u urmãtor lui v în arbore are t1[u] > t1[v], adicã u este vizitat dupã v. De exemplu, t1[1]=1, t1[4]=3 , deci 4 este un descendent al lui 1 în arbore.

Pentru graful anterior vârful 2 nu este punct critic deoarece existã 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 exitã muchia (7,5) de la fiul 7 la pãrintele 5. Vârful 3 este punct de articulare deoarece nu existã o muchie de la fiul 5 la pãrintele sãu 4 si deci eliminarea sa ar întrerupe accesul cãtre vârful 5 si urmãtoarele. La fel 4,5 si 7 sunt puncte critice deoarece nu existã muchie înapoi de la un fiu la un pãrinte.

Un alt exemplu este graful cu 5 vârfuri si muchiile 1-2, 1-3, 2-4, 3-5:



Arborele de explorare dfs din 1 este acelasi cu graful; vârfurile 4 si 5 sunt frunze în arbore, iar 1 este rãdãcinã cu doi fii. Punctele de articulare sunt 1, 2, 3.

Dacã se adaugã muchiile 1-4 si 1-5 la graful anterior atunci 2 si 3 nu mai sunt puncte critice (existã arce înapoi de la succesori la predecesori).

Implementarea algoritmului foloseste 3 vectori de noduri:
d[v] este momentul vizitãrii (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, cãtre care urcã un arc înapoi de la un succesor al lui v.

Vectorul "low" se determinã la vizitarea dfs, iar functia ce determinã punctele de articulare verificã 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 gãsi drumul de cost minim dintre douã noduri oarecare i si j dintr-un graf orientat sau neorientat, cu costuri pozitive.

S-a arãtat cã această problemã nu poate fi rezolvatã mai eficient decât problema gãsirii drumurilor minime dintre nodul i si toate celelalte noduri din graf. De obicei se considerã ca nod sursã i chiar nodul 1 si se determinã lungimile drumurilor minime d[2],d[3],...,d[n] pânã la nodurile 2,3,...n. Pentru memorarea nodurilor de pe un drum minim se foloseste 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 reprezintã acest arbore de cãi în graf).

Cel mai eficient algoritm cunoscut pentru problema drumurilor optime cu o singurã sursã este algoritmul lui Dijkstra, care poate fi descris în mai multe moduri: ca algoritm de tip “greedy” cu o coadã cu prioritãti, ca algoritm ce foloseste operatia de “relaxare” (comunã si altor algoritmi), ca algoritm cu multimi de vârfuri sau ca algoritm cu vectori. Diferentele de prezentare provin din structurile de date utilizate.

In varianta urmãtoare se foloseste un vector D astfel cã d[i] este distanta minimã de la 1 la i, dintre drumurile care trec prin noduri deja selectate. O variabilã S de tip multime memoreazã numerele nodurilor cu distantã minimã fatã de nodul 1, gãsite pânã la un moment dat. Initial S={1} si d[i]=cost[1][i], adicã se considerã arcul direct de la 1 la i ca drum minim între 1 si i. Pe mãsurã ce algoritmul evolueazã, se actualizeazã D si S.

```
S ={1}     // S =multime noduri ptr care s-a determinat dist. minima fata de 1
repetã cât timp S contine mai putin de n noduri {
  gaseste muchia (x,y) cu x în S si y nu în S care face minim d[x]+cost(x,y)
   adauga y la S
   d[y] = d[x] + cost(x,y)
}
```

La fiecare pas din algoritmul Dijkstra:

- Se gãseste dintre nodurile j care nu apartin lui S acel nod "jmin" care are distanta minimã fatã de nodurile din S;
- Se adaugã nodul "jmin" la multimea S;
- Se recalculeazã distantele de la nodul 1 la nodurile care nu fac parte din S, pentru cã distantele la nodurile din S rãmân neschimbate;
- Se retine în p[j] numãrul 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 considerãm un graf orientat cu urmãtoarele 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 intre 1 si 3 si 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 cã într-un drum minim fiecare drum partial este minim; astfel în drumul 1-4-2-3, drumurile partiale 1-4-2 si 1-4 sunt si ele minime.

Evolutia vectorilor D si 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 arãta în final astfel:

p[2]	p[3]	p[4]	p[5]
4	2	1	4

Exemplu de functie 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
    // cautã 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]) {   // daca 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
      }
  }
}
```

In programul principal se apeleazã repetat functia "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 cã valoarea constantei MARE, folositã pentru a marca în matricea de costuri absenta unui arc, nu poate fi mai mare ca jumãtate din valoarea maximã pentru tipul întreg , deoarece la însumarea costurilor a douã drumuri se poate depãsi cel mai mare întreg (se pot folosi pentru costuri si numere reale foarte mari).

Metoda de ajustare treptatã a lungimii drumurilor din vectorul D este o metodã de "relaxare", folositã si în alti algoritmi pentru drumuri minime sau maxime: algoritmul Bellmann-Ford pentru drumuri minime cu o singurã sursã în grafuri cu costuri negative, algoritmul Floyd pentru drumuri minime între oricare pereche de noduri s.a.

Prin relaxarea unei muchii (v,w) se întelege ajustarea costului anterior al drumului cãtre nodul w tinându-se seama si de costul muchiei v-w, deci considerând si un drum cãtre w care trece prin v. Un pas de relaxare pentru drumul minim cãtre nodul w se poate exprima printr-o secventã de forma urmãtoare:

```
      // d[w] = cost drum minim la w fara a folosi si v
    if (d[w] > d[v] + carc(g,v,w) ) {    // daca 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*n) si poate fi redusã la O(m*lg(n)) prin folosirea unei cozi ordonate (min-heap) cu operatie de diminuare a cheii.

In coadã vom pune distanta cunoscutã la un moment dat de la 1 pânã la un alt nod: initial sunt costurile arcelor directe, dupã care se pun costurile drumurilor de la 1 prin nodurile determinate ca fiind cele mai apropiate de 1. Ideea cozii cu diminuarea prioritãtii este cã în coadã vor fi mereu aceleasi elemente (noduri), dar cu prioritãti (distante) modificate de la un pas la altul. In loc sã adãugãm la coadã distante tot mai mici (la aceleasi noduri) vom modifica numai costul drumului deja memorat în coadã. Vom exemplifica cu graful orientat urmãtor: 1-2=4, 1-3=1, 1-4=7, 2-4=1, 3-2=2, 3-4=5, 4-1=7

In coadã vom pune nodul destinatie si distanta de la 1 la acel nod.

In cazul cozii cu prioritãti numai cu operatii de adãugare si eliminare vom avea urmãtoarea evolutie a cozii cu distante 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
```

In cazul cozii cu diminuarea costului drumurilor (prioritãtii) coada va evolua astfel:

pas	coada ordonata	nod proxim (fata de 1)	distanta de la 1
initial	(3,1), (2,4),(4,7)	3	1
prin 3	(2,3), (4,6)	2	3
prin 3 si 2	(4,4)	4	4

Functia urmãtoare foloseste operatia de diminuare a prioritãtii într-un min-heap si actualizează în coadã distantele recalculate :

```
void dijkstra (Net g, int n[], int d[]) {
   // d[k] = distanta minima de la 1 la nodul n[k]
   // pq= Coada cu distante 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 distanta in variabila a
     addpq( &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;                          // distanta 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 distanta 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 acoperã toate nodurile grafului. Un arc "acoperã" nodurile pe care le uneste. Un graf conex are mai multi arbori de acoperire, numãrul acestor arbori fiind cu atât mai mare cu cât numãrul 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 gãsi 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 urmãtoarele 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 listã ordonatã de arce (dupã costuri) si o colectie de multimi disjuncte pentru a verifica dacã urmãtorul arc scos din listã poate fi sau nu adãugat arcelor deja selectate (dacã nu formeazã un ciclu cu arcele din MST).

Algoritmul lui Prim seamãnã cu algoritmul Dijkstra pentru drumuri minime si foloseste o coadã cu prioritãti de arce care leagã vârfuri din MST cu alte vârfuri (coada se modificã pe mãsurã ce algoritmul evolueazã).

Algoritmul lui Prim se bazeazã pe observatia urmãtoare: 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 "tãieturã" în graf: se taie toate arcele care leagã un nod k de restul nodurilor din graf si se determinã arcul de cost minim dintre arcele tãiate; acest arc va face parte din MST si va uni nodul k cu MST al grafului rãmas dupã îndepãrtarea nodului k. La fiecare pas se face o nouã tãieturã în graful rãmas si se determinã un alt arc din MST; proces repetat de n-1 ori (sau pânã când S este vidã).

Fiecare tãieturã în graf împarte multimea nodurilor din graf în douã submultimi S (noduri incluse în MST) si R (restul nodurilor, încã acoperite cu arce). Initial S={1} dacã se porneste cu nodul 1, iar în final S va contine toate nodurile din graf. Tãieturile succesive pentru exemplul considerat sunt:

S (mst)	arce între S si R (arce tãiate)	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 cã o pereche x[i]-y[i] reprezintã un arc din MST. Este posibilã si folosirea unui vector de întregi pentru arborele MST.

Algoritmul Prim este un algoritm greedy, la care lista de candidati este lista arcelor "tãiate", deci arcele care unesc noduri din U cu noduri din V. La fiecare pas se alege arcul de cost minim dintre arcele tãiate si se genereazã o altã listã de candidati.

Vom prezenta douã variante de implementare a acestui algoritm.

Prima variantã traduce fidel descrierea algoritmului folosind multimi, dar nu este foarte eficientã ca timp de executie:

```
// 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 multimi 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,j) && 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 urmãtor, mai eficient, foloseste doi vectori:
p [i] = numãrul nodului din S cel mai apropiat de nodul i din R
c [i] = costul arcului dintre i si p[i]

La fiecare pas se cautã în vectorul “c” pentru a gãsi nodul k din R cel mai apropiat de nodul i din S. Pentru a nu mai folosi o multime S, se atribuie lui c[k] o valoare foarte mare astfel ca nodul k sã nu mai fie luat in considerare în pasii urmãtori.

Multimea S este deci implicit multimea nodurilor i cu c[i] foarte mare. Celelalte noduri formeazã multimea 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 urmãtoarea:

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 douã constante mari: M1 aratã cã nu existã un arc între douã noduri, iar M2 aratã cã acel arc a fost inclus în MST si cã va fi ignorat în continuare.

Vectorul "p" folosit în programul anterior corespunde reprezentãrii unui arbore printr-un singur vector, de predecesori.

Complexitatea algoritmului Prim cu vectori este O(n*n), dar poate fi redusã la O(m*lg(n)) prin folosirea unui heap pentru memorarea costurilor arcelor dintre U si V

Ideia algoritmului Kruskal este de a alege la fiecare pas arcul de cost minim dintre cele rãmase (încã neselectate), dacã el nu formeazã ciclu cu arcele deja incluse în MST (selectate). Conditia ca un arc (x,y) sã nu formeze ciclu cu celelalte arce selectate se poate exprima astfel: nodurile x si y trebuie sã se afle în componente conexe diferite. Initial fiecare nod formeazã o componentã conexã, iar apoi o componentã conexã contine toate nodurile acoperite cu arce din MST, iar nodurile neacoperite formeazã alte componente conexe.

Algoritmul Kruskal pentru gãsirea unui arbore de acoperire de cost minim foloseste douã tipuri abstracte de date: o coadã cu prioritãti si o colectie de multimi disjuncte si poate fi descris astfel :

```
citire date si creare coada de arce
repetã {
   extrage arcul de cost minim din coada
   dacã arc acceptabil atunci {
       afisare arc
      actualizare componente conexe
   }
} pânã când toate nodurile conectate
```

Un arc care leagã douã noduri dintr-o aceeasi componentã conexã va forma un ciclu cu arcele selectate anterior si nu poate fi acceptat. Va fi acceptat numai un arc care leagã între ele noduri aflate în douã componente conexe diferite.

Pentru reteaua 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 urmãtoarea :

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 sã se afle în componentele conexe. Initial sunt atâtea componente (multimi) câte noduri existã. Atunci când un arc este acceptat, se reunesc cele douã multimi (componente) care contin extremitãtile arcului în una singura; în felul acesta numãrul de componente conexe se reduce treptat pânã 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 urmãtor graful este un vector de arce, ordonat crescãtor dupã 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++];                    // urmãtorul 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 colectiei de multimi disjuncte si este în cel mai bun caz O(m*lg(n)) pentru o implementare eficientã 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, fãrã ca graful sã existe efectiv în memorie. Fiecare nod din graf reprezintã o "stare" în care se aflã programul iar arcele modeleazã trecerea dintr-o stare în alta (nu orice tranzitie între stãri este posibilã si graful nu este complet). Vom mentiona douã categorii de algoritmi de acest tip: algoritmi ce modeleazã automate (masini) cu numãr finit de stãri ("Finite State Machine") si algoritmi de optimizare discretã ("backtracking"si alte metode).

Un algoritm de tip "automat finit" trece dintr-o stare în alta ca urmare a intrãrilor furnizate algoritmului, deci prin citirea succesivã a unor date. Exemple sunt programe de receptie a unor mesaje conforme unui anumit protocol de comunicatie, programe de prelucrare expresii regulate, interpretoare si compilatoare ale unor limbaje.

Un analizor sintactic ("parser") poate avea drept stãri: "într-un comentariu" si "în afara unui comentariu", "într-o constantã sir" si "în afara unei constante sir", "într-un bloc de instructiuni si declaratii" sau "terminare bloc", s.a.m.d. Un analizor pentru limbajul C, de exemplu, trebuie sã deosebeascã 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 tranzitiile între stãrile unui parser interesat de recunoasterea comentariilor C sau C++:

Stare curentã	Caracter citit	Starea urmãtoare
între comentarii	/	posibil început de comentariu
posibil inceput de comentariu	/	în comentariu C++
posibil inceput de comentariu	*	în comentariu C
posibil inceput de comentariu	alte caractere	între comentarii
în comentariu C++	\n	între comentarii
în comentariu C++	alte caractere	în comentariu C++
în comentariu C	*	posibil sfârsit comentariu
posibil sfârsit comentariu	/	între comentarii
posibil sfârsit comentariu	alte caractere	în comentariu C

Stãrile pot fi codificate prin numere întregi iar programul contine un bloc *switch* cu câte un caz (*case*) pentru fiecare stare posibilã.

O problemã de optimizare discretã poate avea mai multe solutii vectoriale (sau matriciale) si fiecare solutie are un cost asociat; scopul este gãsirea unei solutii optime pentru care functia de cost este minimã sau maximã.

O serie de algoritmi de optimizare realizeazã o cãutare într-un graf, numit si spatiu al stãrilor. Acest graf este construit pe mãsurã ce algoritmul progreseazã si nu este memorat integral, având în general un numãr foarte mare de noduri (stãri). Graful este de obicei orientat si arcele au asociate costuri.

O solutie a problemei este o cale în graful stãrilor iar costul solutiei este suma costurilor arcelor ce compun calea respectivã.

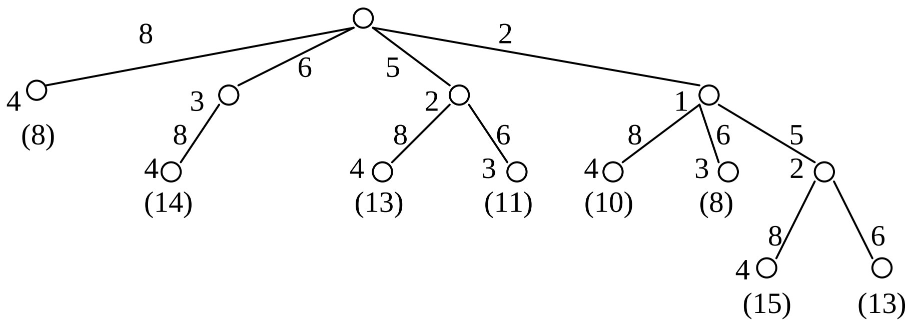
Vom considera douã exemple clasice: problema rucsacului si iesirea din labirint.

Problema rucsacului are ca date n obiecte de greutate g[k] si valoare v[k] fiecare si un sac de capacitate t, iar cerinta este sã selectãm acele obiecte cu greutate totalã mai micã sau egalã cu t pentru care valoarea obiectelor selectate este maximã. Solutia este fie un vector x cu valori 1 sau 0 dupã cum obiectul respectiv a fost sau nu selectat, fie un vector x cu numerele obiectelor din selectia optimã (din rucsac).

Fie cazul concret în care sacul are capacitatea t=15 si exista 4 obiecte de greutãti g[] = {8, 6, 5, 2} si valori unitare egale. Solutia optimã (cu valoare maxima) este cea care foloseste obiectele 1,3 si 4.

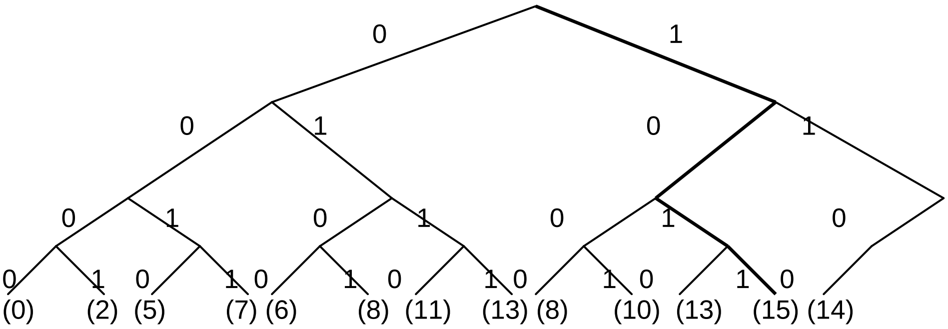
In varianta binara vectorul solutie este x[] = {1,0,1,1}, iar în varianta cu numere de obiecte solutia este x[]={1,2,4}.

Spatiul stãrilor pentru varianta cu x[k] egal cu numãrul obiectului ales în pasul k si dupã eliminarea solutiilor echivalente:



Arcele arborelui de stãri au drept costuri greutãtile (valorile) obiectelor, iar la capãtul fiecãrei ramuri este notatã greutatea selectiei respective (deci costul solutiei). Solutiile optime sunt douã: una care foloseste douã obiecte cu greutãti 6 si 8 si alta care foloseste 3 obiecte cu greutãtile 2,4 si 8.

Spatiul stãrilor în varianta binarã este un arbore binar a cãrui înãltime este egalã cu numãrul de obiecte. Alternativele de pe nivelul k sunt includerea în solutie sau nu a obiectului k. La fiecare cale posibilã este trecut costul însumat al arcelor folosite (valorile obiectelor selectate).

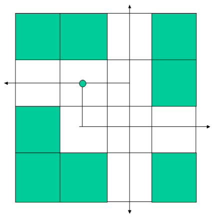


Problema iesirii din labirint ilustreazã o problemã la care spatiul stãrilor nu mai este un arbore ci un graf care poate contine si cicluri.

Un labirint este reprezentat printr-o matrice L de carouri cu m linii si n coloane cu conventia cã L[i][j]=1 dacã caroul din linia i si coloana j este liber (poate fi folosit pentru deplasare) si L[i][j]=0 dacã caroul (i,j) este ocupat (de ziduri despãrtitoare).

Pornind dintr-un carou dat (liber) se cere drumul minim de iesire din labirint sau toate drumurile posibile de iesire din labirint, cu conditia ca un drum sã nu treacã de mai multe ori prin acelasi carou (pentru a evita deplasarea în cerc închis, la infinit). Iesirea din labirint poate însemna cã se ajunge la orice margine sau la un carou dat.

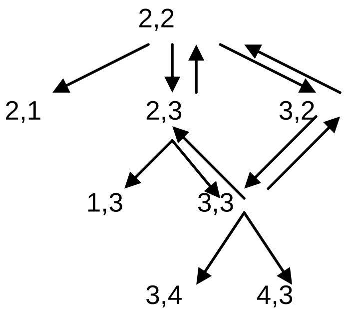
Fie un exemplu de labirint cu 4 linii si 4 coloane si punctul de plecare (2,2).



Câteva trasee de iesire si lungimea lor sunt prezentate mai jos :

(2,2), (2,1)	2
(2,2), (2,3), (1,3)	3
(2,2), (2,3), (3,3), (3,4)	4
(2,2), (3,2), (3,3), (4,3)	4

Graful spatiului stãrilor pentru exemplul de mai sus aratã astfel:



Nodurile fãrã succesori sunt puncte de iesire din labirint. Se observã existenta mai multor cicluri în acest graf.

Explorarea grafului pentru a gãsi o cale cãtre un nod tintã se poate face în adâncime sau în lãrgime.

La explorarea în adâncime se memoreazã (într-o stivã) doar nodurile de pe calea în curs de explorare, deci necesarul de memorie este determinat de cea mai lungã cale din graf. Algoritmul “backtracking” corespunde cãutãrii în adâncime.

La explorarea în lãrgime se memoreazã (într-o coadã) succesorii fiecãrui nod, iar numãrul de noduri de pe un nivel creste exponential cu înãltimea grafului. Din punct de vedere al memoriei necesare cãutarea în adâncime în spatiul stãrilor este preferabilã, dar existã si alte considerente care fac ca în unele situatii sã fie preferatã o variantã de cãutare în lãrgime. Pentru grafuri cu ramuri de lungimi foarte diferite este preferabilã o cãutare în lãrgime. In cazul labirintului, astfel de cãi sunt trasee posibile de lungime foarte mare, dar care nu conduc la o iesire, alãturi de trasee scurte.

Pentru a evita rãmânerea programului într-un ciclu trebuie memorate carourile deja folosite; în principiu se poate folosi o multime de stãri folosite, în care se cautã la fiecare încercare de deplasare din starea curentã. In problema labirintului se foloseste de obicei o solutie ad-hoc, mai simplã, de marcare a carourilor deja folosite, fãrã a mai utiliza o multime separatã.

Timpul de rezolvare a unei probleme prin explorarea spatiului stãrilor depinde de numãrul de noduri si de arce din acest graf, iar reducerea acestui timp se poate face prin reducerea dimensiunii grafului. Graful este generat dinamic, în cursul rezolvãrii problemei prin “expandare”, adicã prin crearea de succesori ai nodului curent.

In graful implicit, un nod (o stare s) are ca succesori stãrile în care se poate ajunge din s, iar aceastã conditie depinde de problema rezolvatã si de algoritmul folosit.

In problema rucsacului, functia “posibil” verificã dacã un nou obiect poate fi luat sau nu în sac (farã a depãsi capacitatea sacului), iar o stare corespunde unei selectii de obiecte. In problema labirintului functia “posibil” verificã dacã este liber caroul prin care încercãm sã ne deplasãm din pozitia curentã.

Graful de stãri se poate reduce ca dimensiune dacã impunem si alte conditii în functia “posibil”. Pentru probleme de minimizare putem compara costul solutiei partiale în curs de generare (costul unei cãi incomplete) cu un cost minim de referintã si sã oprim expandarea cãii dacã acest cost este mai mare decât costul minim stabilit pânã la acel moment. Ideea se poate folosi în problema iesirii din labirint: dacã suntem pe o cale incompletã egalã cu o cale completã anterioarã nu are rost sã mai continuãm pe calea respectivã.

Se poate spune cã diferentele dintre diferiti algoritmi de optimizare discretã provin din modul de expandare a grafului de stãri, pentru minimizarea acestuia.

Cãutarea în adâncime în graful de stãri implicit (metoda “backtracking”) se poate exprima recursiv sau iterativ, folosind o stivã.

Pentru concretizare vom folosi problema umplerii prin inundare a unei suprafete delimitate de un contur oarecare, problemã care seamãnã cu problema labirintului dar este ceva mai simplã. Problema permite vizualizarea diferentei dintre explorarea în adâncime si explorarea în lãtime, prin afisarea suprafetei de colorat dupã fiecare pas.

Datele problemei se reprezintã printr-o matrice pãtraticã de caractere initializatã cu caracterul punct ‘.’, iar punctele colorate vor fi marcate prin caracterul ‘#’. Se dã un punct interior din care începe colorarea (umplerea) spre punctele vecine.

Evolutia unei matrice de 5x5 la explorare în lãrgime a spatiului stãrilor, plecând din punctul (2,2), cu extindere în ordinea sus, dreapta, jos, stânga (primele imagini):

```
. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . # . .   . . # . .
. . . . .   . . # . .   . . # . .   . . # . .   . . # . .   . . # . .   . . # # .
. . # . .   . . # . .   . . # # .   . . # # .   . # # # .   . # # # .   . # # # .
. . . . .   . . . . .   . . . . .   . . # . .   . . # . .   . . # . .   . . # . .
. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .

. . # . .   . . # . .   . . # . .   . . # . .   . . # . .   . . # # .   . # # # .
. # # # .   . # # # .   . # # # .   . # # # .   . # # # .   . # # # .   . # # # .
. # # # .   . # # # #   . # # # #   . # # # #   # # # # #   # # # # #   # # # # #
. . # . .   . . # . .   . . # # .   . . # # .   . # # # .   . # # # .   . # # # .
. . . . .   . . . . .   . . . . .   . . # . .   . . # . .   . . # . .   . . # . .
```

Evolutia matricei 5x5 la explorare în adâncime a spatiului stãrilor, cu ordine de extindere inversã (punctele se scot din stivã în ordine inversã introducerii în stivã) :

```
. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .
. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .
. . # . .   . # # . .   # # # . .   # # # . .   # # # . .   # # # . .   # # # . .
. . . . .   . . . . .   . . . . .   # . . . .   # . . . .   # . . . .   # . . . .
. . . . .   . . . . .   . . . . .   . . . . .   # . . . .   # # . . .   # # # . .

. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .
. . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .   . . . . .
# # # . .   # # # . .   # # # . .   # # # . .   # # # . .   # # # . .   # # # # .
# . . . .   # . . . .   # . . . #   # . . # #   # . # # #   # # # # #   # # # # #
# # # # .   # # # # #   # # # # #   # # # # #   # # # # #   # # # # #   # # # # #
```

In coadã sau în stivã vom pune adrese de puncte (adrese de structuri):

```
typedef struct {  int x,y; } point;
  // construieste o variabila "point"
point * make (int i, int j) {
  point *p= new point;
  p→x=i;p→y=j;
  return p;
}
  // Functie de umplere prin explorare în lãrgime cu coadã :
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
    }
  }
}
```

Functia "posibil" verificã dacã punctul primit este interior conturului si nu este colorat (are culoarea initialã):

```
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 '.'
}
```

Functia de umplere cu exploare în adâncime este la fel cu cea anterioarã, dar foloseste o stivã de pointeri în locul cozii. Functia anterioarã conduce la o crestere rapidã a lungimii cozii (stivei), deoarece acelasi punct necolorat este pus în listã de mai multe ori, ca vecin al punctelor adiacente cu el. Pentru a reduce lungimea listei se poate "colora" fiecare punct pus în listã cu o culoare intermediarã (diferitã de culoarea initialã dar si de cea finalã).

Varianta recursivã 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 cãutare în adâncime într-un arbore (binar sau multicãi). Varianta cu arbore binar (si vector solutie binar) are câteva avantaje: programe mai simple, toate solutiile au aceeasi lungime si nu pot apãrea solutii echivalente (care diferã numai prin ordinea valorilor).

Exemplu de functie recursivã pentru algoritmul "backtracking" (solutii 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;                // incearca 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;                // incearca la dreapta (cu valoarea 0 pe niv. k)
  bkt(k+1);              // cauta in subarborele dreapta
 }
```

Pentru problema rucsacului x[k]=1 semnificã prezenta obiectului k în sac (într-o solutie) iar x[k]=0 semnificã absenta obiectului k dintr-o solutie. Pentru valoarea x[k]=0 nu am verificat dacã solutia este acceptabilã, considerând cã neincluderea unor obiecte în selectia optimã este posibilã întotdeauna.

Ordinea explorãrii celor doi subarbori poate fi modificatã, dar am preferat sã obtinem mai întâi solutii cu mai multe obiecte si valoare mai mare.

Functia "print" fie afiseazã o solutie obtinutã (în problemele de enumerare a tuturor solutiilor posibile), fie comparã costul solutiei obtinute cu cel mai bun cost anterior (în probleme de optimizare, care cer o solutie de cost minim sau maxim).

Capitolul 10

STRUCTURI DE DATE EXTERNE

10.1 SPECIFICUL DATELOR PE SUPORT EXTERN

Principala diferentã dintre memoria internã (RAM) si memoria externã (disc) este modul si timpul de acces la date:
- Pentru acces la disc timpul este mult mai mare decât timpul de acces la RAM (cu câteva ordine de mãrime), dar printr-un singur acces se poate citi un numãr mare de octeti (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). Totusi, nu se folosesc date dispersate într-un fisier din cauza timpului foarte mare de repozitionare pe diferite sectoare din fisier.

In consecintã apar urmãtoarele recomandãri:
- Utilizarea de zone tampon ("buffer") mari, care sã corespundã unui numãr oarecare de sectoare disc, pentru reducerea numãrului de operatii cu discul (citire sau scriere);
- Gruparea fizicã pe disc a datelor între care existã legãturi logice si care vor fi prelucrate foarte probabil împreunã; vom numi aceste grupãri "blocuri" de date ("cluster" sau "bucket").
- Amânarea modificãrilor de articole, prin marcarea articolelor ca sterse si rescrierea periodicã a întregului fisier (în loc de a sterge fizic fiecare articol) si colectarea articolelor care trebuie inserate, în loc de a insera imediat si 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. In principiu se foloseste ideea listelor înlãntuite, care cresc prin alocarea si adãugarea de noi elemente la listã. Practic, nu se folosesc pointeri pentru legarea sectoarelor disc neadiacente dar care fac parte din acelasi fisier; fiecare nume de fisier are asociatã o listã de sectoare disc care apartin fisierului respectiv (ca un vector de pointeri cãtre aceste sectoare). Detaliile sunt mai complicate si depind de sistemul de operare.

Un alt exemplu de adaptare la specificul memoriei externe îl constituie structurile arborescente: dacã un sector disc ar contine mai multe noduri oarecare dintr-un arbore binar, atunci ar fi necesar un numãr mare de sectoare citite (si recitite) pentru a parcurge un arbore. Pentru exemplificare sã considerãm un arbore binar de cãutare cu 4 noduri pe sector, în care ordinea de adãugare a cheilor a condus la urmãtorul continut al sectoarelor disc (numerotate 1,2,3,4) :

50, 30, 40, 20   70, 80, 35, 85   60, 55, 35, 25   65, 75, -, -

Pentru cãutarea valorii 68 în acest arbore ar fi necesarã citirea urmãtoarelor sectoare, în aceastã ordine:

1 (rãdãcina 50), 2 (comparã cu 70), 3 (comparã cu 60), 4 (comparã cu 65)

Solutia gãsitã a fost o structurã arborescentã mai potrivitã pentru discuri, în care un sector (sau mai multe) contine un nod de arbore, iar arborele nu este binar pentru a reduce numãrul de noduri si înãltimea arborelui; acesti arbori se numesc arbori B.

Structurile de date din memoria RAM care folosesc pointeri vor fi salvate pe disc sub o altã formã, farã pointeri; operatia se numeste si "serializare". Serializarea datelor dintr-un arbore, de exemplu, se va face prin traversarea arborelui de la rãdãcinã cãtre frunze (în preordine, de obicei), astfel ca sã fie posibilã reconstituirea legãturilor dintre noduri la o încãrcare ulterioarã a datelor în memorie.

Serializarea datelor dintr-o foaie de calcul ("spreadsheet") se va face scriind pe disc coordonatele (linie,coloana) si continutul celulelor, desi în memorie foaia de calcul se reprezintã ca o matrice de pointeri cãtre continutul celulelor.

De multe ori datele memorate permanent pe suport extern au un volum foarte mare ceea ce face imposibilã memorarea lor simultanã în memoria RAM. Acest fapt are consecinte asupra algoritmilor de sortare externã si asupra modalitãtilor de cãutare rapidã în date pe suport extern.

Sistemele de operare actuale (MS-Windows si Linux) folosesc o memorie virtualã, mai mare decât memoria fizicã RAM, dar totusi limitatã. Memoria virtualã înseamnã extinderea automatã a memoriei RAM pe disc (un fisier sau o partitie de "swap"), dar timpul de acces la memoria extinsã este mult mai mare decât timpul de acces la memoria fizicã si poate conduce la degradarea performantelor unor aplicatii cu structuri de date voluminoase, aparent pãstrate în memoria RAM.

10.2 SORTARE EXTERNÃ

Sortarea externã, adicã sortarea unor fisiere mari care nu încap în memoria RAM sau în memoria virtualã pusã la dispozitie de cãtre sistemul gazdã, este o sortare prin interclasare: se sorteazã intern secvente de articole din fisier si se interclaseazã succesiv aceste secvente ordonate de articole.

Existã variante multiple ale sortãrii externe prin interclasare care diferã prin numãrul, continutul si modul de folosire al fisierelor, prin numãrul fisierelor create.

O secventã ordonatã de articole se numeste si "monotonie" ("run"). Faza initialã a procesului de sortare externã este crearea de monotonii. Un fisier poate contine o singurã monotonie sau mai multe monotonii (pentru a reduce numãrul fisierelor temporare). Interclasarea poate folosi numai douã monotonii ("2-way merge") sau un numãr mai mare de monotonii ("multiway merge"). Dupã fiecare pas se reduce numãrul de monotonii, dar creste lungimea fiecãrei monotonii (fisierul ordonat contine o singurã monotonie).

Crearea monotoniilor se poate face prin citirea unui numãr de articole succesive din fisierul initial, sortarea lor (prin metoda quicksort) si scrierea secventei ordonate ca o monotonie în fisierul de iesire. Pentru exemplificare sã considerãm un fisier ce contine articole cu urmãtoarele chei (în aceastã ordine):

7, 6, 4, 8, 3, 5

Sã considerãm cã se foloseste un buffer de douã articole (în practicã sunt zeci, sute sau mii de articole într-o zonã 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 zonã tampon) se poate face prin metoda selectiei cu înlocuire ("replacement selection") astfel:

- Se alege din buffer articolul cu cea mai micã cheie care este mai mare decât cheia ultimului articol scris în fisier.
- Dacã nu mai existã o astfel de cheie atunci se terminã o monotonie si începe o alta cu cheia minimã din buffer.
- Se scrie în fisierul de iesire articolul cu cheia minimã si se citeste urmãtorul articol din fisierul de intrare.

Pentru exemplul anterior, metoda va crea douã 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 sã se citeascã câte un articol din fiecare monotonie, se va citi câte un grup de articole din fiecare monotonie, ceea ce va reduce numãrul 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 pãstreazã în fisiere si baze de date. O bazã de date reuneste mai multe fisiere necesare unei aplicatii, împreunã cu metadate ce descriu formatul datelor (tipul si lungimea fiecãrui câmp) si cu fisiere index folosite pentru accesul rapid la datele aplicatiei, dupã diferite chei.

Modelul principal utilizat pentru baze de date este modelul relational, care grupeazã datele în tabele, legãturile dintre tabele fiind realizate printr-o coloanã comunã si nu prin adrese disc. Relatiile 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”). In cadrul modelului relational existã o diversitate de solutii de organizare fizicã a datelor, care sã asigure un timp bun de interogare (de regãsire) dupã diverse criterii, dar si modificarea datelor, fãrã degradarea performantelor la cãutare.

Cea mai simplã organizare fizicã a unei baze de date (în dBASE si FoxPro) face din fiecare tabel este un fisier secvential, cu articole de lungime fixã, iar pentru acces rapid se folosesc fisiere index. Metadatele ce descriu fiecare tabel (numele, tipul, lungimea si alte atribute ale coloanelor din tabel) se aflã chiar la începutul fisierului care contine si datele din tabel. Printre aceste metadate se poate afla si numele fisierului index asociat fiecãrei coloane din tabel (dacã a fost creat un fisier index pentru acea coloanã).

Organizarea datelor pe disc pentru reducerea timpului de cãutare este si mai importantã decât pentru colectii de date din memoria internã, datoritã timpului mare de acces la discuri (fatã de memoria RAM) si a volumului mare de date. In principiu existã douã metode de acces rapid dupã o cheie (dupã continut):

- Calculul unei adrese în functie de cheie, ca la un tabel “hash”;
- Crearea si mentinerea unui tabel index, care reuneste cheile si adresele articolelor din fisierul de date indexat.

Prima metodã poate asigura cel mai bun timp de regãsire, dar numai pentru o singurã cheie si fãrã a mentine ordinea cheilor (la fel ca la tabele “hash”).

Atunci când este necesarã cãutarea dupã chei diferite (câmpuri de articole) si când se cere o imagine ordonatã dupã o anumitã cheie a fisierului principal se folosesc tabele index, câte unul pentru fiecare câmp cheie (cheie de cãutare si/sau de ordonare). Aceste tabele index sunt realizate de obicei ca fisiere separate de fisierul principal, ordonate dupã chei.

Un index contine perechi cheie-adresã, unde “adresã” este adresa relativã în fisierul de date a articolului ce contine cheia. Ordinea cheilor din index este în general alta decât ordinea articolelor din fisierul indexat; în fisierul principal ordinea este cea în care au fost adãugate articolele la fisier (mereu la sfârsit de fisier), iar în index este ordinea valorilor cheilor.

Id	Adr
20	
30	
50	
70	
90	

Id	Nume	Marca	Pret	...	
50	aaaaa	AAA	100		
20	dddd	DDD	450		
90	vvvv	VVV	130		
30	cccc	CCC	200		
70	bbbb	BBB	330		

Fisierul index este întotdeauna mai mic decât fisierul indexat, deoarece contine doar un singur câmp din fiecare articol al fisierului principal. Timpul de cãutare va fi deci mai mic în fisierul index decât în fisierul principal, chiar dacã indexul nu este ordonat sau este organizat secvential. De obicei fisierul index este ordonat si este organizat astfel ca sã permitã reducerea timpului de cãutare, dar si a timpului necesar actualizãrii indexului, la modificãri în fisierul principal.

Indexul dat ca exemplu este un index “dens”, care contine câte un articol pentru fiecare articol din fisierul indexat. Un index “rar”, cu mai putine articole decât în fisierul indexat, poate fi folosit atunci când fisierul principal este ordonat dupã cheia continutã de index (situatia când fisierul principal este relativ stabil, cu putine si rare modificãri de articole).

Orice acces la fisierul principal se face prin intermediul fiserului index “activ” la un moment dat si permite o imagine ordonatã a fisierului principal (de exemplu, afisarea articolelor fisierului principal în ordinea din fisierul index).

Mai mult, fisierele index permit selectarea rapidã 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 creeazã aparenta unor noi fisiere, derivate din cele existente, fãrã crearea lor efectivã 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 eficientã a dictionarelor ordonate au fost adaptate pentru fisiere index: arbori binari de cãutare echilibrati (în diferite variante, inclusiv “treap”) si liste skip.

Adaptarea la suport extern înseamnã în principal cã un nod din arbore (sau din listã) nu contine o singurã cheie ci un grup de chei. Mai exact, fiecare nod contine un vector de chei de capacitate fixã, care poate fi completat mai mult sau mai putin. La depãsirea capacitãtii unui nod se creeazã un nou nod.

Cea mai folositã solutie pentru fisiere index o constituie arborii B, în diferite variante (B+, B*).

10.4 ARBORI B

Un arbore B este un arbore de cãutare multicãi echilibrat, adaptat memoriilor externe cu acces direct. Un arbore B de ordinul n are urmãtoarele proprietãti:
- Rãdãcina fie nu are succesori, fie are cel putin doi succesori.
- Fiecare nod interior (altele decât rãdãcina si frunzele) au între n/2 si n succesori.
- Toate cãile de la rãdãcinã la frunze au aceeasi lungime.

Fiecare nod ocupã un articol disc (preferabil un multiplu de sectoare disc) si este citit integral în memorie. Sunt posibile douã variante pentru nodurile unui arbore B:
- Nodurile interne contin doar chei si pointeri la alte noduri, iar datele asociate fiecãrei chei sunt memorate în frunze.
- Toate nodurile au aceeasi structurã, continând atât chei cât si date asociate cheilor.

Fiecare nod (intern) contine o secventã de chei si adrese ale fiilor de forma urmãtoare:

p[0], k[1], p[1], k[2], p[2],...,k[m], p[m]   (n/2 <= m <= n)

unde k[1]<k[2]<...k[m] sunt chei, iar p[0],p[1],..p[m] sunt legãturi cãtre nodurile fii (pseudo-pointeri, pentru cã sunt adrese de octet în cadrul unui fisier disc).

In practicã, un nod contine zeci sau sute de chei si adrese, iar înãltimea arborelui este foarte micã (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, dacã toate nodurile contin numãrul 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<=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^+).
- Numãrul minim de chei pe nod si strategia de spargere a nodurilor (respectiv de contopire noduri cu acelasi pãrinte): n/2 sau 2n/3 sau 3n/4.

Exemplu de arbore B de ordinul 4 (asteriscurile reprezintã adrese de blocuri disc):

```
                        * 18 *
         ______________|      |__________
        |                                |
        * 10 * 12 *                      * 22 * 28 * 34 *
   ____|   __|   _|               ______|  __|     |__   |_______
   |        |      |              |          |         |           |
  4 6 8    10    12 14 16        18 20      22 24 26  28 30 32    34 36 38
```

In desenul anterior nu apar si datele asociate cheilor.

Arborii B au douã utilizãri principale:

- Pentru dictionare cu numãr foarte mare de chei, care nu pot fi pãstrate integral în memoria RAM;
- Pentru fisiere index asociate unor fisiere foarte mari (din baze de date), caz în care datele asociate cheilor sunt adrese disc din fisierul mare indexat.

Cu cât ordinul unui arbore B (numãrul maxim de succesori la fiecare nod) este mai mare, cu atât este mai micã înãltimea arborelui si deci timpul mediu de cãutare.

Cãutarea unei chei date într-un arbore B seamãnã cu cãutarea într-un arbore binar de cãutare BST, dar arborele este multicãi si are nodurile pe disc si nu în memorie.

Nodul rãdãcinã nu este neapãrat primul articol din fisierul arbore B din cel putin douã motive:

- Primul bloc (sau primele blocuri, functie de dimensiunea lor) contin informatii despre structura fisierului (metadate): dimensiune bloc, adresa bloc rãdãcinã, numãrul ultimului bloc folosit din fisier, dimensiune chei, numãr maxim de chei pe bloc, s.a.
- Blocul rãdãcinã se poate modifica în urma cresterii înãltimii arborelui, consecintã a unui numãr mai mare de articole adãugate la fisier.

Fiecare bloc disc trebuie sã continã la început informatii cum ar fi numãrul de chei pe bloc si (eventual) dacã este un nod interior sau un nod frunzã.

Insertia unei chei într-un arbore B începe prin cãutarea blocului de care apartine noua cheie si pot apare douã situatii:

- mai este loc în blocul respectiv, cheia se adaugã si nu se fac alte modificãri;
- nu mai este loc în bloc, se sparge blocul în douã, mutând jumãtate din chei în noul bloc alocat si se introduce o nouã cheie în nodul pãrinte (dacã mai este loc). Acest proces de aparitie a unor noi noduri se poate propaga în sus pânã la rãdãcinã, cu cresterea înãltimii arborelui B. Exemplu de adãugare a cheii 23 la arborele anterior:

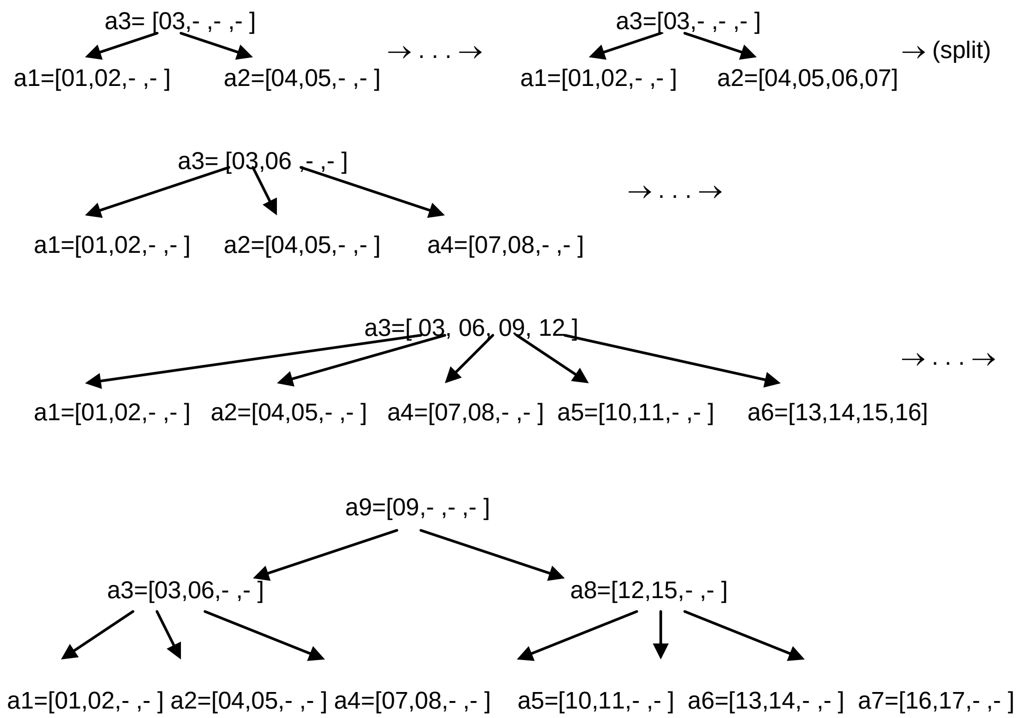
```
                        * 18 *  28  *
         ______________|      |___   |_____________________
        |                         |                          |
        * 10 * 12 *              * 22 * 24 *                 * 34 * 38 *
   ____|   __|   _|          ___|    _|     |        _______|     |     |_____
   |        |      |          |        |      |       |             |          |
  4 6 8   10    12 14 16     18 20   22 23   24 26   28 30 32      34 36      38
```

Propagarea în sus pe arbore a unor modificãri de blocuri înseamnã recitirea unor blocuri disc, deci revenirea la noduri examinate anterior. O solutie mai bunã este anticiparea umplerii unor blocuri: la adãugarea unei chei într-un bloc se verificã dacã blocul este plin si se “sparge” în alte douã blocuri.

Eliminarea unei chei dintr-un arbore B poate antrena comasãri de noduri dacã rãmân prea putine chei într-un nod (mai putin de jumãtate din capacitatea nodului).

Vom ilustra evolutia unui arbore B cu maxim 4 chei si 5 legãturi pe nod (un arbore 2-3-4 dar pe suport extern) la adãugarea unor chei din douã caractere cu valori succesive: 01, 02, ...09, 10, 11,..,19 prin câteva cadre din filmul acestei evolutii. Toate nodurile contin chei si date, iar articolul 0 din fisier contine metadate; primul articol cu date este 1 (notat “a1”), care initial este si rãdãcina arborelui.

a1=[01,- ,- ,-] → [01,02,- ,-] → [01,02,03,-] → [01,02,03,04] → (split)



Secventa de chei ordonate este cea mai rea situatie pentru un arbore B, deoarece rãmân articole numai pe jumãtate completate (cu câte 2 chei), dar am ales-o pentru cã ea conduce repede la noduri pline si care trebuie sparte ("split"), deci la crearea de noi noduri. Crearea unui nod nou se face mereu la sfârsitul fisierului (în primul loc liber) pentru a nu muta date dintr-un nod în altul (pentru minimizarea operatiilor cu discul).

Urmeazã câteva functii pentru operatii cu arbori B si structurile de date folosite de aceste functii:

```
  // 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 fisier, inclusiv primul articol)
typedef struct   {
   int count;                     // Numar de chei dintr-un nod
   Item keys[MaxKeys];           // Chei si date dintr-un nod
   int link[MaxKeys+1];           // Legaturi la noduri fii (int sau long)
} BTNode;
  // zona de lucru comuna functiilor
typedef struct {
  FILE * file;              // Fisierul ce contine arborele B
  char fmode;               // Mod de utilizare fisier ('r' sau 'w')
  int size;                 // Numar de octeti pe nod
  int items;                // Numar total de chei in arbore
  int root;                 // Numar bloc cu radacina arborelui
  int nodes;                // Numar de noduri din arborele B
  BTNode node;              // aici se memoreaza un nod (nodul curent)
} Btree;
```

In aceastã variantã într-un nod nu alterneazã chei si legãturi; existã un vector de chei ("keys") si un vector de legãturi ("link"). link[i] este adresa nodului ce contine 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 legãturi la succesori. Prin “adresã nod” se întelege aici pozitia în fisier a unui articol (relativ la începutul fisierului).

Inaintea oricãrei operatii este necesarã deschiderea fisierului ce contine arborele B, iar la închidere se rescrie antetul (modificat, dacã s-au fãcut adãugãri sau stergeri).

Cãutarea unei chei date în arborele B se face cu functia 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;
}
```

Functia de cãutare 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
}
```

Adãugarea unui nou element la un arbore B este realizatã de câteva functii:
- “addItem” adaugã un element dat la nodul curent (stiind cã este loc)
- “find” cautã nodul unde trebuie adãugat un element, vede dacã nodul este plin, creeazã un nod nou si raporteaza daca trebuie creat nod nou pe nivelul superior
- “split” sparge un nod, creeazã un nod nou si repartizeazã cheilor în mod egal între cele douã noduri
- “insert” foloseste pe “find” si, daca e nevoie, creeazã un alt nod rãdãcinã.

```
void insert(Btree & bt, Item item) {
  int moveUp, newRight ;           // initializate 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 numãrul 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;
   BTNode 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 jumatate 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, BTNode & 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 fãrã clase. In cazul structurilor de date generalizare înseamnã genericitate, adicã 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 legãturilor dintre diferite structuri de date. Astfel, un arbore binar de cãutare devine o clasã derivatã din clasa arbore binar, cu care foloseste în comun o serie de metode (operatii care nu depind de ordinea valorilor din noduri, cum ar fi afisarea arborelui), dar fatã de care posedã metode proprii (operatii specifice, cum ar fi adãugarea de noi noduri sau cãutarea 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 utilizãrii unor biblioteci de clase pentru colectii generice, ceea ce simplificã programarea anumitor aplicatii si înlocuirea unei implementãri cu o altã implementare pentru acelasi tip abstract de date.

Structurile de date se preteazã foarte bine la definirea de clase, deoarece reunesc variabile de diverse tipuri si operatii (functii) asupra acestor variabile (structuri). Nu este întâmplãtor cã singura bibliotecã de clase acceptatã de standardul C++ contine practic numai clase pentru structuri de date
(STL = Standard Template Library).

Programul urmãtor creeazã si afiseazã 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 classei “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 cuvant 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 (dupã o familiarizare cu clasele STL) si eficient, pentru cã dictionarul este implementat ca un arbore binar cu autoechilibrare, de înãltime minimã.

Faptul cã se folosesc aceleasi clase standard în toate programele C++ asigurã acestora un aspect uniform, ceea ce faciliteazã si mai mult întelegerea lor rapidã si modificarea lor fãrã introducere de erori. Programele C sunt mult mai diverse datoritã multiplelor posibilitãti de codificare si de asigurare a genericitãtii, precum si absentei unor biblioteci standard de functii pentru operatii cu structuri de date uzuale.

Pe de altã 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 sã poatã defini noi clase sablon în care sã foloseascã facilitãtile oferite de STL

Clasele STL sunt definite independent unele de altele, fãrã a pune în evidentã relatiile existente între ele. Tehnici specifice programãrii orientate pe obiecte, cum sunt derivarea si mostenirea nu sunt folosite în definirea acestor clase. In schimb, apar functii polimorfice, cu aceeasi formã dar cu implementare diferitã în clase diferite (metode comune claselor container STL).

Clase pentru structuri de date generice se pot realiza în C++ (si în Java) si altfel decât prin clase sablon ("template"), solutie folositã în multe cãrti de structuri de date care folosesc limbajul C++ pentru cã permite definirea unor familii de clase înrudite, folosind derivarea si mostenirea. Existã si biblioteci de clase pentru structuri de date bazate pe aceastã solutie (CLASSLIB din Borland C 3.1), dar nici una care sã se bucure de o recunoastere atât de largã ca biblioteca STL.

Aceastã solutie de definire a unor clase C++ pentru colectii generice seamãnã cu utilizarea de pointeri generici (void*) în limbajul C.

Ideea este ca toate clasele colectie sã continã ca elemente pointeri la obiecte de un tip abstract, foarte general ("object"), iar clasele care genereazã obiecte membre în astfel de colectii sã fie derivate din clasa "object". Deci toate clasele folosite în astfel de aplicatii ar trebui sã fie derivate direct sau indirect dintr-o clasã de bazã "object", aflatã la rãdãcina arborelui de clase.

Un pointer la tipul "object" poate fi înlocuit cu un pointer cãtre orice alt subtip al tipului "object", deci cu un pointer la o clasã derivatã din "object" (derivarea creeazã subtipuri ale tipului de bazã). Pentru a memora date de un tip primitiv (numere, de exemplu) într-o astfel de colectie, va trebui sã dispunem (sau sã definim) clase cu astfel de date si care sunt derivate din clasa "object".

In containerele STL se pot introduce atât valori cât si pointeri, ceea ce permite în final si containere cu obiecte de diferite tipuri înrudite (derivate unele din altele).

11.2 CLASE SI OBIECTE IN C++

Clasele C++ reprezintã o extindere a tipurilor structurã, prin includerea de functii ca membri ai clasei, alãturi de variabilele membre ale clasei. Functiile, numite si metode ale clasei, realizeazã operatii asupra datelor clasei, utile în aplicatii.

O clasã ce corespunde unei structuri de date grupeazã împreunã variabilele ce definesc colectia si operatiile asociate, specifice fiecãrui tip de colectie. De exemplu, o clasã "Stiva" poate avea ca date un vector si un întreg (indice vârf stivã), iar ca metode functii pentru punerea unei valori pe stivã ("push"), scoaterea valorii din vârful stivei ("pop") si altele.

De obicei datele unei clase nu sunt direct accesibile pentru functii din afara clasei având atributul "private" (implicit), ele fiind accesibile numai prin intermediul metodelor publice ale clasei. Aceste metode formeazã "interfata" clasei cu exteriorul. Exemplu de definire a unei clase pentru stive 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 clasã are unul sau mai multi constructori pentru obiectele clasei, sub forma unor functii fãrã nici un tip si cu numele clasei. Constructorul alocã memorie (dacã sunt date alocate dinamic în clasã) si initializeazã variabilele clasei. Exemplu de clasã pentru o stivã cu vector alocat dinamic si 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];        // aloca memorie ptr n intregi
       sp=0;                // initializare varf stiva
     }
    Stiva () {
        s= new int[100];
        sp=0;
    }
    . . .   // metodele clasei
};
```

O functie constructor este apelatã automat în douã situatii:
- La declararea unei variabile de un tip clasã, însotitã de paranteze cu argumente pentru functia 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 si cu functia "alloc"), care poate fi urmat de paranteze si de argumente pentru constrcutor. Exemple:

```
Stiva * ps = new Stiva(20);
Stiva a = * new Stiva(20);
```

Ambele moduri de initializare sunt posibile în C++ si pentru variabile de un tip primitiv (tipuri ale limbajului C). Exemple:

```
int x = *new int(7);
int x(7);       // echivalent cu int x=7
```

In C++ functiile pot avea si argumente cu valori implicite (ultimele argumente); dacã la apel lipseste argumentul efectiv corespunzãtor, atunci se foloseste implicit valoarea declaratã cu argumentul formal corespunzãtor. Aceastã practicã se foloseste si pentru constructori (si 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];             // aloca memorie
      sp=0;
    }
    . . .
};
```

O clasã este un tip de date, iar numele clasei poate fi folosit pentru a declara variabile, pointeri si functii cu rezultat de tip clasã (la fel ca si la tipuri structuri). O variabilã de un tip clasã se numeste si obiect. Sintaxa apelãrii metodelor unei clase C++ diferã de sintaxa apelãrii functiilor C si exprimã actiuni de forma "apeleazã metoda push pentru obiectul stivã 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 urmãtoare, pentru un graf reprezentat printr-o matrice de adiacente, cu câteva operatii strict necesare în aplicatiile 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 operatii cu grafuri pot fi incluse ca metode în clasa "graf", sau pot fi metode ale unei clase derivate din "graf" sau pot fi functii separate, cu un argument "graf". Exemplu de functie independentã pentru vizitarea unui graf în adâncime si 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 si 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 definitia clasei este introdusã într-un fisier antet (de tip H), iar definitiile metodelor sunt continute în fisiere de tip CPP sau sunt deja compilate si introduse în biblioteci statice sau dinamice (LIB sau DLL în sisteme MS-Windows).

Exemplu de clasã pentru obiecte folosite în extragerea cuvintelor dintr-un sir, ca o solutie mai bunã decât functia standard "strtok":

```
 // fisier tokenizer.h
#include <string.h>
#include "stldef.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
};

 // fisier 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 fisierului OBJ cu metodele clasei trebuie sã aparã în acelasi proiect cu numele fisierelor din aplicatia care le foloseste; exceptie face biblioteca standard STL, în care editorul de legãturi cautã 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 utilã în aplicatii, dar suportã si alte variante de definire:

- In locul tipului "char*" sau pe lângã acest tip se poate folosi si tipul "string" definit în biblioteca STL si recomandat pentru sirurile introduse în containere STL.

- In locul unui constructor cu argument sirul analizat se poate defini un constructor fãrã argumente si o metodã care sã preia sirul analizat; în felul acesta nu am mai fi obligati sã declarãm variabila de tip "tokenizer" dupã 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".

Specificã limbajului C++ este supradefinirea operatorilor, care permite extinderea utilizãrii 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 douã caractere speciale, folosite în limbajul C ca operator (unar sau binar). Un operator poate fi definit în câteva moduri:
- ca metodã a unei clase (operator declarat în cadrul clasei);
- ca functie externã clasei (dacã datele clasei sunt publice sau public accesibile);
- ca functie externã declaratã "prieten" ("friend") în clasã, pentru ca functia externã sã aibã acces la datele "private" ale clasei.

Exemplu de supradefinire a operatorului de incrementare prefixat printr-o metodã:

```
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 externã clasei si care nu trebuie declaratã în clasã:

```
ostream & operator << (ostream & os, counter & c) {
   return (os << c.val()) ;
}
```

Exemplu de supradefinire a operatorului de indexare într-o clasã 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 referintã . 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 cãror obiecte se introduc în containere STL trebuie sã aibã definiti 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 clasã A poate contine ca membri obiecte, deci variabile de un tip clasã B. În acest fel metodele clasei A pot (re)folosi metode ale clasei B. De exemplu, o clasã stivã poate contine un obiect de tip vector si refoloseste metode ale clasei vector, cum ar fi extinderea automatã a capacitãtii la umplerea vectorului.

Problema cu aceste clase agregat (compuse) este cã 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 initialã a vectorului este primitã de constructorul obiectelor stivã, pentru cã este si capacitatea initialã a stivei. Un constructor nu poate fi apelat ca o metodã (ca o functie) obisnuitã. Din acest motiv s-a introdus o sintaxã specialã 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 si 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++ fată de limbajul C o constituie si spatiile de nume pentru functii (si clase), în sensul că pot exista functii (si clase) cu nume diferite în spatii de nume diferite. Toate clasele si functiile STL sunt definite în spatiul de nume "std", care trebuie specificat la începutul programelor ce folosesc biblioteca STL:

```
using namespace std;
```

Am mentionat aici acele aspecte ale definirii de clase C++ care sunt folosite si în clasele STL, pentru a facilita întelegerea exemplelor cu clase STL.

In realitate, definitiile de clase sunt mult mai complexe decât exemplele anterioare si folosesc facilităti ale limbajului C++ care nu au fost prezentate.

11.3 CLASE SABLON ("TEMPLATE")

In C++ se pot defini functii si clase sablon având ca parametri tipurile de date folosite în functia sau în clasa respectivă.

Exemplu de functie sablon pentru determinarea valorii minime dintre două variabile de orice tip T, tip neprecizat la definirea functiei:

```
  // definire functie 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 si nu o valoare, dar nu este obligatoriu ca functia "min" să folosească un parametru efectiv de un tip clasă. In functia "min" tipul care va înlocui tipul neprecizat T trebuie să cunoasca operatorul '<' cu rol de comparatie la mai mic. Exemplul următor arată cum se poate folosi functia sablon "min" cu câteva tipuri de date primitive:

```
  // utilizari ale functiei 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 clasã sablon este o clasã în a cãrei definire se folosesc tipuri de date neprecizate pentru variabile si/sau pentru functii membre ale clasei. Toate tipurile neprecizate trebuie declarate într-un preambul al definitiei clasei, care începe prin cuvântul cheie "template" si este urmat, între paranteze ascutite, de o lista 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 clasã sablon trebuie precizat numele tipului efectiv utilizat, între paranteze ascutite, dupã 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 definitiei clasei sablon si a tipului parametrilor efectivi de la instantierea clasei, compilatorul înlocuieste tipul T prin tipul parametrilor efectivi. Pentru fiecare tip de parametru efectiv se genereazã o altã clasã , asa cum se face expandarea unei macroinstructiuni (definitã prin "define"). Definitia clasei este folositã de compilator ca un "sablon" (tipar, model) pentru a genera definitii de clase "normale".

Intre parantezele unghiulare pot fi mai multe tipuri, dacã în clasã se folosesc douã 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 dimensine 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” (clasã STL) si nu “char*” pentru cã are (supra)definiti operatorii de comparatie, utilizati la cãutarea unei chei în dictionar. In general, clasele ale cãror obiecte se introduc în containere STL trebuie sã aibã definiti acesti operatori (==, !=, <, >, <=, >=), pentru ca sã putem exprima la fel comparatiile, indiferent de tipul datelor care se comparã.

O altã problemã, rezolvatã prin exceptii, este “ce rezultat ar trebui sã aibã metoda get atunci când cheia datã nu se aflã în dictionar”, deoarece nu stim nimic despre tipul VT al functiei “get” si deci ce valori speciale de acest tip am putea folosi.

Exemplul urmãtor schiteazã o definitie a clasei “stack” din STL, clasã 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 subclasã derivatã din clasa container. De aceea si operatorii de comparatie trebuie definiti în clasa mstack, chiar dacã definirea lor se reduce la compararea obiectelor container continute de obiectele stivã.

Clasa mstack se numeste si clasã adaptor deoarece nu face decât sã modifice interfata clasei container (alte nume pentru metodele din container), fãrã a modifica si comportamentul clasei (nu existã 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 douã grupe:
- Secvente liniare: clasele *vector, list, deque*
- Containere asociative: *set, multiset,multimap*

Fiecare clasã STL este definitã într-un fisier antet separat, dar fãrã 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 cã 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 mentionãm:

```
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 sfârsitul secventei
void pop_back();                  // eliminare element de la sfârsitul 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 înlãntuite. Toate clasele secventã permit adãugarea de elemente la sfârsitul secventei (metoda "push_back") într-un timp constant (care nu depinde de mãrimea 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 sfârsitul 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 datã a unei secvente.

Clasele *list* si *deque* permit si adãugarea de elemente la începutul secventei în timp O(1) (metoda "push_front"). Exemplu de utilizare a clasei *list* ca o stivã:

```
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 prioritãti) sunt construite pe baza claselor secventã de bazã, fiind liste particulare. Clasa *stack*, de exemplu, adaugã 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 k=1;k<6;k++)
    a.push(k);                         // pune k pe stiva a
```

```
while ( !a.empty()) {
   cout << a.top()<< endl;   a.pop();      // afiseaza si scoate din stiva a
  }
}
```

Metodele "pop", "pop_back", "pop_front" sunt de tip *void* si se folosesc de obicei împreunã (dupã) 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 continutã 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 folositã de container. Pentru clasele iterator sunt supradefiniti operatorii de indirectare (*), comparatie la egalitate (==) si inegalitate(!=), incrementare (++), dar operatorul de decrementare (--) existã numai pentru iteratori bidirectionali.

Prin indirectare se obtine valoarea elementului din pozitia indicatã de iterator. Exemplu de functie sablon pentru cãutarea unei valori date x între douã pozitii date p1 si p2 dintr-un container si care nu depinde de tipul containerului în care se cautã:

```
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 subsecventã dintr-o secventã, iar mai multe functii STL (algoritmi generici) au douã argumente de tip iterator. De exemplu, ordonarea crescãtoare a unui vector v se poate face astfel:

```
sort (v.begin(), v.end());
```

Pentru obiectele de tip *vector* sau *deque* nu se recomandã inserãri si stergeri frecvente (metodele "insert" si "erase"); dacã aplicatia foloseste secvente cu continut dinamic, atunci este mai eficientã secventa *list* (timp de ordinul O(1) pentru orice pozitie din listã).

Pentru liste nu este posibil accesul prin indice (pozitional) si nici ordonarea prin metoda "sort" din biblioteca STL.

In 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 respectã 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 multimi cu chei multiple are ca rezultat pozitia în care a fost inserat noul element, iar dimensiunea multimii creste dupã fiecare apel. Metoda "insert" pentru multimi cu chei unice are ca rezultat o pereche cu primul membru iterator (pozitia în multime) si cu al doilea membru un indicator care aratã dacã s-a modificat continutul (si dimensiunea) multimii (valoarea *true*), sau dacã exista deja un element cu valoarea x si nu s-a modificat multimea (valoarea *false*).

Pentru operatii cu douã multimi (includere, reuniune, intersectie si diferentã) nu existã metode în clasele multime, dar existã functii externe cu argumente iterator. In felul acesta se pot realiza operatii cu submultimi si nu numai cu multimi complete.

Parcurgerea elementelor unei multimi sau unui dictionar (pentru afisarea lor, de exemplu) se poate face numai folosind iteratori, pentru cã nu este posibil accesul prin indici (pozitional) la elementele acestor containere.

Elementele unui dictionar sunt obiecte de tip "pair" (pereche de obiecte), clasã STL definitã 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 dictionare, primul membru al perechii ("first") este cheia, iar al doilea membru ("second") este valoarea asociatã cheii.

Metoda "insert" adaugã o pereche cheie-valoare la dictionar, metoda "find" determinã pozitia acelei perechi care contine o cheie datã, iar obtinerea valorii asociate unei chei nu se face printr-o metodã ci prin operatorul de selectie []. Ideea ar fi cã 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 si afisare a unui dictionar cu frecventa de aparitie a cuvintelor într-un fisier 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 fisier text
  while (in >> cuv) {             // citeste cuvant din fisier
    it=dic.find(cuv);             // cauta cuvant in dictionar
    if (it != dic.end())          // daca exista acel cuvant
      (*it).second++;             // se mãreste numarul de aparitii
    else                          // daca nu exista anterior acel cuvant
       dic.insert ( pair<string,int>(cuv,1));      // atunci se introduce in dictionar
  }

    // afisare continut dictionar
  for (it=dic.begin(); it !=dic.end(); it++)
    cout << (*it).first <<":" << (*it).second << endl;
}
```

Folosind operatorul de selectie [], ciclul principal din programul anterior se poate rescrie astfel:

```
  while (in >> cuv)
    dic[cuv]++;
```

Operatorul de selectie [] cautã cheia datã ca argument iar dacã nu o gãseste introduce automat în dictionar un obiect de tip “pair”, folosind constructorul implicit al acestei clase (care, la rândul lui, apeleazã constructorul implicit pentru tipul “int” si initializeazã cu zero contorul de aparitii).

11.5 UTILIZAREA DE CLASE STL ÎN APLICATII

Aplicatiile care sunt mai usor de scris si de citit cu clase STL sunt cele care folosesc colectii de colectii (vector de liste, de exemplu) si cele în care datele memorate într-o colectie au mai multe componente (cum ar fi un arbore Huffman în care fiecare nod contine un caracter, o frecventã si eventual codul asociat).

Inainte de a relua anumite aplicatii folosind clase STL trebuie spus cã în aceste exemple nu vom folosi toate facilitãtile oferite de STL pentru a face exemplele mai usor de înteles. In aplicatiile STL se foloseste frecvent functia “copy”, inclusiv pentru afisarea continutului unui container pe ecran (în fisierul standard de iesire). 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
```

Aceeasi operatie o vom scrie cu un ciclu explicit folosind un obiect iterator sau operatorul de selectie (numai pentru vectori):

```
vector<string>::iterator it;
 for (it=vs.begin(); it!=vs.end();it++)
    cout << *it << ';' ;
cout << endl;
```

Nu vom utiliza nici alte functii STL (*for_each*, *count*, *equal*, s.a) care pot face programele mai compacte, dar care necesitã explicatii în plus si mãresc diferenta dintre programele C si programele C++ pentru aceleasi aplicatii. Vom mentiona numai functiile *find* si *erase* deoarece sunt folosite într-un exemplu si aratã care stilul de lucru specific STL.

Functia *erase* eliminã o valoare dintr-un container si necesitã ca parametru pozitia din container a valorii eliminate, deci un iterator. Obtinerea pozitiei se poate face prin functia *find*, deci prin cãutarea unei valori într-un container sau, mai exact, între douã pozitii date dintr-un container (într-o subcolectie). Exemplu de functie sablon care eliminã 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 maximã a unui vector, deci vectorul trebuie sã se extindã dinamic. In plus, utilizarea unui obiect *vector* în locul unui vector C ca argument în functii are avantajul reducerii numãrului de argumente (nu mai trebuie datã dimensiunea vectorului ca argument).

De observat cã metoda “size” are ca rezultat numãrul de elemente introduse în vector, iar metoda “capacity” are ca rezultat capacitatea vectorului (care se poate modifica în urma adãugãrilor de elemente la vector). Capacitatea initialã poate fi sau specificatã 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 foloseste un vector de liste de conditionări, unde elementele sortate sunt numere întregi, dar programul se poate modifica usor 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] <<" ";
}
```

In stilul specific programãrii orientate pe obiecte ar trebui definite si utilizate noi clase în loc sã scriem functii ca în C. Programul anterior se poate rescrie cu o clasã pentru obiecte "sortator topologic", având ca metodã principalã pe "topsort".

11.6 DEFINIREA DE NOI CLASE CONTAINER

Biblioteca STL contine un numãr redus de clase container, considerate esentiale, si care pot fi folosite în definirea altor clase.

Un exemplu este definirea unei clase pentru colectii de multimi disjuncte ca vector de multimi:

```
    // fisier DS.H
#include "defstl.h"
class sets {
  vector <set <int> > a;      // vector de multimi de intregi
 public:
  sets(int n);                // constructor
  int find (int x);           // cauta multimea ce contine pe x
  void unif (int x, int y);   // reunire multimi care contin pe x si pe y
  void print();               // afisare colectie de multimi
};
    // 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>();       // construieste o multime a[i]
      a[i].insert(i);         // fiecare a[i] contine initial 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())     // daca x in a[i]
         return i ;                                        // atunci intoarce i
     }
     return -1;               // daca x negasit in colectie
  }
  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 multime ce contine pe x
    for ( p=a[iy].begin(); p!=a[iy].end(); p++)
          a[ix].insert ( *p);    // adauga elementele din a[iy] la a[ix]
    a[iy].clear();               // si goleste multimea 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” definitã anterior, va arãta 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 sã se facã 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.

In exemplul urmãtor se defineste o clasã minimalã pentru arbori binari de cãutare, din care se poate vedea cã pentru metode ce corespund unor algoritmi recursivi trebuie definite functii auxiliare recursive (metodele nu pot fi recursive deoarece nu au ca argument rãdãcina (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=NIL; }                      // 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();
}
```