

# Structuri de Date

Liviu Petrișor Dinu



# Cuprins

<b>1</b>	<b>Structuri de informații</b>	<b>5</b>
1.1	Liste Lineare . . . . .	6
1.1.1	Alocare secvențială . . . . .	7
1.2	Alocare înlănțuită . . . . .	11
1.2.1	Liste circulare . . . . .	12
1.2.2	Liste dublu înlănțuite . . . . .	14
1.2.3	Vectori și liste multilinare . . . . .	14
1.3	Arbori . . . . .	14
1.3.1	Arbori binari . . . . .	15
1.3.2	Reprezentarea arborilor sub forma de arbori binari . .	17
1.3.3	Algoritmul lui Huffman . . . . .	18



# Capitolul 1

## Structuri de informații

Programele pentru calculatoare lucrează de obicei cu *tabele de informații*. În forma sa cea mai simplă, tabelul poate fi o listă lineară de elemente, atunci când proprietățile sale răspund unor întrebări de genul:

1. care este primul element din listă?
2. care este ultimul?
3. ce elemente preced și urmează după un element dat?
4. câte elemente sunt în listă?

În situații mai complicate tabelul ar putea fi un vector bidimensional (matrice) sau un vector  $n$ -dimensional cu  $n > 2$ ; ar putea fi o structură arborescentă; ar putea fi o structură complexă multi-înlănțuită cu un număr imens de legături (e.g. creierul uman).

Pentru o folosire eficientă a calculatorului trebuie să înțelegem atât relațiile structurale ce există între date cât și tehnicile de bază pentru reprezentarea și manevrarea acestor structuri cu ajutorul calculatorului.

Cele mai importante lucruri despre structurile de informații se referă la:

- proprietăți statice și dinamice ale diferitelor tipuri de structuri;
- metode de alocare a spațiului de memorie și de reprezentare a datelor structurate;
- algoritmi eficienți pentru crearea, modificarea, accesul și distrugerea informațiilor structurale.

În cele ce urmează ne vom îndrepta atenția asupra acestor aspecte și vom analiza structurile *așa cum sunt ele reprezentate în interiorul calculatorului*.

## 1.1 Liste Lineare

Pentru o bună proiectare a unui program în general trebuie să știm răspunsul unor întrebări de genul următor:

- ce operații efectuăm asupra datelor?
- cât de mult dintr-o structură trebuie să reprezentăm în tabelele
- cât de accesibile trebuie să fie elementele? noastre?

**Definiția 1.1.1.** *Lista lineară este un șir de  $n \geq 0$  noduri  $X[1]$ ,  $X[2]$ , ...,  $X[n]$ , ale cărui proprietăți structurale esențiale se rezumă la poziția relativă a elementelor așa cum apar ele la rând.*

Operațiile pe care le putem efectua cu astfel de liste sunt:

1. obținerea accesului la nodul al k-lea;
2. inserarea unui nod imediat înaintea sau imediat după nodul al k-lea;
3. ștergerea nodului k;
4. combinarea a două sau mai multe liste lineare într-una singură;
5. partiționarea unei liste lineare în două sau mai multe liste;
6. copierea unei liste lineare;
7. determinarea numărului de noduri dintr-o listă;
8. sortarea nodurilor dintr-o listă pe baza anumitor câmpuri de informații ale nodurilor;
9. căutarea în listă a unui nod cu o anumită valoare;

Listele lineare se diferențiază în funcție de principala operație care se execută asupra componentelor lor.

**Definiția 1.1.2.** *Stiva (stack) este o listă în care toate inserările și ștergerile se fac la unul din capetele sale.*

**Definiția 1.1.3.** *Coadă (queue) este o listă lineară în care toate inserările se fac la unul din capete și toate ștergerile se fac la celălalt capăt.*

### 1.1.1 Alocare secvențială

Cel mai simplu și mai natural mod de a păstra în memoria unui calculator o listă lineară este de a plasa elementele listei în locații consecutive, un nod după altul. Vom avea astfel

$$LOC(X[j+1]) = LOC(X[j]) + c,$$

unde  $c$  este numărul cuvintelor dintr-un nod. (În general  $c=1$ . Când  $c > 1$  este de multe ori mai convenabil să separăm lista inițială în  $c$  liste "paralele".) În general,

$$LOC(X[j]) = L_0 + cj,$$

unde  $L_0$  este o constantă numită adresa de bază, locația unui nod convențional presupus a fi nodul  $X[0]$ .

Alocarea secvențială este foarte convenabilă când lucrăm cu o stivă. Avem nevoie doar de o variabilă  $T$  numită pointerul stivei. Când stiva este vidă îi atribuim lui  $T$  valoarea 0. Pentru a plasa sau șterge un element din stivă aplicăm procedurile următoare:

• **Inserare în stivă** ( $X \Leftarrow Y$ ):

- $T := T + 1$ ;
- dacă  $T > M$  atunci DEPĂȘIRE superioară;
- $X[T] := Y$

• **Ștergerea din stivă** ( $Y \Leftarrow X$ ):

- dacă  $T = 0$  atunci DEPĂȘIRE inferioară;
- $Y := X[T]$ ;
- $T := T - 1$ ;

Reprezentarea unei cozi este ceva mai dificilă și cere mai multă atenție. O soluție evidentă este de a păstra doi pointeri  $R$  și  $F$  (*rear*, respectiv *front*), a.i. inserările să se realizeze cu ajutorul pointerului  $R$  și ștergerile cu ajutorul pointerului  $F$ .

Pentru a preveni risipa de memorie și de timp, vom simula o coadă circulară: vom pune deoparte  $M$  noduri  $X[1]$ ,  $X[2]$ , ...,  $X[M]$  dispuse în cerc, a.i.  $X[1]$  să urmeze după  $X[M]$ . Inserarea, respectiv ștergerea din coadă se realizează cu ajutorul procedurilor următoare:

- **Inserare în coadă ( $X \Leftarrow Y$ ):**
  - dacă  $R = M$  atunci  $R := 1$ ; altfel  $R := R + 1$ ;
  - dacă  $R = F$  atunci DEPĂȘIRE superioară;
  - $X[R] := Y$
- **Ștergerea din coadă ( $Y \Leftarrow X$ ):**
  - $F = R$  atunci DEPĂȘIRE inferioară;
  - dacă  $F = M$  atunci  $F := 1$ ; altfel  $F := F + 1$ ;
  - $Y := X[F]$ ;

**Observația 1.1.1.** *Pointerul  $R$  reprezintă elementul din coadă după care se va insera un nou element; pointerul  $F$  reprezintă elementul din coadă după care se va șterge un element.*

**Observația 1.1.2.** *Pentru o bună comportare a cozii și pentru a nu confunda depășirea inferioară cu cea superioară trebuie să pornim cu inițializarea  $R = F = 1$ .*

O primă întrebare care se pune este ce facem când apare Depășire superioară? Acest caz indică faptul că tabelul este deja plin, dar mai avem informații pe care trebuie să le introducem în el. De obicei lista noastră s-a făcut prea lungă, dar mai există alte liste cu destul spațiu disponibil. Tehnica obișnuită este să se *realoce memoria disponibilă*.

Din nefericire nu există nici o metodă de a memora 3 sau mai multe liste secvențiale de dimensiune variabilă în memorie a.i.: a) Depășirea Superioară să apară doar atunci când mărimea totală a listelor depășește spațiul total; (b) fiecare listă să aibă fixată poziția elementului de la bază. Dacă vrem să satisfacem condiția (a) trebuie să renunțăm la (b), i.e. să permitem elementelor de la baza listei să-și schimbe locul.

Un caz important este cel în care listele sunt stive. Fie  $n$  stive de mărime variabilă. Fiecare stivă va fi referită de baza și vârful său, i.e.  $Base[i]$  și  $Top[i]$ , și presupunem că fiecare nod are lungimea de un cuvânt.

Operațiile de inserare | ștergere devin:

- Inserare:  $Top[i] := Top[i] + 1$ ; dacă  $Top[i] > Base[i + 1]$  atunci Depășire Superioară; altfel atribuie  $Contents(Top[i]) := Y$ .



- Ștergere: dacă  $Top[i]=Base[i]$  atunci Depășire Inferioară; altfel atribuie  $Y:=Contents(Top[i])$ ,  $Top[i]:=Top[i]-1$ .

În cazul Depășirii Superioare vom căuta să redistribuim memoria. Să presupunem că avem  $n$  stive și că lucrăm cu valorile  $Base[i]$ ,  $Top[i]$ , definite ca mai sus. Să presupunem că toate stivele utilizează în comun o zonă de memorie alcătuită din locațiile  $L$  cu  $L_0 < L \leq L_{infy}$ . Putem porni cu toate stivele vide și cu

$$Base[j] = Top[j] = L_0, 1 \leq j \leq n.$$

Punem  $Base[n+1] = L_{infy}$  a.i. operațiile I/S să fie corecte și pentru  $i=n$ .

O soluție pentru a evita DS este ca de fiecare dată să căutăm în sus sau în jos o locație liberă și să deplasăm stivele corespunzător cu o locație.

Multe din depășirile de stivă pot fi eliminat printr-o alegere mai bună a condițiilor inițiale. De exemplu, dacă ne așteptăm ca stivele să fie cam de aceeași dimensiune putem porni cu datele:

$$Base[j] = Top[j] = \lfloor \frac{j-1}{n} (L_{infy} - L_0) \rfloor + L_0, j=1,..n.$$

O metodă mai bună de ameliorare este ca la fiecare redistribuire să se elibereze loc pentru mai mult de un singur element. Garwik a sugerat o redistribuire completă a memoriei la apariția unei depășiri superioare depinzând de modificarea mărimii fiecărei stive de la ultima redistribuire. Algoritmul utilizează un vector suplimentar numit  $OldTop[j]$   $j=1..n$ , care păstrează valorile pe care le avea  $Top[j]$  imediat după alocarea precedentă de memorie. Inițial, tabelele au dimensiunile de mai sus, cu  $OldTop[j] = Top[j]$ .

**Algoritmul G** (Realocarea tabelor secvențiale). Presupunem că a apărut depășire superioară în stiva  $i$ . După executarea algoritmului G fie găsim că a fost depășită capacitatea de memorie, fie se rearanjează memoria a.i. să se poată efectua acțiunea  $NODE(Top[i]):=Y$ .

- G1. (Inițializare) Se dau valorile  $SUM := L_{infy} - L_0$ ;  $INC:=0$ ; se execută pasul 2 pentru  $1 \leq j \leq n$ . După ce s-au efectuat aceste operații se trece la G3.
- G2. (Culegerea datelor statistice) Se face atribuirea  $SUM:=SUM-(Top[j]-Base[j])$ . Dacă  $Top[j] > OldTop[j]$  se dau valorile  $D[j]:=Top[j]-OldTop[j]$  și  $INC:=INC+D[j]$ ; altfel se face atribuirea  $D[j]:=0$ .
- G3 (Memoria este plină?) Dacă  $SUM < 0$  nu mai putem lucra.

- G.4 (Calculul factorilor de realocare) Se dau valorile  $\alpha := 0,1 \times SUM/n$ ,  $\beta := 0,9 \times SUM/INC$  (următorul pas repartizează listelor individuale spațiul disponibil precum urmează: 10% din memoria liberă în prezent va fi împărțită în mod egal celor  $n$  liste, iar restul de 90% vor fi distribuite în mod proporțional cu creșterea în dimensiune a tabelului față de alocarea precedentă)
- G.5 (calculul noilor adrese de bază) Se dau valorile:  $NewBase[1] := Base[1]$  și  $\sigma := 0$ ; apoi, pentru  $j = 2, 3, \dots, n$  se definesc  $\tau := \sigma + \alpha + D[j-1]\beta$ ,  $NewBase[j] := NewBase[j-1] + Top[j-1] - Base[j-1] + \lfloor \tau \rfloor - \lfloor \sigma \rfloor$ , și  $\sigma := \tau$ .
- G6. (Redistribuirea) Se dă valoarea  $Top[i] := Top[i]-1$  Se execută algoritmul R de mai jos după care se revine la valoarea  $Top[i] := Top[i]+1$ . Se face atribuirea  $OldTop[j] := Top[j]$ , pentru  $1 \leq j \leq n$ .

**Algoritmul R** (realocarea tabelor secvențiale). Pentru  $1 \leq j \leq n$  informațiile specificate de  $Base[j]$  și  $Top[j]$  vor fi deplasate în noile locații date de  $NewBase[j]$ , iar valorile  $Base[j]$  și  $Top[j]$  vor fi ajustate corespunzător. Algoritmul se bazează pe faptul că datele care se deplasează în jos nu se vor suprapune peste cele care se deplasează în sus sau stau pe loc.

- R1. (Inițializare) Se dă  $j:=1$ ;
- R2. (Căutarea începutului datelor care trebuie deplasate) (Acum toate listele de la 1 la  $j$  care trebuie mutate în jos au fost aduse pe poziția dorită). Se mărește  $j$  cu câte o unitate până ce se verifică una din relațiile:
- (a)  $NewBase[j] < Base[j]$ ; se trece la R3; sau
  - (b)  $j > n$ : se trece la R4
- R3. (deplasarea listelor în jos). Se fac atribuirile  $\delta := Base[j] - NewBase[j]$  și  $Contents(L - \delta) := Contents(L)$ , pentru  $L = Base[j]+1, Base[j]+2, \dots, Top[j]$ . (Este posibil ca  $Base[j] = Top[j]$ , caz în care nu se solicită nici o acțiune.) Se dau valorile  $Base[j] := NewBase[j]$ ,  $Top[j] := Top[j] - \delta$ . Serevine la R2.
- R4. (Căutarea începutului datelor care trebuie deplasate) (Acum toate listele de la  $j$  la  $n$  care trebuie deplasate în sus au fost aduse pe poziția

dorită). Se micșorează  $j$  cu câte o unitate până ce se verifică una din relațiile:

(a)  $NewBase[j] > Base[j]$ ; se trece la R5; sau

(b)  $j = 1$ : algoritmul se încheie

R5. (deplasarea listelor în sus). Se fac atribuirile  $\delta := NewBase[j] - Base[j]$  și  $Contents(L + \delta) := Contents(L)$ , pentru  $L = Top[j], Top[j] - 1, \dots, Base[j] + 1$ . (Este posibil să nu se solicite nici o acțiune.) Se dau valorile  $Base[j] := NewBase[j]$ ,  $Top[j] := Top[j] - \delta$ . Se revine la R4.

**Observația 1.1.3.** *Stiva 1 nu trebuie niciodată mutată. De aceea este bine să trecem stiva cea mai mare pe prima poziție.*

**Observația 1.1.4.** *În algoritmi  $G$  și  $R$  s-a asigurat în mod deliberat posibilitatea ca*

$$OldTop[j] = D[j] = NewBase[j+1]$$

*pentru  $j=1, \dots, n$ ; cu alte cuvinte aceste 3 tabele pot utiliza în comun locații de memorie deoarece valorile lor nu vor fi niciodată solicitate simultan în momente care să producă conflicte.*

**Observația 1.1.5.** *Experiența a arătat că atunci când memoria este încărcată doar pe jumătate nu va fi nevoie aproape deloc ca tabelele să fie rearanjate cu algoritmul  $G$ . Când tabelele ocupă memoria aproape în întregime algoritmul  $R$  consumă cam mult timp pentru activitatea sa. DS apare mult mai frecvent atunci când memoria este pe sfârșite. Pentru a evita irosirea acestui timp o idee posibilă este aceea ca Algoritmul  $G$  să se oprească la pasul  $G3$  dacă  $SUM$  este mai mică sau egală cu  $S_{min}$ , unde  $S_{min}$  este aleasă de programator.*

## 1.2 Alocare înlănțuită

În loc să menținem o listă lineară în locații succesive de memorie, putem face uz de o schemă mult mai flexibilă, în care fiecare nod conține o legătură la următorul nod al listei.

Alocare secvențială vs. înlănțuită:

- AI solicită spațiu suplimentar pentru legături; totuși , această pierdere nu este foarte mare (de multe ori informațiile din noduri nu ocupă un cuvânt întreg). Mai important, AI poate aduce un câștig de memorie, deoarece tabelele se pot suprapune
- este foarte simplu de șters/inserat un element dintr-o listă înlanțuită;
- referința la un element al listei se eface mai rapid la A.S.
- AI se pretează mai ușor la structuri mai sofisticate, cum ar fi: putem avea un număr oarecare de variabile; orice nod al listei poate fi nod inițial pentru altă listă;

AI implică existența unei liste a spațiului disponibil, pe care o vom nota cu *AVAIL*. Inserarea, respectiv ștergerea unui element în maniera dinamică într-o stivă urmează procedurile de mai jos:

Inserare :  $P \leftarrow AVAIL$ ; Info(P):=Y; Link(P):=T; T:=P.

Ștergere : If  $T = \lambda$ , then listă vidă; altfel P:=T; T:=Link(P); Y:=Info(P);  
 $AVAIL \leftarrow P$

Pentru coadă avem procedurile:

Inserare :  $P \leftarrow AVAIL$ ; Info(P):=Y; Link(P):= $\lambda$ ; dacă  $F=\lambda$  atunci R:=F:=P;  
 altfel Link(R):=P, R:=P.

Ștergere : Dacă  $F = \lambda$ , atunci coadă vidă; altfel P:=F; F:=Link(P); Y:=Info(P);  
 $AVAIL \leftarrow P$

### Sortarea Topologică

Utilă oriunde apare o problemă de ordonare parțială. Aplicații lingvistice, rețele PERT, etc. Exemplu: fie un glosar voluminos cu definițiile diversilor termeni tehnici. Putem scrie  $w_2 < w_1$  iff definiția cuvântului  $w_1$  depinde direct sau indirect de definiția lui  $w_2$ . Problema sortării topologice este găsirea unui mod de aranjare a cuvintelor în glosar a.i. nici un termen să nu apară înainte de a fi definit.

General, problema sortării topologice este de a aranja o ordonare parțială într-o ordine lineară; cu alte cuvinte, de a aranja obiectele într-un sir linear  $a_1, a_2, \dots, a_n$  a.i. ori de câte ori  $a_j < a_k$  să avem  $j < k$ .

Metoda: alegem un obiect care nu este precedat de nici un alt element in cadrul ordonarii. Acest obiect poate fi plasat primul in lista de iesire. Eliminam acum obiectul din multimea initiala; obtinem incontinuare o multime partial ordonata si repetam procedeul pana la sortarea multimii intregi.

Pentru o implementare eficienta trebuie sa putem efectua actiunile descrise mai sus, anume localizarea obiectelor care nu sunt precedate de alte obiecte si eliminarea lor din multime.

Descriere:

Input: perechile:  $9 < 2, 3 < 7, 7 < 5, 5 < 8, 8 < 6, 4 < 6, 1 < 3, 7 < 4, 9 < 5, 2 < 8$ .

Algoritmul va utiliza un tabel secvential  $X[1], X[2], \dots, X[n]$  iar fiecare nod  $X[k]$  va avea forma:

$$| + |0|count[k]|Top[k]|$$

unde  $count[k]$  va fi nr. predecesorilor directi ai obiectului  $k$  (numar de perechi  $j < k$  care apar in datele de intrare) si  $top[k]$  este o legatura la inceputul listei succesorilor directi ai obiectului  $k$ . Aceasta lista contine elemente de forma:

$$| + |0|suc|next|$$

unde  $Suc$  este un succesor direct al lui  $k$  si  $Next$  este urmatorul articol din lista.

**Algorithm:**

- T1. (Initializare) Se introduce  $n$ . Se fac atribuirile  $Count[k] := 0$  si  $Top[k] := \lambda$  pentru  $k = 1..n$ . Se da valoarea  $N := n$
- T2. (Urmatoarea relatie) Se obtine urmatoarea relatie de intrare  $j < k$ . Daca intrarile s-au terminat se trece la T4
- T3. (Inregistrarea relatiei) Se mareste  $Count[k]$  cu 1. Se fac atribuirile:  $P \leftarrow AVAIL$ ,  $Suc(P) := k$ ,  $Next(P) := Top[j]$ ,  $Top[j] := P$ . se revine la T2.
- T4. (Cautarea lui 0). (In acest moment avem reprezentarea in calculator a relatiei. Vom initializa coada de iesire, care este inlantuita cu ajutorul campului  $QLINK$ ). Se dau valorile  $R := 0$  si  $QLINK[0] := 0$ . Pentru  $k = 1..n$ , se examineaza  $Count[k]$  si, daca este 0, se fac atribuirile

$QLINK[R] := k$  și  $R := k$ . După ce s-a efectuat acest lucru pentru toți  $k$ , se da valoarea  $F := QLINK[0]$  (care va conține prima valoare  $k$  întâlnită pentru care  $Count[k] = 0$ ).

T5 (Obținerea elementului din fata al cozii) Se furnizează la ieșire valoarea lui  $F$ . Dacă  $F = 0$ , se trece la T8; altfel se dau valorile  $N := N - 1$  și  $P := Top[F]$  (deoarece tabelele  $QLINK$  și  $Count$  se suprapun, avem  $QLINK[R] = 0$ ; prin urmare condiția  $F = 0$  se verifică atunci când coada este vidă)

T6 Stergerea relațiilor Dacă  $P = \lambda$  se trece la T7. Altfel se micșorează  $Count[Suc(P)]$  cu unu și, dacă a ajuns egal cu 0, se fac atribuiri  $QLINK[R] := Suc(P)$  și  $R := Suc(P)$ . Se da valoarea  $P := Next(P)$  și se reia acest pas (Eliminăm toate relațiile de forma  $F < k$  pentru anumite valori  $k$  din sistem și introducem în coada noi noduri, ai căror predecesori au fost cu toții deja furnizați la ieșire)

T7 (Eliminarea din coada) Se dă valoarea  $F := QLINK[F]$  și se revine la T5.

T8 (sfârșitul procesului) Algoritmul se încheie. Dacă  $N = 0$ , am obținut toate numerele obiectelor în ordinea "topologică" dorită, urmate de un zero. Altfel cele  $N$  obiecte care au mai rămas conțin o buclă, ceea ce contravine definiției ordinii parțiale.

### 1.2.1 Liste circulare

O listă circulară are proprietatea că ultimul ei nod este legat de primul în loc să fie  $\lambda$ . Vom avea astfel acces la toate nodurile listei, indiferent din ce punct plecăm. Să presupunem că avem o variabilă legătură  $PTR$  care indică cel mai din dreapta nod al listei. Cele 3 operații primitive posibile sunt definite după cum urmează:

1. Inserarea lui  $Y$  la stânga:  $P \Leftarrow AVAIL$ ;  $Info(P) := Y$ ; dacă  $PTR = \lambda$  atunci  $PTR := Link(P) := P$ ; altfel  $Link(P) := Link(PTR)$ ,  $Link(PTR) := P$ .
2. Inserarea lui  $Y$  la dreapta: inserez  $Y$  la stânga, apoi  $PTR := P$ .
3. Atribuirea nodului din stânga lui  $Y$  și ștergerea lui din listă: dacă  $PTR = \lambda$  atunci listă vidă; altfel  $P := Link(PTR)$ ,  $Y := Info(P)$ ,  $Link(PTR) := Link(P)$ ; dacă  $PTR = P$ , atunci  $PTR := \lambda$ ,  $AVAIL \Leftarrow P$ .

**Exemplul 1.2.1.** *Adunarea Polinoamelor.*

Ca exemplu de utilizare a listelor circulare vom prezenta adunarea polinoamelor de variabile  $x, y, z$ . Polinomul va fi reprezentat ca o listă în care fiecare nod corespunde unui termen diferit de 0 și are forma:

COEF				
$\pm$	A	B	C	LINK

Aici COEF este coeficientul termenului de forma  $x^A y^B z^C$ .

Algoritmul A. Acest algoritm adună polinomul  $P$  cu polinomul  $Q$ ; lista  $P$  va rămâne neschimbată, în timp ce lista  $Q$  va memora suma.

A1 [Inițializare] Se dau valorile  $P := \text{Link}(P)$ ,  $Q1 := Q$ ;  $Q := \text{Link}(Q)$ ;

A2 [ $ABC(P) ? ABC(Q)$ ] Dacă  $ABC(P) < ABC(Q)$  se fac atribuirile  $Q1 := Q$  și  $Q := \text{LINK}(Q)$  și se reia acest pas; Dacă  $ABC(P) = ABC(Q)$  se trece la Pasul 3. Dacă  $ABC(P) > ABC(Q)$  se trece la Pasul 5.

A3 [Adunarea coeficienților] (Am găsit termeni cu exponenți egali) Dacă  $ABC(P) < 0$  algoritmul se încheie. Altfel, se dă valoarea  $COEF(Q) := COEF(Q) + COEF(P)$ . Acum, dacă  $COEF(Q) = 0$  se trece la A4; în caz contrar, se fac atribuirile  $P := \text{Link}(P)$ ,  $Q1 := Q$ ,  $Q := \text{Link}(Q)$  și se revine la A2.

A4 (Ștergerea termenilor nuli) Se dau valorile  $Q2 := Q$ ,  $\text{Link}(Q1) := Q := \text{Link}(Q)$ ,  $AVAIL \leftarrow Q2$ . Se dă valoarea  $P := \text{Link}(P)$  și se revine la pasul A2.

A5 (Polinomul  $P$  conține un termen care nu apare în  $Q$ , deci îl inserăm). Se fac atribuirile  $Q2 \leftarrow AVAIL$ ,  $COEF(Q2) := COEF(P)$ ,  $ABC(Q2) := ABC(P)$ ,  $\text{Link}(Q2) := Q$ ,  $\text{Link}(Q1) := Q2$ ,  $Q1 := Q2$ ,  $P := \text{Link}(P)$  și se revine la A2.

**1.2.2 Liste dublu înlănțuite**

Pentru o mai mare flexibilitate a manipulării listelor se introduc legături duble pentru fiecare nod: o legătură spre predecesor și una spre succesor. Pentru un nod  $X$ ,  $\text{LLink}(X)$  și  $\text{RLink}(X)$  sunt pointerii ce indică nodul din stânga, respectiv dreapta.

Operațiile tipice de inserare/ ștergere sunt următoarele:

1. **Ștergere**  $RLink(LLink(X)) := RLink(X),$   
 $LLink(RLink(X)) := LLink(X); AVAIL \leftarrow X$
2. **Inserare la dreapta**  $P \leftarrow AVAIL, LLink(P) := X,$   
 $RLink(P) := RLink(X), LLink(RLink(X)) := P, RLink(X) := P$

### 1.2.3 Vectori și liste multilinare

Una din cele mai simple generalizări ale unei liste lineare este un vector de informații bidimensional sau cu mai multe dimensiuni. Reprezentarea acestor structuri poate fi secvențială sau înlanțuită. Pentru al doilea tip, fiecare nod dintr-o matrice va conține 3 cuvinte și 5 câmpuri:

VAL	
ROW	UP
COL	LEFT

Aici ROW și COL sunt indicii nodului pentru linie și coloană; VAL este valoarea memorată de elementul respectiv al matricei; LEFT și UP sunt legături către următorul element nenul din stânga, pe linie, respectiv în sus, pe coloană. Vom avea noduri specifice pentru capetele de listă: BaseRow[i] și BaseCol[j], identificabile prin:

$$COL(LOC(BaseRow[i])) < 0 \text{ și } ROW(LOC(BaseCol[j])) < 0$$

## 1.3 Arbori

Cele mai importante structuri nelineare ce intervin în algoritmi pentru calculatoare sunt arborii. Formal, un arbore este definit ca o mulțime finită T de unul sau mai multe noduri, a.i.:

1. în ea există un nod special numit rădăcina (root) arborelui, root(T);
2. toate celelalte noduri, cu excepția rădăcinii sunt repartizate în  $m \geq 0$  mulțimi disjuncte  $T_1, \dots, T_m$ , fiecare mulțime la rândul său fiind un arbore. Arborii  $T_1, \dots, T_m$  se numesc subarborii rădăcinii.

Pădurea este o mulțime (în general ordonată) de zero sau mai mulți arbori disjuncți.



### 1.3.1 Arbori binari

#### Parcurgerea arborilor binari

##### 1. Preordine

- se vizitează rădăcina
- se parcurge subarborele stâng
- se parcurge subarborele drept

##### 2. Inordine

- se parcurge subarborele stâng
- se vizitează rădăcina
- se parcurge subarborele drept

##### 3. Posordine

- se parcurge subarborele stâng
- se parcurge subarborele drept
- se vizitează rădăcina

În afara acestor parcurgeri recursive există și parcurgeri iterative. Exemplificăm cu inordinea:

Algoritmul T (parcurgerea în inordine a unui arbore binar). Fie T un pointer către un arbore binar.

- T1. [Inițializare]. Se definește stiva A ca vidă și variabila legătură  $P := T$ ;
- T2. [ $P = \lambda$ ?] Dacă  $P = \lambda$  se trece la P4.
- T3. [ $\text{Stivă} \leftarrow P$ ] (Acum P indică un arbore binar nevidcare urmează să fie parcurs) Se efectuează operația  $A \leftarrow P$ , i.e. se introduce P în stiva A. Apoi se dă valoarea  $P := \text{LLink}(P)$  și se revine la T2.
- T4. [ $P \leftarrow \text{Stivă}$ ] Dacă stiva A este vidă, algoritmul se încheie; altfel se efectuează  $P \leftarrow A$ ;
- T5. [Se vizitează P] Se vizitează Node(P). Apoi se face atribuirea  $P := \text{RLink}(P)$  și se revine la pasul T2.

Similar poate fi descrisă și preordinea.

**Observația 1.3.1.** *Un arbore binar este unic determinat de parcurgerile sale în inordine și preordine.*

O reprezentare ingenioasă arborilor binari este cea sub forma unor arbori înșăilați. Pentru aceasta, notăm cu  $P\$$  și  $\$P$  adresa succesorului, respectiv predecesorului, lui  $\text{Node}(P)$  în inordine. Pentru a înșăila un arbore, vom atribui fiecărui nod două legături nevide în felul următor:

Reprezentare fără fire	Reprezentare cu fire	
$LLink(P) = \lambda$	$LTag(P)=1$	$LLink(P)=\$P$
$LLink(P) = Q \neq \lambda$	$LTag(P)=0$	$LLink(P)=Q$
$RLink(P) = \lambda$	$RTag(P)=1$	$RLink(P)=P\$$
$RLink(P) = Q \neq \lambda$	$RTag(P)=0$	$RLink(P)=Q$

Conform acestei definiții, fiecare nouă legătură de tip fir indică direct spre predecesorul sau succesorul nodului despre care este vorba, în ordine simetrică. Câmpurile  $LTag(P)$ ,  $RTag(P)$  ne dau informații dacă legătură stângă, respectiv dreaptă a lui  $P$  este cu fire sau nu.

Pentru a memora un arbore înșăilat folosim un cap de listă cu valorile:  $LLink(Head)=T$ ,  $RLink(Head)=Head$ ,  $RTag(Head)=0$ , unde  $T$  este pointer către arborel binar dacă arborele este nevid,  $LTag(Head)=0$ , altfel  $LLink(Head)=Head$ ,  $LTag(Head)=1$ .

Determinarea succesorului în inordine:

**Algoritm S.** Dacă  $P$  indică spre un nod al unui arbore binar cu fire, algoritmul  $S$  face atribuirea  $Q:=P\$$ :

- S1. [ $RLink(P)$  este fir?] Se dă valoarea  $Q:=RLink(P)$ . Dacă  $RTag(P)=1$ , algoritmul se încheie.
- S2. [Căutare către stânga] Dacă  $LTag(Q)=0$ , se face atribuirea  $Q:=LLink(Q)$  și se reia acest pas. Altfel, algoritmul se încheie.

**Algoritm I.** Acest algoritm atașează un singur nod,  $\text{Node}(Q)$ , ca subarbore drept al nodului  $\text{Node}(P)$ , dacă subarborele drept al acestuia este vid (i.e.  $RTag(P)=1$ ); altfel, inserează nodul  $\text{Node}(P)$  între  $\text{Node}(P)$  și  $\text{Node}(RLink(P))$ , transformându-l pe acesta în copil drept al lui  $\text{Node}(Q)$ .

- I1. [Ajustarea indicatorilor] Se fac atribuirile  $RLink(Q) := RLink(P)$ ,  $RTag(Q) := RTag(P)$ ,  $RLink(P) := Q$ ,  $RTag(P) := 0$ ,  $LLink(Q) := P$ ,  $LTag(Q) := 1$
- I2. [RLink(P) era fir?] Dacă  $RTag(Q) = 0$ , se dă valoarea  $LLink(Q\$) := Q$  (Aici  $Q\$$ ) se determină cu ajutorul algoritmului S. PASul I2 se execută atnci când nu se inserează doar o frunză ci un nod în interiorul arborelui)

Inversând stânga cu dreapta (în particular înlocuind pe  $Q\$$  cu  $\$Q$  la pasul I2) obținem un algoritm care inserează noduri în partea stângă.

### 1.3.2 Reprezentarea arborilor sub forma de arbori binari

O pădure este o mulțime ordonată de 0 sau mai mulți arbori. Subarborii situați imediat sub un nod oarecare al unui arbore formează o pădure. Există un mod natural de a reprezenta o pădure sub forma unui arbore binar. Arborele binar se obține legând unul de altul copiii fiecărei familii și eliminând legăturile verticale cu excepția celei de la un părinte la primul său copil. Reciproc, orice arbore binar corespunde unei unice păduri de arbori ce se obține inversând procedeul.

Riguros formulată, echivalența arată astfel:

Fie  $\mathcal{F} = (T_1, T_2, \dots, T_n)$  o pădure nevidă formată din  $n$  arbori oarecare. Arborele binar  $B(\mathcal{F})$  ce îi corespunde lui  $\mathcal{F}$  poate fi definit astfel:

1. Dacă  $n=0$ ,  $B(\mathcal{F})$  este vid;
2. Dacă  $n > 0$  rădăcina lui  $B(\mathcal{F})$  este rădăcina  $(T_1)$ ; subarboarele stâng al lui  $B(\mathcal{F})$  este  $B(T_{11}, T_{12}, \dots, T_{1m})$ , unde  $T_{11}, T_{12}, \dots, T_{1m}$  sunt subarborii rădăcinii  $(T_1)$ ; subarboarele drept al lui  $B(\mathcal{F})$  este  $B(T_2, \dots, T_n)$

Două metode naturale de traversare a unei păduri:

1. Preordine:
  - (a) se vizitează rădăcina primului arbore,
  - (b) se traversează subarborii primului arbore
  - (c) se traversează ceilalți arbori

2. Postordine:

- (a) se traversează subarborii primului arbore
- (b) se vizitează rădăcina primului arbore,
- (c) se traversează ceilalți arbori

**Observația 1.3.2.** Dacă  $|\mathcal{F}| = 1$ , metodele de mai sus se reduc la traversarea unui arbore oarecare.

**Observația 1.3.3.** Dacă  $\mathcal{F} = (T)$ , și  $T$  este un arbore binar, atunci traversarea pădurii  $\mathcal{F}$  în preordine este identică cu traversarea lui  $T$  în preordine și traversarea lui  $\mathcal{F}$  în postordine este identică cu traversarea lui  $T$  în inordine.

**Propoziția 1.3.1.** Dacă  $u$  și  $v$  sunt două noduri distincte într-o pădure  $\mathcal{F}$ , atunci  $u$  este strămoș al lui  $v$  iff  $u$  îl precede pe  $v$  în preordine și îl succede în postordine.

**Corolar 1.3.1.** Un arbore general tree este unic determinat iff cunoaștem traversările sale în preordine și postordine.

### 1.3.3 Algoritmul lui Huffman

Fie  $T$  un arbore binar. Vom extinde acest arbore la un arbore binar a.i. fiecare nod intern să aibă doi fii și fiecare frunză să aibă de asemenea doi fii (extinderea o facem prin adăugarea unor noi noduri pe care le vom evidenția prin desenarea pătrată a lor). Definim lungimea căii externe  $E$  ca fiind suma -după toate nodurile pătrate- a lungimilor căilor de la rădăcină la fiecare nod. Analog, lungimea căii interne  $I$  este aceeași mărime însumată doar după nodurile interne. Între cele două avem relația:

$$E = I + 2n.$$

Ne interesează construirea unui arbore cu lungimea căii minimă.

Mai general, să presupunem că ni se dau  $m$  numere reale  $w_1, \dots, w_m$ ; problema cere să se determine un arbore binar extins cu  $m$  noduri externe și să se asocieze numerele  $w_1, \dots, w_m$  cu aceste noduri a.i. suma  $\sum w_j l_j$  să fie minimă, unde  $l_j$  este lungimea căii de la rădăcină la nod și suma se efectuează după toate nodurile externe.

Un algoritm elegant de găsim a unui arbore cu lungimea minimă a căii ponderate a fost descoperit de Huffman(1951): mai întâi se găsesc cele mai

mici valori  $w_1$  și  $w_2$ . APoi se rezolvă roblema pentru m-1 ponderi:  $w_1 + w_2, w_3, \dots w_m$  și se înlocuiește nodul  $w_1 + w_2$  cu două frunze egale cu  $w_1, w_2$ .