Structuri de Date

Liviu Petrişor Dinu

Cuprins

Capitolul 1

Structuri de informaţii

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

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

În situaţii 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 strucutră complexă multi-înlănţuită cu un număr imens de legături (e.g. creierul uman).

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

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

- proprietăţi statice şi dinamice ale diferitelor tipuri de structuri;
- metode de alocare a spaţiului de memorie şi de reprezentare a datelor strucutrate;
- algoritmi eficienţi pentru crearea, modificarea, accesul şi distrugerea informaţiilor structurale.

În cele ce urmează ne vom îndrepta atenţia asupra acestor aspecte şi vom analiza structurile *aşa cum sunt ele reprezentate în interiorul calculatorului.*

1.1 Liste Lineare

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

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

Definiţia 1.1.1. *Lista lineară este un şir de $n \geq 0$ noduri X[1], X[2], ..., X[n], ale cărui proprietăţi structurale esenţiale se rezumă la poziţia relativă a elementelor aşa cum apar ele la rând.*

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

1. obţinerea accesului la nodul al k-lea;
2. inserarea unui nod imediat înaintea sau imediat după nodul al k-lea;
3. ştergerea nodului k;
4. combinarea a două sau mai multe liste lineare într-una sngură;
5. partiţionarea 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 informaţii ale nodurilor;
9. căutarea în listă a unui nod cu o anumită valoare;

Listele lineare se diferenţiază în funcţie de principala operaţie care se execută asupra componentelor lor.

Definiţia 1.1.2. *Stiva (stack) este o listă în care toate inserările şi ştergerile se fac la unul din capetele sale.*

Definiţia 1.1.3. *Coada (queue) este o listă lineară în care toate inserările se fac la unul din capete şi toate ştergerile se fac la celălalt capăt.*

1.1.1 Alocare secvenţială

Cel mai simplu şi mai natural mod de a păstra în memoria unui calculator o listă lineară este de a plasa elementele listei în locaţii 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 iniţială în c liste "paralele".) În general,

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

unde L_0 este o constantă numită adresa de bazaă, locaţia unui nod convenţional presupus a fi nodul X[0].

Alocarea secvenţială 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 şterge un element din stivă aplicăm procedurile următoare:

- **Inserare în stivă** ($X \Leftarrow Y$):
  - $T := T + 1$;
  - dacă $T > M$ atunci DEPĂŞIRE superioară;
  - $X[T] := Y$
- **Ştergerea din stivă** ($Y \Leftarrow X$)**:**
  - dacă $T = 0$ atunci DEPĂŞIRE inferioară;
  - $Y := X[T]$;
  - $T := T - 1$;

Reprezentarea unei cozi este ceva mai dificilă şi cere mai multă atenţie. O soluţie evidentă este de a păstra doi pointeri R şi F (*rear*, respectiv *front*), a.i. inserările să se realizeze cu ajutorul pointerului R şi ştergerile cu ajutorul pointerului F.

Pentru a preveni risipa de memorie şi 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 ştergerea 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 DEPĂŞIRE superioară;
  - $X[R] := Y$
- **Ştergerea din coadă** ($Y \Leftarrow X$)**:**
  - $F = R$ atunci DEPĂŞIRE inferioară;
  - dacă $F = M$ atunci F:=1; altfel F:=F+1;
  - $Y := X[F]$;

Observaţia 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 şterge un element.*

Observaţia 1.1.2. *Pentru o bună comportare a cozii şi pentru a nu confunda depăşirea inferioară cu cea superioară trebuie să pornm cu iniţializarea R=F=1.*

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

Din nefericire nu există nici o metodă de a memora 3 sau mai multe liste secvenţiale de dimensiune variabilă în memorie a.i.: a) Depăşirea Superioară să apară doar atunci când mărimea totală a listelor depăşeste spaţiul total; (b) fiecare listă să aibă fixată poziţia elementului de la bază. Dacă vrem să satisfacem condiţia (a) trebuie să renunţăm la (b), i.e. să permitem elementelor de la baza listei să-şi schimbe locul.

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

Operaţiile de inserare | ştergere devin:

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

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

În cazul Depăşirii Superioare vom căuta să redistribuim memoria. Să presupunem că avem n stive şi 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 locaţiile L cu $L_0 < L \leq L_{infty}$. Putem porni cu toate stivele vide şi cu

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

Punem $Base[n+1] = L_{infty}$ a.i. operaţiile I/S să fie corecte şi pentru i=n.

O soluţie pentru a evita DS este ca de fiecare dată să căutăm în sus sau în jos o locaţie liberă şi să deplasăm stivele corespunzător cu o locaţie.

Multe din depăşirile de stivă pot fi eliminat printr-o alegere mai bună a condiţiilor iniţiale. De exemplu, dacă ne aşteptăm ca stivele să fie cam de aceeaşi dimensiune putem porni cu datele:

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

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 apariţia unei depăşiri 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. Iniţital, tabelele au dimensiunile de mai sus, cu OldTop[j] =Top[j].

Algoritmul G (Realocarea tabelelor secvenţiale). Presupunem că a apărut depăşire superioară în stiva i. După executarea algoritmului G fie găsim că a fost depăşită capacitatea de memorie, fie se rearanjează memoria a.i. să se poată efectua acţiunea NODE(Top[i]):=Y.

G1. (Iniţializare) Se dau valorile $SUM := L_{infty} - L_0$; INC:=0; se execută pasul 2 pentru $1 \leq j \geq n$. După ce s-au efectuat aceste operaţii 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] şi INC:=INC+D[j]; altfel se face atribuirea D[j]:=0.

G.3 (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 spaţiul disponibil precum urmează: 10% din memoria liberă în prezent va fi împărţită în mod egal celor n liste, iar restul de 90% vor fi distribuite în mod proporţional cu creşterea în dimensiune a tabelului faţă de alocarea precedentă)

G.5 (calculul noilor adrese de bază) Se dau valorile: NewBase[1]:=Base[1] şi $\sigma := 0$; apoi, pentru $j = 2, 3, ..., n$ se definesc $\tau := \sigma + \alpha + D[j-1]\beta$, $NewBase[j] := NewBase[j-1] + Top[j-1] - Base[j-1] + \lfloor\tau\rfloor - \lfloor\sigma\rfloor$, şi $\sigma := \tau$.

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 $1 \leq j \geq j$.

Algoritmul R (realocarea tabelelor secvenţiale). Pentru $1 \leq j \leq n$ informaţiile specificate de Base[j] şi Top[j] vor fi deplasate în noile locaţii date de NewBase[j], iar valorile Base[j] şi 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. (Iniţializare) 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 poziţia dorită). Se măreşte j cu câte o unitate până ce se verifică una din relaţiile:

(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 $\delta := Base[j] - NewBase[j]$ şi $Contents(L-\delta) := Contents(L)$, pentru L=Base[j]+1, Base[j]+2,...,Top[j]. (Este posibil ca Base[j]=Top[j], caz în care nu se solicită nici o acţiune.) 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 poziţia

dorită). Se micşorează j cu câte o unitate până ce se verifică una din relaţiile:

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

(b) $j = 1$: algoritmul se încheie

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

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

Observaţia 1.1.4. *În algoritmii G şi R s-a asigurat în mod deliberat posibilitatea ca*

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

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

Observaţia 1.1.5. *Experienţa 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 sfârşite. 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 înlănţuită

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

Alocare secvenţială vs. înlănţuită:

- AI solicită spaţiu suplimentar pentru legături; totuşi , această pierdere nu este foarte mare (de multe ori informaţiile din noduri nu ocupă un cuvânt întreg). Mai important, AI poate aduce un câştig de memorie, deoarece tabelele se pot suprapune
- este foarte simplu de şters/inserat un element dintr-o listă înlănţuită;
- referinţa la un element al listei s eface mai rapid la A.S.
- AI se pretează mai uşor la structuri mai sofisticate, cum ar fi: putem avea un număr oarecare de variabile; orice nod al listei poate fi nod iniţial pentru altă listă;

AI implică existenţa unei liste a spaţiului disponibil, pe care o vom nota cu AVAIL. Inserarea, repectiv ştergerea 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.

Ştergere : 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):=λ; dacă F=λ atunci R:=F:=P; altfel Link(R):=P, R:=P.

Ştergere : 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 parţială. Aplicaţii lingvistice, reţele PERT, etc. Exemplu: fie un glosar voluminos cu definiţiile diverşilor termeni tehnici. Putem scrie $w_2 < w_1$ iff definiţia cuvântului w_1 depinde direct sau indirect de definiţia lui w_2. Problema sortării topolgice este găsirea unui mod de aranjare a cuvintelor in glosar a.i. nici un termen sa nu apara inainte de a fi definit.

General, problema sortării topologice este de a aranja o ordonare parţială într-o ordine lineară; cu alte cuvinte, de a aranja obiectele intr-un sir linear $a_1, a_2, \ldots, a_n$ a.i. ori de cate ori $a_j < a_k$ sa 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: $9 < 2, 3 < 7, 7 < 5, 5 < 8, 8 < 6, 4 < 61 < 3, 7 < 4, 9 < 5, 2 < 8$.

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

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

unde count[k] va fi nr. predecesorilor directiai obiectului k (numar de perechi $j < k$ care apar in datele de intrare) si top[k] este 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 s Next este urmatorul articol din lista.

Algoritm:

T1. (Initializare) Se introduce n. Se fac atribuirile Count[k]:=0 si Top[k]:=λ 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 si R:=k. Dupa ce s-a efectuat acest liucru pentru toti k, se da valoarea F:=QLINK[0] (care va contine prima valoare k intalnita pentru care Count[k]=0).

T5 (Obtinerea elementului din fata al cozii) Se furnizeaza la iesire valoarea lui F. Daca F=0, se trece la T8; altfel se dau valorile N:=N-1 si P:=Top[F] (deoarece tabelele QLINK si Count se suprapun, avem QLINK[R]=0; prin urmare conditia F=0 se verifica atunci cand coada este vida)

T6 Stergerea relatiilor Daca $P = \lambda$ se trece la T7. Altfel se micsoreaza Count[Suc(P)] cu unu si, daca a ajuns egal cu 0, se fac atribuirile QLINK[R]:=Suc(P) si R:=Suc(P). Se da valoarea P:=Next(P) si se reia acest pas (Eliminam toate relatiile de dorma $F < k$ pentru anumite valori k din sistem si introducem in coada noi noduri, ai caror predecesori au fost cu totii deja furnizati la iesire)

T7 (Eliminarea din coada) Se davaloarea F:=QLINK[F] si se revine la T5.

T8 (sfarsitul procesului) Algortimul se incheie. Daca N=0, am obtinut toate numerele obiectelor in ordinea "topologica" dorita, urmate de un zero. Altfel cele N obiecte care au mai ramas contin o bucla, ceea ce contravine definitiei ordinii partiale.

1.2.1 Liste circulare

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

1. Inserarea lui Y la stânga: $P \Leftarrow AVAIL$; Info(P):=Y; dacă PTR=λ 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 şi ştergerea 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 şi are forma:

COEF				
±	*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 [Iniţializare] Se dau valorile P:=Link(P), Q1:=Q; Q:=Link(Q);

A2 [ABC(P)?ABC(Q)] Dacă $ABC(P) < ABC(Q)$ *se fac atribuirile Q1:=Q şi Q:=LINK(Q) şi 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 coeficienţilor] (Am găsit termeni cu exponenţi 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:=Link(P), Q1:=Q, Q:=Link(Q) şi se revine la A2.*

A4 (Ştergerea termenilor nuli) Se dau valorile Q2:=Q, Link(Q1):=Q:=Link(Q), $AVAIL \Leftarrow Q2$. *Se dă valoarea P:=Link(P) şi se revine la pasul A2.*

A5 (Polinomul P conţine un termen care nu apare în Q, deci îl inserăm). Se fac atribuirile $Q2 \Leftarrow AVAIL$, *COEF(Q2):=COEF(P), ABC(Q2):=ABC(P), Link(Q2):=Q, Link(Q1):=Q2, Q1:=Q2, P:=Link(P) şi se revine la A2.*

1.2.2 Liste dublu înlănţuite

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

Operaţiile tipice de inserare/ ştergere sunt următoarele:

1. **Ştergere** 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 şi liste multilineare

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

VAL	
ROW	UP
COL	LEFT

Aici ROW şi COL sunt indicii nodului pentru linie şi coloană; VAL este valoarea memorată de elementul respectiv al matricei; LEFT şi 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] şi BaseCol[i], identificabile prin:

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

1.3 Arbori

Cele mai importante structuri nelineare ce intervin în algoritmii pentru calculatoare sunt arborii. Formal, un arbore este definit ca o mulţime 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 exceptţia rădăcinii sunt repartizate în $m \geq 0$ mulţimi disjuncte $T_1, \ldots, T_m$, fiecare mulţime la rândul său fiind un arbore. Arborii $T_1, \ldots, T_m$ se numesc subarborii rădăcinii.

Pădurea este o mulţime (în general ordonată) de zero sau mai mulţi arbori disjuncţi.

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 şi parcurgeri iterative. Exemplificăm cu inordinea:

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

T1. [Iniţializare]. Se defineţe stiva A ca vidă şi variabila legătură P:=T;

T2. [$P = \lambda$?] Dacă $P = \lambda$ se trece la P4.

T3. [Stivă$\Leftarrow P$] (Acum P indică un arbore binar nevidcare urmează să fie parcurs) Se efectuează operaţia $A \Leftarrow P$, i.e. se introduce P în stiva A. Apoi se dă valoarea P:=LLink(P) şi se revine la T2.

T4. [$P \Leftarrow$ 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:=RLink(P) şi se revine la pasul T2.

Similar poate fi descrisă şi preordinea.

Observaţia 1.3.1. *Un arbore binar este unic determinat de parcurgerile sale în inordine şi preordine.*

O reprezentare ingenioasă arborilor binari este cea sub forma unor arbori însăilaţi. Pentru aceasta, notăm cu $P\$$ şi $\$P$ adresa succesorului, respectiv predecesorului, lui Node(P) în inordine. Pentru a însăila 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 definiţii, 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 dqau informaţii dacă legăturq stângă, respectiv dreaptă a lui P este cu fire sau nu.

Pentru a memora un arbore însăilat folosim un cap de listă cu valorile: LLink(Head)=T, RLink(Head)=Head, RTag(Head)=0, unde T este pointer către arborel binarl 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) şi se reia acest pas. Altfel, algoritmul se încheie.

Algoritm I. Acest algoritm ataşează un singur nod, Node(Q), ca subarbore drept al nodului Node(P), dacă subarborele drept al acestuia este vid (i.e. RTag(P)=1); altfel, inserează nodul Node(P) între Node(P) şi Node(RLink(P)), transformându-l pe acesta în copil drept al lui 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) obţinem un algoritm care inserează noduri în partea stângă.

1.3.2 Reprezentarea arborilor sub forma de arbori binari

O pădure este o mulţime ordonată de 0 sau mai mulţi arbori. Subarborii situaţi imediat sub un nod oarecare al unui arbore formează o pădure. Există un mod natural de a repezenta o pădure sub forma unui arbore binar. Arborele binar se obţine legând unul de altul copiii fiecărei familii şi eliminând legăturile verticale cu execpţia celei de la un părinte la primul său copil. Reciproc, orice arbore binar corespunde unei unice păduri de arbori ce se obţine inversând procedeul.

Riguros formulată, echivalenţa arată astfel:

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

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

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 ceilalţi arbori

2. Postordine:

   (a) se traversează subarborii primului arbore

   (b) se vizitează rădăcina primului arbore,

   (c) se traversează ceilalţi arbori

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

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

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

Corolar 1.3.1. *Un arbore general tree este unic determinat iff cunoaţem traversările sale în preordine şi 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 şi fiecare frunză să aibă de asemenea doi fii (extinderea o facem prin adăugarea unor noi noduri pe care le vom evedenţia 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 aceeaşi mărime însumată doar după nodurile interne. Între cele două avem relaţia:

$$\text{E=I+2n.}$$

Ne interesează construirea unui arbore cu lungimea căii minimă.

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

Un algoritm elegant de găsire 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 şi w_2. APoi se rezolvă roblema pentru m-1 ponderi: $w_1 + w_2, w_3, \ldots w_m$ şi se înlocuieşte nodul $w_1 + w_2$ cu două frunze egale cu w_1, w_2.