Cuprins

Capitolul 1

Introducere

Acest manual este dedicat în special studenţilor de la formele de învăţământ ID (Învăţământ la Distanţă) şi PU (Post universitare) de la Facultatea de Informatică a Universităţii "Alexandru Ioan Cuza" din Iaşi. Cartea se doreşte a fi un suport pentru disciplinele Algoritmi şi Programare (ID) şi Structuri de Date şi Algoritmi (PU). Recomandam ca parcurgerea acestui suport sa se facă în paralel cu consultarea materialul electronic aflat pe pagina web a cursului la adresa `http://www.infoiasi.ro/fcs/CS2101.php`. De fapt conţinutul acestei cărţi este o versiune simplificată a celui inclus pe pagina cursului. Din acest motiv unele referinţe apar ca nedefinite (marcate cu "??"). Toate acestea pot fi găsite pe pagina cursului.

Structura manualului este următoarea: Capitolul doi include definiţia limbajului algoritmic utilizat împreună cu definiţiile pentru cele două funcţii principale de măsurare a eficienţei algoritmilor: complexitatea timp şi complexitatea spaţiu. Tipurile de date cele mai utilizate în programare sunt prezentate în capitolul trei. În capitolul patru sunt prezentaţi principalii algoritmi de sortare internă bazaţi pe comparaţii. Capitolul al cincilea este dedicat algoritmilor de căutare şi a principalelor structuri de date utilizate de aceşti algoritmi. Capitolul şase include doi algoritmi de enumerare: enumerarea permutărilor şi enumerarea elementelor produsului cartezian. Fiecare capitol este acopaniat de o listă de exerciţii.

Capitolul 2

Despre algoritmi

2.1 Limbaj algoritmic

2.1.1 Introducere

Un *algoritm* este o secvenţă finită de paşi, aranjată într-o ordine logică specifică care, atunci când este executată, produce o soluţie corectă pentru o problemă precizată. Algoritmii pot fi descrişi în orice limbaj, pornind de la limbajul natural pînă la limbajul de asamblare al unui calculator specific. Un limbaj al cărui scop unic este cel de a descrie algoritmi se numeşte *limbaj algoritmic*. Limbajele de programare sunt exemple de limbaje algoritmice.

În această secţiune descriem limbajul algoritmic utilizat în această carte. Limbajul nostru este tipizat, în sensul că datele sunt organizate în tipuri de date. Un *tip de date* constă dintr-o mulţime de entităţi de tip dată (valori), numită şi *domeniul* tipului, şi o mulţime de operaţii peste aceste entităţi. Convenim să grupăm tipurile de date în trei categorii:

- *tipuri de date elementare*, în care valorile sunt entităţi de informaţie indivizibile;
- *tipuri de date structurate de nivel jos*, în care valorile sunt structuri relativ simple obţinute prin asamblarea de valori elementare sau valori structurate iar operaţiile sunt date la nivel de componentă;
- *tipuri de date structurate de nivel înalt*, în care valorile sunt structuri mai complexe iar operaţiile sunt implementate de algoritmi proiectaţi de către utilizatori.

Primele două categorii sunt dependente de limbaj şi de aceea descrierile lor sunt incluse în aceată secţiune. Tipurile de nivel înalt pot fi descrise într-o manieră independentă de limbaj şi descrierile lor sunt incluse în capitolul 3. Un tip de date descris într-o manieră indepenedentă de reprezentarea valorilor şi implementarea operaţiilor se numeşte *tip de date abstract*.

Paşii unui algoritm şi ordinea logică a acestora sunt descrise cu ajutorul *instrucţiunilor*. O secvenţă de instrucţiuni care acţionează asupra unor structuri de date precizate se numeşte *program*. În secţiunea 2.2 vom vedea care sunt condiţiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm.

2.1.2 Modelarea memoriei

Memoria este reprezentată ca o structură liniară de celule, fiecare celulă având asociată o *adresă* şi putând memora (stoca) o dată de un anumit tip (fig. 2.1). Accesul la memorie este realizat cu ajutorul variabilelor. O *variabilă* este caracterizată de:

- un *nume* cu ajutorul căreia variabila este referită,
- o *adresă* care desemnează o locaţie de memorie şi
- un *tip de date* care descrie natura valorilor memorate în locaţia de memorie asociată variabilei.

Dacă în plus adăugăm şi valoarea memorată la un moment dat în locaţie, atunci obţinem o *instanţă a variabilei*. O variabilă este reprezentată grafic ca în fig. 2.2a. Atunci când tipul se subînţelege din

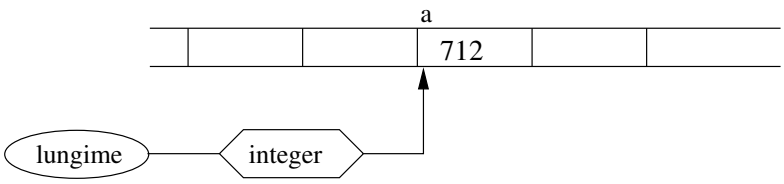


Figura 2.1: Memoria

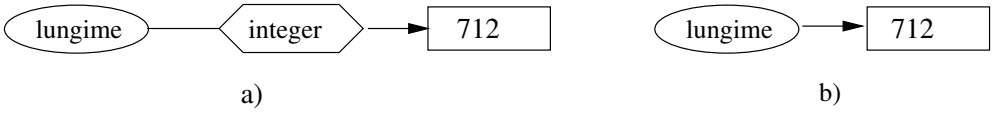


Figura 2.2: Variabilă

context, vom utiliza reprezentarea scurtă sugerată în 2.2b. Convenim să utilizăm fontul `type writer` pentru notarea variabilelor şi fontul *mathnormal* pentru notarea valorilor memorate de variabile.

2.1.3 Tipuri de date elementare

Numere întregi. Valorile sunt numere întregi iar operaţiile sunt cele uzuale: adunarea ($+$), înmulţirea ($*$), scăderea ($-$) etc.

Numere reale. Deoarece prin dată înţelegem o entitate de informaţie reprezentabilă în memoria calculatorului, domeniul tipului numerelor reale este restrâns la submulţimea numerelor raţionale. Operaţiile sunt cele uzuale.

Valori booleene. Domeniul include numai două valori: `true` şi `false`. Peste aceste valori sint definite operaţiile logice `and`, `or` şi `not` cu semnificaţiile cunoscute.

Caractere. Domeniul include litere: `'a'`, `'b'`, ..., `'A'`, `'B'`, ..., cifre: `'0'`, `'1'`, ..., şi caractere speciale: `'+'`, `'*'`, ... Nu există operaţii.

Pointeri. Domeniul unui tip pointer constă din adrese de variabile aparţinând la alt tip. Presupunem existenţa valorii NULL care nu referă nici o variabilă; cu ajutorul ei putem testa dacă un pointer referă sau nu o variabilă. Nu considerăm operaţii peste aceste adrese. Cu ajutorul unei variabile pointer, numită pe scurt şi pointer, se realizează referirea indirectă a unei locaţii de memorie. Un pointer este reprezentat grafic ca în fig. 2.3a. Instanţa variabilei pointer `p` are ca valoare adresa unei variabile de tip întreg. Am notat `integer*` tipul pointer al cărui domeniu este format din adrese de variabile de tip întreg. Această convenţie este extinsă la toate tipurile. Variabila referită de `p` este notată cu `*p`. În fig. 2.3b şi 2.3c sunt date reprezentările grafice simplificate ale pointerilor.

Pointerii sunt utilizaţi la manipularea variabilelor dinamice. O *variabilă dinamică* este o variabilă care poate fi creată şi distrusă în timpul execuţiei programului. Crearea unei variabile dinamice se face cu ajutorul subprogramului `new`. De exemplu, apelul `new(p)` are ca efect crearea variabilei `*p`. Distrugerea (eliberarea spaţiului de memorie) variabilei `*p` se face cu ajutorul apelului `delete(p)` al subprogramului `delete`.

2.1.4 Instrucţiuni

Atribuirea. *Sintaxa*:

⟨variabilă⟩ ← ⟨expresie⟩

unde ⟨variabilă⟩ este numele unei variabile iar ⟨expresie⟩ este o expresie corect formată de acelaşi tip cu ⟨variabilă⟩.

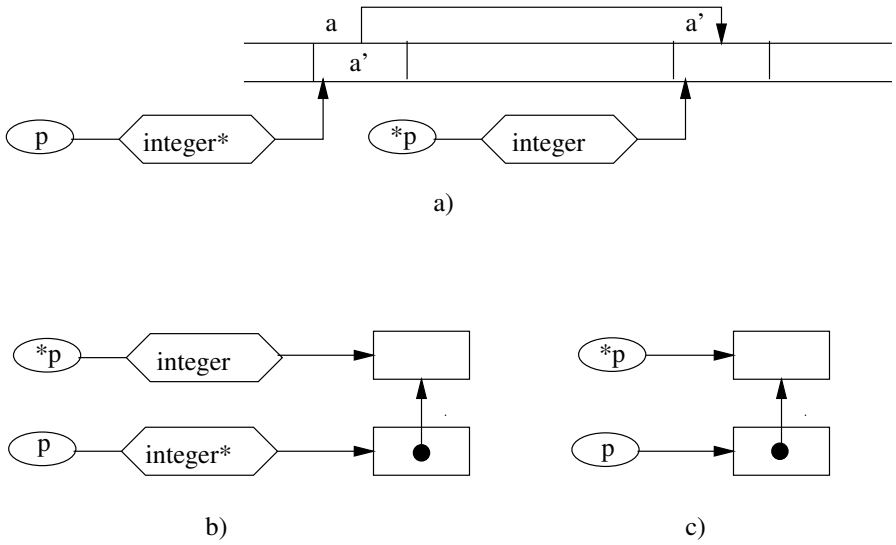


Figura 2.3: Pointer

Semantica: Se evaluează ⟨expresie⟩ şi rezultatul obţinut se memorează în locaţia de memorie desemnată de ⟨variabilă⟩. Valorile tuturor celorlalte variabile rămân neschimbate. Atribuirea este singura instrucţiune cu ajutorul căreia se poate modifica memoria. O reprezentare intuitivă a efectului instrucţiunii de atribuire este dată în fig. 2.4.

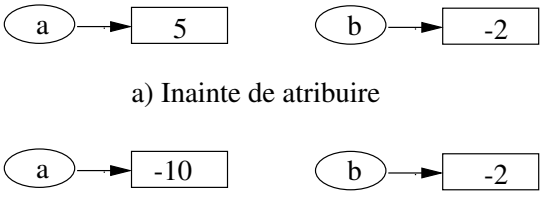


Figura 2.4: Atribuirea

if. *Sintaxa*:

`if` ⟨expresie⟩
`then` ⟨secvenţă-instrucţiuni$_1$⟩
`else` ⟨secvenţă-instrucţiuni$_2$⟩

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni$_i$⟩, $i = 1, 2$, sunt secvenţe de instrucţiuni scrise una sub alta şi aliniate corespunzător. Partea `else` este facultativă. Dacă partea `else` lipseşte şi ⟨secvenţă-instrucţiuni$_1$⟩ este formată dintr-o singură instrucţiune atunci instrucţiunea `if` poate fi scrisă şi pe un singur rând. De asemenea, o expresie în cascadă de forma

```
if (...)
then ...
else if (...)
     then ...
     else if (...)
          then ...
          else ...
```

va fi scrisă sub următoarea formă liniară echivalentă:

```
if (...) then
   ...
else if (...) then
   ...
else if (...) then
   ...
else ...
```

Semantica: Se evaluează ⟨expresie⟩. Dacă rezultatul evaluării este `true` atunci se execută ⟨secvenţă-instrucţiuni$_1$⟩ după care execuţia instrucţiunii `if` se termină; dacă rezultatul evaluării este `false` atunci se execută ⟨secvenţă-instrucţiuni$_2$⟩ după care execuţia instrucţiunii `if` se termină.

while. *Sintaxa*:

`while` ⟨expresie⟩ `do`
⟨ secvenţă-instrucţiuni⟩

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: 1. Se evaluează ⟨expresie⟩.
2. Dacă rezultatul evaluării este `true` atunci se execută ⟨secvenţă-instrucţiuni⟩ după care se reia procesul începând cu pasul 1. Dacă rezultatul evaluării este `false` atunci execuţia instrucţiunii `while` se termină.

for. *Sintaxa*:

`for` ⟨variabilă⟩ $\leftarrow$ ⟨expresie$_1$⟩ `to` ⟨expresie$_2$⟩ `do`
⟨ secvenţă-instrucţiuni⟩

sau

`for` ⟨variabilă⟩ $\leftarrow$ ⟨expresie$_1$⟩ `downto` ⟨expresie$_2$⟩ `do`
⟨secvenţă-instrucţiuni⟩

unde ⟨variabilă⟩ este o variabilă de tip întreg, ⟨expresie$_i$⟩, $i = 1, 2$, sunt expresii care prin evaluare dau valori întregi, ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: Instrucţiunea

`for i` $\leftarrow$ e_1 `to` e_2 `do`
S

simulează execuţia următorului program:

`i` $\leftarrow$ e_1
`temp` $\leftarrow$ e_2
`while (i` $\leq$ `temp) do`
S
`i` $\leftarrow$ `i+1`

iar instrucţiunea

`for i` $\leftarrow$ e_1 `downto` e_2 `do`
S

simulează execuţia următorului program:

`i` $\leftarrow$ e_1
`temp` $\leftarrow$ e_2
`while (i` $\geq$ `temp) do`
S
`i` $\leftarrow$ `i-1`

repeat. *Sintaxa*:

```
repeat
    ⟨secvenţă-instrucţiuni⟩
until ⟨expresie⟩
```

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: Instrucţiunea

```
repeat
    S
until e
```

simulează execuţia următorului program:

```
S
while (not e) do
    S
```

Excepţii. *Sintaxa*:

```
throw  ⟨mesaj⟩
```

unde ⟨mesaj⟩ este un şir de caractere (un text).
Semantica: Execuţia programului se opreşte şi este afişat textul ⟨mesaj⟩. Cel mai adesea, `throw` este utilizată împreună cu `if`:

```
if ⟨expresie⟩ then throw  ⟨mesaj⟩
```

Obţinerea rezultatului `true` în urma evaluării expresiei are ca semnificaţie apariţia unei excepţii, caz în care execuţia programului se opreşte. Un exemplu de excepţie este cel când procedura `new` nu poate aloca memorie pentru variabilele dinamice:

```
new(p)
if (p = NULL) then throw 'memorie insuficienta'
```

2.1.5 Subprograme

Limbajul nostru algoritmic este unul modular, unde un modul este identificat de un subprogram. Există două tipuri de subprograme: proceduri şi funcţii.

Proceduri. Interfaţa dintre o procedură şi modulul care o apelează este realizată numai prin parametri şi variabilele globale. De aceea apelul unei proceduri apare numai ca o instrucţiune separată. Forma generală a unei proceduri este:

```
procedure ⟨nume⟩(⟨lista-parametri⟩)
begin
    ⟨secvenţă-instrucţiuni⟩
end
```

Lista parametrilor este opţională. Considerăm ca exemplu o procedură care interschimbă valorile a două variabile:

```
procedure swap(x, y)
begin
    aux ← x
    x ← y
    y ← aux
end
```

Permutarea circulară a valorilor a trei variabile `a, b, c` se face apelând de două ori procedura `swap`:

```
swap(a, b)
swap(b, c)
```

Funcţii. În plus faţă de proceduri, funcţiile întorc valori calculate în interiorul acestora. De aceea apelul unei funcţii poate participa la formarea de expresii. Forma generală a unei funcţii este:

```
function ⟨nume⟩(⟨lista-parametri⟩)
begin
   ⟨secvenţă-instrucţiuni⟩
   return ⟨expresie⟩
end
```

Lista parametrilor este opţională. Valoarea întoarsă de funcţie este cea obţinută prin evaluarea expresiei. O instrucţiune `return` poate apărea în mai multe locuri în definiţia unei funcţii. Considerăm ca exemplu o funcţie care calculează maximul dintre valorile a trei variabile:

```
function max3(x, y, z)
begin
   temp ← x
   if (y > temp) then temp ← y
   if (z > temp) then temp ← z
   return temp
end
```

Are sens să scriem `2*max3(a, b, c)` sau `max3(a, b, c) < 5`.

2.1.5.1 Comentarii

Comentariile sunt notate similar ca în limbajul C, utilizând combinaţiile de caractere `/*` şi `*/`. Comentariile au rolul de a introduce explicaţii suplimentare privind descrierea algoritmului:

```
function absDec(x)
begin
   if (x > 0) /* testeaza daca x este pozitiv */
   then x ← x-1 /* decrementeaza x*/
   else x ← x+1 /* incrementeaza x*/
end
```

2.1.6 Tipuri de date structurate de nivel jos

2.1.6.1 Tablouri

Un *tablou* este un ansamblu omogen de variabile, numite *componentele* tabloului, în care toate variabilele componente aparţin aceluiaşi tip şi sunt identificate cu ajutorul indicilor. Un *tablou 1-dimensional (unidimensional)* este un tablou în care componentele sunt identificate cu ajutorul unui singur indice. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru indice este $\{0, 1, 2, \ldots, n-1\}$, atunci variabilele componente sunt `a[0], a[1], a[2], ..., a[n-1]`. Memoria alocată unui tablou 1-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea indicilor. Tablourile 1-dimensionale sunt reprezentate grafic ca în fig. 2.5. Operaţiile asupra tablourilor se realizează prin intermediul componentelor. Prezentăm ca exemplu iniţializarea tuturor componentelor cu 0:

```
for i ← 0 to n-1 do
    a[i] ← 0
```

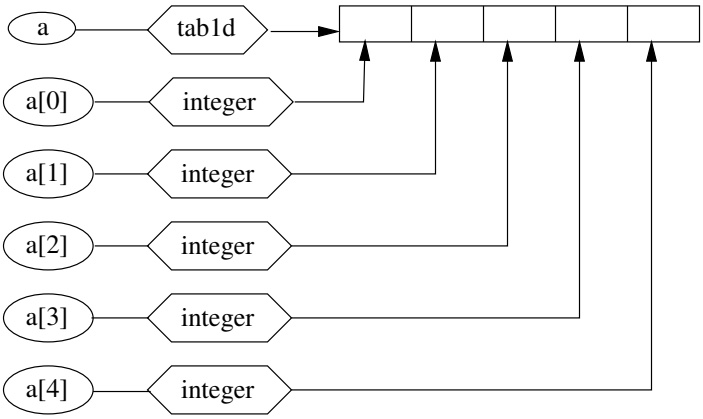


Figura 2.5: Tablou 1-dimensional

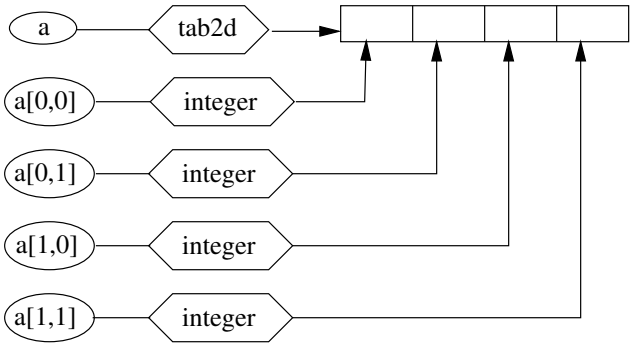


Figura 2.6: Tablou 2-dimensional

Un *tablou 2-dimensional (bidimensional)* este un tablou în care componentele sunt identificate cu ajutorul a doi indici. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru primul indice este $\{0, 1, \ldots, m-1\}$ iar mulţimea valorilor pentru cel de-al doilea indice este $\{0, 1, \ldots, n-1\}$ atunci variabilele componente sunt `a[0,0], a[0,1],..., a[0,m-1], ..., a[m-1,0], a[m-1,1], ..., a[m-1,n-1]`. Ca şi în cazul tablourilor 1-dimensionale, memoria alocată unui tablou 2-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea lexicografică definită peste indici. Tablourile 2-dimensionale sunt reprezentate grafic ca în fig. 2.6. Aşa cum o matrice poate fi văzută ca fiind un vector de linii, tot aşa un tablou 2-dimensional poate fi văzut ca fiind un tablou 1-dimensional de tablouri 1-dimensionale. Din acest motiv, componentele unui tablou 2-dimensional mai pot fi notate prin expresii de forma `a[0][0]`, `a[0][1]`, etc (a se vedea şi fig. 2.7).

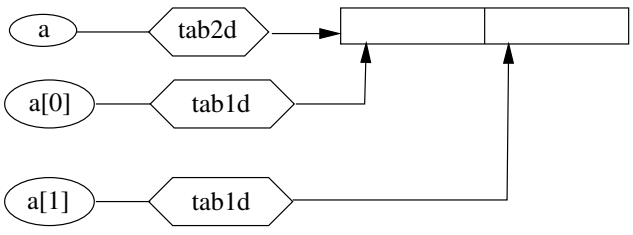


Figura 2.7: Tablou 2-dimensional văzut ca tablou de tablouri 1-dimensionale

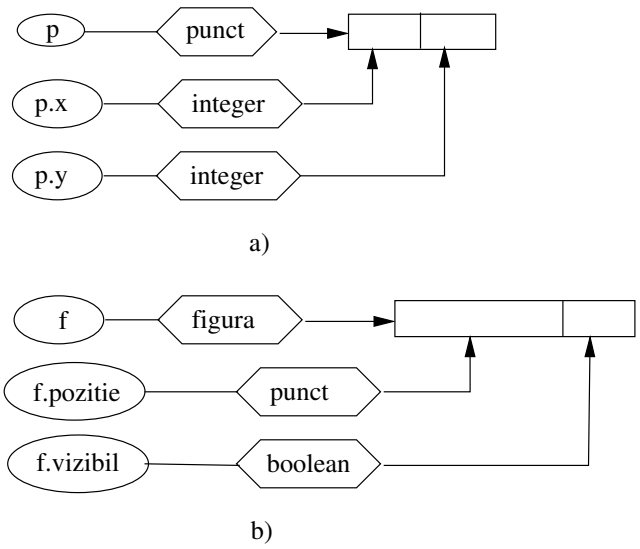


Figura 2.8: Structuri

2.1.6.2 Şiruri de caractere.

Şirurile de caractere pot fi gândite ca fiind tablouri unidimensionale a căror elemente sunt caractere. O constantă şir de caractere este notată utilizând convenţia din limbajul C: ‘‘`exemplu de sir`’’. Peste şiruri sunt definite următoarele operaţii:

- concatenarea, notată cu `+`: ’’`unu`’’ `+` ‘‘`doi`’’ are ca rezultat ’’`unudoi`’’;
- `strcmp(sir1, sir2)` - întoarce rezultatul compărării lexicografice a celor două şiruri: -1 dacă `sir1` < `sir2`, 0 dacă `sir1` = `sir2`, şi +1 dacă `sir1` > `sir2`;
- `strlen(sir)` - întoarce lungimea şirului dat ca parametru;
- `strcmp(sir1, sir2)` - realizează copierea şirului `sir2` în `sir1`.

2.1.6.3 Structuri

O *structură* este un ansamblu eterogen de variabile, numite *câmpuri*, în care fiecare câmp are propriul său nume şi propriul său tip. Numele complet al unui câmp se obţine din numele structurii urmat de caracterul “.” şi numele câmpului. Memoria alocată unei structuri este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare câmp. Ordinea de memorare a câmpurilor corespunde cu ordinea de descriere a acestora în cadrul structurii. Ca exemplu, presupunem că o figură `f` este descrisă de două câmpuri: `f.pozitie` - punctul care precizează poziţia figurii, şi `f.vizibil` - valoare booleana care precizează dacă figura este desenată sau nu. La rândul său, punctul poate fi văzut ca o structură cu două câmpuri - câte unul pentru fiecare coordonată (considerăm numai puncte în plan având coordonate întregi). Structurile sunt reprezentate grafic ca în fig. 2.8. Pentru identificarea câmpurilor unei structuri referite indirect prin intermediul unui pointer vom utiliza o notaţie similară celei utilizate în limbajul C (a se vedea şi fig. 2.9).

2.1.6.4 Liste liniare simplu înlănţuite

O *listă liniară simplu înlănţuită* este o înlănţuire de structuri, numite *noduri*, în care fiecare nod, exceptând ultimul, “cunoaşte” adresa nodului de după el (nodul succesor). În forma sa cea mai simplă, un nod `v` este o structură cu două câmpuri: un câmp `v->elt` pentru memorarea informaţiei şi un câmp `v->succ` care memorează adresa nodului succesor. Se presupune că se cunosc adresele primului şi respectiv ultimului nod din listă. O listă liniară simplu înlănţuită este reprezentată grafic ca în fig. 2.10. Listă liniară simplu înlănţuită este o structură de date dinamică în sensul că pot fi inserate sau eliminate noduri cu condiţia să fie păstrată proprietatea de înlănţuire liniară. Operaţiile elementare ce se pot

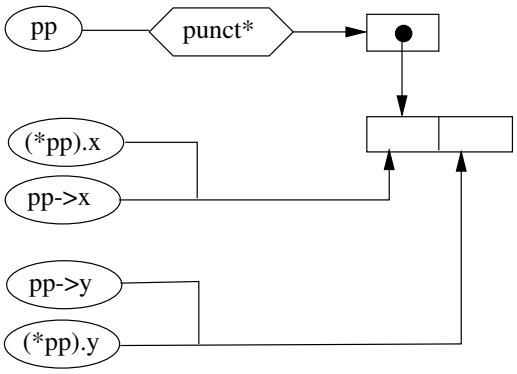


Figura 2.9: Structuri şi pointeri

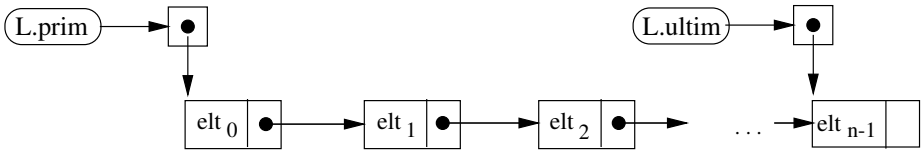


Figura 2.10: Listă liniară simplu înlănţuită

efectua asupara unei liste simplu înlănţuite sunt:

- adăugarea unui nod la început (a se vedea figura 2.11):

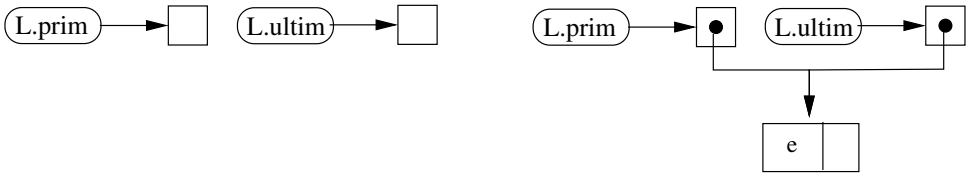
```
procedure adLaInc(L, e)
begin
  new(v)
  v->elt ← e
  if (L = NULL)
  then L.prim ← v        /* lista vida */
       L.ultim ← NULL
       v->succ ← NULL
  else v->succ ← L.prim  /* lista nevida */
       L.prim ← v
end
```

- adăugarea unui nod la sfârşit (a se vedea figura 2.12):

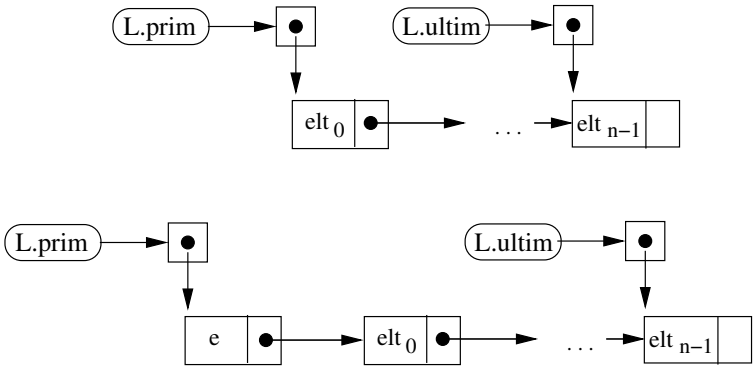
```
procedure adLaSf(L, e)
begin
  new(v)
  v->elt ← e
  v->succ ← NULL
  if (L = NULL)
  then L.prim ← v         /* lista vida */
       L.ultim ← NULL
  else L.ultim->succ ← v  /* lista nevida */
       L.ultim ← v
end
```

- ştergerea nodului de la început:

```
procedure stLaInc(L)
```



a) Lista initiala vida



b) Lista initiala nevida

Figura 2.11: Adăugarea unui nod la începutul unei liste

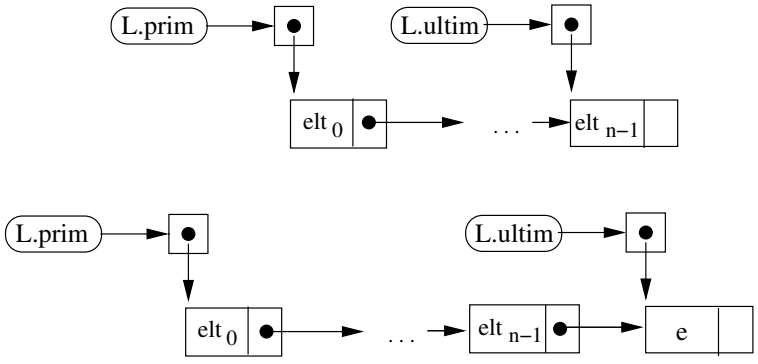


Figura 2.12: Adăugarea unui nod la sfârşitul unei liste

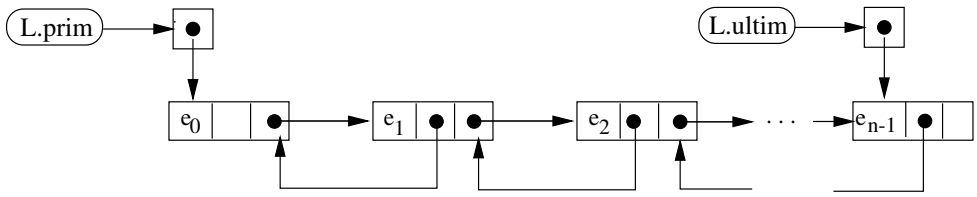


Figura 2.13: Listă liniară dublu înlănţuită

```
begin
  if (L ≠ NULL)
  then v ← L.prim  /* lista nevida */
       L.prim ← L.prim->succ
       if (L.ultim = v) then L.ultim ← NULL
       delete v
end
```

- ştergerea nodului de la sfârşit:

```
procedure stLaSf(L, e)
begin
  if (L ≠ NULL)
  then v ← L.prim     /* lista nevida */
       if (L.ultim = v)       /* un singur nod */
       then L.prim ← NULL
            L.ultim ← NULL
            delete v
       else /* mai multe noduri */
            /* determina penultimul */
            while (v->succ ≠ NULL) do
               v ← v->succ
            L.ultim ← v
            delete v->succ
end
```

Această structură va fi utilizată la reprezentarea obiectelor de tip dată a tipurilor de date de nivel înalt din capitolul 3.

2.1.6.5 Liste liniare dublu înlănţuite

Listele liniare dublu înlănţuite sunt asemănătoare celor simplu înlănţuite cu deosebirea că, în plus, fiecare nod, exceptând primul, "cunoaşte" adresa nodului din faţa sa (nodul predecesor). Astfel, un nod `v` are un câmp în plus `v->pred` care memorează adresa nodului predecesor. O listă liniară dublu înlănţuită este reprezentată grafic ca în fig. 2.13.

2.1.7 Calculul unui program

Intuitiv, calculul (execuţia) unui program constă în succesiunea de paşi elementari determinaţi de execuţiile instrucţiunilor ce compun programul. Fie, de exemplu, următorul program:

```
x ← 0
i ← 1
while (i < 10) do
   x ← x*10+i
   i ← i+2
```

Calculul descris de acest program ar putea fi descris de următorul tabel:

Pasul	Instrucţiunea	i	x
0	`x` ← `0`	–	–
1	`i` ← `1`	–	0
2	`x` ← `x*10+i`	1	0
3	`i` ← `i+2`	1	1
4	`x` ← `x*10+i`	3	1
5	`i` ← `i+2`	3	13
6	`x` ← `x*10+i`	5	13
7	`i` ← `i+2`	5	135
8	`x` ← `x*10+i`	7	135
9	`i` ← `i+2`	7	1357
10	`x` ← `x*10+i`	9	1357
11	`i` ← `i+2`	9	13579
12		11	13579

Acest calcul este notat formal printr-o secvenţă $c_0 \vdash c_1 \vdash \cdots \vdash c_{12}$. Prin c_i am notat configuraţiile ce intervin în calcul. O *configuraţie* include instrucţiunea curentă (starea programului) şi starea memoriei (valorile curente ale variabilelor din program). În exemplul de mai sus, o configuraţie este reprezentată de o linie în tabel. Relaţia $c_{i-1} \vdash c_i$ are următoarea semnificaţie: prin execuţia instrucţiunii din c_{i-1}, se transformă c_{i-1} în c_i. c_0 se numeşte *configuraţie iniţială* iar c_{12} *configuraţie finală*. Notăm că pot exista şi calcule infinite. De exemplu instrucţiunea

```
while (true) do
   i ← i+1
```

generează un calcul infinit.

2.1.8 Exerciţii

Exerciţiul 2.1.1. O *secţiune* a tabloului `a`, notată `a[i..j]`, este formată din elementele `a[i]`, `a[i` + `1]`, ..., `a[j]`, $i \leq j$. *Suma* unei secţiuni este suma elementelor sale. Să se scrie un program care, aplicând tehnica de la problema platourilor [Luc93], determină secţiunea de sumă maximă.

Exerciţiul 2.1.2. Să se scrie o funcţie care, pentru un tablou`b`, determină valoarea predicatului:

$$P \equiv \forall i, j : 1 \leq i < j \leq n \Rightarrow b[i] \leq b[j]$$

Să se modifice acest subprogram astfel încât să ordoneze crescător elementele unui tablou.

Exerciţiul 2.1.3. Să se scrie un subprogram tip funcţie care, pentru un tablou de valori booleene `b`, determină valoarea predicatului:

$$P \equiv \forall i, j : 1 \leq i \leq j \leq n \Rightarrow b[i] \Rightarrow b[j])$$

Exerciţiul 2.1.4. Se consideră tabloul `a` ordonat crescător şi tabloul `b` ordonat descrescător. Să se scrie un program care determină cel mai mic x, când există, ce apare în ambele tablouri.

Exerciţiul 2.1.5. Se consideră două tablouril `a` şi `b`. Ambele tablouri au proprietatea că oricare două elemente sunt distincte. Să se scrie un program care determină elementele x, când există, ce apar în ambele tablouri. Să se compare complexitatea timp acestui algoritm cu cea a algoritmului din 2.1.4. Ce se poate spune despre complexităţile celor două probleme?

Exerciţiul 2.1.6. Să se proiecteze structuri pentru reprezentarea punctelor şi respectiv a dreptelor din plan. Să se scrie subprograme care să rezolve următoarele probleme:

(i) Apartenenţa unui punct la o dreaptă:

Instanţă	O dreaptă d şi un punct P.
Întrebare	$P \in d$?

(ii) Intersecţia a două drepte:

Intrare	Două drepte d_1 şi d_2.
Ieşire	$d_1 \cap d_2$.

(iii) Test de perpendicularitate:

Instanţă	Două drepte d_1 şi d_2.
Întrebare	$d_1 \perp d_2$?

(iv) Test de paralelism:

Instanţă	Două drepte d_1 şi d_2.
Întrebare	$d_1 \| d_2$?

Exerciţiul 2.1.7. Să se proiecteze o structură pentru reprezentarea numerelor complexe. Să se scrie subprograme care realizează operaţii din algebra numerelor complexe.

Exerciţiul 2.1.8. Presupunem că numerele complexe sunt reprezentate ca în soluţia exerciţiului 2.1.7. Un polinom cu coeficienţi complecşi poate fi reprezentat ca un tablou de articole. Să se scrie un subprogram care, pentru un polinom cu coeficienţi complecşi P şi un număr complex z date, calculează $P(z)$.

Exerciţiul 2.1.9. O fişă într-o bibliotecă poate conţine informaţii despre o carte sau o revistă. Informaţiile care interesează despre o carte sunt: autor, titlu, editură şi an apariţie, iar cele despre o revistă sunt: titlu, editură, an, volum, număr. Să se definească un tip de date articol cu variante pentru reprezentarea unei fişe.

Exerciţiul 2.1.10. Să se proiecteze o structură de date pentru reprezentarea datei calendaristice. Să se scrie subprograme care realizează următoarele:

(i) Decide dacă valoarea unei variabile din structură conţine o dată validă.

(ii) Având la intrare o dată calendaristică oferă la ieşire data calendaristică următoare.

(iii) Având la intrare o dată calendaristică oferă la ieşire data calendaristică precedentă.

Exerciţiul 2.1.11. Să se scrie tipurile şi instrucţiunile care construiesc listele înlănţuite din fig. 2.14. Se va utiliza cel mult o variabilă referinţă suplimentară.

Exerciţiul 2.1.12. Să se scrie un subprogram care inversează sensul legăturilor într-o listă simplu înlănţuită astfel încât primul nod devine ultimul şi ultimul nod devine primul.

Exerciţiul 2.1.13. Se consideră un tablou unidimensional de dimensiune mare care conţine elemente alocate (ocupate) şi elemente disponibile. Ne putem imagina că acest tablou reprezintă memoria unui calculator (element al tabloului = cuvânt de memorie). Zonele ocupate (sau zonele libere) din tablou sunt gestionate cu ajutorul unei liste înlănţuite. Asupra tabloului se execută următoarele operaţii:

- *Alocă*(m) - determină o secvenţă de elemente succesive de lungime m, apoi adresa şi lungimea acesteia le adaugă ca un nou element la lista zonelor ocupate (sau o elimină din lista zonelor libere) şi întoarce adresa zonei determinate; dacă nu se poate aloca o asemenea secvenţă întoarce -1 (sau altă adresă invalidă în tablou).
- *Eliberează*(a, l) - disponibilizează secvenţa de lungime l începând cu adresa a; lista zonelor ocupate (sau a zonelor libere) va fi actualizată corespunzător.
- *Colectează* - reaşează zonele ocupate astfel încît să existe o singură zonă disponibilă (compactificarea zonelor disponibile); lista zonelor ocupate (ţsau a zonelor libere) va fi actualizată corespunzător.

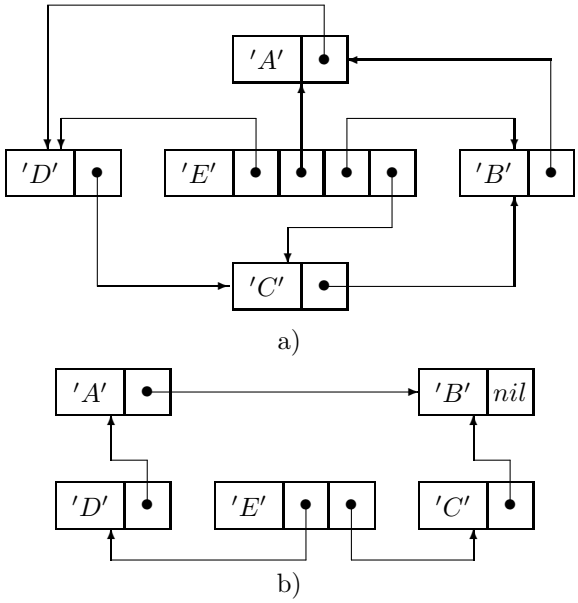


Figura 2.14:

Observaţie. Pentru funcţia de alocare se consideră următoarele două strategii:

- *FirstFit* - alocă prima zonă disponibilă de lungime $\geq m$;
- *BestFit* - alocă cea mai mică zonă de mărime $\geq m$.

Să se proiecteze structurile de date şi subprogramele care să realizeze operaţiile descrise mai sus.

Exerciţiul 2.1.14. Se consideră problema alocării memoriei din exerciţiul 2.1.13. Să se proiecteze o structură de date pentru gestionarea memoriei când toate cererile au aceeşi lungime k. Mai este necesară colectarea zonelor neutilizate? Să se scrie proceduri care alocă şi eliberează zone de memorie utilizând această structură.

2.2 Probleme şi programe

Un algoritm constituie soluţia unei probleme specifice. Descrierea algoritmului într-un limbaj algoritmic se face prin intermediul unui program. În această secţiune vom preciza condiţiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm, adică ce înseamnă că un program descrie o soluţie pentru o problemă dată.

2.2.1 Noţiunea de problemă

O problemă are două componente: domeniul, care descrie elementele care intervin în problemă şi relaţiile dintre aceste elemente, şi o întrebare despre proprietăţile anumitor elemente sau o cerinţă de determinare a unor elemente ce au o anumită proprietate. În funcţie de scopul urmărit, există mai multe moduri de a formaliza o problemă. Noi vom utiliza numai două dintre ele.

Intrare/Ieşire. Dacă privim un program ca o cutie neagră care transformă datele de intrare în date de ieşire atunci putem formaliza problema rezolvată de program ca o pereche (*Intrare*, *Ieşire*). Componenta *Intrare* descrie datele de intrare iar componenta *Ieşire* descrie datele de ieşire. Un exemplu simplu de problemă reprezentată astfel este următorul:

Intrare: Un număr întreg pozitiv x.
Ieşire: Cel mai mare număr prim mai mic decât sau egal cu x.

Problemă de decizie. Este un caz particular de problemă când ieşirea este de forma 'DA' sau 'NU'. O astfel de problemă este reprezentată ca o pereche (*Instanţă*, *Întrebare*) unde componenta *Instanţă* descrie datele de intrare iar componenta *Întrebare* se referă, în general, la existenţa unui obiect sau a unei proprietăţi. Un exemplu tipic îl reprezintă următoarea problemă:

Instanţă: Un număr întreg x.
Întrebare: Este x număr prim?

Problemele de decizie sunt preferate atât în teoria complexităţii cât şi în teoria calculabilităţii datorită reprezentării foarte simple a ieşirilor. Facem observaţia că orice problemă admite o reprezentare sub formă de problemă de decizie, indiferent de reprezentarea sa iniţială. Un exemplu de reprezentare a unei probleme de optim ca problemă de decizie este dat în secţiunea **??**.

De obicei, pentru reprezentarea problemelor de decizie se consideră o mulţime A, iar o instanţă este de forma $B \subseteq A, x \in A$ şi întrebarea de forma $x \in B$?.

2.2.2 Problemă rezolvată de un program

Convenim să considerăm întotdeauna intrările p ale unei probleme P ca fiind instanţe şi, prin abuz de notaţie, scriem $p \in P$.

Definiţia 2.1. *Fie S un program şi P o problemă. Spunem că o configuraţie iniţială c_0 a lui S* include *instanţa $p \in P$ dacă există o structură de dată* `inp`*, definită în S, astfel încât valoarea lui* `inp` *din c_0 constituie o reprezentare a lui p. Analog, spunem că o configuraţie finală c_n a unui program S* include *ieşirea $P(p)$ dacă există o structură de dată* `out`*, definită în S, astfel încât valoarea lui* `out` *din c_n constituie o reprezentare a lui $P(p)$.*

Definiţia 2.2. *(Problemă rezolvată de un program)*
1. Un program S rezolvă o problemă *P în sensul corectitudinii totale dacă pentru orice instanţă p, calculul unic determinat de configuraţia iniţială ce include p este finit şi configuraţia finală include ieşirea $P(p)$.*
2. Un program S rezolvă o problemă *P în sensul corectitudinii parţiale dacă pentru orice instanţă p pentru care calculul unic determinat de configuraţia iniţială ce include p este finit, configuraţia finală include ieşirea $P(p)$.*

Ori de câte ori spunem că un program S rezolvă o problemă P vom înţelege de fapt că S rezolvă o problemă P în sensul corectitudinii totale.

Definiţia 2.3. *(Problemă rezolvabilă/nerezolvabilă)*
1. O problemă P este rezolvabilă *dacă există un program care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este* decidabilă*.*
2. O problemă de decizie P este semidecidabilă *sau* parţial decidabilă *dacă există un program S care rezolvă P în sensul corectitudinii parţiale astfel încât calculul lui S este finit pentru orice instanţă p pentru care răspunsul la întrebare este 'DA'.*
3. O problemă P este nerezolvabilă *dacă nu este rezolvabilă, adică nu există un program care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este* nedecidabilă*.*

2.2.3 Un exemplu de problemă nedecidabilă

Pentru a arăta că o problemă este decidabilă este suficient să găsim un program care să o rezolve. Mai complicat este cazul problemelor nedecidabile. În legătură cu acestea din urmă se pun, în mod firesc, următoarele întrebări: Există probleme nedecidabile? Cum putem demonstra că o problemă nu este decidabilă? Răspunsul la prima întrebare este afirmativ. Un prim exemplu de problemă necalculabilă este cea cunoscută sub numele de *problema opririi*. Notăm cu A mulţimea perechilor de forma $\langle S, \overline{x} \rangle$ unde S este un program şi $\overline{x}$ este o intrare pentru S, iar B este submulţimea formată din acele perechi $\langle S, \overline{x} \rangle$ pentru care calculul lui S pentru intrarea $\overline{x}$ este finit. Dacă notăm prin $S(\overline{x})$ (a se citi $S(\overline{x}) = true$) faptul că $\langle S, \overline{x} \rangle \in B$, atunci problema opririi poate fi scrisă astfel:

Problema opririi
Instanţă: Un program S, $\overline{x} \in \mathbb{Z}^*$.
Întrebare: $S(\overline{x})$?

Teorema 2.1. *Problema opririi nu este decidabilă.*

Demonstraţie. Un program Q, care rezolvă problema opririi, are ca intrare o pereche $\langle S, \overline{x} \rangle$ şi se opreşte întotdeauna cu răspunsul 'DA', dacă S se opreşte pentru intrarea $\overline{x}$, sau cu răspunsul 'NU', dacă S nu se opreşte pentru intrarea $\overline{x}$. Fie Q' următorul program:

```
while  (Q(x,x) = 'DA') do
  /*nimic*/
```

Reamintim că $Q(\overline{x}, \overline{x}) = 'DA'$ înseamnă că programul reprezentat de $\overline{x}$ se opreşte pentru intrarea $\overline{x}$, adică propria sa codificare. Presupunem acum că $\overline{x}$ este codificarea lui Q'. Există două posibilităţi.

1. Q' se opreşte pentru intrarea $\overline{x}$. Rezultă $Q(\overline{x}, \overline{x}) = 'NU'$, adică programul reprezentat de $\overline{x}$ nu se opreşte pentru intrarea $\overline{x}$. Dar programul reprezentat de $\overline{x}$ este chiar Q'. Contradicţie.
2. Q' nu se opreşte pentru intrarea $\overline{x}$. Rezultă $Q(\overline{x}, \overline{x}) = 'DA'$, adică programul reprezentat de $\overline{x}$ se opreşte pentru intrarea $\overline{x}$. Contradicţie.

Aşadar, nu există un program Q care să rezolve problema opririi. sfdem

Observaţie: Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. "Există un oraş cu un bărbier care bărbiereşte pe oricine ce nu se bărbiereşte singur. Cine bărbiereşte pe bărbier?" sfobs

2.3 Măsurarea performanţelor unui algoritm

Fie P o problemă şi A un algoritm pentru P. Fie $c_0 \vdash_A c_1 \cdots \vdash_A c_n$ un calcul finit al algoritmului A. Notăm cu $t(c_i)$ timpul necesar obţinerii configuraţiei c_i din $c_{i-1},\ 1 \leq i \leq n$, şi cu $s(c_i)$ spaţiul de memorie ocupat în configuraţia $c_i,\ 0 \leq i \leq n$.

Definiţia 2.4. *Fie A un algoritm pentru problema P, $p \in P$ o instanţă a problemei P şi $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ calculul lui A corespunzător instanţei p.* Timpul *necesar algoritmului A pentru rezolvarea instanţei p este:*

$$T_A(p) = \sum_{i=1}^{n} t(c_i)$$

Spaţiul *(de memorie) necesar algoritmului A pentru rezolvarea instanţei p este:*

$$S_A(p) = \max_{0 \leq i \leq n} s(c_i)$$

În general este dificil de calculat cele două măsuri în funcţie de instanţe. Acesta poate fi simplificat astfel. Asociem unei instanţe $p \in P$ o mărime $g(p)$, care, în general, este un număr natural, pe care o numim *mărimea* instanţei p. De exemplu, $g(p)$ poate fi suma lungimilor reprezentărilor corespunzând datelor din instanţa p. Dacă reprezentările datelor din p au aceeaşi lungime, atunci se poate lua $g(p)$ egală cu numărul datelor. Astfel dacă p constă dintr-un tablou atunci se poate lua $g(p)$ ca fiind numărul de elemente ale tabloului; dacă p constă dintr-un polinom se poate lua $g(p)$ ca fiind gradul polinomului (= numărul coeficienţilor minus 1); dacă p este un graf se poate lua $g(p)$ ca fiind numărul de vârfuri sau numărul de muchii etc.

Definiţia 2.5. *Fie A un algoritm pentru problema P.*

1. *Spunem că A rezolvă P în* timpul $T_A^{fav}(n)$ *dacă:*

$$T_A^{fav}(n) = \inf\{T_A(p) \mid p \in P, g(p) = n\}$$

2. *Spunem că A rezolvă P în* timpul $T_A(n)$ *dacă:*

$$T_A(n) = \sup\{T_A(p) \mid p \in P, g(p) = n\}$$

3. *Spunem că A rezolvă P în spaţiul $S_A^{fav}(n)$ dacă:*

$$S_A^{fav}(n) = \inf\{s_a(p) \mid p \in P, g(p) = n\}$$

4. *Spunem că A rezolvă P în spaţiul $S_A(n)$ dacă:*

$$S_A(n) = \sup\{s_a(p) \mid p \in P, g(p) = n\}$$

Funcţia T_A^{fav} (S_A^{fav}) se numeşte complexitatea timp (spaţiu) a algoritmului A pentru cazul cel mai favorabil *iar funcţia T_A (S_A) se numeşte* complexitatea timp (spaţiu) a algoritmului A pentru cazul cel mai nefavorabil.

Exemplu: Considerăm problema căutării unui element într-un tablou:

Problema P
Intrare: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.
Ieşire: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Presupunem că secvenţa $(a_1, \ldots, a_n)$ este memorată în tabloul $(\texttt{a[i]} \mid 1 \leq i \leq n)$. Algoritmul descris de următorul program rezolvă P:

```
i ← 0
while (a[i] ≠ z) and (i < n-1) do
   i ← i+1
if (a[i] = z)
then poz ← i
else poz ← -1
```

Convenim să notăm acest algoritm cu A. Considerăm ca dimensiune a problemei P numărul n al elementelor din secvenţa în care se caută. Deoarece suntem cazul când toate datele sunt memorate pe câte un cuvânt de memorie, vom presupune că toate operaţiile necesită o unitate de timp. Mai întâi calculăm complexităţile timp. Cazul cel mai favorabil este obţinut când $a[0] = z$ şi se efectuează trei comparaţii şi două atribuiri. Rezultă $T_A^{fav}(n) = 3 + 2 = 5$. Cazul cel mai nefavorabil se obţine când $z \notin \{a_0, \ldots, a_{n-1}\}$ sau $z = a[n-1]$ şi în acest caz sunt executate $2n+1$ comparaţii şi $n+1$ atribuiri. Rezultă $T_A(n) = 3n+2$. Pentru simplitatea prezentării, nu au mai fost luate în considerare operaţiile `and` şi operaţiile de adunare şi scădere. Complexitatea spaţiu pentru ambele cazuri este $n + 6$. sfex

Observaţie: Complexităţile pentru cazul cel mai favorabil nu oferă informaţii relevante despre eficienţa algoritmului. Mult mai semnificative sunt informaţiile oferite de complexităţile în cazul cel mai nefavorabil: în toate celelalte cazuri algoritmul va avea performanţe mai bune sau cel puţin la fel de bune.

Pentru complexitatea timp nu este necesar totdeauna să numărăm toate operaţiile. Pentru exemplul de mai sus, observăm că operaţiile de atribuire (fără cea iniţială) sunt precedate de comparaţii. Astfel că putem număra numai comparaţiile, pentru că numărul acestora domină numărul atribuirilor. Putem merge chiar mai departe şi să numărăm numai comparaţiile între z şi componentele tabloului. sfobs

Există situaţii când instanţele p cu $g(p) = n$ pentru care $T_A(p)$ este egală cu $T_A(n)$ sau ia valori foarte apropiate de $T_A(n)$ apar foarte rar. Pentru aceste cazuri este preferabil să calculăm comportarea în medie

a algoritmului. Pentru a putea calcula comportarea în medie este necesar să privim mărimea $T_A(p)$ ca fiind o variabilă aleatoare (o experienţă = execuţia algoritmului pentru o instanţă p, valoarea experienţei = durata execuţiei algoritmului pentru instanţa p) şi să precizăm legea de repartiţie a acestei variabile aleatoare. Apoi, *comportarea în medie (complexitatea medie)* se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul complexităţii timp):

$$T_A^{med}(n) = M(\{T_A(p) \mid p \in P \wedge g(p) = n\})$$

Dacă mulţimea valorilor variabilei aleatoare $T_A(p)$ este finită , $x_1, \ldots, x_k$, şi probabilitatea ca $T_A(p) = x_i$ este p_i, atunci media variabilei aleatoare T_A (complexitatea medie) este:

$$T_A^{med}(n) = \sum_{i=1}^{k} x_i \cdot p_i$$

Exemplu: Considerăm problema P din exemplul anterior. Mulţimea valorilor variabilei aleatoare $T_A(p)$ este $\{3i+2 \mid 1 \leq i \leq n\}$. În continuare trebuie să stabilim legea de repartiţie. Facem următoarele presupuneri: probabibilitatea ca $z \in \{a_1, \ldots, a_n\}$ este q şi $poz = i$ cu probabilitatea $\frac{q}{n}$ (indicii i candidează cu aceeaşi probabilitate pentru prima apariţie a lui z). Rezultă că probabilitatea ca $z \notin \{a_1, \ldots, a_n\}$ este $1-q$. Acum probabilitatea ca $T_A(p) = 3i+2$ ($poz = i$) este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $T_A(p) = 3n+2$ este $p_n = \frac{q}{n} + (1-q)$ (probabilitatea ca $poz = n$ sau ca $z \notin \{a_1, \ldots, a_n\}$). Complexitatea medie este:

$$\begin{aligned}
T_A^{med}(n) &= \sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n-1} \frac{q}{n} \cdot (3i+2) + (\frac{q}{n} + (1-q)) \cdot (3n+2) \\
&= \frac{3q}{n} \cdot \sum_{i=1}^{n} i + \frac{q}{n} \sum_{i=1}^{n} 2 + (1-q) \cdot (3n+2) \\
&= \frac{3q}{n} \cdot \frac{n(n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\
&= \frac{3q \cdot (n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\
&= 3n - \frac{3nq}{2} + \frac{3q}{2} + 2
\end{aligned}$$

Pentru $q = 1$ (z apare totdeauna în secvenţă) avem $T_A^{med}(n) = \frac{3n}{2} + \frac{7}{2}$ şi pentru $q = \frac{1}{2}$ avem $T_A^{med}(n) = \frac{9n}{4} + \frac{11}{4}$. sfex

Exemplu: Fie P' următoarea problemă: dat un număr natural $n \leq NMax$, să se determine cel mai mic număr natural x cu proprietatea $n \leq 2^x$. P' poate fi rezolvată prin metoda căutării secvenţiale:

```
x ← 0
doilax ← 1
while (n > doilax) do
   x ← x+1
   doilax ← doilax * 2
```

Dacă se ia dimensiunea problemei egală cu n, atunci există o singură instanţă a problemei P' pentru un n fixat şi deci cele trei complexităţi sunt egale. Dacă se ia dimensiunea problemei ca fiind $NMax$, atunci cele trei complexităţi se calculează într-o manieră asemănătoare cu cea de la exerciţiul precedent. sfex

2.3.1 Calcul asimptotic

În practică, atât $T_A(n)$ cât şi $T_A^{med}(n)$ sunt dificil de evaluat. Din acest motiv se caută, de multe ori, margini superioare şi inferioare pentru aceste mărimi. Următoarele clase de funcţii sunt utilizate cu succes în stabilirea acestor margini:

$$\begin{aligned}
O(f(n)) &= \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\} \\
\Omega(f(n)) &= \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\} \\
\Theta(f(n)) &= \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0) c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}
\end{aligned}$$

Cu notaţiile O, Ω şi Θ se pot forma expresii şi ecuaţii. Considerăm numai cazul O, celelalte tratându-se similar. Expresiile construite cu O pot fi de forma:

$$O(f_1(n)) \text{ op } O(f_2(n))$$

unde "op" poate fi $+, -, *$ etc. şi notează mulţimile:

$$\{g(n) \mid (\exists g_1(n), g_2(n), c_1, c_2 > 0, n_1, n_2 > 1)((\forall n) g(n) = g_1(n) \text{ op } g_2(n) \wedge$$

$$(\forall n \geq n_1) g_1(n) \leq c_1 f_1(n) \wedge (\forall n \geq n_2) g_2(n) \leq c_2 f_2(n))\}$$

De exemplu:

$$O(n) + O(n^2) = \{f(n) = f_1(n) + f_2(n) \mid (\forall n \geq n_1) f_1(n) \leq c_1 n \wedge (\forall n \geq n_2) f_2(n) \leq c_2 n^2\}$$

Utilizând regulile de asociere, se obţin expresii de orice lungime:

$$O(f_1(n)) \text{ op}_1 O(f_2(n)) \text{ op}_2 \cdots$$

Orice funcţie $f(n)$ o putem gândi ca o notaţie pentru mulţimea cu un singur element $f(n)$ şi deci putem forma expresii de forma:

$$f_1(n) + O(f_2(n))$$

ca desemnând mulţimea:

$$\{f_1(n) + g(n) \mid (\exists c > 0, n_0 > 1)(\forall n \geq n_0) g(n) \leq c \cdot f_2(n)\}$$

Peste expresii considerăm "ecuaţii" de forma:

$$expr1 = expr2$$

cu semnificaţia că mulţimea desemnată de $expr1$ este inclusă în mulţimea desemnată de $expr2$. De exemplu, avem:

$$n \log n + O(n^2) = O(n^2)$$

pentru că $(\exists c_1 > 0, n_1 > 1)(\forall n \geq n_1) n \log n \leq c_1 n^2$, dacă $g_1(n) \in O(n^2)$ atunci $(\exists c_2 > 0, n_2 > 1)(\forall n \geq n_2) g_1(n) \leq c_2 n^2$ şi de aici $(\forall n \geq n_0) g(n) = n \log n + g_1(n) \leq n \log n + c_2 n^2 \leq (c_1 + c_2) n^2$, unde $n_0 = \max\{n_1, n_2\}$. De remarcat nesimetria ecuaţiilor: părţile stânga şi cea dreaptă joacă roluri distincte. Ca un caz particular, notaţia $f(n) = O(g(n))$ semnifică de fapt $f(n) \in O(g(n))$.

Notaţiile O, Ω şi Θ oferă informaţii cu care putem aproxima comportarea unei funcţii. Pentru ca această aproximare să aibă totuşi un grad de precizie cât mai mare, este necesar ca mulţimea desemnată de partea dreaptă a ecuaţiei să fie cât mai mică. De exemplu, avem atât $3n = O(n)$ cât şi $3n = O(n^2)$. Prin incluziunea $O(n) = O(n^2)$ rezultă că prima ecuaţie oferă o aproximare mai bună. De fiecare dată vom căuta mulţimi care aproximează cel mai bine comportarea unei funcţii.

Cu notaţiile de mai sus, două programe, care rezolvă aceeaşi problemă, pot fi comparate numai dacă au complexităţile în clase diferite. De exemplu, un algoritm A cu $T_A(n) = O(n)$ este mai eficient decât un algoritm A' cu $T_{A'}(n) = O(n^2)$. Dacă cei doi algoritmi au complexităţile în aceeaşi clasă, atunci compararea lor devine mai dificilă pentru că trebuie determinate şi constantele cu care se înmulţesc reprezentanţii clasei.

2.3.2 Complexitatea problemelor

În continuare extindem noţiunea de complexitate la cazul problemelor.

Definiţia 2.6. *Problema P are* complexitatea timp (spaţiu) $O(f(n))$ în cazul cel mai nefavorabil *dacă există un algoritm A care rezolvă problema P în timpul $T_A(n) = O(f(n))$ (spaţiul $S_A(n) = O(f(n))$).*
Problema P are complexitatea timp (spaţiu) $\Omega(f(n))$ în cazul cel mai nefavorabil *dacă orice algoritm A pentru P are complexitatea timp $T_A(n) = \Omega(f(n))$ (spaţiu $S_A(n) = \Omega(f(n))$).*
Problema P are complexitatea $\Theta(f(n))$ în cazul cel mai nefavorabil *dacă are simultan complexitatea $\Omega(f(n))$ şi complexitatea $O(f(n))$.*

Observaţie: Definiţia de mai sus necesită câteva comentarii. Pentru a arăta că o anumită problemă are complexitatea $O(f)$, este suficient să găsim un algoritm pentru P care are complexitatea $O(f(n))$. Pentru a arăta că o problemă are complexitatea $\Omega(f(n))$ este necesar de arătat că orice algoritm pentru P are complexitatea în cazul cel mai nefavorabil în clasa $\Omega(f(n))$. În general, acest tip de rezultat este dificil de arătat, dar este util atunci când dorim să arătăm că un anumit algoritm este optimal. Complexitatea unui algoritm optimal aparţine clasei de funcţii care dă limita inferioară pentru acea problemă. sfobs

2.3.3 Calculul complexităţii timp pentru cazul cel mai nefavorabil

Un algoritm poate avea o descriere complexă şi deci evaluarea sa poate pune unele probleme. De aceea prezentăm câteva strategii ce sunt aplicate în determinarea complexităţii timp pentru cazul cel mai nefavorabil. Deoarece orice algoritmul este descris de un program, în cele ce urmează considerăm S o secvenţă de program. Regulile prin care se calculează complexitatea timp sunt date în funcţie de structura lui S:

1. S este o instrucţiune de atribuire. Complexitatea timp a lui S este egală cu complexitatea evaluării expresiei din partea dreaptă.
2. S este forma:

   S_1
   S_2

   Complexitatea timp a lui S este egală cu suma complexităţilor programelor S_1 şi S_2.
3. S este de forma `if` e `then` S_1 `else` S_2. Complexitatea timp a lui S este egală cu maximul dintre complexităţile programelor S_1 şi S_2 la care se adună complexitatea evaluării expresiei e.
4. S este de forma `while` e `do` S_1. Se determină cazul când se execută numărul maxim de execuţii ale buclei `while` şi se face suma timpilor calculaţi pentru fiecare iteraţie. Dacă nu este posibilă determinarea timpilor pentru fiecare iteraţie, atunci complexitatea timp a lui S este egală cu produsul dintre maximul dintre timpii execuţiilor buclei S_1 şi numărul maxim de execuţii ale buclei S_1.

Plecând de la aceste reguli de bază, se pot obţine în mod natural reguli de calcul a complexităţii pentru toate instrucţiunile. Considerăm că obţinerea acestora constituie un bun exrciţiu pentru cititor.

2.3.4 Exerciţii

Exerciţiul 2.3.1. Să se arate că:

1. $7n^2 - 23n = \Theta(n^2)$.
2. $n! = O(n^n)$.
3. $n! = \Theta(n^n)$.
4. $5n^2 + n \log n = \Theta(n^2)$.
5. $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$.
6. $\frac{n^k}{\log n} = O(n^k)$ $(k > 0)$.
7. $n^5 + 2^n = O(2^n)$.
8. $\log_5 n = O(\log_2 n)$.

Exerciţiul 2.3.2. Să se determine complexitatea timp, pentru cazul cel mai nefavorabil, a algoritmului descris de următorul program:

```
x ← a
y ← b
z ← 1
while (y > 0) do
   if (y impar) then z ← z*x
   y ← y div 2
   x ← x*x
```

Indicaţie. Se va ţine cont de faptul că programul calculează $z = a^b \overset{\text{not}}{=} f(a,b)$ după formula

$$f(u,v) = \begin{cases} 1 & \text{, dacă } v = 0 \\ u^{v-1} * u & \text{, dacă v este impar} \\ (u^{\frac{v}{2}})^2 & \text{, dacă v este par.} \end{cases}$$

Exerciţiul 2.3.3. Să se determine complexitatea timp a algoritmului descris de următorul program:

```
x ← a
y ← b
s ← 0
while x<>0 do
   while not odd(x) do
      y ← 2*y
      x ← x div 2
   s ← s+y
   x ← x-1
```

Exerciţiul 2.3.4. Să se scrie un program care pentru un tablou unidimensional dat, determină dacă toate elementele tabloului sunt egale sau nu. Să se determine complexităţile timp pentru cazul cel mai nefavorabil şi în medie ale algoritmului descris de program.

Exerciţiul 2.3.5. Se consideră polinomul cu coeficienţi reali $p = a_0 + a_1 x + \cdots + a_n x^n$ şi punctul x_0.

a) Să se scrie un program care să calculeze valoarea polinomului p în x_0, utilizând formula:

$$p(x_0) = \sum_{i=0}^{n} a_i \cdot x_0^i$$

b) Să se îmbunătăţească programul de mai sus, utilizând relaţia $x_0^{i+1} = x_0^i \cdot x_0$.

c) Să se scrie un program care să calculeze valoarea polinomului p în x_0, utilizând formula (schema lui Horner):

$$p(x_0) = a_0 + (\cdots + (a_{n-1} + a_n \cdot x_0) \cdots)$$

Pentru fiecare dintre cele trei cazuri, să se determine numărul de înmulţiri şi de adunări şi să se compare. Care metodă este cea mai eficientă?

Exerciţiul 2.3.6. Să se scrie un program care caută secvenţial într-un tablou ordonat. Să se determine complexităţile timp pentru cazul cel mai nefavorabil şi în medie, ale algoritmului descris de program.

Capitolul 3

Tipuri de date de nivel înalt

3.1 Lista liniară

3.1.1 Tipul de date abstract LLIN

3.1.1.1 Descrierea obiectelor de tip dată

O *listă liniară* este o secvenţă finită $(e_0, \ldots, e_{n-1})$ în care elementele e_i aparţin unui tip dat Elt. Componentele unei liste nu trebuie să fie neapărat distincte; adică, putem avea $i \neq j$ şi $e_i = e_j$. *Lungimea* unei liste $L = (e_0, \ldots, e_{n-1})$, notată cu Lung$(L)$, este dată de numărul de componente din listă: Lung$(L) = n$. *Lista vidă* ( ) este lista fără nici o componentă (de lungime 0).

3.1.1.2 Operaţii.

ListaVidă. Întoarce lista vidă.

Intrare: – nimic;
Ieşire: – lista vidă ( ).

EsteVidă. Testează dacă o listă dată este vidă.

Intrare: – o listă liniară L;
Ieşire: – *true* dacă $L = ( )$,
– *false* dacă $L \neq ( )$.

Lung. Întoarce lungimea unei listei date.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$ cu $n \geq 0$;
Ieşire: – n.

Copie. Întoarce o copie distinctă a unei liste date.

Intrare: – o listă liniară L;
Ieşire: – o copie L' distinctă a lui L.

Egal. Testează dacă două liste liniare sunt egale.

Intrare: – două liste $L = (e_0, \ldots, e_{n-1})$ şi $L' = (e'_0, \ldots, e'_{n'-1})$;
Ieşire: – *true* dacă $n = n'$ şi $e_0 = e'_0, \ldots, e_{n-1} = e'_{n-1}$,
– *false*, în celelalte cazuri.

Poz. Caută dacă un element dat x apare într-o listă liniară dată L.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi un element x din Elt;
Ieşire: – adresă invalidă (-1 sau NULL) dacă x nu apare în L,
– adresa elementului e_i dacă $e_i = x$ şi $(\forall j) 0 \leq j < i \Rightarrow e_j \neq x$.

Parcurge. Parcurge o listă liniară dată efectuând o prelucrare uniformă asupra componentelor acesteia.

Intrare: – o listă liniară L şi o procedură Vizitează(e);
Ieşire: – lista L dar ale cărei componente au fost prelucrate de procedura Vizitează.

Citeşte. Întoarce elementul de la o poziţie dată dintr-o listă dată.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi o poziţie $k \geq 0$;
Ieşire: – e_k dacă $0 \leq k < n$,
– *eroare*, în celelalte cazuri.

Inserează. Adaugă un element dat într-o listă liniară dată la o poziţie dată.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$, un element e din Elt şi o poziţie $k \geq 0$;
Ieşire: – $L = (e_0, \ldots, e_{k-1}, e, e_k, \ldots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \ldots, e_{n-1}, e)$ dacă $k \geq n$.

EliminăDeLaK. Elimină elementul de la o poziţie dată dintr-o listă liniară dată.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi o poziţie $k \geq 0$;
Ieşire: – $L = (e_0, \ldots, e_{k-1}, e_{k+1}, \ldots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \ldots, e_{n-1})$ dacă $k \geq n$.

Elimină. Elimină un element dat dintr-o listă liniară dată.

Intrare: – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi un element e din Elt;
Ieşire: – lista L din care s-au eliminat toate elementele e_i egale cu e.

3.1.2 Implementarea dinamică cu liste simplu înlănţuite

3.1.2.1 Reprezentarea obiectelor de tip dată

O listă liniară $L = (e_1, \ldots, e_n)$ este reprezentată printr-o listă ca cea din fig. 3.1. Fiecare componentă a listei liniare este memorată într-un nod al listei simplu înlănţuite. Există doi pointeri `L.prim` şi `L.ultim` care fac referire la primul şi respectiv ultimul nod.

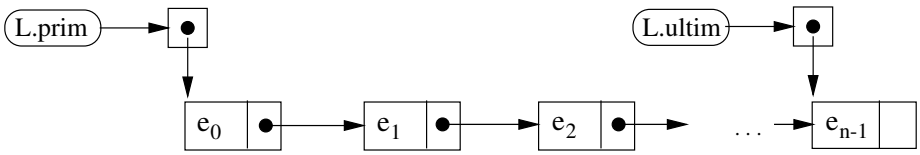


Figura 3.1: Reprezentarea listei liniare cu listă simplu înlănţuită

3.1.2.2 Implementarea operaţiilor

ListaVidă. Este reprezentată de lista fără nici un nod: `L.prim` = `L.ultim` = `NULL`.

EsteVidă. Este suficient să se testeze dacă pointerul `L.prim` este egal cu `NULL`.

Lung. Parcurge lista şi contorizează fiecare nod parcurs. Timpul necesar determinării lungimii este $O(n)$. Acesta poate fi redus la $O(1)$ dacă lungimea ar fi inclusă în reprezentarea listei. Aceasta ar presupune ca toate operaţiile care modifică lungimea listei să actualizeze corespunzător variabila responsabilă cu memorarea lungimii.

```
function lung(L)
begin
   lg ← 0
   p ← L.prim
   while (p ≠ NULL) do
      lg ← lg+1
      p ← p->succ
   return lg
end
```

Copie. Parcurge lista sursă şi adaugă la lista destinaţie o copie a fiecărui nod parcurs. Dacă presupunem că operaţia de copiere a unui nod se face în timpul $O(t)$ atunci copierea întregii liste se realizează în timpul $O(n \cdot t)$.

```
procedure copie(L, Lcopie)
begin
   if (L.prim = NULL)
   then Lcopie.prim ← NULL
        Lcopie.ultim ← NULL
   else new(Lcopie.prim)
        Lcopie.prim->elt ← L.prim->elt
        Lcopie.ultim ← Lcopie.prim
        p ← L.prim->succ
        while (p ≠ NULL) do
           new(Lcopie.ultim->succ)
           Lcopie.ultim ← Lcopie.ultim->succ
           Lcopie.ultim->elt ← p->elt
           p ← p->succ
        Lcopie.ultim->succ ← NULL
end
```

Egal. Parcurge simultan cele două liste şi compară perechile de noduri corespunzătoare. Dacă presupunem că operaţia de comparare a unei perechi de noduri se face în timpul $O(1)$ atunci compararea listelor se realizează în timpul $O(n)$ în cazul cel mai nefavorabil, unde n este valoarea minimă dintre lungimile celor două liste.

```
function egal(L1, L2)
begin
   p1 ← L1.prim
   p2 ← L2.prim
   while ((p1 ≠ NULL) and (p2 ≠ NULL)) do
      if (p1->elt ≠ p2->elt) then return false
      p1 ← p1->succ
      p2 ← p2->succ
   if ((p1 = NULL) and (p2 = NULL))
   then return true
   else return false
end
```

Poz. Se aplică tehnica căutării secvenţiale: se pleacă din primul nod şi se interoghează nod cu nod până când este găsit primul care memorează pe x sau au fost epuizate toate nodurile. Plecând de la ipoteza

că interogarea unui nod se face în timpul $O(1)$, rezultă că operaţia de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
function poz(L, x)
begin
   p ← L.prim
   while (p ≠ NULL) do
      if (p->elt = x) then return p
      p ← p->succ
   return NULL
end
```

Parcurge. Se parcurge lista nod cu nod şi se aplică procedura Vizitează. Se obţine un grad mai mare de aplicabilitate a operaţiei dacă procedurile de tip Vizitează au ca parametru adresa unui nod şi nu informaţia memorată în nod. Dacă t este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci complexitatea timp a operaţiei de parcurgere este $O(n \cdot t)$.

```
procedure parcurge(L, viziteaza())
begin
   p ← L.prim
   while (p ≠ NULL) do
      viziteaza(p)
      p ← p->succ
end
```

Citeşte. Pentru a putea citi informaţia din cel de-al k-lea nod, este necesară parcurgerea secvenţială a primelor k-noduri. Deoarece $k < n$, rezultă că avem o complexitate în cazul cel mai nefavorabil egală cu $O(n)$.

```
function citeste(L, k)
begin
   p ← L.prim
   i ← 0
   while ((p ≠ NULL) and (i < k)) do
      i ← i + 1
      p ← p->succ
   if (p = NULL) then throw 'eroare'
   return p->elt
end
```

Inserează. Ca şi în cazul operaţiei de citire, este necesară parcurgerea secvenţială a primelor $k-1$ noduri (noul nod va fi al k-lea). Dacă avem $k = 0$ sau $k \geq \mathsf{Lung}(L)$ atunci pointerii `prim` şi respectiv `ultim` se actualizează corespunzător. Deoarece operaţia de adăugare a unui nod după o adresă dată se realizează în timpul $O(1)$, rezultă că operaţia de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure insereaza(L, k, e)
begin
   if (k < 0) then throw 'eroare'
   new(q)
   q->elt ← e
   if ((k = 0) or (L.prim = NULL))
   then q->succ ← L.prim
        L.prim ← q
        if (L.ultim = NULL) then L.ultim ← q
   else p ← L.prim
        i ← 0
        while ((p ≠ L.ultim) and (i < k-1)) do
```

```
            i ← i + 1
            p ← p->succ
         q->succ ← p->succ
         p->succ ← q
         if (p = L.ultim) then L.ultim ← q
end
```

ElimināDeLaK. Se realizează într-o manieră asemănătoare cu cea a operaţiei de inserare.

```
procedure eliminaDeLaK(L, k)
begin
   if ((k < 0) or (L.prim = NULL)) then throw 'eroare'
   if (k = 0)
   then p ← L.prim
        L.prim ← L.prim->succ
        delete(p)
   else p ← L.prim
        i ← 0
        while ((p ≠ NULL) and (i < k-1)) do
           i ← i + 1
           p ← p->succ
        if (p ≠ NULL)
        then q ← p->succ
             p->succ ← q->succ
             if (L.ultim = q) then L.ultim ← p
             delete(q)
end
```

Elimină. Se parcurge lista secvenţial şi fiecare nod care memorează un element egal cu e este eliminat. Eliminarea unui nod presupune cunoaşterea adresei nodului anterior; aceasta este determinată în timpul parcurgerii secvenţiale. Dacă se elimină primul sau ultimul nod atunci pointerii `L.prim` şi respectiv `L.ultim` se actualizează corespunzător. Operaţia de eliminare a unui nod se realizează în timpul $O(1)$. Dacă operaţia de comparare se realizează în timpul $O(1)$ atunci operaţia de eliminare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure elimina(L, e)
begin
   while ((L.prim ≠ NULL) and (L.prim->elt = e)) do
      q ← L.prim
      L.prim ← q->succ
      delete(q)
   if (L.prim = NULL)
   then L.ultim ← NULL
   else p ← L.prim
        while (p ≠ NULL) do
           q ← p->succ
           if ((q ≠ NULL) and (q->elt = e))
           then p->succ ← q->succ
                if (L.ultim = q) then L.ultim ← p
                delete(q)
           p ← p->succ
end
```

3.1.3 Implementarea statică cu tablouri

3.1.3.1 Reprezentarea obiectelor de tip dată

O listă liniară $L = (e_0, \ldots, e_{n-1})$ este reprezentată printr-un tablou `L.tab` şi o variabilă indice `L.ultim` ca în fig. 3.2. Cele n componente a listei liniare sunt memorate în primele n componente ale tabloului. Dimensiunea maximă a tabloului este MAX astfel că vor putea fi reprezentate numai liste de lungime $\leq MAX$. Pointerul indice `L.ultim` memorează adresa în tablou a ultimului element din listă; el coincide şi cu lungimea listei.

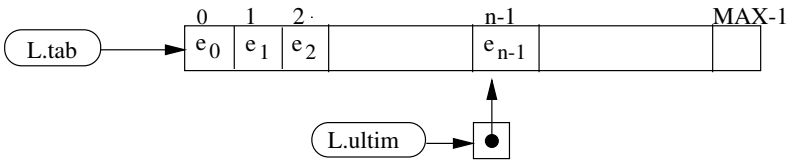


Figura 3.2: Reprezentarea listei liniare cu tablouri

3.1.3.2 Implementarea operaţiilor

ListaVidă. Este reprezentată faptul că valoarea pointerului `ultim` este -1.

EsteVidă. Se testează dacă pointerul `L.ultim` este egal cu -1.

Lung. Întoarce valoarea pointerului `L.utim` adunată cu 1. Deci este realizată în timpul $O(1)$.

```
function lung(L)
begin
   return L.ultim+1
end
```

Copie. Parcurge tabloul sursă şi copie fiecare componentă parcursă. Dacă presupunem că operaţia de copiere a unui nod se face în timpul $O(t)$ atunci copierea întregii liste se realizează în timpul $O(n \cdot t)$.

```
procedure copie(L, Lcopie)
begin
   Lcopie.ultim ← L.ultim
   for i ← 0 to L.ultim do
       Lcopie.tab[i] ← L.tab[i]
end
```

Egal. Parcurge simultan cele două tablouri şi compară perechile de componente corespunzătoare. Dacă presupunem că operaţia de comparare a unei perechi de elemente se face în timpul $O(1)$ atunci compararea listelor se realizează în timpul $O(n)$ în cazul cel mai nefavorabil, unde n este valoarea minimă dintre lungimile celor două liste.

```
function egal(L1, L2)
begin
   if (L1.ultim ≠ L2.ultim)
   then return false
   else for i ← 0 to L1.ultim do
            if (L1.tab[i] ≠ L2.tab[i]) then return false
        return true
end
```

Poz. Se aplică tehnica căutării secvenţiale: se pleacă din prima componentă a tabloului şi se interoghează componentă cu componentă până când este găsit prima care memorează pe x sau au fost epuizate toate componentele de adresă $\leq$ L.ultim. Plecând de la ipoteza că interogarea unei componente se face în timpul $O(1)$ rezultă că operaţia de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
function poz(L, x)
begin
   i ← 0
   while ((i < L.ultim) and (L.tab[i] ≠ x)) do
      i ← i+1
   if (L.tab[i] = x)
   then return i
   else return -1
end
```

Parcurge. Se parcurge tabloul componentă cu componentă şi se aplică procedura Vizitează. Ca şi în cazul implementării cu liste simplu înlănţuite, se obţine un grad mai mare de aplicabilitate a operaţiei dacă procedurile de tip Vizitează au ca parametru adresa în tablou. Dacă t este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci complexitatea timp a operaţiei de parcurgere este $O(n \cdot t)$.

```
procedure parcurge(L, viziteaza())
begin
   for i ← 0 to L.ultim do
       viziteaza(L.tab, i)
end
```

Citește. Întoarce valoarea din componenta k a tabloului. Este realizată în timpul $O(1)$.

```
function citeste(L, k)
begin
   if ((k >L.ultim) or (k < 0)) then throw 'eroare'
   return L.tab[k]
end
```

Inserează. Elementele memorate în componentele $k, k+1, \ldots, ultim$ trebuie deplasate (şiftate) la dreapta cu o poziţie pentru a face componenta k liberă. Valoarea pointerului ultim este incrementată cu o unitate. Cum k poate fi şi 1, rezultă că operaţia de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure insereaza(L, k, e)
begin
   if (k < 0) then throw 'eroare'
   if (L.ultim = MAX-1) then throw 'memorie insuficienta'
   L.ultim ← L.ultim+1
   if (k ≥ L.ultim)
   then L.tab[L.ultim] ← e
   else for i ← L.ultim downto k+1 do
            L.tab[i] ← L.tab[i-1]
        L.tab[k] ← e
end
```

ElimināDeLaK. Elementele memorate în componentele $k, k+1, \ldots, ultim$ trebuie deplasate (şiftate) la stânga cu o poziţie. Valoarea pointerului ultim este decrementată cu o unitate. Complexitatea timp în cazul cel mai nefavorabil este $O(n)$.

```
procedure eliminaDeLaK(L, k)
begin
   if (k < 0) then throw 'eroare'
```

```
   if (L.ultim = -1) then throw 'lista vida'
   L.ultim ← L.ultim-1
   for i ← k to L.ultim do
       L.tab[i] ← L.tab[i+1]
end
```

Elimină. Se parcurge tabloul secvenţial şi fiecare componentă care memorează un element egal cu e este eliminată prin deplasarea la stânga a elementelor memorate după ea. La fiecare eliminare pointerul `L.ultim` se actualizează corespunzător. Operaţia de eliminare a unui nod se realizează în timpul $O(k)$, unde k este adresa în tablou a elementului. Dacă operaţia de comparare se realizează în timpul $O(1)$ atunci operaţia de eliminare necesită timpul $O(n^2)$ în cazul cel mai nefavorabil (cînd toate elementele sunt egale cu e).

```
procedure elimina(L, e)
begin
   if (k < 0) then throw 'eroare'
   k ← 0
   while (k ≤ L.ultim) do
      if (L.tab[k] = e)
      then L.ultim ← L.ultim-1
           for i ← k to L.ultim do
               L.tab[i] ← L.tab[i+1]
end
```

3.1.4 Exerciţii

Exerciţiul 3.1.1. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea polinoamelor rare cu o singură variabilă şi cu coeficienţi întregi (polinoame de grade mari cu mulţi coeficienţi egali cu zero). Să se scrie proceduri care, utilizând această structură, realizează operaţiile algebrice cu polinoame şi operaţiile de citire şi scriere de polinoame.

Exerciţiul 3.1.2. Acelaşi lucru ca la problema 3.1.1 dar considerând polinoame cu trei variabile.

Exerciţiul 3.1.3. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea întregilor mari într-o bază b. Să se scrie proceduri care realizează operaţii aritmetice cu întregi reprezentaţi astfel.

Exerciţiul 3.1.4. Fie A un alfabet. Dacă $w = a_1 \dots a_n$ este un şir din A^* atunci prin $\overline{w}$ notăm oglinditul lui w, $\overline{w} = a_n \dots a_1$. Presupunem că şirurile din A^* sunt reprezentate prin liste liniare simplu înlănţuite. Să se scrie subprograme care să realizeze:

- dat un şir w determină oglinditul său $\overline{w}$;
- dat un şir w decide dacă w este de forma $u\overline{u}$.
- date două şiruri w şi w', decide dacă $w \preceq w'$ (se consideră ordonarea lexicografică).

Exerciţiul 3.1.5. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea mulţimilor finite de numere complexe. Să se scrie proceduri care să realizeze operaţii cu mulţimi de numere complexe reprezentate astfel.

Exerciţiul 3.1.6. Să se scrie un program care să creeze o listă liniară ordonată crescător cu toate numerele de forma $2^i \cdot 3^j \cdot 5^k$ (i, j, k întregi pozitivi) mai mici decît un n dat. (În legătură cu problema lui Hamming [Luc93].)

Exerciţiul 3.1.7. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea numerelor raţionale mari şi cu multe zecimale. Să se scrie proceduri care să realizeze operaţii aritmetice cu numere raţionale reprezentate astfel.

Exerciţiul 3.1.8. Se consideră şiruri peste un alfabet A reprezentate prin liste liniare. Să se scrie un subprogram care având la intrare două şiruri $x = x_1 \dots x_m$ şi $y = y_1 \dots y_m$, $x_i, y_i \in A$, construieşte şirul $m(x, y) = x_1 y_1 \dots x_m y_m$. Să se definească o operaţie asemănătoare când x şi y au lungimi diferite şi să se scrie subprogramul care realizează această nouă operaţie.

3.2 Lista liniară ordonată

3.2.1 Tipul de date abstract LLINORD

3.2.1.1 Descrierea obiectelor de tip dată

O *listă liniară ordonată* este o listă liniară $(e_0, \ldots, e_{n-1})$ în care elementele e_i aparţin unei mulţimi total ordonate date Elt şi sunt ordonate crescător: $e_0 < \cdots < e_{n-1}$.

3.2.1.2 Operaţii

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie, Parcurge şi Poz sunt aceleaşi ca la lista liniară.

Inserează. Adaugă un element dat într-o listă liniară ordonată astfel încât după inserare lista rămîne ordonată crescător.

Intrare: – o listă liniară ordonată $L = (e_0 < \cdots < e_{n-1})$ şi un element e din Elt;
Ieşire: – $L = (e_0 < \cdots < e_{k-1} < e < e_k < \cdots < e_{n-1})$.

Elimină. Elimină un element dat dintr-o listă liniară ordonată dată.

Intrare: – o listă liniară ordonată $L = (e_0 < \cdots < e_{n-1})$ şi un element e din Elt;
Ieşire: – $L = (e_0 < \cdots < e_{k-1} < e_{k+1} < \cdots < e_{n-1})$ dacă e apare în l pe locul k $(e = e_k)$,
– lista L neschimbată dacă e nu apare în L.

3.2.2 Implementarea dinamică cu liste simplu înlănţuite

3.2.2.1 Reprezentarea obiectelor de tip dată

Similară cu cea a listelor liniare.

3.2.2.2 Implementarea operaţiilor

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie și Parcurge au aceleaşi implementări ca cele de la lista liniară.

Poz. Algoritmul de căutare secvenţială poate fi îmbunătăţit în sensul următor: dacă s-a întâlnit un element $e_k > x$ atunci toate elementele care urmează sunt mai mari decît x şi procesul de căutare se termină fără succes (întoarce valoarea -1). Complexitatea timp în cazul cel mai nefavorabil rămâne $O(n)$.

```
function poz(L, x)
begin
   p ← L.prim
   while (p ≠ NULL) do
      if (p->elt = x) then return p
      if (p->elt > x) then return NULL
      p ← p->succ
   return NULL
end
```

Inserează. Se disting următoarele cazuri:

1. Lista L este vidă. Se creează un nod în care va fi memorat e. Pointerii `L.prim` şi `L.ultim` vor referi în continuare acest nod.
2. $e < L.prim$->elt. Se adaugă un nod la începutul listei şi se memorează e în acest nod. Pointerul `L.prim` va referi în continuare acest nod.

3. $e > L.ultim$->elt. Se adaugă un nod la sfîrşitul listei şi se memorează e în acest nod. Pointerul `L.ultim` va referi în continuare acest nod.

4. $L.prim$->$elt \leq e \leq L.ultim$->elt. Algoritmul de inserare are două subetape:

   4.1. Caută poziţia pe care trebuie memorat e. Dacă în timpul căutării este întîlnit un $e_k = e$ atunci procesul de inserare se termină fără a modifica lista.

   4.2. Dacă procesul de căutare s-a terminat pe prima poziţie k astfel încât $e > e_k$, atunci inserează un nod în faţa celui pe care s-a terminat procesul de căutare (acesta memorează cel mai mic element mai mare decît e) şi memorează e în acest nou nod.

Complexitatea timp în cazul cel mai nefavorabil a algoritmului este $O(n)$.

```
procedure insereaza(L, e)
begin
   new(q)
   q->elt ← e
   if (L.prim = NULL) then /* cazul 1 */
        q->succ ← NULL
        L.prim ← q
        L.ultim ← q
   else if (e < L.prim->elt) then /* cazul 2 */
        q->succ ← L.prim
        L.prim ← q
   else if (e > L.ultim->elt) then /* cazul 3 */
        q->succ ← NULL
        L.ultim->succ ← q
        L.ultim ← q
   else p ← L.prim /* cazul 4 */
        while (p->elt < e) do
           p ← p->succ
        if (p->elt ≠ e) /* subcazul 4.2 */
        then q->succ ← p->succ
             p->succ ← q
             q->elt ← p->elt
             p->elt ← e
        else delete(q)
end
```

Elimină. Caută nodul care memorează e ca la implementarea operaţiei Poz. În timpul procesului de parcurgere şi căutare se memorează într-o variabilă pointer adresa nodului ce precede nodul curent (predecesorul). Dacă a fost găsit un astfel de nod (e apare în listă), atunci îl elimină. Dacă e apare pe prima sau ultima poziţie, atunci trebui actualizaţi pointerii `L.prim` şi respectiv `L.ultim`. Complexitatea timp în cazul cel mai nefavorabil a algoritmului este $O(n)$.

```
procedure elimina(L, e)
begin
   if (L.prim = NULL) then throw 'eroare'
   p ← L.prim
   if (p->elt = e)
   then L.prim ← p->succ
        delete(p)
   else while ((p ≠ NULL) and (p->elt ≠ e)) do
           predp ← p
           p ← p->succ
        if (p ≠ NULL)
        then predp->succ ← p->succ
```

```
            if (p = L.ultim) then L.ultim ← predp
            delete(p)
end
```

3.2.3 Implementarea statică cu tablouri

3.2.3.1 Reprezentarea obiectelor de tip dată

Similară cu cea a listelor liniare.

3.2.3.2 Implementarea operaţiilor

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie şi Parcurge au aceleaşi implementări ca cele de la lista liniară.

Poz. **(Căutarea binară)**. Accesul direct la componentele listei şi ordonarea crescătoare a acestora permite o metodă de căutare foarte eficientă, cunoscută sub numele de *căutare binară*. Descrierea metodei are un caracter recursiv. Generalizăm presupunând că procesul de căutare a lui x se face cercetând componentele de la poziţia p la poziţia q. Căutarea în lista L corespunde cazului particular $p = 0$ şi $q = \texttt{lung}(L) - 1$. Există următoarele cazuri:

1. $p > q$. Subtabloul în care se caută este vid.
2. $p \leq q$. Se determină poziţia de la (cea mai apropiată de) mijlocul subtabloului:

$$m \leftarrow \left[\frac{p+q}{2}\right]$$

   2.1 $x = L.tab[m]$. Căutarea se termină cu suces: Poz $\leftarrow m$.
   2.2 $x < L.tab[m]$. Deoarece toate elementele aflate la dreapta lui m sunnt mai mari decît x rezultă că singurul loc unde avem şanse să-l găsim pe x este la stînga lui m. Se modifică limita dreaptă a subtabloului în care se caută, $q \leftarrow m - 1$, şi se reia procesul de căutare.
   2.3 $x > L.tab[m]$. Deoarece toate elementele aflate la stânga lui m sunt mai mici decît x rezultă că singurul loc unde avem şanse să-l găsim pe x este la dreapta lui m. Se modifică limita stângă a subtabloului în care se caută, $p \leftarrow m + 1$, şi se reia procesul de căutare.

Complexitatea timp în cazul cel mai nefavorabil a algoritmului este $O(\log_2 n)$. Demonstraţia acestui rezultat este prezentată în secţiunea **??**.

```
function poz(L, x)
begin
   p ← 0
   q ← L.ultim
   m ← ⌈(p+q)/2⌉
   while ((x ≠ L.tab[m]) and (p < q)) do
      if (x < L.tab[m])
      then q ← m-1
      else p ← m+1
      m ← ⌈(p+q)/2⌉
   if (x = L.tab[m])
   then return m
   else return -1
end
```

Inserează. Se realizează în doi paşi:

1 Se caută binar poziţia k unde urmează a fi memorat e.
2. Se translatează la dreapta toate elementele de la poziţia k la poziţia $L.ultim$ şi se memorează e pe poziţia k.

Dacă în timpul căutării este întîlnită o componentă ce conţine e (e este deja în listă) atunci operaţia se termină fără a modifica lista. Etapa de căutare necesită $O(\log_2 n)$ timp în cazul cel mai nefavorabil, dar operaţia de translatare necesită $O(n)$. Astfel că algoritmul care realizează operaţia de inserare are complexitatea timp în cazul cel mai nefavorabil $O(n)$.

```
procedure insereaza(L, e)
begin
   p ← 0
   q ← L.ultim
   k ← ⌈(p+q)/2⌉
   while ((e ≠ L.tab[k]) and (p < q)) do
      if (e < L.tab[k])
      then q ← k-1
      else p ← k+1
      k ← ⌈(p+q)/2⌉
   if (e ≠ L.tab[k])
   then if (e > L.tab[k]) then k ← k+1
        if (L.ultim = MAX) then throw 'memorie insuficienta'
        L.ultim ← L.ultim+1
        for i ← L.ultim downto k
            L.tab[i] ← L.tab[i-1]
        L.tab[k] ← e
end
```

Elimină. Se realizează în doi paşi:

1 Se caută binar, apelând Poz, poziţia k unde este memorat e. Dacă Poz întoarce zero atunci e nu apare în listă şi operaţia se termină fără a modifica lista.
2. Se translatează la stînga toate elementele de la poziţia $k+1$ la poziţia $L.ultim$. Ca şi în cazul inserării, algoritmul care realizează operaţia de eliminare are complexitatea timp în cazul cel mai nefavorabil $O(n)$.

```
procedure elimina(L, e)
begin
   p ← 1
   q ← L.ultim
   k ← ⌈(p+q)/2⌉
   while ((e ≠ L.tab[k]) and (p < q)) do
      if (e < L.tab[k])
      then q ← k-1
      else p ← k+1
      k ← ⌈(p+q)/2⌉
   if (e = L.tab[k])
      L.ultim ← L.ultim-1
      for i ← k to L.ultim
          L.tab[i] ← L.tab[i+1]
end
```

3.3 Stiva

3.3.1 Tipul de date abstract STIVA

3.3.1.1 Descrierea obiectelor de tip dată

Stivele sunt liste cu proprietatea că se cunoaşte "vechimea" fiecărui element în listă, i.e., se cunoaşte ordinea introducerii elementelor în listă. La orice moment, ultimul element introdus în listă este primul candidat la operaţiile de citire şi eliminare. Din acest motiv se mai numesc şi liste LIFO (Last In, First Out). Presupunem că elementele unei stive aparţin tipului abstract Elt.

3.3.1.2 Operaţii

StivaVidă. Întoarce stiva vidă.

Intrare: – nimic;
Ieşire: – stiva fără nici un element.

EsteVidă. Testează dacă o stivă este vidă.

Intrare: – o stivă S;
Ieşire: – *true* dacă S este stiva vidă,
– *false* în caz contrar.

Push. Scrie un element în stivă.

Intrare: – o stivă S şi un element e din Elt;
Ieşire: – stiva S la care s-a adăugat e. Elementul e este ultimul introdus şi va fi primul candidat la operaţiile de citire şi eliminare.

Pop. Elimină ultimul element introdus în stivă.

Intrare: – o stivă S;
Ieşire: – stiva S din care s-a eliminat ultimul element introdus dacă S nu este vidă,
– stiva vidă dacă S este stiva vidă.

Top. Citeşte ultimul element introdus într-o stivă.

Intrare: – o stivă S;
Ieşire: – ultimul element introdus în S dacă S nu este stiva vidă,
– un mesaj de eroare dacă S este stiva vidă.

Observaţie: Tipul abstract STIVA poate fi privit ca un caz particular al tipului abstract LLIN în sensul că o stivă poate fi privită ca o listă liniară şi că operaţiile stivei pot fi exprimate cu ajutorul celor de la lista liniară:

STIVA.Push(S, e) = LLIN.Inserează(S, Lungime(S), e)
STIVA.Top(S) = LLIN.Citeşte(S, Lungime(S))
STIVA.Pop(S) = LLIN.Elimină(S, Lungime(S))

Astfel, o stivă poate fi reprezentată printr-o secvenţă $(e_1, \ldots, e_n)$ unde e_n este elementul din vârful stivei (cu vechimea cea mai mică). Ca o consecinţă, obţinem următoarea interpretare pentru operaţii:

StivaVidă : $\mapsto ()$
Push : $((e_0, \ldots, e_{n-1}), e) \mapsto (e_0, \ldots, e_{n-1}, e)$
Pop : $(e_0, \ldots, e_{n-1}) \mapsto (e_0, \ldots, e_{n-2})$
Pop : $() \mapsto eroare$
Top : $(e_0, \ldots, e_{n-1}) \mapsto e_{n-1}$
Top : $() \mapsto eroare$

sfobs

3.3.2 Implementarea dinamică cu liste simplu înlănţuite

3.3.2.1 Reprezentarea obiectelor de tip dată

O stivă constă dintr-o pereche formată din o listă simplu înlănţuită, care conţine elementele memorate în stivă la un moment dat, şi o variabilă referinţă care indică vârful stivei, adică ultimul element introdus (fig. 3.3).

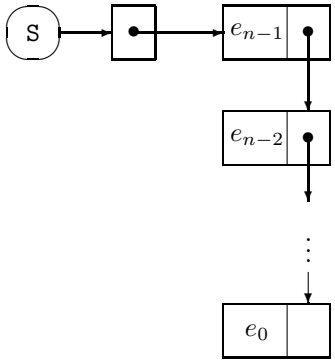


Figura 3.3: Stiva modelată ca listă înlănţuită

3.3.2.2 Implementarea operaţiilor

StivaVidă. Constă în crearea listei simplu înlănţuite vide.

EsteVidă. Testează dacă vârful S este egal cu `NULL`.

Push. Constă în adăugarea unui nod la începutul listei şi "mutarea" vîrfului S la nodul adăugat. Implementarea operaţiei necesită timpul $O(1)$.

```
procedure push(S, e)
begin
   new(q)
   q->elt ← e
   q->succ ← S
   S ← q
end
```

Pop. Constă în eliminarea nodului referit de pointerul S şi "mutarea" vîrfului la următorul nod. Operaţia este descrisă de o funcţie booleană care întoarce *true* dacă şi numai dacă stiva era nevidă înainte de eliminare. Ca şi implementarea operaţiei Push, necesită timpul $O(1)$.

```
procedure pop(S)
begin
   if (S = NULL) then throw 'eroare'
   q ← S
   S ← S->succ
   delete(q)
end
```

Top. Dacă stiva nu este vidă, furnizează informaţia memorată în nodul referit de vârful S.

```
function top(S)
begin
   if (S = NULL) then throw 'eroare'
```

```
   return S->elt
end
```

3.3.3 Implementarea statică cu tablouri

3.3.3.1 Reprezentarea obiectelor de tip dată

O stivă `S` este o pereche formată dintr-un tablou `S.tab`, care memorează elementele stivei, şi o variabilă indice (vârful) `S.varf` care indică poziţia ultimului element introdus în stivă (fig. 3.4).

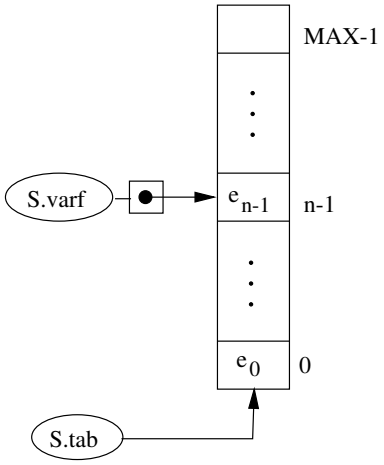


Figura 3.4: Stiva modelată ca tablou

3.3.3.2 Implementarea operaţiilor

StivaVidă. Vârful, care memorează atât adresa ultimului element introdus cât şi numărul elementelor existente în stivă, este pus pe zero.

EsteVidă. Testează dacă vârful este zero.

Push. Incrementează vârful şi memorează noul element în tablou la noua adresă dată de vârf. Deoarece dimensiunea maximă a tabloului care memorează stiva este fixă, este necesar să se verifice dacă nu se depăşeşte această limită. Operaţia este realizată în timpul $O(1)$.

```
procedure push(S, e)
begin
   if (S.varf = MAX) then throw 'memorie insuficienta'
   S.varf ← S.varf+1
   S.tab[S.varf] ← e
end
```

Pop. Dacă vârful este mai mare ca zero atunci îl decrementează. Evident, realizarea operaţiei se face în timpul $O(1)$.

```
procedure pop(S)
begin
   if (S.varf = 0) then throw 'eroare'
   S.varf ← S.varf-1
end
```

Top. Dacă vârful este mai mare ca zero atunci întoarce elementul memorat în tablou la adresa dată de vârf.

```
function top(S)
begin
   if (S.varf = 0) then throw 'eroare'
   return S.tab[S.varf]
end
```

3.3.4 Exerciţii

Exerciţiul 3.3.1. [?] Se consideră o maşină compusă din trei memorii: o secţiune de intrare şi o secţiune de ieşire constând dintr-o secvenţă de n celule fiecare (o celulă poate memora un număr întreg) şi o stivă cu memorie infinită. Controlul finit al maşinii este format dintr-un program construit cu următoarele instrucţiuni:

`Read` - introduce valoarea din prima celulă din secţiunea de intrare în stivă apoi deplasează conţinutul secţiunii de intrare astfel încât conţinutul din celula $k+1$ trece în celula k. Conţinutul din prima celulă se pierde iar cel din ultima celulă devine nedefinit;

`Write` - deplasează conţinutul secţiunii de ieşire astfel încât conţinutul din celula k trece în celula $k+1$. Conţinutul din ultima celulă se pierde apoi, extrage un element din stivă şi-l memorează în prima celulă din secţiunea de ieşire.

a) Presupunând că secţiunea de intrare conţine n valori distincte, să se determine numărul a_n al permutărilor ce pot fi obţinute în secţiunea de ieşire.

b) Să se scrie un program care listează toate programele maşinilor ce realizează permutări.

3.4 Coada

3.4.1 Tipul de date abstract COADA

3.4.1.1 Descrierea obiectelor de tip dată

Ca şi stivele, *cozile* sunt liste cu proprietatea că se cunoaşte vechimea fiecărui element în listă. La orice moment, primul element introdus în listă este primul candidat la operaţiile de citire şi eliminare. Din acest motiv se mai numesc şi liste FIFO (First In, First Out). Presupunem că elementele unei stive aparţin tipului abstract Elt.

3.4.1.2 Operaţii

CoadaVidă. Întoarce coada vidă.

Intrare: – nimic;
Ieşire: – coada fără nici un element.

EsteVidă. Testează dacă o coadă este vidă.

Intrare: – o coadă C;
Ieşire: – *true* dacă C este coada vidă,
– *false* în caz contrar.

Inserează. Scrie un element în coadă.

Intrare: – o coadă C şi un element e din Elt;
Ieşire: – coada C la care s-a adăugat e. Elementul e este ultimul introdus şi va fi ultimul candidat la operaţiile de citire şi eliminare.

Elimină. Elimină primul element introdus în coadă.

Intrare: – o coadă C;
Ieşire: – coada C din care s-a eliminat primul element introdus dacă C nu este vidă,
– coada vidă dacă C este coada vidă.

Citeşte. Citeste primul element introdus în coadă.

Intrare: – o coadă C;
Ieşire: – primul element introdus în C dacă C nu este coada vidă,
– un mesaj de eroare dacă C este coada vidă.

Observaţie: Ca şi în cazul stivei, tipul abstract COADA poate fi privit ca un caz particular al tipului abstract LLIN în sensul că o coadă este privită ca o listă liniară şi că operaţiile cozii pot fi exprimate cu ajutorul celor de la lista liniară:

$$\mathsf{COADA.Inserează(C, e)} = \mathsf{LLIN.Inserează(C, Lungime(C), e)}$$
$$\mathsf{COADA.Citeşte(C)} = \mathsf{LLIN.Citeşte(C, 0)}$$
$$\mathsf{COADA.Elimină(C)} = \mathsf{LLIN.Elimină(S, 0)}$$

Astfel, o coadă poate fi reprezentată printr-o secvenţă $(e_1, \ldots, e_n)$, unde e_1 reprezintă capul cozii (elementul cu vechimea cea mai mare) iar e_n este elementul de la sfârşitul cozii (cu vechimea cea mai mică). Utilizând reprezentarea prin secvenţe, operaţiile se rescriu astfel:

$\mathsf{CoadaVidă} : \mapsto ()$
$\mathsf{Inserează} : \langle (e_0, \ldots, e_{n-1}), e \rangle \mapsto (e_0, \ldots, e_{n-1}, e)$
$\mathsf{Citeşte} : (e_0, \ldots, e_{n-1}) \mapsto e_0$
$\mathsf{Citeşte} : () \mapsto eroare$
$\mathsf{Elimină} : (e_0, \ldots, e_{n-1}) \mapsto (e_1, \ldots, e_{n-1}$
$\mathsf{Elimină} : () \mapsto eroare$

sfobs

3.4.2 Implementarea dinamică cu liste simplu înlănţuite

3.4.2.1 Reprezentarea obiectelor de tip dată

O coadă este formată dintr-o listă simplu înlănţuită C, care conţine elementele memorate la un moment dat în coadă, o variabilă referinţă `C.prim` care face referire la nodul cu vechimea cea mai mare (candidatul la ştergere şi la interogare) şi o variabilă referinţă `C.ultim` care face referire la ultimul nod introdus în coadă (nodul cu vechimea cea mai mică) (fig. 3.5).

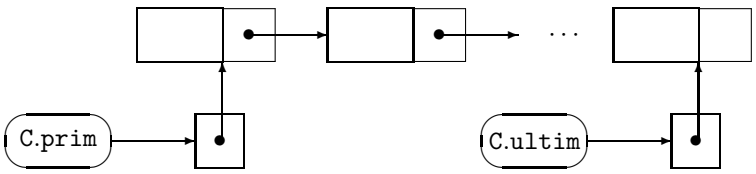


Figura 3.5: Coada reprezentată ca listă simplu înlănţuită

3.4.2.2 Implementarea operaţiilor

CoadaVidă. Creează lista simplu înlănţuită vidă.

EsteVidă. Testează dacă pointerul `C.prim` este egal cu `NULL`.

Citește. Dacă lista nu este vidă, furnizează informaţia din nodul cu vechimea cea mai mare (referit de C.prim).

```
function citeste(C)
begin
   if (C.prim = NULL) then throw 'eroare'
   return C.prim->elt
end
```

Inserează. Adaugă un nod după cel referit de C.ultim şi "mută" referinţa C.ultim la nodul nou introdus. În cazul când s-a adăugat la coada vidă, se actualizează şi referinţa C.prim.

```
procedure insereaza(C, e)
begin
   new(q)
   q->elt ← e
   q->succ ← NULL
   if (C.ultim ≠ NULL)
   then C.ultim->succ ← q
        C.ultim ← q
   else C.prim ← q
        C.ultim ← q
end
```

Elimină. În cazul în care coada nu este vidă, elimină nodul referit de C.prim (cel cu vechimea cea mai mare) şi "mută" referinţa C.prim la nodul următor. Dacă lista avea un singur nod atunci după eliminare va deveni vidă şi cei doi pointeri vor fi actualizaţi corespunzător.

```
procedure elimina(C)
begin
   if (C.prim = NULL) then throw 'eroare'
   p ← C.prim
   C.prim ← C.prim->succ
   if (C.prim = NULL) then C.ultim ← NULL
   delete(p)
end
```

3.4.3 Implementarea statică cu tablouri

3.4.3.1 Reprezentarea obiectelor de tip dată

Elementele unei cozi sunt memorate într-un tablou. Pointerul C.prim, care acum este adresă în tablou, referă cel mai "vechi" element în coadă iar pointerul C.ultim cel mai nou element în coadă (fig. 3.6). Pentru o mai bună utilizare a spaţiului de memorie vom conveni ca, atunci cînd cel mai nou element este memorat pe poziţia ultimă Max din tablou, următorul element să fie memorat pe prima poziţie din tablou, dacă aceasta este liberă (fig. 3.6b). Aceasta conduce la utilizarea unei aritmetici modulare pentru pointerii C.prim şi C.ultim. Mai rămâne de rezolvat problema reprezentării cozii vide. Aceasta devine foarte simplă dacă vom considera o variabilă contor suplimentară C.nrElem care să memoreze numărul elementelor din coadă. În plus, pentru că valoarea lui C.ultim poate fi dedusă din valorile C.prim şi C.nrElem, utlizarea câmpului C.ultim devine acum opţională.

3.4.3.2 Implementarea operaţiilor

CoadaVidă. Variabila nr_elem este pusă pe zero. Este convenabil ca pointerul C.ultim să fie făcut egal cu MAX-1 iar C.prim să fie făcut egal cu prima adresă din tablou.

EsteCoadaVidă. Testează dacă variabila C.nrElem este zero.

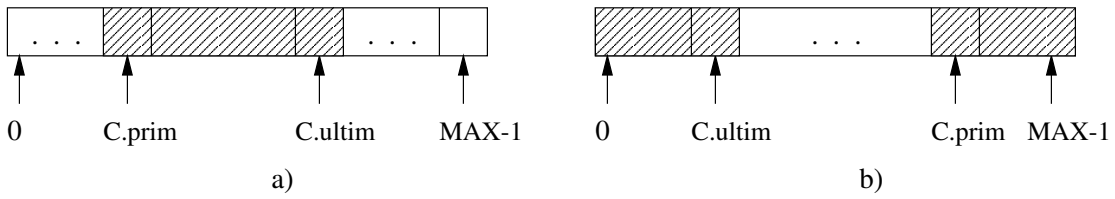


Figura 3.6: Reprezentarea cozii printr-un tablou

Citește. Dacă coada nu este vidă, furnizează informaţia de la adresa din tablou dată de pointerul `C.prim`.

```
function citeste(C)
begin
   if (C.nrElem = 0) then throw 'eroare'
   return C.tab[C.prim]
end
```

Inserează. Dacă mai este loc în tablou, `nr_elem` $<$ `Max`, atunci incrementează `C.ultim` modulo `MAX` şi memorează noul element e la noua adresă dată de `C.ultim`. Variabila contor `C.nrElem` este incrementată.

```
procedure insereaza(C, e)
begin
   if (C.nrElem = MAX) then throw 'memorie insuficienta'
   C.ultim ← (C.ultim + 1) % MAX
   C.tab[C.ultim] ← e
   C.nrElem ← C.nrElem + 1
end
```

Elimină. În cazul în care coada nu este vidă, incrementează pointerul `C.prim` modulo `MAX` şi decrementează variabila contor `C.nrElem`.

```
procedure pop(S)
begin
   if (C.nrElem = 0) then throw 'eroare'
   C.prim ← (C.prim + 1) % MAX
   C.nrElem ← C.nrElem - 1
end
```

3.4.4 Exerciţii

Exerciţiul 3.4.1. O *coadă completă cu restricţie la intrare* este o listă liniară abstractă cu proprietatea că elementele se pot insera numai la unul din capete iar ştergerile/citirile se pot efectua la oricare din capete.

(ii) Să se implementeze această structură cu ajutorul listelor înlănţuite.

(iii) Să se implementeze această structură cu ajutorul tablourilor.

Exerciţiul 3.4.2. O *coadă completă cu restricţie la ieşire* este o listă liniară abstractă cu proprietatea că elementele se pot insera la oricare din capete iar ştergerile/citirile se pot efectua numai la unul din capete.

(ii) Să se implementeze această structură cu ajutorul listelor înlănţuite.

(iii) Să se implementeze această structură cu ajutorul tablourilor.

Exerciţiul 3.4.3. O *coadă completă* ("deque") este o listă liniară abstractă cu proprietatea că elementele se pot insera, şterge şi citi la ambele capete.

(ii) Să se implementeze această structură cu ajutorul listelor înlănţuite.

(iii) Să se implementeze această structură cu ajutorul tablourilor.

Exerciţiul 3.4.4. Se consideră un tablou unidimensional cu 10000 de componente. Să se scrie subprograme care să realizeze:

- memorarea eficientă în tablou a două stive;
- memorarea în tablou a două cozi. Se pot memora două cozi la fel de eficient ca două stive? Dacă da atunci să se arate cum, dacă nu să se argumenteze de ce.

3.5 Arbori binari

3.5.1 Tipul de date abstract ABIN

3.5.1.1 Descrierea obiectelor de tip dată

Mulţimea *arborilor binari* peste Elt este definită recursiv astfel:

- arborele vid [] este un arbore binar,
- arborele t format dintr-un nod distinct, numit *rădăcină*, ce memorează un element e din Elt, şi din doi arbori binari t_1 şi t_2, numiţi *subarborele stâng* şi respectiv *subarborele drept* (ai rădăcinii), este un arbore binar; notăm acest arbore prin $[e](t_1, t_2)$.

Convenim ca arborele cu un singur nod $[e]([\,],[\,])$ să mai fie notat şi prin notaţia mai simplă $[e]$. Fie v un nod într-un arbore binar t şi v_1 şi v_2 rădăcinile subarborilor stâng şi respectiv drept ai nodului v. Atunci spunem că v este *nod tată* pentru v_1 şi v_2, v_1 este *fiul stâng* al lui v iar v_2 este *fiul drept* al lui v. Un nod pentru care ambii subarbori sunt vizi este numit *frunză*. Totalitatea nodurilor frunză formează *frontiera* arborelui. Un *drum* într-un arbore este o succesiune de noduri $v_1, \ldots, v_n$ astfel încât v_i este tatăl lui v_{i+1} pentru $i = 1, \ldots, n-1$. *Lungimea* unui drum $e_1, \ldots, e_n$ este $n-1$. *Dimensiunea* unui arbore t este egală cu numărul de noduri din t. *Înălţimea* unui arbore t este lungimea celui mai lung drum de la rădăcina lui t la un nod frunză. Prin definiţie, înălţimea arborelui vid este -1.

Exemplu: Fie $a, b, c, d, e, f, g, h, i$ nouă elemente în Elt. Aplicând de mai multe ori partea a doua a definiţiei obţinem succesiv următorii arbori binari: $[a]([\,],[d])$, $[b]([f],[i])$, $[g]([h],[\,])$, $[e]([b]([f],[i]),[g]([h],[\,]))$, $[c]([a]([\,],[d]),[e]([b]([f],[i]),[g]([h],[\,])))$. Ultimul arbore binar este reprezentat în fig. 3.7. De exemplu, e este tatăl nodurilor b şi g, b este fiul stâng al lui e iar g este fiul drept al lui e. Înălţimea arborelui este 3 iar dimensiunea sa este 9. sfex

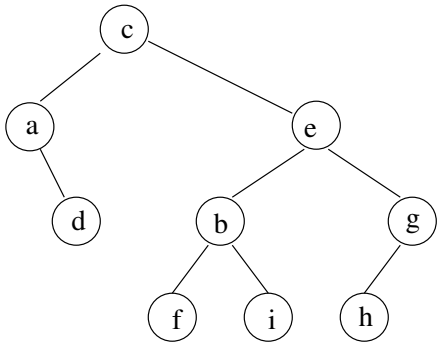


Figura 3.7: Exemplu de arbore binar

3.5.1.2 Operaţii

ArbBinVid.

Intrare: – nimic;
Ieşire: – arborele binar vid.

EsteVid.

Intrare: – un arbore binar t;
Ieşire: – *true* dacă t este vid,
– *false* în caz contrar.

Dimensiune.

Intrare: – un arbore binar t;
Ieşire: – dimensiunea lui t.

Înălţime.

Intrare: – un arbore binar t;
Ieşire: – înălţimea arborelui t.

Copie.

Intrare: – un arbore binar t;
Ieşire: – o copie distinctă a lui t.

Egal.

Intrare: – doi arbori binari t_1 şi t_2;
Ieşire: – *true* dacă cei doi arbori sunt egali, i.e., elementul memorat în rădăcina lui t_1 este egal cu cel memorat de rădăcina lui t_2 şi subarborele stâng al lui t_1 este egal cu subarborele stâng al lui t_2 şi subarborele drept al lui t_1 este egal cu subarborele drept al lui t_2. Doi arbori vizi sunt egali prin definiţie. Formal, $t_1 = t_2$ dacă şi numai dacă:

- $t_1 = [\,] = t_2$ sau
- $t_1 = [a](t_1', t_1'')$, $t_2 = [a](t_2', t_2'')$ şi $t_1' = t_2'$, $t_1'' = t_2''$.

– *false* în caz contrar.

ParcurgePreordine.

Intrare: – un arbore binar t şi o procedură Vizitează(adr_nod);
Ieşire: – arborele t dar în care nodurile au fost procesate în ordinea dată de parcurgerea preordine a lui t. Parcurgerea *preordine* a unui arbore t este obţinută aplicând recursiv următorul proces:

- vizitează rădăcina;
- vizitează subarborele stâng;
- vizitează subarborele drept.

De exemplu, parcurgerea preordine a arborelui din fig. 3.7 produce lista: $c, a, d, e, b, f, i, g, h$.

ParcurgeInordine.

Intrare: – un arbore binar t şi o procedură Vizitează(adr_nod);

Ieşire: – arborele t dar în care nodurile au fost procesate în ordinea dată de parcurgerea inordine a lui t. Parcurgerea *inordine* a unui arbore t este obţinută aplicând recursiv următorul proces:

- vizitează subarborele stâng;
- vizitează rădăcina;
- vizitează subarborele drept.

De exemplu, parcurgerea inordine a arborelui din fig. 3.7 produce lista: $a, d, c, f, b, i, e, h, g$.

ParcurgePostordine.

Intrare: – un arbore binar t şi o procedură Vizitează(adr_nod);

Ieşire: – arborele t dar în care nodurile au fost procesate în ordinea dată de parcurgerea postordine a lui t. Parcurgerea *postordine* a unui arbore t este obţinută aplicând recursiv următorul proces:

- vizitează subarborele stâng;
- vizitează subarborele drept;
- vizitează rădăcina.

De exemplu, parcurgerea postordine a arborelui din fig. 3.7 produce lista: $d, a, f, i, b, h, g, e, c$.

ParcurgeBFS.

Intrare: – un arbore binar t şi o procedură Vizitează(adr_nod);

Ieşire: – arborele t dar în care nodurile au fost procesate în ordinea dată de parcurgerea BFS a lui t. *Parcurgerea BFS* (Breadth First Search) a unui arbore t constă în: vizitarea rădăcinii, apoi a fiilor rădăcinii (nodurile aflate la distanţa 1 faţă de rădăcină), apoi a nodurilor aflate la distanţa 2 faţă de rădăcină ş.a.m.d. De exemplu, parcurgerea BFS a arborelui din fig. 3.7 produce lista: $c, a, e, d, b, g, f, i, h$.

3.5.2 Implementarea dinamică

3.5.2.1 Reprezentarea obiectelor de tip dată

Fiecare nod al arborelui binar este reprezentat printr-o structură care, în forma sa cea mai simplă, are trei câmpuri: un câmp `elt` pentru memorarea informaţiei din nod, un câmp `stg` pentru memorarea adresei rădăcinii subarborelui stâng şi un câmp `drp` pentru memorarea adresei rădăcinii subarborelui drept. Accesul la întreaga structură de date este realizat prin intermediul rădăcinii. Figura 3.8 include reprezentarea dinamică a arborelui binar din fig. 3.7.

3.5.2.2 Implementarea operaţiilor

ArbBinVid. Constă în crearea listei vide.

EsteVid. Testează dacă rădăcina t face referire la vreun nod.

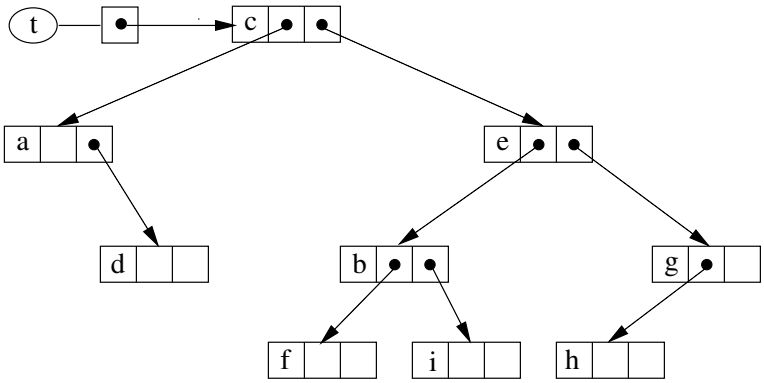


Figura 3.8: Reprezentarea dinamică a arborelui binar

Dimensiune. Descrierea funcţiei care calculează dimensiunea unui arbore binar are un caracter recursiv: dacă arborele este vid, atunci întoarce 0; dacă arborele nu este vid, atunci întoarce suma dintre dimensiunea subarborelui stâng şi dimensiunea subarborelui drept la care se adaugă 1 (rădăcina). Complexitatea timp pentru cazul cel mai nefavorabil este $O(n)$, unde n este chiar dimensiunea arborelui.

```
function dimens(t)
begin
   if (t = NULL)
   then return 0
   else d1 ← dimens(t->stg)
        d2 ← dimens(t->drp)
        return d1 + d2 + 1
end
```

Înălţime. Funcţia care calculează înălţimea are o descriere asemănătoare celei care calculează dimensiunea. Complexitatea timp pentru cazul cel mai nefavorabil este $O(n)$.

```
function inalt(t)
begin
   if (t = NULL)
   then return -1
   else h1 ← inalt(t->stg)
        h2 ← inalt(t->drp)
        return max(h1, h2) + 1
end
```

Copie. Şi această funcţie are o descriere recursivă: se realizează o copie a rădăcinii după care se realizează prin apeluri recursive copii ale subarborilor rădăciniii. Lanţul apelurilor recursive se termină atunci când este întâlnit subarborele vid. Complexitatea timp pentru cazul cel mai nefavorabil este $O(n)$.

```
function copie(t)
begin
   if (t = NULL)
   then return NULL
   else new(t_copie)
        t_copie->elt ← t->elt
        t_copie->stg ← copie(t->stg)
        t_copie->drp ← copie(t->drp)
        return t_copie
end
```

Egal. Deoarece descrierea funcţiei `egal` este foarte simplă, omitem scrierea ei.

ParcurgePreordine. Cea mai simplă descriere este cea recursivă:

```
procedure preordine(t, viziteaza())
begin
   if (t ≠ NULL)
   then viziteaza(t)
        preordine(t->stg, viziteaza)
        preordine(t->drp, viziteaza)
end
```

ParcurgeInordine. Similară parcurgerii preordine.

ParcurgePostordine. Similară parcurgerii preordine.

ParcurgeBFS. Pentru implementarea acestei operaţii se utilizează o coadă C cu următoarele proprietăţi:

1. iniţial coada conţine numai rădăcina subarborelui;
2. la momentul curent se vizitează nodul extras din C;
3. atunci când un nod este vizitat, fii acestuia sunt adăugaţi în C.

Complexitatea timp pentru cazul cel mai nefavorabil este $O(n)$.

```
procedure parcurgeBFS(t, viziteaza())
begin
   if (t ≠ NULL)
   then C ← coadaVida()
        insereaza(C,t)
        while not esteVida(C) do
           v ← citeste(C)
           elimina(C)
           viziteaza(v)
           if (t->stg ≠ NULL) then insereaza(C, t->stg)
           if (t->drp ≠ NULL) then insereaza(C, t->drp)
end
```

3.5.3 Implementarea statică cu tablouri

Nodurile arborelui binar sunt memorate într-un tablou de structuri. Acum, valorile câmpurilor de legătură nu mai sunt adrese de variabile ci adrese în tablou. O reprezentare statică a arborelui din fig. 3.7 este dată în fig. 3.9. Valoarea -1 joacă rolul pointerului NULL. Descrierile operaţiilor sunt asemănătoare cu cele de la implementarea dinamică. Prezentăm, ca exemplu, parcurgerea inordine:

```
procedure inordine(t, i, viziteaza())
begin
   if (i ≠ -1)
   then inordine(t, t.tab[i].stg, viziteaza)
        viziteaza(t, i)
        inordine(t, t.tab[i].drp, viziteaza)
end
```

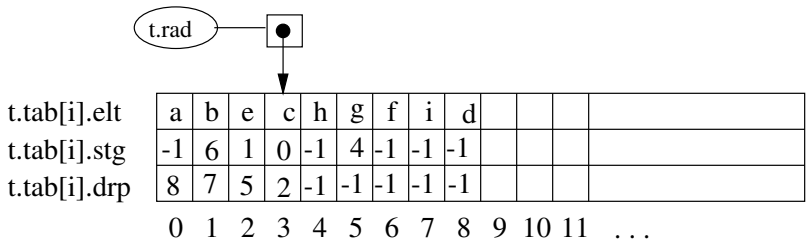


Figura 3.9: Reprezentarea statică a arborelui binar

3.5.4 Exerciţii

Exerciţiul 3.5.1. Să se scrie proceduri care implementează operaţiile pentru arborii binari reprezentaţi cu tablouri.

Exerciţiul 3.5.2. Asociem vârfurilor unui arbore binar adrese simbolice, ce sunt secvenţe de 0 şi 1 construite astfel: rădăcina are adresa simbolică 1; dacă vârful v are adresa simbolică ρ atunci fiul aflat la stânga lui v are adresa simbolică $\rho 0$ iar fiul aflat la dreapta are adresa $\rho 1$.

a) Să se scrie un program care, pentru un arbore dat, afişează adresele simbolice ale tuturor vârfurilor în preordine, inordine şi postordine.

b) Să se scrie un program care, având la intrare adresele simbolice ale tuturor vârfurilor, construieşte o listă înlănţuită care reprezintă arborele.

Exerciţiul 3.5.3. Să se scrie un program care, având la intrare listele liniare ale nodurilor unui arbore binar în preordine, respectiv în inordine, construieşte lista înlănţuită care reprezintă arborele. Mai este posibil acelaşi lucru dacă se consideră la intrare oricare altă pereche de liste (preordine şi postordine sau inordine şi postordine)?

Exerciţiul 3.5.4. Fie t un arbore binar cu n noduri, $v_1, \ldots, v_n$ lista nodurilor în preordine şi $v_{p_1}, \ldots, v_{p_n}$ lista nodurilor în inordine. Să se arate că permutarea $p_1, \ldots, p_n$ se poate obţine cu o maşină de la exerciţiul 3.3.1 şi reciproc, orice permutare obţinută cu o maşină de la exerciţiul 3.3.1 corespunde unui arbore binar.

Exerciţiul 3.5.5. Peste arborii binari se defineşte relaţia $\preceq$ astfel: $t_1 \preceq t_2$ dacă şi numai dacă:

- $t_1 = [\,]$, sau
- $t_1 = [a_1](t_1', t_1'')$, $t_2 = [a_2](t_2', t_2'')$ şi
  - $a_1 < a_2$ sau
  - $a_1 = a_2$ şi $t_1' \preceq t_2'$ sau
  - $a_1 = a_2$ şi $t_1' = t_2'$ şi $t_1'' \preceq t_2''$.

a) Să se arate că, pentru orice doi arbori t_1 şi t_2, are loc $t_1 \preceq t_2$ sau $t_2 \preceq t_1$.

b) Să se scrie un program care, având la intrare doi arbori t_1 şi t_2, decide dacă $t_1 \prec t_2$ sau $t_2 \prec t_1$ sau $t_1 = t_2$.

Exerciţiul 3.5.6. Doi arbori binari t şi t' se numesc *echivalenţi* dacă:

(i) lista inordine a lui t este $(u_1, \ldots, u_n)$,

(ii) lista inordine a lui t' este $u_1', \ldots, u_n'$, şi

(iii) informaţia din u_i şi u_i' este aceeaşi, pentru orice $i = 1, \ldots, n$.

Să se scrie o procedură care testează dacă doi arbori binari sunt echivalenţi.

Exerciţiul 3.5.7. (*Arbori binari însăilaţi la dreapta.*) Presupunem că structura unui nod cuprinde un câmp suplimentar `tdrp` cu următoarea semnificaţie:

- v->$tdrp = false$ semnifică faptul că v nu are succesor dreapta iar v->drp va conţine adresa succesorului lui v din lista inordine. Un astfel de pointer se numeşte *însăilare*.
- v->$tdrp = true$ semnifică faptul că v are succesor dreapta.

În plus, se presupune că primul nod este succesorul-inordine al ultimului nod din lista inordine. Un astfel de arbore se numeşte *însăilat la dreapta*.

a) Să se scrie un subprogram `sucInord(p, t)` care determină succesorul-inordine al unui nod arbitrar p în arborele t. Procedura va utiliza $O(1)$ spaţiu suplimentar. Care este complexitatea timp a algoritmului descris de subprogram pentru cazul cel mai nefavorabil?

b) Este posibil să se scrie o procedură `sucInord` pentru arborii binari neînsăilaţi la dreapta? Dacă da, să se arate cum, iar dacă nu să se spună de ce.

c) Să se descrie o procedură de parcurgere a arborilor binari însăilaţi la dreapta, utilizând procedura `sucInord`. Să se arate că algoritmul de parcurgere descris necesită $O(n)$ timp (n este numărul de noduri din arbore).

Exerciţiul 3.5.8. Se consideră următoarea modificare a structurii de arbore binar:

– zona de legături a fiecărui nod conţine două cîmpuri suplimentare: `tstg` şi `tdrp` cu semnificaţiile:
  – v->$tstg = true$ dacă v are subarbore stânga,
  – v->$tstg = false$ dacă v nu are subarbore stânga şi în acest caz v->$lstg$ este adresa predecesorului lui v în inordine,
  – v->$tdrp = true$ dacă v are subarbore dreapta,
  – v->$tdrp = false$ dacă v nu are subarbore dreapta şi în acest caz v->drp este adresa succesorului lui v în inordine.

Un asemenea arbore se numeşte *însăilat*.

(i) Să se scrie o procedură de parcurgere în inordine a arborilor însăilaţi.

(ii) Să se scrie o procedură care copie un arbore binar obişnuit într-un arbore însăilat.

Exerciţiul 3.5.9. Să se scrie o procedură care să numere nodurile de pe frontiera arbore binar.

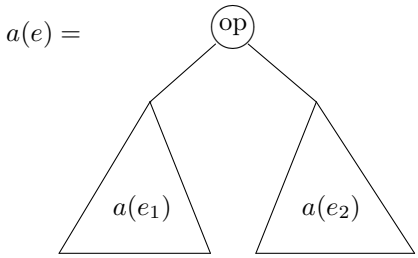
Exerciţiul 3.5.10. Se consideră arbori binar care au ca informaţie în noduri numere întregi. *Lungimea externă ponderată* a arborelui t este $\sum\{v\text{->}inf * lung(v) \mid v$ este nod pe frontiera lui $t\}$, unde $lung(v)$ este lungimea drumului de la rădăcină la nodul v. Să se scrie un subprogram care determină lungimea externă a unui astfel de arbore.

Exerciţiul 3.5.11. (*Reprezentarea expresiilor aritmetice*) Considerăm expresii formate numai din numere întregi şi operatorii binari $+$, $-$, $/, *$, %. Unei expresii e i se poate asocia un arbore ninar $a(e)$ astfel:

- dacă e este un număr întreg atunci $a(e)$ este format dintr-un singur nod $[e]$:

$$a(e) = \textcircled{e}$$

- dacă $e = e_1 \,\mathrm{op}\, e_2$ atunci $a(e)$ este arborele $[](a(e_1), a(e_2))$:



Rădăcina lui $a(e)$ are eticheta "op", subarborele din stânga rădăcinii este $a(e_1)$ iar subarborele din dreapta rădăcinii este $a(e_2)$.

Să se scrie un subprogram care are la intrare o listă liniară ce reprezintă o expresie şi oferă la ieşire arborele binar asociat expresiei. Arborele asociat este unic? Să se justifice cum rezolvă programul problema unicităţii.

Exerciţiul 3.5.12. Aşa cum am văzut în exerciţiul 3.5.11, orice expresie aritmetică poate fi reprezentată prin arbori. Notaţia *postfixată* a unei expresii se obţine prin parcurgerea în postordine a arborelui. De exemplu, expresia $a/b*2+b*d-a*c$ are următoarea formă postfixată: $a\ b\ 2\ *\ /b\ d*+a\ c*-$. Următorul algoritm realizează conversia unei expresii din notaţia infixată în notaţia postfixată (listele cu formele infixate şi respectiv infixate sunt reprezentate prin cozi):

```
S ← stivaVida()
while not esteVida(listaInfix) do
   x ← citeste(listaInfix)
   if (x este operand)
   then insereaza(listaPostfix, x)
   else if (x = ')')
        then while (top(S) ≠ '(') do
                y ← top(S)
                pop(S)
                insereaza(listaPostfix, y)
             pop(S)
        else while ((isp(top(S)) ≥ icp(x)) and not esteVida(S)) do
                y ← top(S)
                pop(S)
                insereaza(listaPostfix, y)
             push(S, x)
while (not esteVida(S)) do
   x ← top(S)
   pop(S)
   insereaza(listaPostfix, x)
```

unde `isp` şi `icp` sunt operatorii de prioritate (în stivă şi respectiv în evaluarea expresiei) şi sunt daţi prin următorul tabel:

Simbol	isp	icp
)	–	–
$*, /, \%$	2	2
$+, -$(binari)	1	1
(	0	4

Să se descrie execuţia algoritmului pentru expresia $12 + 45 * 2/6 - 23 \,\%\, 5 * 3$.

Exerciţiul 3.5.13. Pentru notaţia postfixată există un algoritm simplu de evaluare a expresiei. Acest algoritm utilizează o stivă S şi are următoarea descriere:

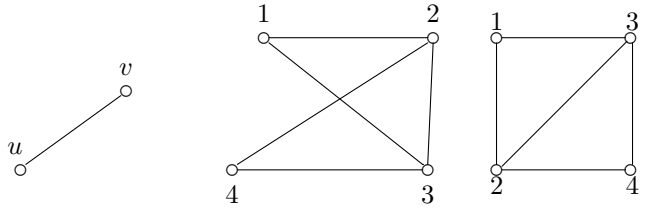
```
S ← stivaVida()
while (not esteVida(listaPostfix) do
  x ← citeste(listaPostfix)
  if (x este operand)
  then push(S, x)
  else y2 ← top(S)
       pop(S)
       y1 ← top(S)
       pop(S)
       push(S, y1 op(x) y2)
return top(S)
```

Să se descrie execuţia algoritmului pentru forma postfixată a expresiei $12 + 45 * 2/6 - 23 \,\%\, 5 * 3$.

3.6 Grafuri

3.6.1 Definiţii

Prezentarea de aici se bazează pe cea din [Cro92]. Un *graf* este o pereche $G = (V, E)$, unde V este o mulţime ale cărei elemente le numim *vârfuri*, iar E este o mulţime de perechi neordonate $\{u, v\}$ de vârfuri, pe care le numim *muchii*. Aici considerăm numai cazul când V şi E sunt finite. Dacă $e = \{u, v\}$ este o muchie, atunci u şi v se numesc extremităţile muchiei e; mai spunem că e este *incidentă* în u şi v sau că vârfurile u şi v sunt *adiacente (vecine)*. În general, muchiile şi grafurile sunt reprezentate grafic ca în fig. 3.10. Dacă G conţine muchii de forma $\{u, u\}$, atunci o asemenea muchie se numeşte *buclă*, iar graful se numeşte *graf general* sau *pseudo-graf*. Dacă pot să existe mai multe muchii $\{u, v\}$ atunci G se numeşte *multigraf*. Denumirea provine de la faptul că, în acest caz, E este o multi-mulţime. Două grafuri $G = (V, E), G' = (V', E')$ sunt *izomorfe* dacă există o funcţie $f : V \to V'$ astfel încât $\{u, v\}$ este muchie în G dacă şi numai dacă $\{f(u), f(v)\}$ este muchie în G'. Un graf $G' = (V', E')$ este *subgraf* al lui $G = (V, E)$ dacă $V \subseteq V', E \subset E'$. Un *subgraf parţial* G' al lui G este un subgraf cu proprietatea $V' = V$. Dacă G este un graf şi $X \subseteq V$ o submulţime de vârfuri, atunci *subgraful indus* de X are ca mulţime de vârfuri pe X şi mulţimea de muchii formată din toate muchiile lui G incidente în vârfuri din X.



Muchie Două reprezentări ale aceluiaşi graf

Figura 3.10: Reprezentarea grafurilor

Un *mers* de la u la v (de extremităţi u şi v) în graful G este o secvenţă

$$u = v_0, \{v_o, v_1\}, v_1, \ldots, \{v_{n-1}, v_n\}, v_n = v$$

unde $v_i, 0 \leq i \leq n$ sunt vârfuri, iar $\{v_{i-1}, v_i\}, 1 \leq i \leq n$ sunt muchii. Uneori, un mers se precizează numai prin muchiile sale sau numai prin vârfurile sale. În exemplul din fig. 3.10, secvenţa $4, \{4, 3\}, 3, \{3, 1\}, 1, \{1, 2\}, 2, \{2, 3\}, 3, \{3, 1\}, 1$ este un mers de la vârful 4 la vârful 1. Acest mers mai este notat prin $\{4, 3\}, \{3, 1\}, \{1, 2\}, \{2, 3\}, \{3, 1\}$ sau prin $4, 3, 1, 2, 3, 1$. Dacă într-un mers toate muchiile sunt distincte, atunci el se numeşte *parcurs*, iar dacă toate vârfurile sunt distincte, atunci el se numeşte *drum*. Mersul din exemplul de mai sus nu este nici parcurs şi nici drum. Un mers în care extremităţile coincid se numeşte *închis*, sau *ciclu*. Dacă într-un mers toate vârfurile sunt distincte, cu excepţia extremităţilor, se numeşte *circuit (drum închis)*. Un graf G se numeşte *conex* dacă între oricare două vârfuri ale sale există un drum. Un graf care nu este conex se numeşte *neconex*. Orice graf se poate

exprima ca fiind reuniunea disjunctă de subgrafuri induse, conexe şi maximale cu această proprietate; aceste subgrafuri sunt numite *componente conexe*. De fapt, o componentă conexă este subgraful indus de o clasă de echivalenţă, unde relaţia de echivalenţă este dată prin: vârfurile u şi v sunt echivalente dacă şi numai dacă există un drum de la u la v. Graful din fig. 3.11 are două componente conexe: $G_1 = (\{1,3,4,6\}, \{\{1,4\},\{3,6\},\{4,3\},\{6,3\}\})$ şi $G_2 = (\{2,5,7\}, \{\{2,7\},\{\{5,2\},\{\{7,5\}\})$.

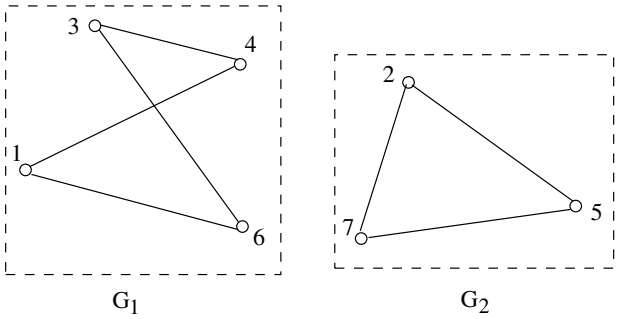


Figura 3.11: Componente conexe

Un *digraf* este o pereche $D = (V, A)$ unde V este o mulţime de vârfuri iar A este o mulţime de perechi ordonate (u, v) de vârfuri. Considerăm cazul când V şi A sunt finite. Elementele mulţimii A se numesc *arce*. Dacă $a = (u, v)$ este un arc, atunci spunem că u este *extremitatea iniţială (sursa)* a lui a şi că v este *extremitatea finală (destinaţia)* lui a; mai spunem că u este in *predecesor imediat* al lui v şi că v este un *succesor imediat* al lui u. Formulări echivalente: a este *incident din* u şi *incident în (spre)* v, sau a este *incident cu* v *spre interior* şi a este *incident cu* u *spre exterior*, sau a *pleacă din* u *şi soseşte în* v. Arcele şi digrafurile sunt reprezentate ca în fig. 3.12. Orice digraf D defineşte un graf, numit *graf suport al lui* D, obţinut prin înlocuirea oricărui arc (u, v) cu muchia $\{u, v\}$ (dacă mai multe arce definesc aceeaşi muchie, atunci se consideră o singură muchie). Graful suport al digrafului din fig. 3.12 este cel reprezentat în fig. 3.10. Mersurile, parcursurile, drumurile, ciclurile şi circuitele se definesc ca în cazul grafurilor, dar considerând arce în loc de muchii. Un digraf se numeşte *tare conex* dacă pentru orice două vârfuri u şi v, există un drum de la u la v şi un drum de la v la u. Ca şi în cazul grafurilor, orice digraf este reuniunea disjunctă de subgrafuri induse, tare conexe şi maximale cu această proprietate, pe care le numim *componente tare conexe*. O componentă tare conexă este subgraful indus de o clasă de echivalenţă, unde relaţia de echivalenţă de această dată invocă definiţia de drum într-un digraf. Digraful din fig. 3.13 are trei componente tare conexe: $G_1 = (\{1,3\}, \{(1,3),(3,1)\})$, $G_2 = (\{2\}, \emptyset)$, $G_3 = (\{4,5,6\}, \{(4,5),(5,6),(6,4)\})$. Un digraf D este *conex* dacă graful suport al lui D este conex.

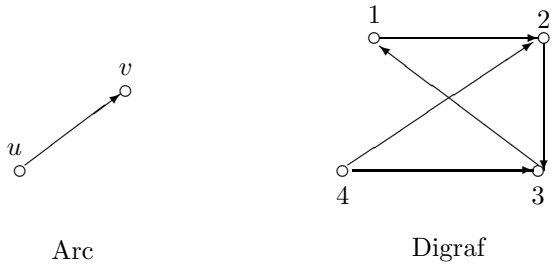


Figura 3.12: Reprezentarea digrafurilor

Un *(di)graf etichetat pe vârfuri* este un (di)graf $G = (V, E)$ împreună cu o mulţime de etichete L şi o funcţie de etichetare $\ell : V \to L$. În fig. 3.14a este reprezentat un graf etichetat, în care funcţia de etichetare este $\ell(1) = A, \ell(2) = B, \ell(3) = C$. Un *(di)graf etichetat pe muchii (arce)* este un (di)graf $G = (V, E)$ îm preună cu o mulţime de etichete L şi o funcţie de etichetare $\ell : E \to L$. În fig. 3.14b este reprezentat un graf etichetat pe arce, în care funcţia de etichetare este $\ell(\{1,2\}) = A, \ell(\{2,3\}) = B, \ell(\{3,1\}) = C$. Dacă mulţimea de etichete este $\mathcal{R}$ (mulţimea numerelor reale) atunci (di)graful se mai numeşte *ponderat*.

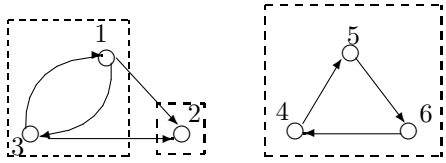


Figura 3.13: Componente tare conexe

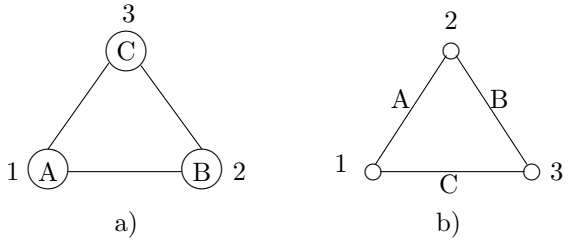


Figura 3.14: Grafuri etichetate

Un *arbore* este un graf conex fără circuite. Un *arbore cu rădăcină* este un digraf conex, fără circuite, în care există un vârf r, numit *rădăcină*, cu proprietatea că, pentru orice alt vârf v, există un drum de la r la v. Un *arbore cu rădăcină şi ordonat* este un arbore cu rădăcină cu proprietatea că orice vârf u, exceptând rădăcina, are exact un predecesor imediat şi, pentru orice vârf, mulţimea succesorilor imediaţi este total ordonată. Într-un arbore cu rădăcină ordonat, succesorii imediaţi ai vârfului u se numesc şi *fii* ai lui u, iar predecesorul imediat al lui u se numeşte *tată* lui u. Un caz particular de arbore ordonat este arborele binar, unde relaţia de ordine peste mulţimea fiilor vârfului u este precizată prin (fiul stâng, fiul drept). Exemple de arbori sunt date în fig. 3.15.

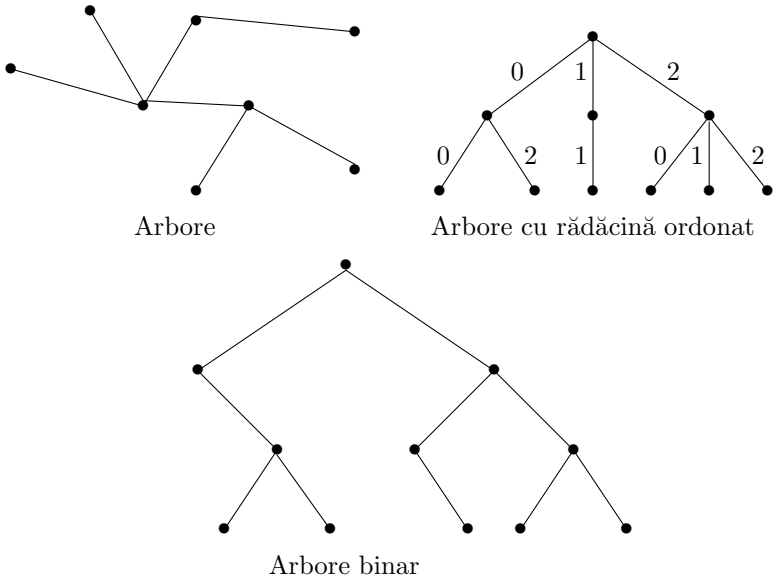


Figura 3.15: Exemple de arbori

3.6.2 Tipul de dată abstract Graf

3.6.2.1 Descrierea obiectelor de tip dată

Obiectele de tip dată sunt grafuri $G = (V, E)$, definite în mod unic până la un izomorfism, unde mulţimea de vârfuri V este o submulţime finită a tipului abstract Vârf, iar mulţimea de muchii E este o submulţime

a tipului abstract Muchie. Fără să restrângem generalitatea, presupunem că mulţimile de vârfuri V sunt mulţimi de forma $\{0, 1, \ldots, n-1\}$, unde $n = \#V$, iar mulţimile de muchii sunt submulţimi ale mulţimii $\{\{i, j\} \mid i, j \in \{0, 1, \ldots, n-1\}\}$. Dacă $G = (V, E)$ este oarecare cu $\#V = n$, atunci putem defini o funcţie bijectivă $index_G : V \to \{0, 1, \ldots, n-1\}$ şi graful $G' = (\{0, 1, \ldots, n-1\}, E')$ cu $\{index_G(u), index_G(v)\} \in E' \iff \{u, v\} \in E$. Evident, G şi G' sunt izomorfe şi deci putem lucra cu G' în loc de G. Întoarcerea de la G' la G se poate face via funcţia inversă lui $index_G$, $nume_G : \{0, 1, \ldots, n-1\} \to V$ dată prin $nume_G(i) = v \iff index_G(v) = i$.

3.6.2.2 Operaţii

GrafVid.

Intrare: – nimic;
Ieşire: – graful vid $G = (\emptyset, \emptyset)$.

EsteGrafVid.

Intrare: – un graf $G = (V, E)$;
Ieşire: – *true* dacă G este vid,
– *false* în caz contrar..

Inserează Vârf.

Intrare: – un graf $G = (V, E)$ cu $V = \{0, 1, \ldots, n-1\}$;
Ieşire: – graful G la care se adaugă vârful n ca vârf izolat (nu există muchii incidente în n).

Inserează Muchie.

Intrare: – un graf $G = (V, E)$ şi două vârfuri $i, j \in V$;
Ieşire: – graful G la care se adaugă muchia $\{i, j\}$.

Șterge Vârf.

Intrare: – un graf $G = (V, E)$ şi un vârf $k \in V$;
Ieşire: – graful G din care au fost eliminate toate muchiile incidente în k (au o extremitate în k) iar vîrfurile $i > k$ sunt redenumite ca fiind $i - 1$.

Șterge Muchie.

Intrare: – un graf $G = (V, E)$ şi două vârfuri $i, j \in V$;
Ieşire: – graful G din care a fost eliminată muchia $\{i, j\}$.

Lista De Adiacenţă.

Intrare: – un graf $G = (V, E)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor adiacente cu vârful i.

Lista Vârfurilor Accesibile.

Intrare: – un graf $G = (V, E)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor accesibile din vârful i. Un vârf j este accesibil din i dacă şi numai dacă există un drum de la i la j.

3.6.3 Tipul de dată abstract Digraf

3.6.3.1 Descrierea obiectelor de tip dată

Obiectele de tip dată sunt digrafuri $D = (V, A)$, unde mulţimea de vârfuri V o considerăm de forma $\{0, 1, \ldots, n-1\}$, iar mulţimea de arce este o submulţime a produsului cartezian $V \times V$. Tipul Vârf are aceeaşi semnificaţie ca în cazul modelului constructiv Graf, iar tipul Arc este produsul cartezian Vârf×Vârf.

3.6.3.2 Operaţii

DigrafVid.

Intrare: – nimic;
Ieşire: – digraful vid $D = (\emptyset, \emptyset)$.

EsteDigrafVid.

Intrare: – un digraf $D = (V, A)$;
Ieşire: – *true* dacă D este vid,
– *false* în caz contrar..

Inserează Vârf.

Intrare: – un digraf $D = (V, A)$ cu $V = \{0, 1, \ldots, n-1\}$;
Ieşire: – digraful D la care s-a adăugat vârful n.

Inserează Arc.

Intrare: – un digraf $D = (V, A)$ şi două vârfuri $i, j \in V$;
Ieşire: – digraful D la care s-a adăugat arcul (i, j).

Șterge Vârf.

Intrare: – un digraf $D = (V, A)$ şi un vârf $k \in V$;
Ieşire: – digraful D din care au fost eliminate toate arcele incidente în k (au o extremitate în k) iar vîrfurile $i > k$ sunt redenumite ca fiind $i - 1$.

Șterge Arc.

Intrare: – un digraf $D = (V, A)$ şi două vârfuri $i, j \in V$;
Ieşire: – digraful D din care s-a eliminat arcul (i, j).

ListaDeAdiacenţăInterioară.

Intrare: – un digraf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea surselor (vârfurilor de plecare ale) arcelor care sosesc în vârful i.

ListaDeAdiacenţăExterioară.

Intrare: – un digraf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea destinaţiilor (vârfurilor de sosire ale) arcelor care pleacă din vârful i.

ListaVârfurilorAccesibile.

Intrare: – un graf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor accesibile din vârful i. Un vârf j este accesibil din i dacă şi numai dacă există un drum de la i la j.

3.6.4 Reprezentarea grafurilor ca digrafuri

Orice graf $G = (V, E) \in$ Graf poate fi reprezentat ca un digraf $D = (V, A) \in$ Digraf considerând pentru fiecare muchie $\{i, j\} \in E$ două arce $(i, j), (j, i) \in A$. Cu aceste reprezentări, operaţiile tipului Graf pot fi exprimate cu ajutorul celor ale tipului Digraf. Astfel, inserarea/ştergerea unei muchii este echivalentă cu inserarea/ştergerea a două arce, iar celelalte operaţii coincid cu cele de la digrafuri. Această reprezentare ne va permite să ne ocupăm în continuare numai de implementări ale tipului abstract Digraf.

3.6.5 Implementarea cu matrice de adiacenţă (incidenţă)

3.6.5.1 Reprezentarea obiectelor de tip dată

Digraful D este reprezentat printr-o structură cu trei câmpuri: `D.n`, numărul de vârfuri, `D.m`, numărul de arce, şi un tablou bidimensional `D.a` de dimensiune $n \times n$ astfel încât:

$$D.a[i, j] = \begin{cases} 0 & , (i, j) \notin A \\ 1 & , (i, j) \in A \end{cases}$$

pentru $i, j = 0, \ldots, n-1$. Dacă `D` este reprezentarea unui graf atunci matricea de adiacenţă este simetrică:

$$D.a[i, j] = D.a[j, i] \quad \text{pentru orice } i, j.$$

Observaţie: În loc de valorile 0 şi 1 se pot considera valorile booleene $false$ şi respectiv $true$. sfobs

3.6.5.2 Implementarea operaţiilor

DigrafVid. Este reprezentat de orice variabilă `D` cu $D.n = 0$ şi $D.m = 0$.

esteDigrafVid. Se testează dacă `D.m` şi `D.n` sunt egale cu zero.

Inserează Vârf. Constă în adăugarea unei linii şi a unei coloane la matricea de adiacenţă.

```
procedure insereazaVarf(D)
begin
   D.n ← D.n+1
   for i ← 0 to D.n-1 do
       D.a[i,n-1] ← 0
       D.a[n-1,i] ← 0
end
```

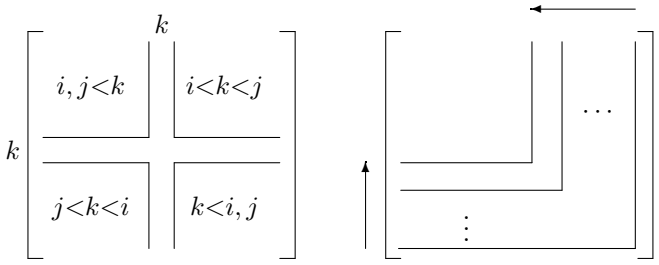
Inserează Arc. Inserarea arcului (i, j) presupune numai actualizarea componentei `D.a[i, j]`.

Şterge Vârf. Notăm cu a valoarea matricei `D.a` înainte de ştergerea vârfului k şi cu a' valoarea de după ştergere. Au loc următoarele relaţii (a se vedea şi fig. 3.16):

1. dacă $i, j < k$ atunci $a'_{i,j} = a_{i,j}$;
2. dacă $i < k \leq j$ atunci $a'_{i,j} = a_{i,j+1}$;
3. dacă $j < k \leq i$ atunci $a'_{i,j} = a_{i+1,j}$;
4. dacă $k \leq i, j$ atunci $a'_{i,j} = a_{i+1,j+1}$.

Din aceste relaţii deducem următorul algoritm:

```
procedure stergeVarf(D, k)
begin
   D.n ← D.n-1
   for i ← 0 to D.n-1 do
       for j ← 0 to D.n-1 do
           if (i ≥ k and j ≥ k) then
                D.a[i,j] ← D.a[i+1,j+1]
           else if (i ≥ k) then
                D.a[i,j] ← D.a[i+1,j]
           else if (j ≥ k) then
                D.a[i,j] ← D.a[i,j+1]
end
```



a) b)

Figura 3.16: Ştergerea unui vârf

ŞtergeArc. Asemănător operaţiei Inserează̆Arc.

ListaDeAdiacInt şi ListaDeAdiacExt. Lista vârfurilor destinatare ale arcelor care "pleacă" din i este reprezentată de linia i, iar lista vârfurilor sursă ale arcelor care sosesc în i este reprezentată de coloana i. Dacă `D` este reprezentarea unui graf atunci lista vârfurilor adiacente se obţine numai prin consultarea liniei (sau numai a coloanei) i.

ListaVârfurilorAccesibile. Reamintim că un vârf j este accesibil în D din i dacă există în D un drum de la i la j. Dacă $i = j$ atunci evident j este accesibil din i (există un drum de lungime zero). Dacă $i \neq j$ atunci există drum de la i la j dacă există arc de la i la j, sau există k astfel încât există drum de la i la k şi drum de la k la j. De fapt, cele de mai sus spun că relaţia "există drum de la i la j", definită peste V, este închiderea reflexivă şi tranzitivă a relaţiei "există arc de la i la j". Ultima relaţie este reprezentată de matricea de adiacenţă, iar prima relaţie se poate obţine din ultima prin următorul algoritm datorat lui Warshall (1962):

```
procedure detInchReflTranz(D, b)
begin
   for i ← 0 to D.n-1 do
       for j ← 0 to D.n-1 do
           b[i,j] ← D.a[i,j]
           if (i = j) then b[i,j] ← 1
   for k ← 0 to D.n-1 do
       for i ← 0 to D.n-1 do
           if (b[i,k] = 1)
           then for j ← 0 to D.n-1 do
                    if (b[k,j] = 1) then b[i,j] ← 1
end
```

Lista vârfurilor accesibile din vârful i este reprezentată acum de linia i a tabloului bidimensional `b`. În continuare vom arăta că subprogramul `detInchReflTranz` determină într-adevăr închiderea reflexivă şi

tranzitivă. Se observă uşor că reflexivitatea este rezolvată de primele două instrucţiuni `for` care realizează şi copierea `D.a` în `b`. În continuare ne ocupăm de tranzitivitate. Fie i şi j două vârfuri astfel încât j este accesibil din i. Există un k şi un drum de la i la j cu vârfuri intermediare din mulţimea $X = \{0, \ldots, k\}$. Vom arăta că după k iteraţii avem $b[i, j] = 1$. Procedăm prin inducţie după $\#X$. Dacă $X = \emptyset$, atunci $b[i, j] = D.a[i, j] = 1$. Presupunem $\#X > 0$. Rezultă că există un drum de la i la k cu vârfuri intermediare din $\{0, \ldots, k-1\}$ şi un drum de la k la j cu vârfuri intermediare tot din $\{0, \ldots, k-1\}$. Din ipoteza inductivă rezultă că după $k-1$ execuţii ale buclei `for k ...` avem $b[i, k] = 1$ şi $b[k, j] = 1$. După cea de-a k-a execuţie a buclei obţinem $b[i, j] = 1$, prin execuţia ramurii `then` a ultimei instrucţiuni `if`.

3.6.6 Implementarea cu liste de adiacenţă dinamice

3.6.6.1 Reprezentarea obiectelor de tip dată

Un digraf D este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacenţă dar unde matricea de adiacenţă este înlocuită cu un tablou unidimensional de n liste liniare, implementate prin liste simplu înlănţuite şi notate cu `D.a[i]` pentru $i = 0, \ldots, n-1$, astfel încât lista `D.a[i]` conţine vârfurile destinatare ale arcelor care pleacă din i (= lista de adiacenţă exterioară).

Exemplu: Fie G graful reprezentat în fig. 3.17a. Tabloul listelor de adiacenţă corepunzătoare lui G este reprezentat în fig. 3.17b. Pentru digraful D din figura 3.18a, tabloul listelor de adiacenţă înlănţuite este reprezentat în fig. 3.18b. sfex

De multe ori este util ca, atunci când peste mulţimea vârfurilor este dată o relaţie de ordine, componentele listelor de adiacenţă să respecte această ordine.

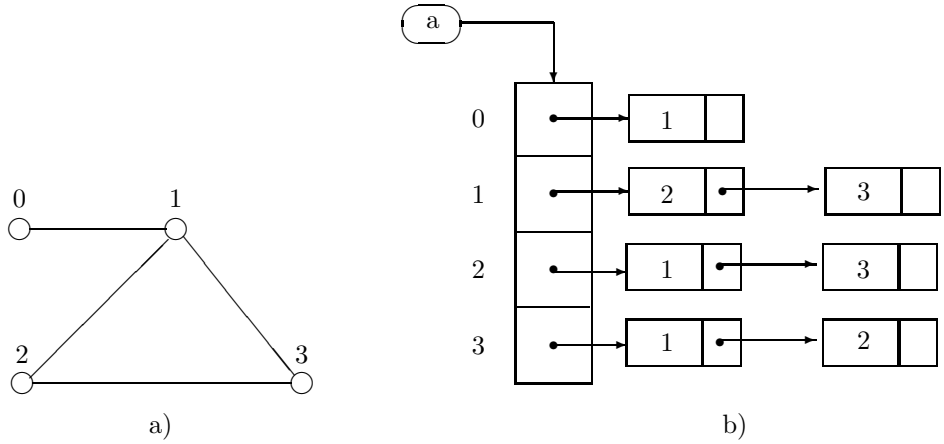


Figura 3.17: Graf reprezentat prin liste de adiacenţă înlănţuite

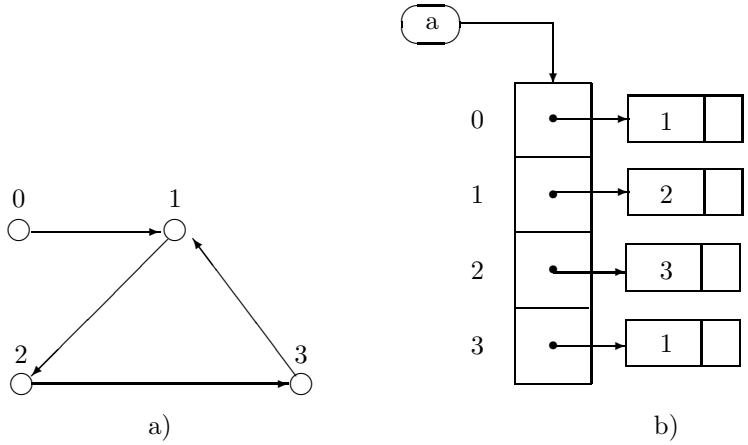


Figura 3.18: Digraf reprezentat prin liste de adiacenţă înlănţuite

3.6.6.2 Implementarea operaţiilor

DigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

EsteDigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

Inserează Vârf. Se incrementează `D.n` şi se iniţializează `D.a[n]` cu lista vidă:

```
D.n ← D.n+1
D.a[n-1] ← listaVida()
```

Inserează Arc. Adăugarea arcului (i, j) constă în inserarea unui nou nod în lista `D.a[i]` în care se memorează valoarea j:

```
insereaza(D.a[i], 0, j)
```

Complexitatea timp în cazul cel mai nefavorabil este $O(1)$.

Şterge Vârf. Ştergerea vârfului i constă în eliminarea listei `D.a[i]` din tabloul `D.a` şi parcurgerea tuturor listelor pentru a elimina nodurile care memorează i şi pentru a redenumi vârfurile $j > i$ cu $j - 1$.

```
procedure stergeVarf(D, i)
begin
   while (D.a[i].prim ≠ NULL) do
      p ← D.a[i].prim
      D.a[i].prim ← p->succ
      delete(p)
      D.a[i].ultim ← NULL
   for j ← 0 to D.n-1 do
       elimina(D.a[j], i)
   D.n ← D.n-1
   for j ← i to D.n-1 do
       D.a[j] ← D.a[i+1]
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(m)$, unde m este numărul de arce din digraf ($m \leq n^2$).

Şterge Arc. Ştergerea arcului (i, j) constă în eliminarea nodului care memorează valoarea j din lista `D.a[i]`.

```
procedure stergeArc(D, i, j)
begin
   elimina(D.a[i], j)
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(n)$.

Lista De Adiacenţă Interioară. Determinarea listei de adiacenţă interioară pentru vârful i constă în selectarea acelor vîrfuri j cu proprietatea că i apare în lista `D.a[j]`. Presupunem că lista de adiacenţă interioară este reprezentată de o coadă `C`:

```
procedure listaAdInt(D, i, C)
begin
   C ← coadaVida()
   for j ← 0 to D.n-1 do
       p ← D.a[j].prim
```

```
        while (p ≠ NULL) do
           if (p->elt = i)
           then insereaza(C, j)
                p ← NULL
           p ← p->succ
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(m)$, unde m este numărul de arce din digraf ($m \leq n^2$).

ListaDeAdiacenţăExterioară. Lista de adiacenţă exterioară a vârfului i este `D.a[i]`.

ListaVârfurilorAccesibile. Notăm cu i_0 vârful din (di)graful D pentru care se doreşte să se calculeze lista vârfurilor accesibile. Algoritmul pe care-l propunem va impune un proces de vizitare succesivă a vecinilor imediaţi lui i_0, apoi a vecinilor imediaţi ai acestora şi aşa mai departe. În timpul procesului de vizitare vor fi gestionate două mulţimi:

- S – mulţimea vârfurilor accesibile din i_0 vizitate până în acel moment,
- SB – o submulţime de vârfuri din S pentru care este posibil să existe vârfuri adiacente accesibile din i_0 nevizitate încă.

Vom utiliza, de asemenea, un tablou de pointyeri ($\texttt{p[i]} \mid 0 \leq i < n$) cu care ţinem evidenţa primului vârf nevizitat încă din fiecare listă de adiacenţă. Mai precis, dacă $p[i] \neq NULL$ şi $i \in S$ atunci $i \in SB$. Algoritmul constă în execuţia repetată a următorului proces:

– se alege un vârf $i \in SB$,

– dacă primul element neprocesat încă din lista `D.a[i]` nu este în S atunci se adaugă atât la S cât şi la SB,

– dacă lista `D.a[i]` a fost parcursă complet, atunci elimină i din SB.

Descrierea schematică a algoritmului de explorare a unui digraf este:

```
procedure explorareDigraf(D, i0, viziteaza(), S)
begin
   for  ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
   S ← {i0}
   SB ← {i0}
   viziteaza(i0)
   while (SB ≠ ∅) do
      i ← citeste(SB)
      if (p[i] = NULL)
      then SB ← SB \ {i}
      else j ← p[i]->elt
           p[i] ← p[i]->succ
           if j ∉ S
           then viziteaza(j)
                S ← S ∪ {j}
                SB ← SB ∪ {j}
end
```

Teorema 3.1. *Procedura ExplorareGraf determină vârfurile accesibile din i_0 în timpul $O(\max(n, m_0))$, unde m_0 este numărul muchiilor accesibile din i_0, şi utilizând spaţiul $O(\max(n, m))$.*

Demonstraţie. Corectitudinea rezultă din faptul că următoarea proprietate este invariantă:

S conţine o submulţime de noduri accesibile din i_0 vizitate deja, $SB \subseteq S$ şi mulţimea vârfurilor accesibile din i_0 nevizitate încă este inclusă în mulţimea vârfurilor accesibile din SB.

Complexitatea timp $O(\max(n, m_0))$ rezultă în ipoteza că operaţiile asupra mulţimilor S şi SB se realizează în timpul $O(1)$. Se observă imediat că partea de iniţializare necesită $O(n)$ timp iar bucla-while se repetă de $O(m_0)$ ori. Complexitatea spaţiu rezultă imediat din declaraţii. sfdem

Funcţie de structura de date utilizată pentru gestionarea mulţimii SB, obţinem diferite strategii de explorare a digrafurilor.

Explorarea DFS (Depth First Search). Se obţine din `ExplorareGraf` prin reprezentarea mulţimii `SB` printr-o stivă. Presupunem că mulţimea S este reprezentată prin vectorul său caracteristic:

$$S[i] = \begin{cases} 1 & \text{, dacă } i \in S \\ 0 & \text{, altfel.} \end{cases}$$

Înlocuim operaţiile scrise în dreptunghiuri din procedura `ExplorareGraf` cu operaţiile corespunzătoare stivei şi obţinem procedura `DFS` care descrie strategia DFS:

```
procedure DFS(D, i0, viziteaza(), S)
begin
   for i ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
       S[i]  ← 0
   SB ← stivaVida()
   push(SB, i0)
   S[i0] ← 1
   viziteaza(i0)
   while (not esteStivaVida(SB)) do
      i ← top(SB)
      if (p[i] = NULL)
      then pop(SB)
      else j ← p[i]->elt
           p[i] ← p[i]->succ
           if S[j] = 0
           then viziteaza(j)
                S[j] ← 1
                push(SB, j)
end
```

Notăm faptul că implementarea respectă cerinţa ca operaţiile peste mulţimile S şi SB să se execute în timpul O(1).

Exemplu: Considerăm digraful din fig. 3.20. Presupunem $i_0 = 0$. Calculul procedurii DFS este descris în tabelul din fig. 3.19. Descrierea pe scurt a acestui calcul este: se vizitează vârful 0, apoi primul din lista vârfului 0 – adică 1, după care primul din lista lui 1 – adică 2. Pentru că 2 nu are vârfuri adiacente spre exterior, se întoarce la 1 şi vizitează al doilea din lista vârfului 1 – adică 4. Analog ca la 2, pentru că 4 nu are vârfuri adiacente spre exterior se întoarce la 1 şi pentru că lista lui 1 este epuizată se întoarce la lista lui 0. Al doilea din lista lui 0 este 2, dar acesta a fost deja vizitat şi deci nu mai este luat în considerare. Următorul din lista lui 0 este vârful 3 care nu a mai fost vizitat, după care se ia în considerare primul din lista lui 3 – adică 1, dar acesta a mai fost vizitat. La fel şi următorul din lista lui 3 – 4. Aşadar lista ordonată dată de explorarea DFS a vârfurilor accesibile din 0 este: (0,1,2,4,3). sfex

Metodei îi putem asocia un arbore, numit *arbore parţial DFS*, în modul următor: Notăm cu T mulţimea arcelor (i, j) cu proprietatea că j este vizitat prima dată parcurgând acest arc. Pentru a obţine T este suficient să înlocuim în procedura `DFS` apelul `Viziteaza(j)` cu o instrucţiune de forma "adaugă (i, j) la T". Arborele parţial DFS este $S(D, i_0) = (V_0, T)$, unde V_0 este mulţimea vârfurilor accesibile din i_0. Definiţia este valabilă şi pentru cazul când argumentul procedurii `DFS` este reprezentarea

i	j	S	SB
		$\{0\}$	(0)
0	1	$\{0,1\}$	$(0,1)$
1	2	$\{0,1,2\}$	$(0,1,2)$
2	–	$\{0,1,2\}$	$(0,1)$
1	4	$\{0,1,2,4\}$	$(0,1,4)$
4	–	$\{0,1,2,4\}$	$(0,1)$
1	–	$\{0,1,2,4\}$	(0)
0	2	$\{0,1,2,3\}$	(0)
0	3	$\{0,1,2,3,4\}$	$(0,3)$
3	1	$\{0,1,2,3,4\}$	$(0,3)$
3	4	$\{0,1,2,3,4\}$	$(0,3)$
3	–	$\{0,1,2,3,4\}$	(0)
0	–	$\{0,1,2,3,4\}$	$(\ )$

Figura 3.19: Un calcul al procedurii DFS

unui graf, doar cu precizarea că se consideră muchii în loc de arce. În fig. 3.20a este reprezentat arborele parţial DFS pentru digraful din fig. 3.20b şi vârful 0. Arborele parţial DFS este util în multe aplicaţii ce necesită parcurgeri de grafuri.

Explorarea BFS (Breadth First Search). Se obţine din `ExplorareGraf` prin reprezentarea mulţimii SB printr-o coadă.

Exerciţiul 3.6.1. Să se înlocuiască operaţiile scrise în dreptunghiuri din procedura `ExplorareGraf` cu operaţiile corespunzătoare cozii. Noua procedură se va numi `BFS`.

Exemplu: Considerăm digraful din exemplul precedent. Presupunem de asemenea $i_0 = 1$. Calculul procedurii BFS este descris în tabelul din fig. 3.21. Descrierea sumară a acestui calcul este: se vizitează vârful 0, apoi toate vârfurile din lista lui 0, apoi toate cele nevizitate din lista lui 1, apoi cele nevizitate din lista lui 2 şi aşa mai departe. Lista ordonată dată de parcurgerea BFS este (0,1,2,3,4). sfex

Ca şi în cazul parcurgerii DFS, metodei i se poate ataşa un arbore, numit *arbore parţial BFS*. În fig. 3.20c este reprezentat arborele parţial BFS din exemplul de mai sus.

Sugerăm cititorului să testeze cele două strategii pe mai multe exemple pentru a vedea clar care este diferenţa dintre ordinile de parcurgere a vârfurilor.

Exerciţiul 3.6.2. Am văzut că, în cazul reprezentării digrafurilor prin matrice de adiacenţă, liste de adiacenţă exterioară sunt incluse în liniile matricei. Să se rescrie subprogramele `DFS` şi `BFS` pentru cazul când digraful este reprezentat prin matricea de adiacenţă.

3.6.7 Implementarea cu liste de adiacenţă statice

Listele de adiacenţă ale digrafului $D = \langle V, A\rangle$ sunt reprezentate prin două tablouri `D.a[i]`, $i = 0, \ldots, D.n$, şi `D.s[j]`, $j = 0, \ldots, D.m$, care satisfac proprietăţile:

- `D.a[i]` este adresa j în tabloul $D.s$ a primului element cu proprietatea $\langle i, s[j]\rangle \in A$;
- vârfurile destinatare ale arcelor care pleacă din i sunt memorate în componente consecutive din `D.s`;
- `D.a[0]` = `0` şi `D.a[n]` = `D.m`.

Exemplu: În fig. 3.22 este dat un exemplu de digraf reprezentat prin liste de acest fel. Din vârful 0 pleacă trei arce cu destinaţiile 1,2 şi 3 - ce sunt memorate în primele trei componente ale tabloului `D.s`. Din vârful 1 pleacă două arce cu destinaţiile 2 şi 3 - ce sunt memorate în `D.s[3]` şi `D.s[4]`. Din vârful 2 nu pleacă nici un arc şi de aceea `D.a[2]` = `D.a[3]`. Din vârful 3 pleacă un singur arc ce este memorat în `D.s[5]`. Elementul `D.a[4]` are rolul de a marca locul unde se termină lista de adiacenţă exterioară a vârfului 3. sfex

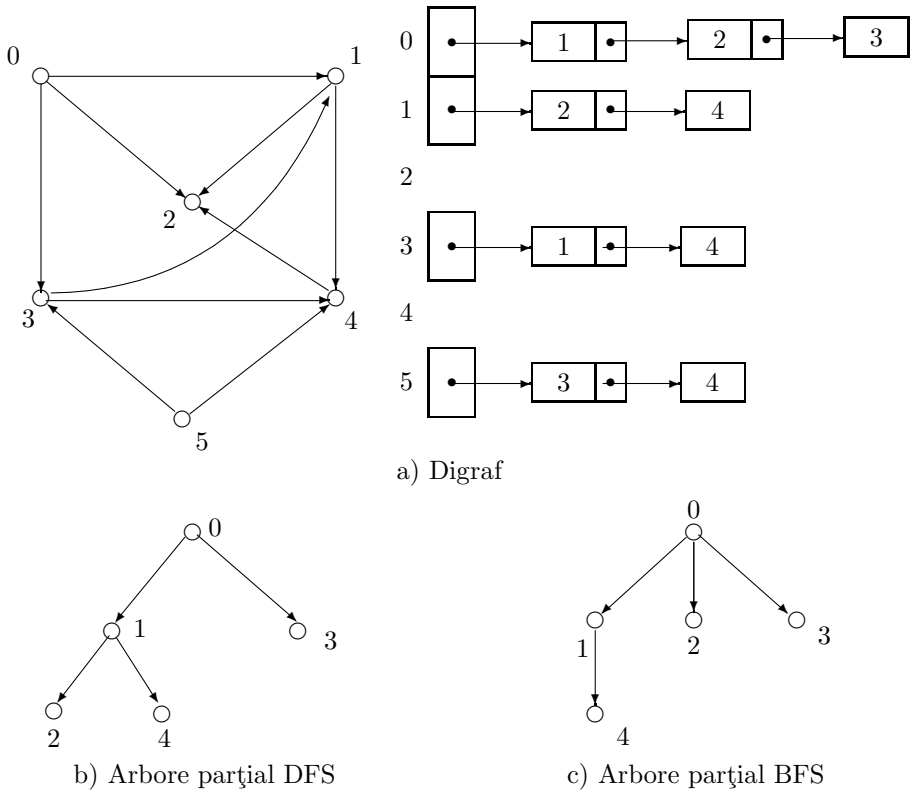


Figura 3.20: Explorarea unui digraf

Exerciţiul 3.6.3. Să se descrie implementările operaţiilor tipului Digraf pentru cazul când digrafurile sunt reprezentate prin liste de adiacenţă statice.

3.6.8 Exerciţii

Exerciţiul 3.6.4. Dacă G este un graf atunci gradul $\rho(i)$ al unui vârf i este egal cu numărul de vârfuri adiacente cu i. Dacă D este un digraf, atunci gradul extern ($\rho^+(i)$) al unui vârf i este egal cu numărul de vârfuri adiacente cu i spre exterior, iar gradul intern ($\rho^-(i)$) este numărul de vârfuri adiacente cu i spre interior.

Să se scrie o procedură `detGrad(D, tip, din,dout)` care determină gradele vârfurilor digrafului D (cazul când $tip = true$) sau ale grafului reprezentat de `D` (cazul când $tip = false$).

Exerciţiul 3.6.5. O alegere de drumuri care unesc toate perechile de vârfuri conectabile i, j într-un digraf poate fi memorată într-o matrice p cu semnificaţia: $p[i, j]$ este primul vârf intermediar întâlnit după i pe drumul de la i la j.

1. Să se modifice subprogramul `detInchReflTranz` astfel încât să determine şi o alegere de drumuri care unesc vârfurile conectabile.

2. Să se scrie un subprogram care, având date matricea drumurilor şi două vârfuri i, j, determină un drumul de la i la j.

Exerciţiul 3.6.6. Un digraf D poate fi reprezentat şi prin *listele de adiacenţă interioară*, unde `D.a[i]` este lista vârfurilor sursă ale arcelor care sosesc în i. Să se scrie procedurile care implementează operaţiile tipului Digraf pentru cazul când digrafurile sunt reprezentate prin listele de adiacenţă interioară.

Exerciţiul 3.6.7. Să se scrie o procedură `Conex(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este conex sau nu.

i	j	S	SB
		$\{0\}$	(0)
0	1	$\{0,1\}$	$(0,1)$
0	2	$\{0,1,2\}$	$(0,1,2)$
0	3	$\{0,1,2,3\}$	$(0,1,2,3)$
0	–	$\{0,1,2,3\}$	$(1,2,3)$
1	2	$\{0,1,2,3\}$	$(1,2,3)$
1	4	$\{0,1,2,3,4\}$	$(1,2,3,4)$
1	–	$\{0,1,2,3,4\}$	$(2,3,4)$
2	–	$\{0,1,2,3,4\}$	$(3,4)$
3	1	$\{0,1,2,3,4\}$	$(3,4)$
3	4	$\{0,1,2,3,4\}$	$(3,4)$
3	–	$\{0,1,2,3,4\}$	(4)
4	–	$\{0,1,2,3,4\}$	$(\,)$

Figura 3.21: Un calcul al procedurii BFS

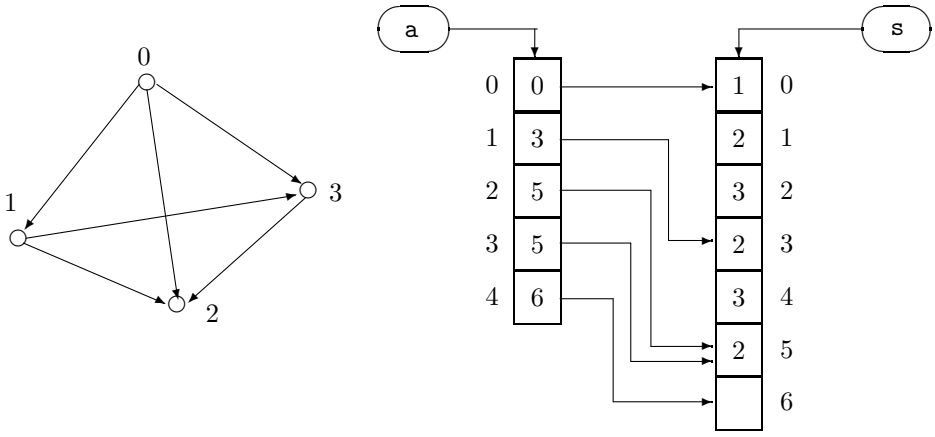


Figura 3.22: Digraf reprezentat prin liste de adiacenţă tablouri

Exerciţiul 3.6.8. Să se scrie o procedură `Arbore(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este arbore sau nu.
Indicaţie. Se poate utiliza faptul că un graf cu n vârfuri este arbore dacă şi numai dacă este conex şi are $n-1$ muchii [Cro92].

Exerciţiul 3.6.9. Să se proiecteze o structură de date pentru reprezentarea componentelor conexe ale unui graf şi să se scrie o procedură care, având la intrare reprezentarea unui graf, construieşte componentele conexe ale acestuia.

Exerciţiul 3.6.10. Fie $D = (V, A)$ un digraf cu n vârfuri. Un vârf i se numeşte *groapă* ("sink") dacă pentru orice alt vârf $j \neq i$ există un arc $(j, i) \in A$ şi nu există arc de forma (i, j). Să se scrie o funcţie `Groapa(D, g)` care decide dacă digraful reprezentat de `D` are o groapă sau nu; dacă da, atunci variabila `g` va memora o asemenea groapă. Algoritmul descris de program va avea complexitatea $O(n)$. Pot fi mai multe gropi?

Exerciţiul 3.6.11. Se consideră un arbore G (graf conex fără circuite) cu muchiile colorate cu culori dintr-un alfabet A. Să se scrie un program care, pentru un şir $a \in A^*$ dat, determină dacă există un drum în G etichetat cu a.

Exerciţiul 3.6.12. Mulţimea *grafurilor serie-paralel* (G, s, t), unde G este un multigraf (graf cu muchii multiple) iar s şi t sunt vârfuri în G numite *sursă* respectiv *destinaţie*, este definită recursiv astfel:

- Orice muchie $G = \{u, v\}$ defineşte două grafuri serie-paralel: (G, u, v) şi (G, v, u).

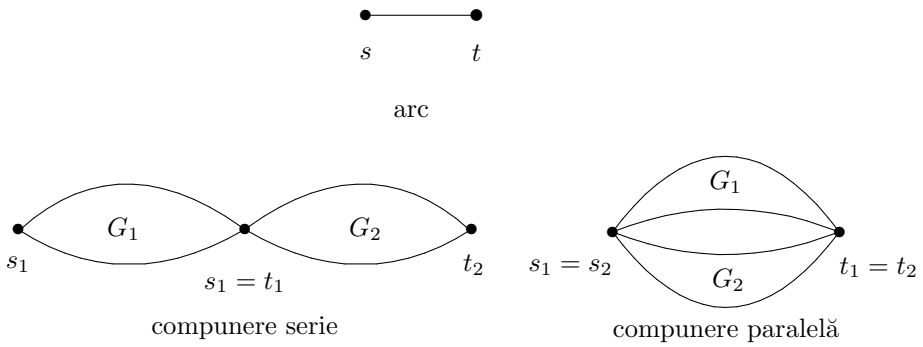


Figura 3.23: Grafuri serie-paralel

- Dacă (G_1, s_1, t_1) şi (G_2, s_2, t_2) sunt două grafuri serie-paralel atunci:
  - *compunerea serie* $G_1G_2 = (G, s_1, t_2)$, obţinută prin reuniunea disjunctă a grafurilor G_1 şi G_2 în care vârfurile s_2 şi t_1 sunt identificate, este graf serie-paralel;
  - *compunerea paralelă* $G_1 \| G_2 = (G, s_1 = s_2, t_1 = t_2)$, obţinută prin reuniunea disjunctă a grafurilor G_1 şi G_2 în care sursele s_1 şi s_2 şi respectiv destinaţiile t_1 şi t_2 sunt identificate, este graf serie-paralel.

Definiţia este sugerată grafic în fig. 3.23.

Să se scrie un program care decide dacă un graf dat este serie-paralel.

Exerciţiul 3.6.13. Să se scrie un program care, pentru un graf $G = (V, E)$ şi $i_0 \in V$ date, enumără toate drumurile maximale care pleacă din i_0.

Exerciţiul 3.6.14. Se consideră problema din exerciţiul 3.6.13. Presupunem că muchiile grafului G sunt etichetate cu numere întregi. Să se modifice programul de la 3.6.13 astfel încât drumurile să fie enumerate în ordine lexicografică.

3.7 "Heap"-uri

3.7.1 Tipul de date abstract "coadă cu priorităţi"

3.7.1.1 Obiectele de tip dată

O *coadă cu priorităţi* este o structură de date în care elementele sunt numite *atomi* iar fiecare atom conţine un câmp-cheie care ia valori dintr-o mulţime total ordonată. Valoarea acestui câmp-cheie se numeşte *prioritate*. Operaţiile de citerie/eliminare se refră întotdeauna la atomul cu prioritatea cea mai mare. Interpretarea noţiunii de prioritate poate diferi de la caz la caz. Există situaţii când atomii cei mai prioritari sunt cei cu cheile valorilor mai mici şi există situaţii când atomii cei mai prioritari sunt cei cu cheile valorilor mai mari. Noi considerăm aici ultimul caz.

3.7.1.2 Operaţii

CoadaVidă.

Intrare: – nimic;
Ieşire: – coada cu priorităţi vidă.

Elimină.

Intrare: – o coadă cu priorităţi Q;
Ieşire: – Q din care s-a eliminat atomul cu cheia cea mai mare.

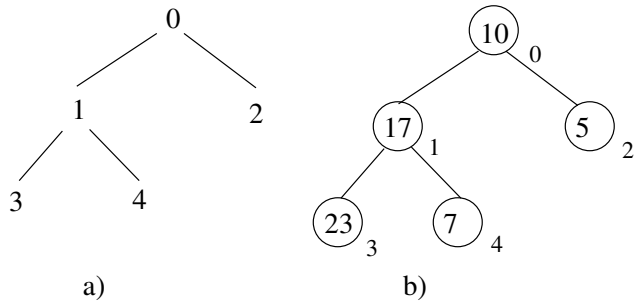


Figura 3.24: Arbore binar complet

Inserează.

Intrare: – o coadă cu priorităţi Q şi un atom a;
Ieşire: – Q la care s-a adăugat a.

Citeşte.

Intrare: – o coadă cu priorităţi Q;
Ieşire: – atomul cu cheia cea mai mare din coada Q.

3.7.2 Implementarea cu max-"heap"-uri

3.7.2.1 Descrierea max-"heap"-urilor

Definiţia 3.1. *Un* max-"heap" *este un arbore binar complet cu proprietăţile:*

1. *informaţiile din noduri sunt valori dintr-o mulţime total ordonată numite* chei*;*
2. *pentru orice nod intern* v*, cheia memorată în* v *este mai mare decât sau egală cu cheia oricăruia dintre fii.*

În continuare arătăm cum max-"heap" sunt reprezentate prin tablouri 1-dimensionale.

Definiţia 3.2. *Pentru un $n \in N^*$,* arborele binar complet *ataşat lui n este definit prin*

$$H_n = (\{0, \ldots, n-1\}, E)$$

unde $E = \{(\left[\frac{i-1}{2}\right], i) \mid i = 1, \ldots, n-1\}$.

Exemplu: Pentru $n = 5$ arborele H_5 este reprezentat în fig. 3.24a. sfex

Lema 3.1. *Au loc următoarele proprietăţi ale arborelui H_n:*

1. *Vârful i are succesorii $2i+1$ şi $2i+2$.*
2. *Vârful i are ca vârf-tată pe $\left[\frac{i-1}{2}\right]$, dacă $i \geq 2$.*
3. *Arborele are $[\log_2 n] + 1$ nivele.*
4. *Pe nivelul k se găsesc vârfurile $2^k - 1, 2^k, \ldots, \min\{2^{k+1} - 2, n-1\}$.*[1]

Definiţia 3.3. *Fie $a = (a_0, \ldots, a_{n-1})$. Prin $H_n(a)$ notăm arborele obţinut din H_n prin etichetarea vârfurilor i cu elementele a_i în mod corespunzător.*

[1] Aici nivelurile sunt numerotate începând cu 0; rădăcina este pe nivelul 0.

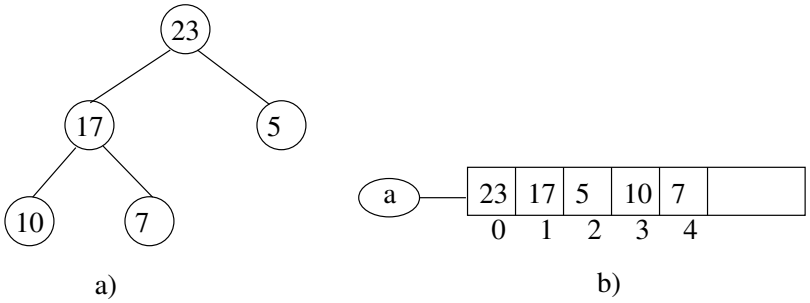


Figura 3.25: Exemplu de max-"heap"

Exemplu: Dacă $a = (10, 17, 5, 23, 7)$ atunci arborele $H_5(a)$ este reprezentat în fig. 3.24b. sfex

Definiţia 3.4. a) *Tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are proprietatea* MAX-HEAP *dacă* $(\forall k)(1 \leq k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$. *Notăm această proprietate prin* MAX-HEAP(a).

b) *Tabloul* a *are proprietatea* MAX-HEAP *începând cu poziţia* ℓ *dacă* $(\forall k)(\ell \leq \frac{k-1}{2} < k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$. *Notăm această proprietate prin* MAX-HEAP$(\mathtt{a}, \ell)$.

Vom da câteva proprietăţi ale predicatelor MAX-HEAP(a) şi MAX-HEAP$(\mathtt{a}, \ell)$.

Lema 3.2. *1. Tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are proprietatea* MAX-HEAP *(adică* MAX-HEAP(a) = *true) dacă şi numai dacă pentru orice vârf* $a[i]$ *din* $H_n(a)$, $a[i]$ *este mai mare decât sau egal cu orice succesor* $a[j]$ *al său în* $H_n(a)$.

2. Pentru orice tablou $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are loc* MAX-HEAP$(\mathtt{a}, \left[\frac{n}{2}\right])$.

3. Dacă MAX-HEAP$(\mathtt{a}, \ell)$, *atunci pentru orice* $j > \ell$ *are loc* MAX-HEAP$(\mathtt{a}, j)$.

4. Dacă MAX-HEAP(a) *atunci* $a[0]$ *este elementul maxim din tablou.*

5. Dacă MAX-HEAP(a) *atunci* $H_n(a)$ *este un max-"heap".*

În fig. 3.25 este arătat un exemplu de max-"heap" şi tabloul cu reprezentarea sa.

3.7.2.2 Implementarea operaţiilor

CoadaVidă. Este reprezentată de tabloul vid.

Elimină. Atomul cu cea mai mare cheie se află în rădăcina max-"heap"-ului (prima poziţie în tablou). Ştergerea acestuia lasă un loc liber în rădăcină în care copiem atomul de pe ultima poziţie din tablou. Dimensiunea max-"heap"-ului este decrementată cu 1. Refacerea proprietăţii MAX-HEAP se realizează prin parcurgerea unui drum de la rădăcină spre frontieră conform următorului algoritm:

1. Se consideră vârful curent j. Iniţial avem $j = 0$.
2. Determină vârful cu valoarea maximă dintre fiii lui j. Fie acesta k.
3. Dacă $a[j] < a[k]$ atunci interschimbă a[j] cu a[k].
4. Dacă $2 * j \leq n$ atunci repetă paşii 2-4 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de eliminare este:

```
procedure elimina(a, n)
begin
   a[0] ← a[n-1]
   n ← n-1
   j ← 0
   esteHeap ← false
```

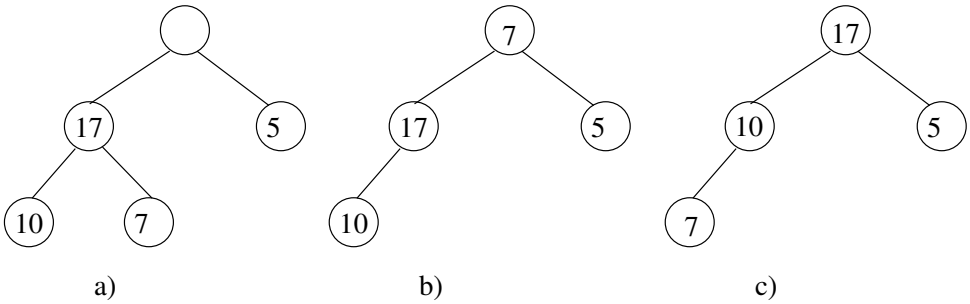


Figura 3.26: Eliminarea din max-"heap"-ul din fig. 3.25

```
   while ((2*j+1 ≤ n-1) and not esteHeap)
      k ← 2*j+1
      if ((k < n-1) and (a[k] < a[k+1])) then k ← k+1
      if (a[j] < a[k])
      then swap(a[j], a[k])
      else esteHeap ← true
      j ← k
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(\log_2 n)$. În fig. 3.26 este arătat modul în care este eliminat atomul cu cheia cea mai mare din max-"heap"-ul din fig. 3.25.

Inserează. Dacă max-"heap"-ul are n elemente, atunci se meorează noul atom pe poziţia $n+1$ şi se incrementează dimensiunea max-"heap"-ului. Apoi se reface proprietatea MAX-HEAP parcurgând un drum de la nodul ce memorează noul atom spre rădăcină, conform următorului algoritm:

1. Se consideră vârful curent `j`. Iniţial avem $j = n - 1$.
2. Fie k poziţia nodului tată. Dacă $a[j] > a[k]$ atunci interschimbă `a[j]` cu `a[k]`.
3. Dacă $j > 1$ atunci repetă paşii 2-3 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de inserare este:

```
procedure insereaza(a, n, o_cheie)
begin
   n ← n+1
   a[n-1] ← o_cheie
   j ← n
   esteHeap ← false
   while ((j > 0) and not esteHeap)
      k ← [j/2]
      if(a[j] > a[k])
      then swap(a[j], a[k])
      else esteHeap ← true
      j ← k
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(\log_2 n)$. În fig. 3.27 este arătată inserarea unui atom cu cheia egală cu 20 în max-"heap"-ul din fig. 3.25.

Citeşte. Întoarce primul element din tablou. Complexitatea timp în cazul cel mai nefavorabil este $O(1)$.

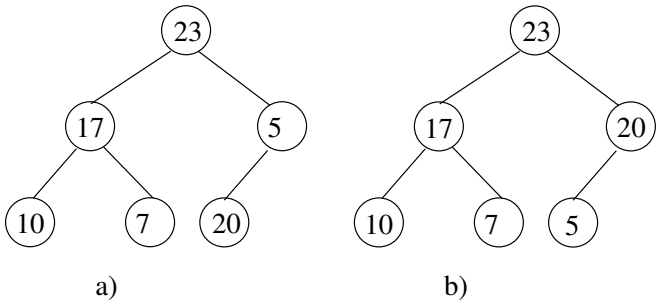


Figura 3.27: Inserarea cheii 20 în max-"heap"-ul din fig. 3.25

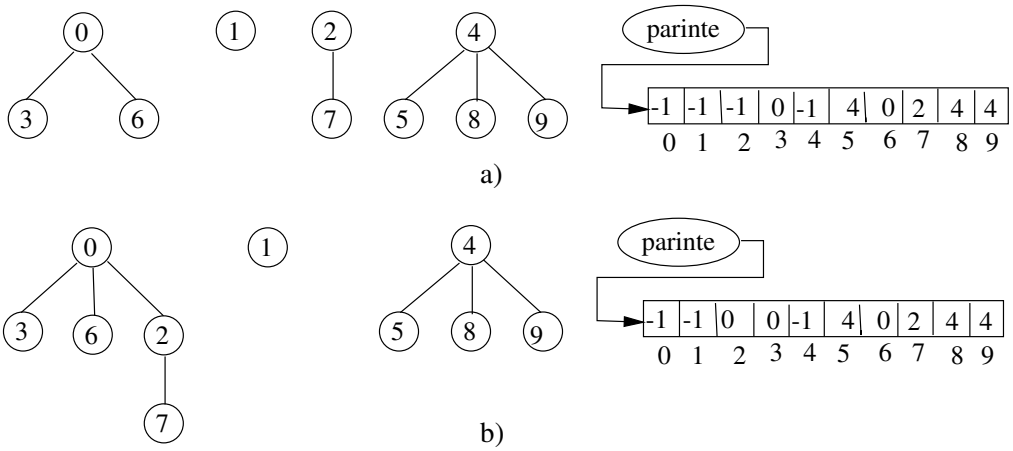


Figura 3.28: Structuri "union-find"

3.8 "Union-find"

Grafurile pot reprezenta colecţii de mulţimi disjuncte într-un mod foarte natural: vârfurile corespund obiectelor iar o muchie are semnificaţia că extremităţile sale sunt în aceeaşi mulţime. Această reprezentare permite rezolvarea eficientă a multor probleme legate de mulţimi. De exemplu, a răspunde la întrebarea "La ce mulţime aparţine obiectul x" (operaţia "find") presupune de fapt a preciza la ce componentă conexă aparţine vârful x. Există şi un inconvenient al reprezentării: aceeaşi mulţime poate fi reprezentată prin mai multe grafuri conexe. De aceea, de exemplu, informaţiile oferite de parcurgerile sistematice are o utilitate mai redusă.

Un caz aparte se obţine când se dă un aspect dinamic problemei: la diferite momente, două mulţimi se pot reuni într-una singură (operaţia "union"). Formularea unei cereri de acest tip este de forma: "reuneşte mulţimea la care aparţine i cu mulţimea la care aparţine j". Aceasta operaţie se poate realiza foarte simplu prin adăugarea unei muchii care să unească un vârf din componenta conexă corespunzătoare primei mulţimi cu un vârf din componenta corespunzătoare celei de-a doua mulţimi. Rămâne de stabilit care vârfuri candidează la momentul respectiv pentru unire printr-o muchie. Această alegere va trebui să permită construirea de algoritmi eficienţi pentru ambele operaţii: "union" şi "find". Având în vedere că topologia componentelor conexe nu este importantă, vom alege structura de tip arbore cu rădăcină pentru reprezentarea unei mulţimi. Colecţia de mulţimi este reprezentată de o colecţie de arbori (o *pădure*). Pentru reprezentarea unei păduri vom utiliza un tablou `parinte` cu semnificaţia că $parinte[i]$ este predecesorul imediat (părintele) vârfului i. Mulţimea univers, peste care se consideră colecţia de mulţimi, este aplicată prin funcţia *index* într-un interval de numere întregi $[0, n-1]$. Astfel o colecţie C va fi reprezentată de numărul întreg n şi tabloul *parinte*. Dacă i este rădăcina unui arbore, atunci $parinte[i] = -1$. Un exemplu de colecţie reprezentată astfel este arătat în fig. 3.28a.

Numele unei mulţimi este dat de rădăcina arborelui ce o reprezintă. Crearea unei mulţimi cu un singur element este banală:

```
procedure singleton(C, i)
begin
   C.parinte[i] ← -1
end
```

Determinarea mulţimii la care aparţine i este echivalentă cu determinarea rădăcinii:

```
function find(C, i)
begin
   temp ← i
   while (C.parinte[temp] > 0) do
      temp ← C.parinte[temp]
   return temp
end
```

Reuniunea a două mulţimi înseamnă ducerea unui arc de la rădăcina unui arbore la rădăcina celuilalt:

```
procedure union(C, i, j)
begin
   r1 ← find(i)
   r2 ← find(j)
   if (r1 ≠ r2) then C.parinte[r2] ← r1
end
```

Executând operaţia $union(C, 7, 8)$ pentru colecţia din fig. 3.28a obţinem structura din fig. 3.28b.

Prin execuţia repetată a operaţiei "union" se pot obţine arbori dezechilibraţi, în care determinarea rădăcinii pentru anumite vârfuri să dureze mult. Structura ar putea fi îmbunătăţită dacă s-ar putea menţine o formă cât mai aplatizată a rborilor, adică să nu existe noduri aflate la distanţe mari de rădăcină. Aceasta se poate realiza prin memorarea în rădăcini a numărului de vârfuri din arbore (greutatea arborelui). Aceasta va fi memorată cu semnul minus pentru a distinge rădăcinile de celelalte noduri. În fig. 3.29 este reprezentată o asemenea structură. În plus, procedura `union` va realiza mai întâi o "aplatizare" a arborilor şi apoi duce un arc de la rădăcina arborelui cu greutatea mai mare la rădăcina celui cu greutatea mai mică.

```
union(C, i, j)
begin
   r1 ← find(i)
   r2 ← find(j)
   while (C.parinte[i] > 0) do
      temp ← i
      i ← C.parinte[i]
      C.parinte[temp] ← r1
   while (C.parinte[j] > 0) do
      temp ← j
      j ← C.parinte[j]
      C.parinte[temp] ← r2
   if (C.parinte[r1] > C.parinte[r2]) then
        C.parinte[r2] ← C.parinte[r1]+C.parinte[r2]
        C.parinte[r1] ← r2
   else if (C.parinte[r1] < C.parinte[r2]) then
        C.parinte[r1] ← C.parinte[r1]+C.parinte[r2]
        C.parinte[r2] ← r1
end
```

O soluţie alternativă la cea de mai sus este să memorăm înălţimea arborelui în loc de greutate. Operaţia `union` va ancora arborele cu înălţime mai mică de cel cu înălţimea mai mare.

Tipul de date definit mai sus este cunoscut în literatură sub numele de *structura "union-find"*.

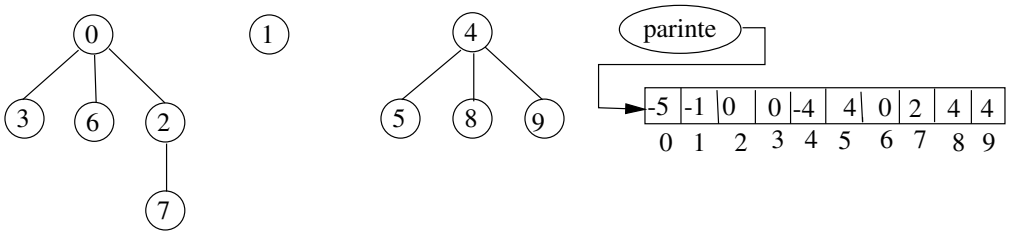


Figura 3.29: Structură "union-find" ponderată

Teorema 3.2. *O secvenţă de m operaţii* `singleton`, `union` *şi* `find` *din care n sunt operaţii* `singleton` *poate fi realizată în timpul $O(m \cdot \log^* n)$ în cazul cel mai nefavorabil, unde $\log^* n$ este cel mai mic număr natural k cu proprietatea $\log^{(k)} n \leq 1$ şi $\log^{(k)}$ desemnează funcţia $\log$ compusă cu ea însăşi de k ori: $\log^{(0)} n = n$, $\log^{(i+1)} n = \log \log^{(i)} n$.*

Demonstraţia acestei teoreme poate fi găsită în [CLR93].

Capitolul 4

Sortare internă

Alături de căutare, sortarea este una dintre problemele cele mai importante atât din punct de vedere practic cât şi teoretic. Ca şi căutarea, sortarea poate fi formulată în diferite moduri. Cea mai generală formulare şi mai des utilizată este următoarea:

> Fie dată o secvenţă $(v_0, \ldots, v_{n-1})$ cu componentele v_i dintr-o mulţime total ordonată. Problema sortării constă în determinarea unei permutări π astfel încât $v_{\pi(0)} \leq v_{\pi(1)} \leq \cdots \leq v_{\pi(n-1)}$ şi în rearanjarea elementelor din secvenţă în ordinea dată de permutare.

O altă formulare, echivalentă cu cea de mai sus, este următoarea:

> Fie dată o secvenţă de înregistrări $(R_0, \ldots, R_{n-1})$, unde fiecare înregistrare R_i are o valoare cheie K_i. Peste mulţimea cheilor K_i este definită o relaţie de ordine totală. Problema sortării constă în determinarea unei permutări π astfel încât $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ şi în rearanjarea înregistrărilor în ordinea $(R_{\pi(0)}, \ldots, R_{\pi(n-1)})$.

Pentru ambele formulări presupunem că secvenţa dată este reprezentată printr-o listă liniară. Dacă această listă este memorată în memoria internă a calculatorului atunci avem *sortare internă* şi dacă se găseşte într-un fişier memorat pe un periferic atunci avem *sortare externă*. În acest capitol ne ocupăm numai de sortarea internă.

Vom simplifica formularea problemei presupunând că secvenţa dată este un tablou unidimensional. Acum problema sortării se reformulează astfel:

> *Intrare*: n şi tabloul $(\texttt{a[i]} \mid i = 0, \ldots, n-1)$ cu $a[i] = v_i$, $i = 0, \ldots, n-1$.
> *Ieşire*: tabloul `a` cu proprietăţile: $a[i] = w_i$ pentru $i = 0, \ldots, n-1$, $w_0 \leq \cdots \leq w_{n-1}$ şi $(w_0, \ldots, w_{n-1})$ este o permutare a secvenţei $(v_1, \ldots, v_n)$; convenim să notăm această proprietate prin $(w_0, \ldots, w_{n-1}) = \mathrm{Perm}(v_0, \ldots, v_{n-1})$.

Există foarte mulţi algoritmi care rezolvă problema sortării interne. Nu este în intenţia noastră a-i trece în revistă pe toţi. Vom prezenta numai câteva metode pe care le considerăm cele mai semnificative. Două dintre acestea, anume sortarea prin interclasare şi sortarea rapidă, vor fi prezentate în capitolul dedicat metodei divide-et-impera.

4.1 Sortare bazată pe comparaţii

În această secţiune am grupat metodele de sortare bazate pe următoarea tehnică: determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ şi $a[j]$ ale tabloului supus sortării. Scopul comparării poate fi diferit: pentru a rearanja valorile celor două componente în ordinea firească (sortare prin interschimbare), sau pentru a insera una dintre cele două valori într-o subsecvenţă ordonată deja (sortare prin inserţie), sau pentru a selecta o valoare ce va fi pusă pe poziţia sa finală (sortare prin selecţie). Decizia că o anumită metodă aparţine la una dintre subclasele de mai sus are un anumit grad de subiectivitate. De exemplu selecţia naivă ar putea fi foarte bine considerată ca fiind o metodă bazată pe interschimbare.

4.1.1 Sortarea prin interschimbare

Vom prezenta aici strategia cunoscuta sub numele de *sortare prin metoda bulelor* (bubble sort).

Notăm cu $SORT(\texttt{a})$ predicatul care ia valoarea *true* dacă si numai dacă tabloul `a` este sortat. Metoda bubble sort se bazează pe următoarea definiţie a predicatului $SORT(\texttt{a})$:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

O pereche (i, j) cu $i < j$ formează o *inversiune (inversare)* dacă $a[i] > a[j]$. Astfel, pe baza definiţiei de mai sus vom spune că tabloul `a` este sortat dacă si numai dacă nu există nici o inversiune de elemente vecine. Metoda "bubble sort" propune parcurgerea iterativă a tabloului `a` şi, la fiecare parcurgere, ori de câte ori se întâlneşte o inversiune $(i, i+1)$ se procedează la interschimbarea $\texttt{a}[\texttt{i}] \leftrightarrow \texttt{a}[\texttt{i}+1]$. La prima parcurgere, elementul cel mai mare din secvenţă formează inversiuni cu toate elementele aflate după el şi, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este şi locul său final. La iteraţia următoare, la fel se va intâmpla cu cel de-al doilea element cel mai mare. În general, dacă subsecvenţa $\texttt{a}[\texttt{r}+1..\texttt{n}-1]$ nu are nici o inversiune la iteraţia curentă, atunci ea nu va avea inversiuni la nici una din iteraţiile următoare. Aceasta permite ca la iteraţia următoare să fie verificată numai subsecvenţa $\texttt{a}[0..\texttt{r}]$. Terminarea algoritmului este dată de faptul că la fiecare iteraţie numărul de interschimbari este micşorat cu cel puţin 1.

Descrierea Pascal a algoritmului este următoarea:

```
procedure bubbleSort(a, n)
begin
   ultim ← n-1
   while (ultim > 0) do
      n1  ← ultim - 1
      ultim  ← 0
      for i ← 0 to n1 do
          if (a[i] > a[i+1])
          then swap(a[i], a[i+1])
               ultim  ← i
end
```

Evaluarea algoritmului Cazul cel mai favorabil este intâlnit atunci când secvenţa de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operaţii. Cazul cel mai nefavorabil este obţinut când secvenţa de intrare este ordonată descrescător şi, în acest caz, procedura execută $O(n^2)$ operaţii.

4.1.2 Sortare prin inserţie

Una din familiile importante de tehnici de sortare se bazează pe metoda "jucătorului de bridge" (atunci când îşi aranjează cărţile), prin care fiecare element este inserat în locul corespunzător în raport cu elementele sortate anterior.

4.1.2.1 Sortare prin inserţie directă

Principiul de bază al algoritmului de sortare prin inserţie este următorul: Se presupune că subsecvenţa $(\texttt{a}[0], \ldots, \texttt{a}[\texttt{j}-1])$ este sortată. Se caută în această subsecvenţă locul i al elementului $a[j]$ şi se inserează $a[j]$ pe poziţia i. Poziţia i este determinată astfel:

- $i = 0$ dacă $a[j] < a[0]$;
- $0 < i < j$ şi satisface $a[i-1] \leq a[j] < a[i]$;
- $i = j$ dacă $a[j] \geq a[j-1]$.

Determinarea lui i se poate face prin căutare secvenţială sau prin căutare binară. Considerăm cazul când poziţia i este determinată prin căutare secvenţială (de la dreapta la stânga) simultan cu deplasarea elementelor mai mari decât $a[j]$ cu o poziţie la dreapta. Această deplasare se realizează prin interschimbări astfel încât valoarea $a[j]$ realizează câte o deplasare la stânga până ajunge la locul ei final.

```
procedure insertSort(a, n)
begin
   for j ← 1 to n-1 do
       i ← j-1
       temp ← a[j]
       while ((i ≥ 0) and (temp < a[i])) do
          a[i+1] ← a[i]
          i ← i-1
       if (i ≠ j-1) then a[i+1] ← temp
end
```

Evaluarea Căutarea poziţiei i în subsecvenţa $\texttt{a}[0..\texttt{j}-1]$ necesită $O(j-1)$ timp. Rezultă că timpul total în cazul cel mai nefavorabil este $O(1+\cdots+n-1) = O(n^2)$. Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, complexitatea timp este $O(n)$.

Exerciţiul 4.1.1. Complexitatea algoritmului de sortare prin inserţie poate fi îmbunătăţită considerând secvenţa de sortat reprezentată printr-o listă simplu înlănţuită. Să se rescrie algoritmul `InsertSort` corespunzător acestei reprezentări. Să se precizeze complexitatea timp a noului algoritm.

4.1.2.2 Metoda lui Shell

În algoritmul precedent elementele se deplasează numai cu câte o poziţie o dată şi prin urmare timpul mediu va fi proporţional cu n^2, deoarece fiecare element călătoreşte în medie $n/3$ poziţii în timpul procesului de sortare. Din acest motiv s-au căutat metode care să îmbunătăţească inserţia directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi în loc de paşi mici. O asemenea metodă a fost propusă in anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micşorarea incrementului*. Următorul exemplu ilustrează ideea generală care stă la baza metodei.

Exemplu: Presupunem $n = 16$. Sunt executaţi următorii paşi:

1. *Prima trecere*. Se impart cele 16 elemente în 8 grupe de cîte două înregistrări (valoarea incrementului $h_0 = 8$): $(\texttt{a}[0], \texttt{a}[8]), (\texttt{a}[1], \texttt{a}[9]), \ldots, (\texttt{a}[7], \texttt{a}[15])$. Fiecare grupă este sortată separat, astfel că elementele mari se deplasează spre dreapta.

2. *A doua trecere*. Se împart elementele în grupe de câte 4 (valoarea incrementului $h_1 = 4$) care se sortează separat:
$$(\texttt{a}[0], \texttt{a}[4], \texttt{a}[8], \texttt{a}[12]), \ldots, (\texttt{a}[3], \texttt{a}[7], \texttt{a}[11], \texttt{a}[15])$$

3. *A treia trecere*. Se grupează elementele în două grupe de câte 8 elemente (valoarea incrementului $h_2 = 2$): $(\texttt{a}[0], \texttt{a}[2], \ldots, \texttt{a}[14])$, $(\texttt{a}[1], \texttt{a}[3], \ldots, \texttt{a}[15])$ şi se sortează separat.

4. *A patra trecere*. Acest pas termină sortarea prin considerarea unei singure grupe care conţine toate elementele. În final cele 16 elemente sunt sortate.

sfex

Fiecare din procesele intermediare de sortare implică, fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că, inserţia directă poate fi utilizată cu succes pentru fiecare operaţie de sortare. Prin aceste inserţii intermediare, elementele tind să conveargă rapid spre destinaţia lor finală. Secvenţa de incremente 8, 4, 2, 1 nu este obligatorie; poate fi utilizată orice secvenţă $h_i > h_{i-1} > \cdots > h_0$, cu condiţia ca ultimul increment h_0 să fie 1.

Presupunem că numărul de incremente este memorat de variabila `nincr` şi că acestea sunt memorate în tabloul $(\texttt{kval}[\texttt{h}] \mid 0 \le h \le nincr - 1)$. Subprogramul care descrie metoda lui Shell este:

```
procedure ShellSort(a, n)
begin
   for h ← nincr-1 downto 0 do
       k ← kval[h]
       for i ← k to n-1 do
```

```
            temp ← a[i]
            j ← i-k
            while ((j ≥ 0) and (temp < a[j])) do
               a[j + k] ← a[j]
               j ← j - k
            if (j+k ≠ i) then a[j+k] ← temp
end
```

Evaluarea metodei lui Shell Pentru evaluare vom presupune că elementele din secvenţă sunt diferite şi dispuse aleator. Vom denumi operaţia de sortare corespunzătoare primei treceri h_t-*sortare*, apoi h_{t-1}-*sortare*, etc.. O subsecvenţă pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, va fi denumită *h-ordonată*. Vom considera pentru început cea mai simplă generalizare a inserţiei directe şi anume cazul când avem numai două incremente: $h_1 = 2$ şi $h_0 = 1$. Cazul cel mai favorabil este obţinut când secvenţa de intrare este ordonată crescător şi sunt executate $\frac{n}{2} - 1 + n - 1$ comparaţii şi nici o deplasare. Cazul cel mai nefavorabil, când secvenţa de intrare este ordonată descrescător, necesită $\frac{1}{4}n(n-2) + \frac{n}{2}$ comparaţii şi tot atâtea deplasări (s-a presupus n număr par). În continuare ne ocupăm de comportarea în medie. În cea de-a doua trecere avem o secvenţă 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$. Este uşor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulţimii $\{0, 1, \ldots, n-1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{[\frac{n}{2}]}$ deoarece obţinem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse in poziţii impare $1, 3, \ldots$. Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvenţă aleatoare a fost 2-ordonată. Determinăm numărul mediu de inversări între astfel de permutări. Fie A_n numărul total de inversări peste toate permutările 2-ordonate de $\{0, 1, \ldots, n-1\}$. Relaţiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele şase cazuri 2-ordonate

$$0\,1\,2\,3 \quad 0\,2\,1\,3 \quad 0\,1\,3\,2 \quad 1\,0\,2\,3 \quad 1\,0\,3\,2 \quad 2\,0\,3\,1$$

vom găsi $A_4 = 0 + 1 + 1 + 2 + 3 = 8$. În urma calculelor, care sunt un pic dificile (a se vedea [Knu76] pag. 87), se obţine pentru A_n o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

De aceea numărul mediu de inversări într-o permutare aleatoare 2-ordonată este

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{[\frac{n}{2}]}}$$

După aproximarea lui Stirling aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

Teorema 4.1. *Numărul mediu de inversări executate de algoritmul lui Shell pentru secvenţa de incremente* $(2,1)$ *este* $O(n^{\frac{3}{2}})$.

Se pune problema dacă în loc de secvenţa de incremente $(2,1)$ se consideră $(h,1)$, atunci pentru ce valori ale lui h se obţine un timp cât mai mic pentru cazul cel mai nefavorabil. Are loc următorul rezultat ([Knu76], pag. 89).

Teorema 4.2. *Dacă* $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$ *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil.*

Pentru cazul general, când secvenţa de incremente pentru algoritmul lui Shell este $h_{t-1}, \ldots, h_0$, se cunosc următoarele rezultate. Primul dintre ele pune în evidenţă o alegere nepotrivită pentru incremente.

Teorema 4.3. *Dacă secvenţa de incremente* $h_{t-1}, \ldots, h_0$ *satisface condiţia*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1$$

atunci complexitatea timp pentru cazul cel mai nefavorabil este $O(n^2)$.

O justificare intuitivă a teoremei de mai sus este următoarea. De exemplu dacă $h_s = 2^s$, $0 \leq s \leq 3$ atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacţiune între elementele de pe poziţiile pare si impare. De aceea, trecerii finale de 1-sortare îi vor reveni $O(n^{\frac{3}{2}})$ inversări. Dar să observăm că o 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât trecerea finală de 1-sortare nu va găsi mai mult de $2n$ inversări. Astfel are loc următoarea teoremă.

Teorema 4.4. *Complexitatea timp în cazul cel mai nefavorabil a algoritmului* `ShellSort` *este* $O(n^{\frac{3}{2}})$ *când* $h_s = 2^s,\ 0 \leq s \leq t-1 = [\log_2 n]$.

4.1.3 Sortarea prin selecţie

Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă : la pasul curent se selectează un element din secvenţă şi se plasează pe locul său final. Procedeul continuă până când toate elementele sunt plasate pe locurile lor finale. După modul în care se face selectarea elementului curent, metoda poate fi mai mult sau mai puţin eficientă. Noi ne vom ocupa doar de două strategii de sortare prin selecţie.

4.1.3.1 Selecţia naivă

Este o metodă mai puţin eficientă dar foarte simplă în prezentare. Se bazează pe următoarea caracterizare a predicatului $SORT(\texttt{a})$:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n) \Rightarrow a[i] = \max\{a[0], \ldots, a[i]\}$$

Ordinea în care sunt aşezate elementele pe poziţiile lor finale este $n-1, n-2, \ldots, 0$. O formulare ehivalentă este:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n) : a[i] = \min\{a[i], \ldots, a[n]\}$$

caz în care ordinea de aşezare este $0, 1, \ldots, n-1$.

Subprogramul `naivSort` determină de fiecare dată locul valorii maxime:

```
procedure naivSort(a, n)
begin
   for i ← n-1 downto 1 do
       locmax ← 0
       maxtemp ← a[0]
       for j ← 1 to i do
           if (a[j] > maxtemp)
           then locmax ← j
                maxtemp ← a[j]
       a[locmax] ← a[i]
       a[i] ← maxtemp
end
```

Evaluare algoritmului descris de procedura `NaivSort` este simplă şi conduce la o complexitate timp $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are complexitatea $\Theta(n^2)$. Este interesant de comparat `BubbleSort` cu `NaivSort`. Cu toate că sortarea prin metoda bulelor face mai puţine comparaţii decât selecţia naivă, ea este aproape de două ori mai lentă decât selecţia naivă, datorită faptului că realizează multe schimbări în timp ce selecţia naivă implică o mişcare redusă a datelor. În tabelul din fig. 4.1 sunt redaţi timpii de execuţie (în sutimi de secunde) pentru cele două metode obţinuţi în urma a 10 teste pentru $n = 1000$.

4.1.3.2 Selecţia sistematică

Se bazează pe structura de date de tip max-"heap" 3.7.2. Metoda de sortare prin selecţie sistematică constă în parcurgerea a două etape:

I Construirea pentru secvenţa curentă a proprietăţii MAX-HEAP(`a`).

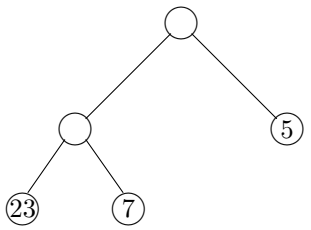
Nr. test	BubbleSort	NaivSort
1	71	33
2	77	27
3	77	28
4	94	38
5	82	27
6	77	28
7	83	32
8	71	33
9	71	39
10	72	33

Figura 4.1: Compararea algoritmilor BubbleSort şi NaivSort

II Selectarea în mod repetat a elementului maximal din secvenţa curentă şi refacerea proprietăţii MAX-HEAP pentru secvenţa rămasă.

Etapa I Considerăm că tabloul a are lungimea n. Iniţial are loc MAX-HEAP(a, $\frac{n}{2}$).

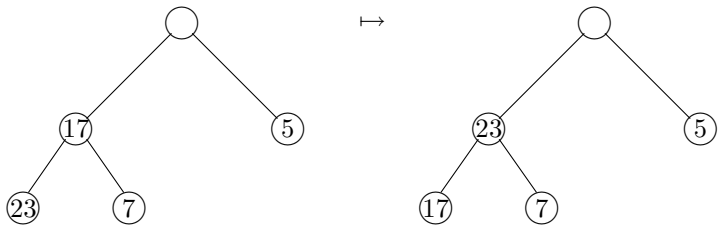
Exemplu: Presupunem că valoarea tabloului a este $a = (10, 17, 5, 23, 7)$. Se observă imediat că are loc MAX-HEAP(a, 2):



sfex

Dacă are loc MAX-HEAP(a, $\ell + 1$) atunci se procedează la introducerea lui $a[\ell]$ în grămada deja construită a[$\ell + 1$..n $- 1$], astfel încât să obţinem MAX-HEAP(a, ℓ). Procesul se repetă până când ℓ devine 0.

Exemplu: (Continuare) Avem $\ell = 1$ şi introducem pe $a[1] = 17$ în grămada a[2..4]:



Se obţine secvenţa $(10, 23, 5, 17, 7)$ care are proprietatea MAX-HEAP începând cu 2. Considerăm $\ell = 1$ şi introducem $a[1] = 10$ în grămada a[2..5]. Se obţine valoarea $(23, 17, 5, 10, 7)$ pentru tabloul a, valoare care verifică proprietatea MAX-HEAP. sfex

Algoritmul de introducerea elementului $a[\ell]$ în grămada a[ℓ+1..n$-$1], pentru a obţine MAX-HEAP(a, ℓ), este asemănător celui de inserare într-un max-"heap":

```
procedure intrInGr(a, n, ℓ)
begin
   j ← ℓ
   esteHeap ← false
```

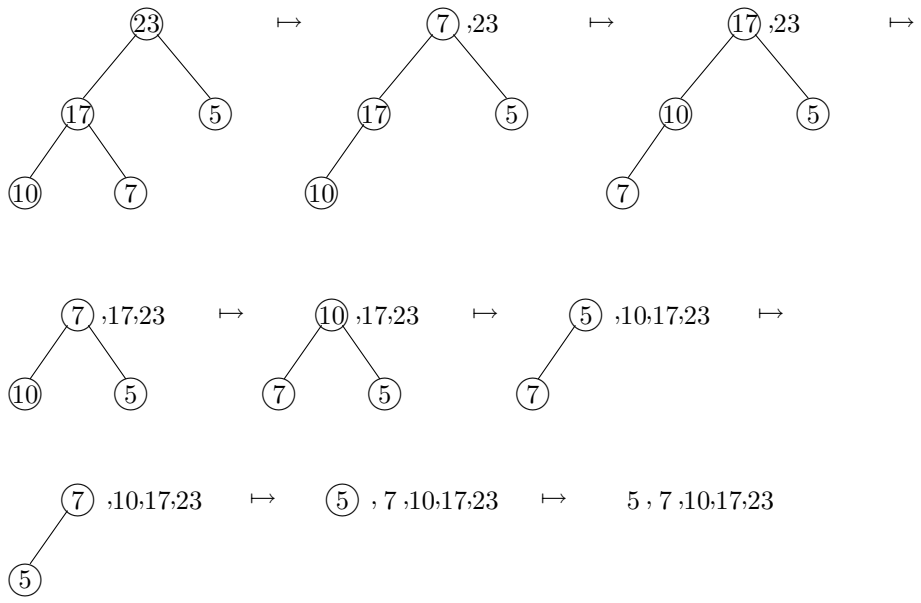


Figura 4.2: Etapa a doua

```
   while ((2*j+1 ≤ n-1) and not esteHeap)
      k ← j*2+1
      if((k < n-1) and (a[k] < a[k+1])
      then k ← k+1
      if(a[j] < a[k])
      then swap(a[j], a[k])
      else esteHeap ← true
      j ← k
end
```

Etapa II Deoarece iniţial avem MAX-HEAP(a) rezultă că pe primul loc se găseşte elementul maximal din a[0..n − 1]. Punem acest element la locul său final prin interschimbarea a[0] ↔ a[n − 1]. Acum a[0..n − 2] are proprietatea MAX-HEAP începând cu 1. Refacem MAX-HEAP pentru această secventă prin introducerea lui $a[0]$ în grămada a[0..n − 2], după care îl punem pe locul său final cel de-al doilea element cel mai mare din secvenţa de sortat. Procedeul continuă până când toate elementele ajung pe locurile lor finale.

Exemplu: Etapa a doua pentru exemplul anterior este arătată în fig. 4.2. sfex

Algoritmul de sortare prin selecţie sistematică este descris de subprogramul `heapSort`:

```
procedure HeapSort(a,n)
begin
   n1 ← ⌈(n − 1)/2⌉
   for ℓ ← n1 downto 0 do
       intrInGr(a, n, ℓ)
   r ← n-1
   while (r ≥ 1) do
       swap(a[0], a[r])
       intrInGr(a, r, 0)
       r ← r-1
end
```

Evaluarea algoritmului heapSort Considerăm $n = 2^k - 1$. În faza de construire a proprietăţii MAX-HEAP pentru toată secvenţa de intrare sunt efectuate următoarele operaţii:

- se construiesc vârfurile de pe nivelele $k-2, k-3, \ldots$;
- pentru construirea unui vârf de pe nivelul i se vizitează cel mult câte un vârf de pe nivelele $i+1, \ldots, k-1$;
- la vizitarea unui vârf sunt executate 2 comparaţii.

Rezultă că numărul de comparaţii executate în prima etapă este cel mult:

$$\sum_{i=0}^{k-2} 2(k-i-1)2^i = (k-1)2 + (k-2)2^2 + \cdots + 1 \cdot 2^{k-1} = 2^{k+1} - 2(k+1)$$

În etapa a II-a, dacă presupunem că $a[r]$ se găseşte pe nivelul i, introducerea lui $a[0]$ în grămada `a[1..r]` necesită cel mult $2i$ comparaţii. Deoarece r ia valori de la 1 la $n-1$ rezultă că în această etapă numărul total de comparatii este cel mult:

$$\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

Numărul total de comparaţii este cel mult:

$$\begin{aligned} C(n) &= 2^{k+1} - 2(k+1) + (k-2)2^{k+1} + 4 \\ &= 2^{k+1}(k-1) - 2(k-1) \\ &= 2k(2^k - 1) - 2(2^k - 1) \\ &= 2n \log_2 n - 2n \end{aligned}$$

De unde rezultă că numărul de comparaţii este $C(n) = O(n \log_2 n)$.

4.1.4 Exerciţii

Exerciţiul 4.1.2. Se consideră un tablou de structuri $(\texttt{a[i]} \mid 0 \leq i < n)$ şi un al doilea tablou $(\texttt{a[i]} \mid 0 \leq i < n)$ care conţine o permutare a mulţimii $\{0, 1, \ldots, n-1\}$. Să se scrie un program care rearanjează componentele tabloului `a` conform cu permutarea dată de `b`.

Exerciţiul 4.1.3. Să se modifice `ShellSort` astfel încât să utilizeze întotdeauna secvenţa de incremente $(h_0, \ldots, h_{k-1})$ dată prin recurenţa: $h_0 = 1; h_{i+1} = 3h_i + 1$ şi h_{k-1} cel mai mare increment de acest fel mai mic decât $n-1$. Apoi se va încerca găsirea de alte secvenţe de incremente care să producă algoritmi de sortare mai eficienţi.

Exerciţiul 4.1.4. Să se scrie o variantă recursivă a algoritmului de inserare într-un heap `intrInGr`. Care dintre cele două variante este mai eficientă?

Exerciţiul 4.1.5. Care este timpul de execuţie al algoritmului `heapSort` dacă secvenţa de intrare este ordonată crescător? Dar dacă este ordonată descrescător?

Exerciţiul 4.1.6. Să se proiecteze un algoritm care să determine cel de-al doilea cel mai mare element dintr-o listă. Algoritmul va executa $n + \lceil \log n \rceil - 2$ comparaţii.
Indicaţie. Fie $(a_0, \ldots, a_{n-1})$ secvenţa de intrare. Cel mai mare element din listă se poate face prin $n-1$ comparaţii. Se va utiliza următoarea metodă pentru determinarea maximului:

- se determină $b_0 = \max(a_0, a_1), b_1 = \max(a_2, a_3), \ldots$.
- se determină $c_0 = \max(b_0, b_1), c_1 = \max(b_2, b_3), \ldots$.
- etc.

Metodei de mai sus i se poate ataşa un arbore binar complet de mărime n, fiecare vârf intern reprezentând o comparaţie iar rădăcina corespunzând celui mai mare element. Pentru a determina cel de-al doilea cel mai mare element este suficient să se considere numai elementele care au fost comparate cu maximul. Numărul acestora este $\log_2 n - 1$. Va trebui proiectată o structură de date pentru memorarea acestor elemente.

Exerciţiul 4.1.7. (Selecţie.) Să se generalizeze metoda din exerciţiul anterior pentru a determina cel de-al k-lea cel mai mare element dintr-o listă. Care este complexitatea timp algoritmului? (Se ştie că există algoritmi care rezolvă această problemă în timpul $\Theta(n + \min(k, n-k)\log n)$).

Exerciţiul 4.1.8. (Sortare prin metoda turneelor.) Să se utilizeze algoritmul din exerciţiul precedent pentru sortarea unei liste. Care este complexitatea algoritmului de sortare?

Exerciţiul 4.1.9. Să se proiecteze programarea unui turneu de tenis la care participă 16 jucători astfel încât numărul de meciuri să fie minim iar primii doi să fie corect clasaţi relativ la relaţia de ordine "a mai bun ca b", definită astfel:

- dacă a învinge pe b atunci a mai bun ca b;
- dacă a mai bun ca b şi b mai bun ca c atunci a mai bun ca c.

Capitolul 5

Căutare

Alături de sortare, căutarea în diferite structuri de date constituie una din operaţiile cele mai des utilizate în activitatea de programare. Problema căutării poate îmbrăca diferite forme particulare:

- dat un tablou $(\texttt{s[i]} \mid 0 \leq i < n)$ şi un element a, să se decidă dacă există $i \in \{0, \ldots, n-1\}$ astfel încât $s[i] = a$;
- dată o listă înlănţuită (liniară, arbore, etc.) şi un element a, să se decidă dacă există un nod în listă a cărui informaţie este egală cu a;
- dat un fişier şi un element a, să se decidă dacă există o componentă a fişierului care este egală cu a;
- etc.

În plus, fiecare dintre aceste structuri poate avea sau nu anumite proprietăţi:

- informaţiile din componente sunt distincte două câte două sau nu;
- componentele sunt ordonate în conformitate cu o relaţie de ordine peste mulţimea informaţiilor sau nu;
- căutarea se poate face pentru toată informaţia memorată într-o componentă a structurii sau numai pentru o parte a sa numită *cheie*;
- cheile pot fi unice (ele identifică în mod unic componentele) sau multiple (o cheie poate identifica mai multe componente);
- între oricare două căutări structura de date nu suferă modificări (aspectul static) sau poate face obiectul operaţiilor de inserare/ştergere (aspectul dinamic).

Aici vom discuta numai o parte dintre aceste aspecte. Mai întâi le considerăm pe cele incluse în următoarea formulare abstractă:

Instanţă o mulţime univers $\mathbb{U}$, o submulţime $S \subseteq \mathbb{U}$ şi un element a din $\mathbb{U}$;
Întrebare $x \in S$?

Aspectul *static* este dat de cazul când între oricare două căutări mulţimea S nu suferă nici o modificare. Aspectul *dinamic* este obţinut atunci când între două căutări mulţimea S poate face obiectul următoarelor operaţii:

- Inserare.

  Intrare S, x;
  Ieşire $S \cup \{x\}$.

- Ştergere.

  Intrare S, x;
  Ieşire $S \setminus \{x\}$.

Evident, realizarea eficientă a căutării şi, în cazul aspectului dinamic, a operaţiilor de inserare şi de ştergere, depinde de structura de date aleasă pentru reprezentarea mulţimii S.

Tip de date	Implementare	Căutare	Inserare	Ştergere
Listă liniară	Tablouri	$O(n)$	$O(1)$	$O(n)$
	Liste înlănţuite	$O(n)$	$O(1)$	$O(1)$
Listă liniară ordonată	Tablouri	$O(\log_2 n)$	$O(n)$	$O(n)$
	Liste înlănţuite	$O(n)$	$O(n)$	$O(1)$

Figura 5.1: Complexitatea pentru cazul cel mai nefavorabil

5.1 Căutare în liste liniare

Mulţimea S este reprezentată printr-o listă liniară. Dacă mulţimea $\mathcal{U}$ este total ordonată, atunci S poate fi reprezentată de o listă liniară ordonată. Algoritmii corespunzători celor trei operaţii au fost deja prezentaţi în secţiunile 3.1 şi respectiv 3.2. Complexitatea timp pentru cazul cel mai nefavorabil este dependentă de implementarea listei. Tabelul 5.1 include un sumar al valorilor acestei complexităţi. Facem observaţia că valorile pentru operaţiile de inserare şi şteregere nu presupun şi componenta de căutare. În mod obişnuit, un element x este adăugat la S numai dacă el nu apare în S; analog, un element x este şters din S numai dacă el apare în S. Deci ambele operaţii ar trebui precedate de căutare. În acest caz, la valorile complexităţilor pentru inserare şi ştergere se adaugă şi valoarea corespunzătoare pentru căutare. De exemplu, complexitatea în cazul cel mai nefavorabil pentru inserare în cazul în care S este reprezentată prin tablouri neordonate devine $O(n)$ iar pentru cazul tablourilor ordonate rămâne aceeaşi, $O(n)$.

Pentru calculul complexităţii medie vom presupune că $a \in S$ cu probabilitatea q şi că a poate apărea în S la adresa adr cu aceeaşi probabilitate $\frac{q}{n}$. Complexitatea medie a căutărilor cu succes (a este găsit în S) este:

$$T^{med,s}(n) = \frac{3q(1+2+\cdots+n)}{n} + 2q = \frac{3q(n+1)}{2} + 2q$$

iar în cazul general avem:

$$T^{med}(n) = 3n - \frac{3nq}{2} + \frac{3q}{2} + 2$$

Cazul când se ia în considerare frecvenţa căutărilor. Presupunem că x_i este căutat cu frecvenţa f_i. Se poate demonstra că se obţine o comportare în medie bună atunci când $f_1 \geq \cdots \geq f_n$. Dacă aceste frecvenţe nu se cunosc aprioric, se pot utiliza *tablourile cu auto-organizare*. Într-un tablou cu auto-organizare, ori de câte ori se caută pentru un $a = s[i]$, acesta este deplasat la începutul tabloului în modul următor: elementele de pe poziţiile $1, \ldots, i-1$ sunt deplasate la dreapta cu o poziţie după care se pune a pe prima poziţie. Dacă în loc de tablouri se utilizează liste înlănţuite, atunci deplasările la dreapta nu mai sunt necesare. Se poate arăta [Knu76] că pentru tablourile cu auto-organizare complexitatea medie este:

$$T^{med,s}(n) \approx \frac{2n}{\log_2 n}$$

5.2 Arbori binari de căutare

Arborii $T(0, n-1)$ din definiţia arborilor de decizie pentru căutare sunt transformaţi în structuri de date înlănţuite asemănătoare cu cele definite în secţiunea **??**. Aceste structuri pot fi definite într-o manieră independentă:

Definiţia 5.1. *Un* arbore binar de căutare *este un arbore binar cu proprietăţile:*

1. *informaţiile din noduri sunt elemente dintr-o mulţime total ordonată;*
2. *pentru fiecare nod v, elementele memorate în subarborele stâng sunt mai mici decât valoarea memorată în v iar elementele memorate în subarborele drept sunt mai mari decât valoarea memorată în v.*

În continuare descriem implementările celor trei operaţii peste această structură.

Căutare. Operaţia de căutare într-un asemenea arbore este descrisă de următoarea procedură:

```
function poz(t, a)
begin
   p ← t
   while ((p ≠ NULL) and (a ≠ p->elt)) do
   if (a < p->elt)
   then p ← p->stg
   else p ← p->drp
   return p
end
```

Funcţia `poz` ia valoarea $NULL$ dacă $a \notin S$ şi adresa nodului care conţine pe a în caz contrar. Operaţiile de inserare şi de ştergere trebuie să păstreze invariantă următoarea proprietate:

valorile din lista inordine a nodurilor arborelui trebuie să fie în ordine crescătoare.

Pentru a realiza operaţia de inserare se caută intervalul la care aparţine x. Dacă în timpul procesului de căutare se găseşte un nod $*p$ cu p->$elt = x$ atunci arborele nu suferă nici o modificare (deoarece $x \in S$ implică $S \cup \{x\} = S$). Fie p adresa nodului de pe frontieră care defineşte intervalul. Dacă $x < p$->elt atunci x se adaugă ca succesor la stânga; în caz contrar se adaugă ca succesor la dreapta. Un exemplu este arătat în fig. 5.2.

Algoritmul care realizează operaţia de inserare are următoarea descriere:

```
procedure insereaza(t, x)
begin
   if (t = NULL)
   then t ← nodNou(x)
   else q ← t
        while (q ≠ NULL) do
           p ← q
           if (x < q->elt)
           then q ← q->stg
           else if (x > q->elt)
                then q ← q->drp
                else q ← NULL
        if (p->elt ≠ x)
        then q ← nodNou(x)
             if (x < p->elt)
             then p->stg ← q
             else p->drp ← q
end
```

Funţia `nodNou()` creează un nod al arborelui binar şi întoarce adresa acestuia:

```
function nodNou(x)
begin
   new(p)
   p->elt ← x
   p->stg ← NULL
   p->drp ← NULL
   return p
end
```

Operaţia de ştergere se realizează într-un mod asemănător. Se caută x în arborele care reprezintă mulţimea S. Dacă nu se găseşte un nod $*p$ cu p->$elt = x$ atunci arborele rămâne neschimbat (deoarece $x \notin S$ implică $S \setminus \{x\} = S$). Fie p referinţa la nodul care conţine pe x. Se disting următoarele trei cazuri:

1. $*p$ nu are fii (fig. 5.3a). Se şterge nodul $*p$.

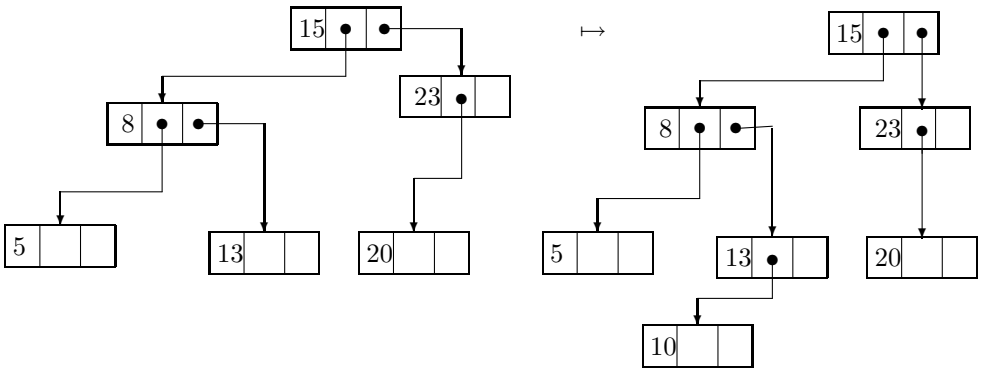


Figura 5.2: Inserare într-un arbore binar de căutare

2. $*p$ are un singur fiu (fig. 5.3b). Părintele lui $*p$ este legat direct la fiul lui $*p$.

3. $*p$ are doi fii (fig. 5.3c). Se determină cea mai mare valoare dintre cele mai mici decât x. Fie aceasta y. Ea se găseşte memorată în ultimul nod $*q$ din lista inordine a subarborelui din stânga lui $*p$. Se transferă informaţia y în nodul $*p$ după care se elimină nodul $*q$ ca în primele două cazuri.

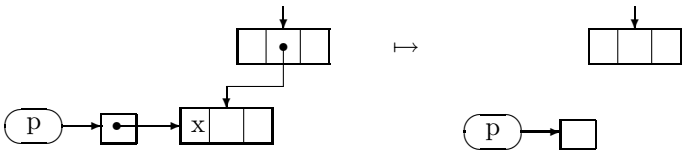
În n cazurile 1 şi 2 trebuie prevăzute şi situaţiile când $p = t$ ($*p$ este rădăcina arborelui). Următoarea procedură descrie algoritmul care rezolvă aceste două cazuri:

```
procedure elimCaz1sau2(prep, p)
begin
   if (p=t)
   then if (t->stg ≠ NULL)
        then t ← t->stg
        else t ← t->drp
   else if (p->stg ≠ NULL)
        then if (predp->stg = p)
             then predp->stg ← p->stg
             else predp->drp ← p->stg
        else if (predp->stg = p)
             then predp->stg ← p->drp
             else predp->drp ← p->drp
   delete(p)
end
```

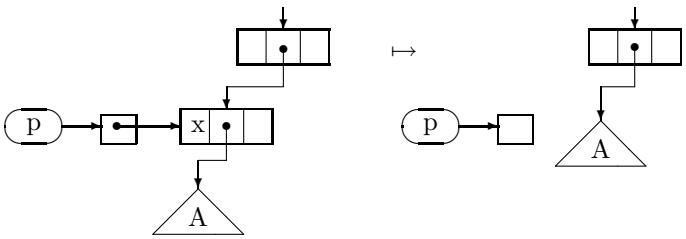
Acum algoritmul de căutare are următoarea descriere:

```
procedure elimina(t, x)
begin
   if (t ≠ NULL)
   then p ← t
        while ((p ≠ NULL) and (x ≠ p->elt)) do
           predp ← p
           if (x < p->elt)
           then p ← p->stg
           else p ← p->drp
        if (p ≠ NULL)
        then if (p->stg = NULL) or (p->drp = NULL)
             then elimCaz1sau2(prep, p)
             else q ← p->stg
                  predq ← p
                  while (q->drp ≠ NULL) do
```

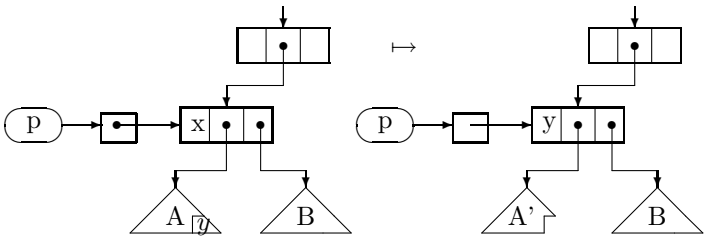
```
            predq ← q
            q ← q->drp
          p->elt ← q->elt
          elimCaz1sau2(predq, q)
end
```



a) Fara fii



b) Un fiu



c) Doi fii

Figura 5.3: Ştergere dintr-un arbore binar de căutare

De remarcat că ambele operaţii, inserarea şi eliminarea, necesită o etapă de căutare.

Observaţie: Structura de date utilizată aici se mai numeşte şi *arbore binar de căutare orientat pe noduri interne*. Această denumire vine de la faptul că elementele lui S corespund nodurilor interne ale arborelui de căutare. Nodurile externe ale acestuia, care au ca informaţie intervalele deschise corespunzătoare căutărilor fără succes, nu au mai fost incluse în definiţia structurii de date din următoarele motive: simplitate a prezentării, economie de memorie şi, atunci când interesează, intervalele pot fi determinate în timpul procesului de căutare. Sugerăm cititorului, ca exerciţiu, să modifice algoritmul de căutare astfel încât, în cazul căutărilor fără succes, să ofere la ieşire intervalul deschis la care aparţine a.

Mai există o variantă a structurii, numită *arbore binar de căutare orientat pe frontieră*, în care elementele lui S sunt memorate atât în nodurile interne cât şi în nodurile de pe frontieră. Informaţiile din nodurile de pe frontieră sunt în ordine crescătoare de la stânga la dreapta şi putem gândi că ele corespund intervalelor $(-\infty, x_0], (x_0, x_1], \ldots, (x_{n-1} + \infty)$. Algoritmul de căutare într-o astfel de structură va face testul $p\text{->}val = a$ numai dacă $*p$ este un nod pe frontieră. În caz când avem egalitate rezultă că a aparţine mulţimii S; altfel a aparţine intervalului deschis mărginit la dreapta de $p\text{->}elt$. Pentru $S = \{5, 8, 13, 15, 20, 23\}$, structura de date este reprezentată schematic în fig. 5.4. sfobs

Exerciţiul 5.2.1. Să se descrie proceduri pentru operaţiile de căutare, inserare şi ştergere pentru arbori binari de căutare orientaţi pe frontieră. Operaţiile vor păstra proprietatea ca fiecare nod să aibă exact doi fii.

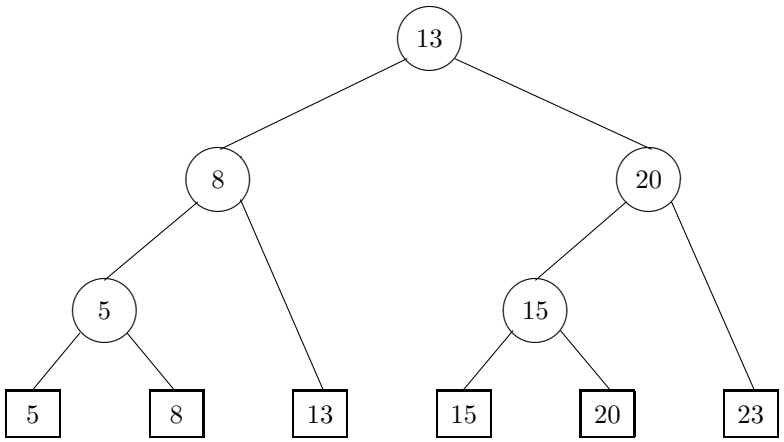


Figura 5.4: Arbore orientat pe frontieră

5.2.0.1 Exerciţii

Exerciţiul 5.2.2. Un *nod complet* este un nod care are doi fii. Să se arate că într-un arbore binar numărul nodurilor complete este egal cu numărul nodurilor frunză minus 1.

Exerciţiul 5.2.3. Să se scrie un subprogram care determină cel mai mare element dintr-un arbore binar de căutare.

Exerciţiul 5.2.4. Să se scrie un subprogram care determină cel mai mic element dintr-un arbore binar de căutare.

Exerciţiul 5.2.5. Presupunem că se doreşte efectuarea unui experiment care să verifice problemele care apar la execuţiile aleatoare de perechi de operaţii inserare/ştergere. Următoarae strategie nu este perfect aleatoare dar e destul de aproape. Se construieşte un arbore cu n elemente numere întregi alese aleator din intervalul $[1, m]$, unde $m = \alpha \cdot n$. Apoi sunt realizate n^2 perechi de inserări urmate de ştergeri. Presupunem că există un subprogram `randInt(`a, b`)` care întoarce un întreg ales aleator uniform din intervalul $[a, b]$.

1. Să se arate cum se generează aleator un număr întreg din intervalul $[1, m]$ care **nu** este deja în arbore. Ce se poate spune despre timpul de execuţie al acestei operaţii?
2. Să se arate cum se generează aleator un număr întreg din intervalul $[1, m]$ care este deja în arbore. Este utilă o astfel de operaţie? Ce se poate spune despre timpul de execuţie al acestei operaţii?
3. Care este o bună alegere pentru α? De ce?

Exerciţiul 5.2.6. Să se scrie un subprogram care listează elementele k dintr-un arbore binar de căutare cu proprietatea $k_1 \leq k \leq k_2$ pentru k_1 şi k_2 daţi. Care este timpul de execuţie a subprogramului?

Capitolul 6

Enumerare

Apar adesea situaţii când în descrierea soluţiei unei probleme este necesară enumerarea elementelor unei mulţimi. Un exemplu îl constituie algoritmii bazaţi pe tehnica backtracking, unde enumerarea completă este transformată într-o enumerare parţială. În acest capitol considerăm numai trei studii de caz: enumerarea permutărilor, enumerarea elementelor produsului cartezian a unei mulţimi cu ea însăşi de n ori şi enumerarea arborilor parţiali într-un graf.

6.1 Enumerarea permutărilor

În această secţiune ne ocupăm de generarea listei cu cele $n!$ permutări ale mulţimii $\{0, 1, \ldots, n-1\}$. Notăm cu S_n mulţimea acestor permutări.

6.1.1 Enumerarea recursivă

Scrierea unui program recursiv pentru generarea permutărilor trebuie să aibă ca punct de plecare o definiţie recursivă pentru S_n. Dacă $(i_0, \ldots, i_{n-1})$ este o permutare din S_n cu $i_k = n-1$ atunci $(\ldots i_{k-1}, i_{k+1}, \ldots)$ este o permutare din S_{n-1}. Deci orice permutare din S_n se obţine dintr-o permutare din S_{n-1} prin inserţia lui n în una din cele n poziţii posibile. Evident, permutări distincte din S_{n-1} vor produce permutări diferite în S_n. Aceste observaţii conduc la următoarea definiţie recursivă:

$$S_1 = \{(0)\}$$
$$S_n = \{(i_0, \ldots, i_{n-2}, n-1), \ldots, (n-1, i_0, \ldots, i_{n-2}) \mid (i_0, \ldots, i_{n-2}) \in S_{n-1}\}$$

Generalizăm prin considerarea mulţimii $S_n(\pi, k)$ a permutărilor din S_n ce pot fi obţinute din permutarea $\pi \in S_k$. Pentru $S_n(\pi, k)$ avem următoarea definiţie recursivă:

$$S_n(\pi, k) = S_n((i_0, \ldots, i_{k-1}, k), k) \cup \cdots \cup S_n((k, i_0, \ldots, i_{k-1}), k)$$

unde $\pi = (i_0, \ldots, i_{k-1})$. Are loc $S_n = S_n((0), 0)$ şi $S_n(\pi, n-1) = \{\pi\}$. Vom scrie un subprogram recursiv pentru calculul mulţimii $S_n(\pi, k)$ şi apoi vom apela acest subprogram pentru determinarea lui S_n. Pentru reprezentarea permutărilor utilizăm tablouri unidimensionale. Subprogramul recursiv care calculează mulţimea $S_n(\pi, k)$ are următoarea descriere:

```
procedure genPermRec(p, k)
begin
   if (k = n-1)
   then scriePerm(p,n)
   else p[k] ← k
        for i ← k-1 downto 0 do
            genPermRec(p,k+1)
            swap(p[i+1],p[i])
        genPermRec(p,k+1)
end
```

Enumerarea tuturor celor $n!$ permutări se realizează prin execuţia următoarelor două instrucţiuni:

```
p[0] ← 0
genPermRec(p, 0)
```

6.1.2 Enumerarea nerecursivă

Metodei recursive i se poate ataşa un arbore ca în fig. 6.1. Fiecare vârf intern din arbore corespunde unui apel recursiv. Vârfurile de pe frontieră corespund permutărilor din S_n. Ordinea apelurilor recursive coincide cu ordinea dată de parcurgerea DFS a acestui arbore. Dacă vom compara programul recursiv care generează permutările cu varianta recursivă a algoritmului DFS, vom observa că ele au structuri asemănătoare. Şi este normal să fie aşa, pentru că programul de generare a permutărilor realizează acelaşi lucru: parcurgerea mai întâi în adâncime a arborelui din fig. 6.1. Deci şi varianta nerecursivă a algoritmului de generare a permutărilor poate fi obţinut din varianta nerecursivă a algoritmului DFS. Locul tabloului p din DFS este luat de o funcţie $f(k, i, p)$ care pentru o permutare $(p[0], \ldots, p[k-1])$ (aflată pe nivelul $k-1$ în arbore) determină al i-lea succesor, $0 \leq i \leq k$. Deoarece pentru orice permutare, corespunzătoare unui vârf de pe nivelul $k \geq 1$ în arbore, putem determina permutarea din vârful tată, rezultă că nu este necesară memorarea permutărilor în stivă. Astfel, stiva va memora, pentru fiecare nivel din arbore, indicele succesorului ce urmează a fi vizitat.

```
procedure genPerm(n)
begin
   k ← 0
   S[0] ← 0
   while (k ≥ 0) do
      if (S[k] ≥ 0)
      then f(k, S[k], p)
           S[k-1] ← S[k-1]-1
           if (k = n-1)
           then scriePerm(p,n)
           else k ← k+1
                S[k] ← k
      else aux ← p[0]
           for i ← 0 to k-1 do
               p[i] ← p[i+1]
           p[k] ← aux
           k ← k-1
end
```

Funcţia $f(k, i, p)$ este calculată de următorul subprogram:

```
function f(k, i, p)
begin
   if (i = k)
   then p[k] ← k
   else aux ← p[i+1]
        p[i+1] ← p[i]
        p[i] ← aux
end
```

Exerciţiul 6.1.1. Să se scrie un subprogram care, pentru numerele naturale i şi n date, determină a i-a permutare (în ordinea lexicografică) din S_n.

Observaţie: Algoritmul de mai sus poate fi îmbunătăţit din punctul de vedere al complexităţii timp. Mai întâi să notăm faptul că orice algoritm de enumerare a permutărilor are complexitatea $O(n!) = O(n^n)$. Ideea este de a găsi un algoritm care să efectueze cât mai puţine operaţii pentru determinarea permutării succesoare. Execuţia a $c' \cdot n!$ operaţii în loc de $c \cdot n!$ cu $c' < c$, semnifică, de fapt, o reducere cu $(c - c') \cdot n!$

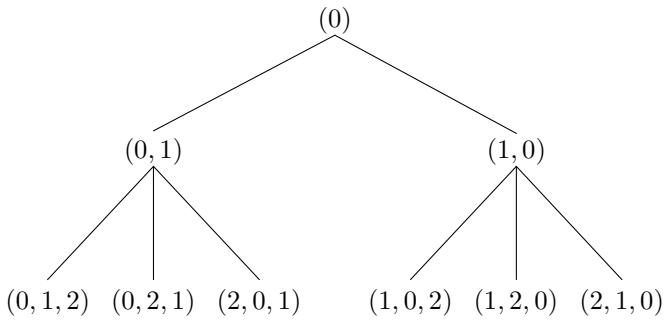


Figura 6.1: Arborele permutărilor generat de metoda recursivă

a complexităţii timp. Un astfel de algoritm este obţinut după cum urmează. În arborele din figura 6.1 se schimbă ordinea succesorilor permutării $(1, 0)$. Ordinea permutărilor de pe orice nivel din noul arbore are proprietatea că oricare două permutări succesive diferă printr-o transpoziţie de poziţii vecine. Dacă se reuşeşte generararea permutărilor direct în această ordine, fără a simula parcurgerea arborelui, atunci se obţine un program care generează permutările cu număr minim de operaţii. Regula generală prin care se obţine această ordine este următoarea (fig. 6.2):

> La fiecare nivel din arborele apelurilor recursive, succesorii vârfurilor de rang par îşi schimbă ordinea astfel încât cel mai din stânga devine cel mai din dreapta şi cel mai din dreapta devine cel mai din stânga.

Evitarea simulării parcurgerii arborelui se realizează prin utilizarea unui vector de "direcţii", $d = (d[k] \mid 0 \leq k < n)$, cu următoarea semnificaţie:

- $d[k] = +1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(p[0], \ldots, p[k-1], k), \ldots, (k, p[0], \ldots, p[k-1])$;
- $d[k] = -1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(k, p[0], \ldots, p[k-1]), \ldots, (p[0], \ldots, p[k-1], k)$;

În acest mod, vectorul d descrie complet drumul de la rădăcină la un grup de permutări pentru care transpoziţia se aplică în aceeaşi direcţie. Determinarea indicelui i la care se aplică transpoziţia se poate face utilizând un tablou care memorează permutarea inversă. Dacă notăm acest tablou cu $pinv$ atunci, utilizând relaţia $p[pinv[k]] = k$, obţinem că locul i unde se află k este $pinv[k]$. Noua poziţie a lui k va fi $i + d[k]$. sfobs

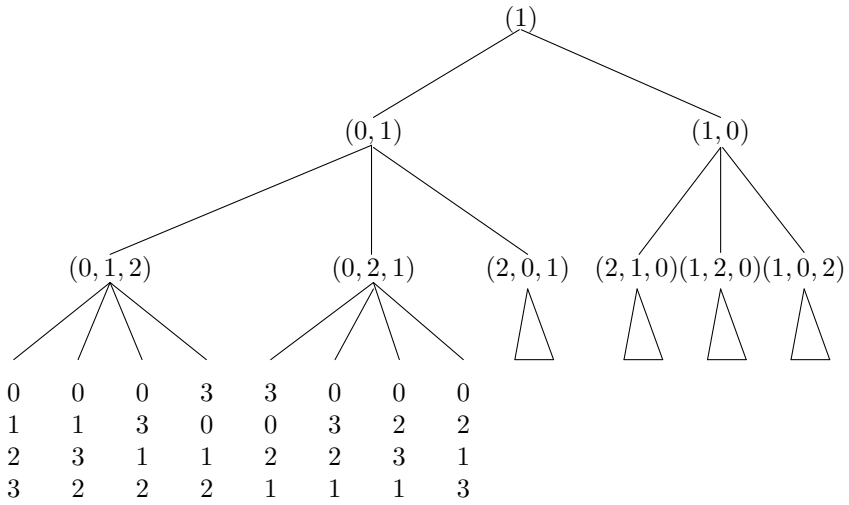


Figura 6.2: Arborele permutărilor modificat

6.2 Enumerarea elementelor produsului cartezian

Considerăm următoarea problemă:

> Date două numere întregi pozitive n şi m, să se genereze toate elementele produsului cartezian $\{0,\ldots,m-1\}^n = \{0,\ldots,m-1\} \times \cdots \times \{0,\ldots,m-1\}$ (de n ori).

Mulţimea $\{0,\ldots,m-1\}^n$ poate fi reprezentată printr-un arbore cu n nivele în care fiecare vârf intern are exact m succesori iar vârfurile de pe frontieră corespund elementelor mulţimii. De exemplu, arborele corespunzător cazului $m = 2$ şi $n = 3$ este reprezentat grafic în fig. 6.3.

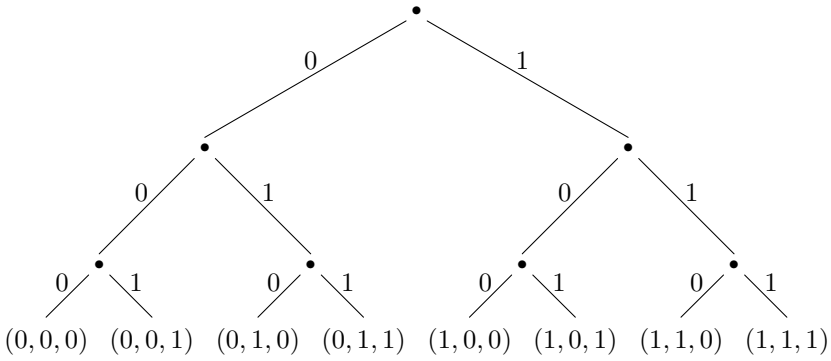


Figura 6.3: Arborele produsului cartezian

Fiecare vârf în arbore este identificat de drumul de la rădăcină la el: notăm acest drum cu $(x_0,\ldots,x_k)$. Pentru vârfurile de pe frontieră avem $k = n-1$ iar drumul $(x_0,\ldots,x_{n-1})$ este un element al produsului cartezian. Algoritmul pe care-l prezentăm va simula generarea acestui arbore prin parcurgerea DFS. Deoarece "adresele" vârfurilor succesoare pot fi determinate printr-un calcul foarte simplu, stiva este reprezentată de o variabilă simplă k, care indică poziţia vârfului stivei.

```
procedure genProdCart(n, m)
begin
   k ← 0
   x[0] ← -1
   while (k ≥ 0) do
      if (x[k] < m-1)
      then x[k] ← x[k]+1
           if (k = n)
           then scrieElement(x,n)
           else k ← k+1
                x[k] ← -1
      else k ← k-1
end
```

Varianta recursivă a programului `GenProdCart` este următoarea:

```
procedure genProdCartRec(x, k)
begin
   for j ← 0 to m-1 do
       x[k] ← j
       if (k = n-1)
       then scrieElement(x, n)
       else genProdCartRec(x, k+1)
end
```

Enumerarea tuturor elementelor produsului cartezian se face executând apelul:

```
genProdCartRec(x, 0)
```

Bibliografie

[CLR93] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1993.

[Cro92] Cornelius Croitoru. *Tehnici de bază în optimizarea combinatorie*. Editura Universităţii "Al.I.Cuza", Iaşi, 1992.

[HS84] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1984.

[HSAF93] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, 1993.

[Knu76] D.E. Knuth. *Sortare şi căutare*, volume 3 of *Tratat de programarea calculatoarelor*. Editura tehnică, Bucureşti, 1976.

[Luc93] D. Lucanu. *Programarea algoritmilor: Tehnici elementare*. Editura Universităţii "Al.I.Cuza", Iaşi, 1993.

[MS91] B.M.E. Morret and H.D. Shapiro. *Algorithms from P to NP: Design and Efficiency*. The benjamin/Cummings Publishing Company, Inc., 1991.