



XEL LINUX

CE2BF3B0
CABBA11B
UBA1406463
F942ECD33B5
[HTTP://www.GANGSTERSERVER.COM](http://www.GANGSTERSERVER.COM)



Dorel Lucanu • Mitică Craus

• PROIECTAREA • ALGORITMILOR

sw & pici

"You can go a long way with a smile.
You can go a lot farther with a smile and a
gun." (Al Capone)

Be the best, fuck the rest!

Dorel Lucanu a absolvit secția de Informatică a Facultății de Matematică, Universitatea „A.I.I. Cuza” Iași; este doctor în științe al Institutului de Matematică al Academiei Române. În prezent, este profesor universitar la Facultatea de Informatică, Universitatea „A.I.I. Cuza” Iași, unde predă discipline de programare, algoritmi și metode formale în ingineria software de la înființarea acesteia (1992). Este laureat al premiului „Grigore Moisil” acordat de Academia Română.

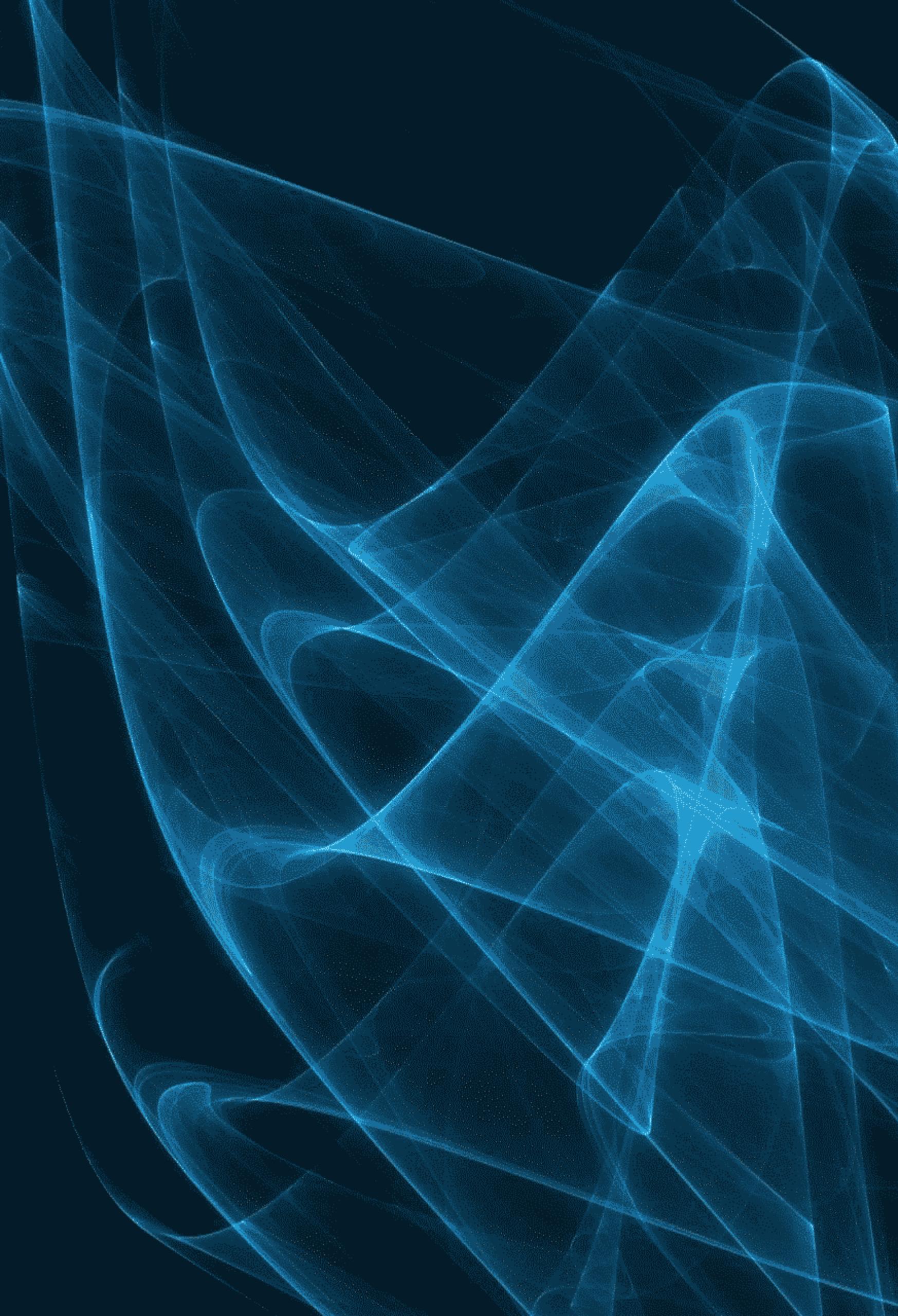
Mitică Craus a absolvit secția de Informatică a Facultății de Matematică, Universitatea „A.I.I. Cuza” Iași; este doctor în științe al Facultății de Informatică din cadrul aceleiași universități. În prezent, este profesor universitar la Facultatea de Automatică și Calculatoare, Universitatea Tehnică „Gheorghe Asachi” Iași, unde susține cursuri de algoritmi sevențiali și paraleli, structuri de date, programare, limbaje formale și translatoare din 1997.

Cuprins

Prefață	9
1 Despre algoritmi	13
1.1 Limbaj algoritmic	13
1.2 Probleme, algoritmi și programe	27
1.3 Măsurarea performanțelor unui algoritm	30
1.4 Referințe bibliografice	40
2 Tipuri de date de nivel înalt	41
2.1 Liste liniare	41
2.2 Liste liniare ordonate	51
2.3 Liste circulare	55
2.4 Stive	57
2.5 Cozi	62
2.6 Liste generalizate	66
2.7 Arbori binari	71
2.8 Coada cu priorități	79
2.9 Grafuri	83
2.10 <i>Union-find</i>	101
2.11 Analiza amortizată	104
2.12 Referințe bibliografice	108
3 Derecursivare	109
3.1 Parcurgerea în inordine a arborilor binari	109
3.2 Recursia liniară	113
3.3 Recursia în cascadă	116
3.4 Memorarea într-un tabel a rezultatelor subproblemelor	120
3.5 Referințe bibliografice	122
4 Sortare internă	123
4.1 Sortare bazată pe comparații	124
4.2 Sortare prin distribuire	138
4.3 Sortare prin numărare	142
4.4 Sortare topologică	144

4.5	Referințe bibliografice	146
5	Căutare	147
5.1	Căutare în liste liniare	148
5.2	Modelul de calcul al arborilor de decizie pentru căutare	149
5.3	Arbore binari de căutare	155
6	Tipuri de date avansate pentru căutare	161
6.1	Arbore echilibrați	161
6.2	Dispersia (hashing)	184
6.3	Arbore digitali (tries)	190
6.4	Referințe bibliografice	198
7	Căutare peste siruri	201
7.1	Căutarea naivă	202
7.2	Algoritmul Knuth-Morris-Pratt	202
7.3	Algoritmul Boyer-Moore	205
7.4	Algoritmul Rabin-Karp	206
7.5	Expresii regulate	208
7.6	Mai multe patternuri și înlocuire	211
7.7	Exerciții	214
7.8	Referințe bibliografice	215
8	Despre paradigmile de proiectare	217
8.1	Aspecte generale	217
8.2	Un exemplu simplu de paradigmă: eliminarea	218
8.3	Alte considerații privind paradigmile de proiectare	222
8.4	Referințe bibliografice	223
9	Algoritmi greedy	225
9.1	Memorarea eficientă a programelor	225
9.2	Prezentare intuitivă a paradigmiei	226
9.3	Arbore ponderați pe frontieră optimi	227
9.4	Arbore Huffman	232
9.5	Interclasare optimă pe două căi	234
9.6	Problema rucsacului I (varianta continuă)	236
9.7	Secvențializarea optimă a activităților	239
9.8	Problema instructorului de schi	242
9.9	Arborele parțial de cost minim	244
9.10	Prezentare formală a paradigmiei	246
9.11	Exerciții	248
9.12	Referințe bibliografice	251

10 Divide-et-impera	253
10.1 Prezentare generală	253
10.2 Sortare prin interclasare (<i>Merge Sort</i>)	255
10.3 Sortarea rapidă (<i>Quick Sort</i>)	258
10.4 Selecționare	262
10.5 Linia orizontului	264
10.6 Transformată Fourier discretă	270
10.7 Exerciții	274
10.8 Referințe bibliografice	275
11 Programare dinamică	277
11.1 Exemplu: drum optim într-o rețea piramidală de numere	277
11.2 Prezentarea intuitivă a paradigmiei	278
11.3 Alocarea resurselor	279
11.4 Drumurile cele mai scurte între oricare două vârfuri ale unui digraf	282
11.5 Problema rucsacului II (varianta discretă)	285
11.6 Subsecvența crescătoare maximală	293
11.7 Distanța între siruri	295
11.8 Arbori binari de căutare optimali	298
11.9 Prezentarea formală a paradigmiei	301
11.10 Exerciții	304
11.11 Referințe bibliografice	308
12 Backtracking și branch-and-bound	309
12.1 Organizarea spațiului soluțiilor candidat	310
12.2 <i>Backtracking</i>	319
12.3 <i>Branch-and-bound</i>	320
12.4 Colorarea grafurilor	325
12.5 Problema celor n regine	327
12.6 Submultime de sumă dată	330
12.7 Problema rucsacului II (continuare)	332
12.8 Perspico	337
12.9 Exerciții	340
12.10 Referințe bibliografice	342
13 Probleme NP-complete	343
13.1 Algoritmi nedeterminiști	343
13.2 Clasele \mathbb{P} și \mathbb{NP}	344
13.3 Probleme \mathbb{NP} -complete	348
13.4 Exerciții	354
13.5 Referințe bibliografice	354
Bibliografie	355
Index	359



Prefață

Această carte are o perioadă de elaborare foarte lungă, greu de justificat, ce își găsește originile în volumele doi și trei ale cursului multiplicat [Luc96] și ale cursului [CB02]. Cartea are la bază experiența acumulată de-a lungul a optzece ani în cazul primului autor și a doisprezece ani în cazul celui de-al doilea autor în predarea cursurilor de strucruri de date și algoritmi, ce au purtat diverse denumiri de-a lungul anilor, la Facultatea de Informatică a Universității „Al. I. Cuza”, respectiv la Facultatea de Automatizări și Calculatoare a Universității Tehnice „Gh. Asachi”, ambele din Iași.

De ce o nouă carte despre algoritmi?

Studierea algoritmilor este o componentă obligatorie atât a programelor școlare liceale, cât și a programelor universitare ale liceelor și facultăților ce includ clase, respectiv secții cu profil de informatică. Ca atare, au apărut multe manuale și cărți despre algoritmi. În aceste condiții se pune firesc întrebarea: ce poate aduce în plus sau original o nouă carte de algoritmi?

Asistăm în această perioadă la două fenomene majore: dezvoltarea tehnologică foarte rapidă și accesul la un volum mare de informații. În fiecare zi apar situații în care trebuie să învățăm din mers noi tehnologii, noțiuni sau concepte. Datorită grabei permanente, de multe ori suntem superficiali în înțelegerea acestora. Această superficialitate poate fi de înțeles atunci când dorim să știm doar despre ce este vorba, eventual pentru a putea utiliza un nou produs sau o nouă tehnologie, dar nu este nici justificabilă și nici scuzabilă atunci când dorim să ne formăm ca specialisti ai domeniului respectiv. Din păcate, mulți autori de cărți și manuale, din dorința de a avea succes, au căzut în capcana acestui sistem de „învățat din fugă”. Cartea de față se dorește a fi un contrabalans al acestei tendințe. Gândirea algoritmică trebuie să fie un pilon de bază pentru oricine dorește să devină un specialist în dezvoltarea de software (produse soft, programe de calculator). Gândirea algoritmică nu poate fi însușită doar prin învățarea unei colecții de algoritmi; ea presupune însușirea unui limbaj specific care trebuie să fie independent de limbajul de programare, abilitatea de a abstractiza pentru a vedea ce este esențial în descrierea unui algoritm pentru a-l putea aplica în situații similare, cunoașterea de instrumente prin care să se poată analiza algoritmul sau problema din punctul de vedere al corectitudinii, timpului de execuție și/sau a spațiului de memorie necesar. Toate aceste aspecte le întâlnim în

toate etapele dezvoltării unei aplicații soft. Când spunem specialist în dezvoltarea de software, ne referim la ceea ce se înțelege de fapt prin *inginer software*, care e diferit de un *tehnician software*: un tehnician software este capabil să scrie un program pe baza unor specificații date sau a unui algoritm dat; un inginer software utilizează în plus instrumente matematice cu ajutorul cărora proiectează soluția și dovedește că soluția găsită este corectă și performantă. Din acest punct de vedere, cartea de față a fost scrisă cu gândul la inginerul software.

Gândirea algoritmică este una interdisciplinară. Această interdisciplinaritate are două surse. Prima se referă la diversitatea domeniilor problemelor care necesită o soluție algoritmică. A doua sursă este dată de partea de analiză a algoritmilor: aici vedem de ce este util să stăpânim instrumente învățate la logică, grafuri, algebră, analiză matematică, probabilități etc. Credeam că această interdisciplinaritate este unul dintre elementele principale care dau frumusețe acestui domeniu.

Sumarizând, considerăm că am scris această carte cu gândul de a fi utilă pentru înșurarea pe îndelete a unei gădiri algoritmice, care să permită descoperirea frumuseții acestei discipline, utilizând instrumentele cele mai potrivite.

Cui se adresează această carte?

În termeni generali, răspunsul la această întrebare este dat de secțiunea precedentă. Aici încercăm să identificăm câteva situații concrete în care această carte este utilă.

În primul rând, cartea poate fi utilizată ca suport sau ca material consultativ la cursurile despre structuri de date și algoritmi de la facultățile sau secțiile cu profil informatic, atât de la universitățile clasice, cât și de la cele cu profil tehnic.

Elevii de liceu găsesc în această carte multe dintre temele discutate la orele de informatică, descrise într-o manieră mai riguroasă și cu informații suplimentare sau complementare. Multe dintre problemele de la concursurile de informatică își au sursa de inspirație în paradigmile prezentate aici.

Pentru profesorii de liceu care predau disciplina informatică, cartea poate constitui o sursă de inspirație pentru orele de predare sau un suport pentru perfecționare.

Orice specialist în dezvoltarea de software poate găsi în această carte o sursă de inspirație pentru găsirea de soluții la probleme concrete.

Ce nu se găsește în această carte...

Lumea algoritmilor este una foarte complexă și cuprinzătoare, a cărei descriere nu poate fi inclusă într-o singură carte, fie ea și de natură enciclopedică așa cum este [Kao08]. Deoarece punem un accent deosebit pe conceptul de paradigmă de proiectare a algoritmilor, în sensul de model de gădere algoritmică, următoarele aspecte, importante în studiul algoritmilor, nu au putut fi incluse în prezența carte.

Unul dintre domeniile cele mai proifice pentru gădirea algoritmilor este teoria grafurilor și combinatorica. Deși cartea include multe exemple sau studii de caz din acest domeniu, nu am inclus un capitol dedicat algoritmilor specifici, deoarece considerăm că el constituie, de obicei, o disciplină separată.

Problemele NP-dificele și problemele NP-complete sunt prezentate succint. Nu există un capitol special dedicat euristicilor și algoritmilor de aproximare pentru aceste probleme. Este în intenția autorilor de a elabora, eventual, în colaborare cu alți colegi, o continuare a prezentei cărți cu alte paradigme de algoritmi: probabilisti, genetici, paraleli, distribuiți,

Analiza algoritmilor presupune studierea și a altor clase de complexitate care includ probleme practice de mare importanță. Prezentarea acestora nu intră în scopul prezentei cărți.

Multumiri

Apariția unei cărți este rezultatul muncii mai multor persoane, chiar dacă pe copertă apar scrise numele a numai doi autori. Multe dintre secțiunile acestei cărți sunt rodul discuțiilor cu domnii profesori Cornelius Croitoru și Corneliu Bârsan (în prezent senior SDE la Microsoft). Capitolul despre structuri de date a căpătat actualul contur după multe discuții cu Simona Orzan (actualmente la Universitatea din Eindhoven). Forma finală a soluțiilor unor studii de caz de la paradigmă este rezultatul discuțiilor avute cu Laurențiu Iancu (în prezent la Microsoft). Încurajările date de către domnii profesori Dan Cristea, Henri Luchian și Sabin Buraga ne-au fost de mare ajutor.

Dacă există acum mai puține greșeli, se datorează și foștilor noștri studenți Dana Buzdugan, Laurențiu Vornicu, Ciprian Bacalu, Georgiana Caltais, Eugen Goria, Ionuț Gavrilă și Alexandru Archip.



Capitolul 1

Despre algoritmi

1.1 Limbaj algoritmic

1.1.1 Introducere

Un *algoritm* este o secvență finită de operații (pași) care, atunci când este executată, produce o soluție corectă pentru o problemă precizată. Tipul operațiilor și ordinea lor în secvență respectă o logică specifică.

Algoritmii pot fi descriși în orice limbaj, pornind de la limbajul natural până la limbajul nativ al unui calculator specific. Un limbaj al cărui scop unic este cel de a descrie algoritmi se numește *limbaj algoritmic*. Limbajele de programare sunt exemple de limbaje algoritmice.

În această secțiune 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 mulțime de entități (componente) de tip dată (informație reprezentabilă în memoria unui calculator), numită și *domeniul* tipului, și o mulțime de operații peste aceste entități. Convenim să grupăm tipurile de date în trei categorii:

- *tipuri de date elementare*, în care entitățile sunt indivizibile;
- *tipuri de date structurate de nivel jos*, în care entitățile sunt structuri relativ simple obținute prin asamblarea de date elementare sau date structurate, iar operațiile sunt definite la nivel de componentă;
- *tipuri de date structurate de nivel înalt*, în care componentele sunt structuri mai complexe, iar operațiile sunt implementate de algoritmi proiectați de către utilizatori.

Primele două categorii sunt dependente de limbaj și de aceea descrierile lor sunt incluse în această secțiune. Tipurile de nivel înalt pot fi descrise într-o manieră independentă de limbaj și descrierile lor sunt incluse în capitolul 2. Un tip de date descris într-o manieră independentă de reprezentarea valorilor și implementarea operațiilor se numește *tip de date abstract*.

Pașii unui algoritm și ordinea logică a acestora sunt descrise cu ajutorul *instrucțiunilor*. O secvență de instrucțiuni care acționează asupra unor structuri de date

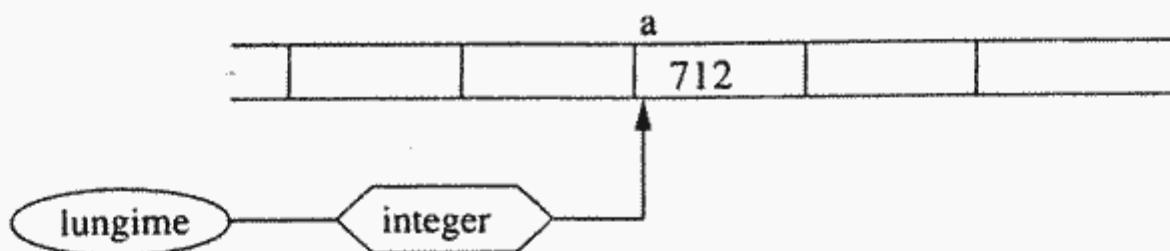


Figura 1.1: Memoria

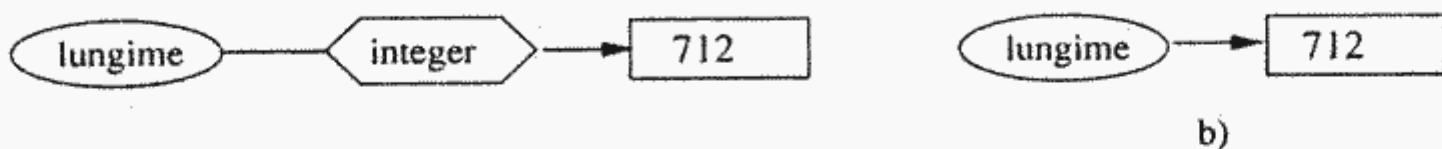


Figura 1.2: Variabilă

precizate se numește *program*. În secțiunea 1.2 vom vedea care sunt condițiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm.

1.1.2 Modelarea memoriei

Memoria este reprezentată ca o secvență de celule (locații), fiecare celulă având asociată o *adresă* și putând memora (stoca) o dată de un anumit tip (figura 1.1). Accesul la memorie este realizat cu ajutorul variabilelor. O *variabilă* este caracterizată de:

- un *nume* cu ajutorul căruia variabila este referită,
- o *adresă* care desemnează o locație de memorie și
- un *tip de date* care descrie natura datelor memorate în locația de memorie asociată variabilei.

Dacă în plus adăugăm și data memorată la un moment dat în locație, atunci obținem o *instanță a variabilei*. O variabilă este reprezentată grafic ca în figura 1.2.a. Atunci când tipul se subînțelege din context, vom utiliza reprezentarea scurtă sugerată în 1.2.b. Convenim să utilizăm fontul **type writer** pentru notarea variabilelor și fontul **mathnormal** pentru notarea valorilor memorate de variabile.

1.1.3 Tipuri de date elementare

Numere întregi. Valorile sunt numere întregi, iar operațiile sunt cele uzuale: adunarea (+), înmulțirea (*), scăderea (-) etc.

Numere reale. Deoarece prin dată înțelegem o entitate de informație reprezentabilă în memoria calculatorului, domeniul tipului numerelor reale este restrâns la submulțimea numerelor raționale. Operațiile sunt cele uzuale.

Valori booleene. Domeniul include numai două valori: **true** și **false**. Peste aceste valori sunt definite operațiile logice **and**, **or** și **not** cu semnificațiile cunoscute.

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

Pointeri. Domeniul unui tip pointer constă din adrese de variabile aparținând altui tip. Presupunem existența valorii NULL care nu referă nici o variabilă; cu ajutorul ei putem testa dacă un pointer referă sau nu o variabilă. Nu considerăm operații peste aceste adrese. Cu ajutorul unei variabile pointer, numită pe scurt și pointer, se realizează referirea indirectă a unei locații de memorie. Un pointer este reprezentat grafic ca în figura 1.3.a. Instanța variabilei pointer p are ca valoare adresa unei variabile de tip întreg. Am notat cu integer^* tipul pointer al cărui domeniu este format din adrese de variabile de tip întreg. Această convenție este extinsă la toate tipurile. Variabila referată de p este notată cu $*p$. În figurile 1.3.b și 1.3.c sunt date reprezentările grafice simplificate ale pointerilor.

Pointerii sunt utilizati la manipularea variabilelor dinamice. O *variabilă dinamică* este o variabilă care poate fi creată și distrusă în timpul execuției programului. Crearea unei variabile dinamice se face cu ajutorul instrucțiunii `new`. De exemplu, `new(p)` are ca efect crearea variabilei $*p$. Distrugerea (eliberarea memoriei) variabilei $*p$ se face cu ajutorul instrucțiunii `delete(p)`.

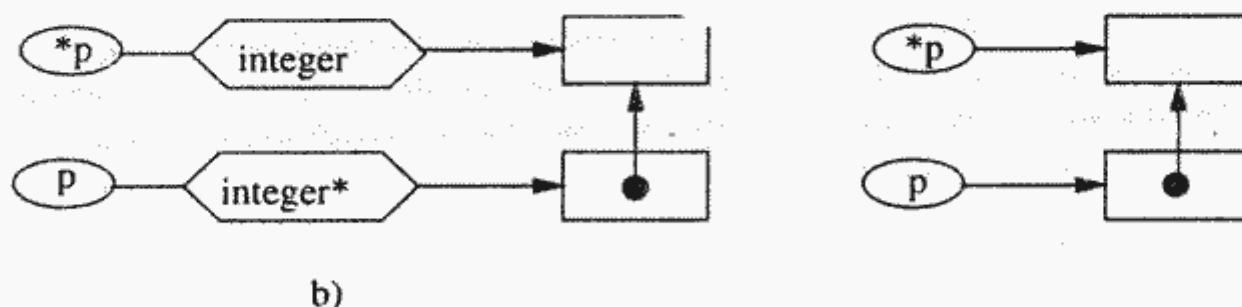
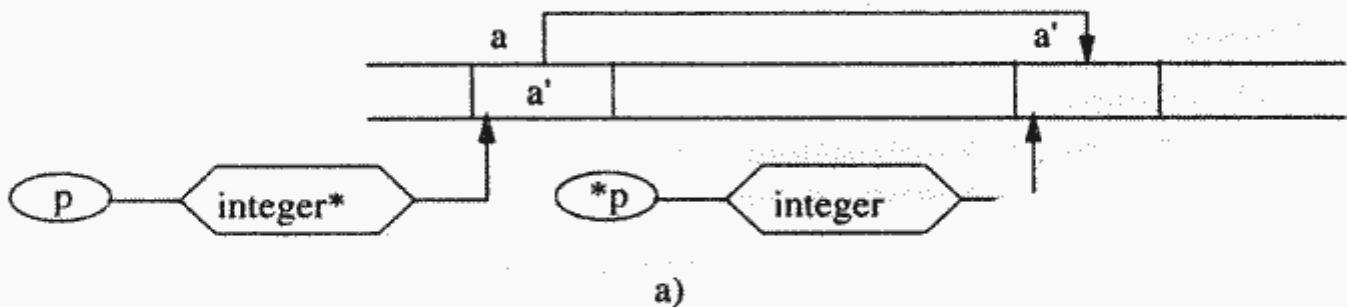


Figura 1.3: Pointer

1.1.4 Instrucțiuni

Atribuirea. Sintaxa:

$\langle \text{variabilă} \rangle \leftarrow \langle \text{expresie} \rangle$

unde $\langle \text{variabilă} \rangle$ este numele unei variabile, iar $\langle \text{expresie} \rangle$ este o expresie corect formată, de același tip cu $\langle \text{variabilă} \rangle$.

Semantica: Se evaluatează *(expresie)* și rezultatul obținut se memorează în locația de memorie desemnată de *(variabilă)*. Valorile tuturor celorlalte variabile rămân neschimbate. Atribuirea este singura instrucțiune cu ajutorul căreia se poate modifica memoria. O reprezentare intuitivă a efectului instrucțiunii de atribuire este dată în figura 1.4.

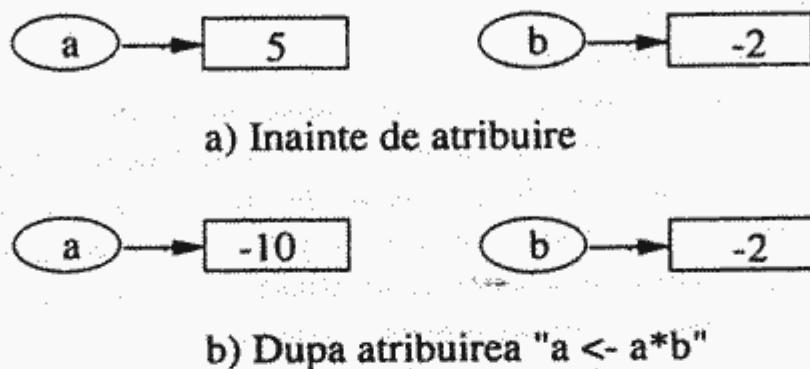


Figura 1.4: Atribuirea

if. Sintaxa:

```
if <expresie>
    then <secvență-instrucțiuni1>
    else <secvență-instrucțiuni2>
```

unde *(expresie)* este o expresie care prin evaluare dă rezultat boolean, iar *(secvență-instrucțiuni_i)*, $i = 1, 2$, sunt secvențe de instrucțiuni scrise una sub alta și aliniate corespunzător. Partea *else* este facultativă. Dacă partea *else* lipsește și *(secvență-instrucțiuni₁)* este formată dintr-o singură instrucțiune, atunci instrucțiunea *if* poate fi scrisă și pe un singur rând.

Semantica: Se evaluatează *(expresie)*. Dacă rezultatul evaluării este *true*, atunci se execută *(secvență-instrucțiuni₁)*, după care execuția instrucțiunii *if* se termină; dacă rezultatul evaluării este *false*, atunci se execută *(secvență-instrucțiuni₂)* după care execuția instrucțiunii *if* se termină.

Observație. O instrucțiune *if* de forma:

```
if (e1)
    then i1
    else if (e2)
        then i2
        else if (e3)
            then i3
            else if (e4)
                ...
                else in
```

va fi scrisă mai simplu astfel:

```

switch
  case ( $e_1$ )
     $i_1$ 
  case ( $e_2$ )
     $i_2$ 
  case ( $e_3$ )
     $i_3$ 
  case ( $e_4$ )
    ...
  otherwise
     $i_n$ 

```

sfobs

while. *Sintaxa:*

```

while <expresie> do
  <secvență-instrucțiuni>

```

unde <expresie> este o expresie care prin evaluare dă rezultat boolean, iar <secvență-instrucțiuni> este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.

Semantica: 1. Se evaluează <expresie>.

2. Dacă rezultatul evaluării este **true**, atunci se execută <secvență-instrucțiuni>, după care se reia procesul începând cu pasul 1. Dacă rezultatul evaluării este **false**, atunci execuția instrucțiunii **while** se termină.

for. *Sintaxa:*

```

for <variabilă>  $\leftarrow$  <expresie1> to <expresie2> do
  <secvență-instrucțiuni>

```

sau

```

for <variabilă>  $\leftarrow$  <expresie1> downto <expresie2> do
  <secvență-instrucțiuni>

```

unde <variabilă> este o variabilă de tip întreg, <expresie_i>, $i = 1, 2$, sunt expresii care prin evaluare dau valori întregi, <secvență-instrucțiuni> este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.

Semantica: Instrucțiunea

```

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

```

simulează execuția următorului program:

```

i  $\leftarrow$   $e_1$ 
temp  $\leftarrow$   $e_2$ 
while ( $i \leq \text{temp}$ ) do
  S
  i  $\leftarrow \text{succ}(i)$ 
  /* succ(i) întoarce successorul numărului întreg i */

```

iar instrucțiunea:

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

simulează execuția următorului program:

```
i  $\leftarrow e_1$ 
temp  $\leftarrow e_2$ 
while (i  $\geq$  temp) do
    S
    i  $\leftarrow \text{pred}(i)$ 
    /* pred(i) intoarce predecesorul numărului întreg i */
```

O formă mai puțin convențională a instrucțiunii **for**, dar frecvent utilizată în programe este **for each**:

for each. *Sintaxa:*

```
for each <variabilă>  $\in$  <mulțime> do
    <secvență-instrucțiuni>
```

unde <variabilă> este o variabilă de tip T , <mulțime> este o mulțime finită de valori din T , <secvență-instrucțiuni> este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.

Semantica: Instrucțiunea

```
for each x  $\in T$  do
    S
```

simulează execuția următorului program:

```
while ( $T \neq \emptyset$ ) do
    x  $\leftarrow$  un element din  $T$ 
    S
     $T \leftarrow T \setminus \{x\}$ 
```

repeat. *Sintaxa:*

```
repeat
    <secvență-instrucțiuni>
until <expresie>
```

unde <expresie> este o expresie care prin evaluare dă rezultat boolean, iar <secvență instrucțiuni> este o secvență de instrucțiuni scrise una sub alta și aliniate corespunzător.

Semantica: Instrucțiunea

```
repeat
    S
until e
```

simulează execuția următorului program:

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

Excepții. Sintaxa:

```
throw <mesaj>
```

unde <mesaj> este un sir de caractere (un text).

Semantica: Execuția programului se oprește și este afișat textul <mesaj>. Cel mai adesea, **throw** este utilizată împreună cu **if**:

```
if <expresie> then throw <mesaj>
```

Obținerea rezultatului **true** în urma evaluării expresiei are ca semnificație apariția unei excepții, caz în care execuția programului se oprește. Un exemplu de excepție este cel când procedura **new** nu poate aloca memorie pentru variabilele dinamice:

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

1.1.5 Subprograme

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

Proceduri. Interfața dintre o procedură și modulul care o apelează este realizată numai prin parametri și variabilele globale. De aceea, apelul unei proceduri apare ca o instrucțiune separată. Forma generală a unei proceduri este:

```
procedure <nume>(<lista-parametri>)
    <secvență-instrucțiuni>
end
```

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

```
procedure swap(x, y)
    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)
```

Funcții. În plus față de proceduri, o funcție întoarce o valoare asociată cu numele funcției. De aceea, apelul unei funcții poate participa la formarea de expresii. Forma generală a unei funcții este:

```
function <nume>(<lista-parametri>)
    <secvență-instrucțiuni>
    return <expresie>
end
```

Lista parametrilor este optională. Valoarea întoarsă de funcție este cea obținută prin evaluarea expresiei. O instrucțiune `return` poate apărea în mai multe locuri în definiția unei funcții și execuția ei implică terminarea evaluării funcției. Considerăm ca exemplu o funcție care calculează maximul dintre valorile a trei variabile:

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

Are sens să scriem $2 * \text{max3}(a, b, c)$ sau $\text{max3}(a, b, c) < 5$.

1.1.5.1 Comentarii

Comentariile sunt notate în mod similar cu limbajul C, utilizând combinațiile de caractere `/*` și `*/`. Comentariile au rolul de a introduce explicații suplimentare privind descrierea algoritmului:

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

1.1.6 Tipuri de date structurate de nivel jos

1.1.6.1 Tablouri

Un *tablou* este un ansamblu omogen de variabile, numite *componentele* tabloului, în care toate variabilele componente aparțin aceluiași tip și sunt identificate cu ajutorul indicilor. Un *tablou 1-dimensional (unidimensional)* este un tablou în care componente sunt identificate cu ajutorul unui singur indice. De exemplu, dacă numele variabilei tablou este *a* și mulțimea valorilor pentru indice este $\{0, 1, 2, \dots, n - 1\}$, atunci variabilele componente sunt $a[0], a[1], a[2], \dots, a[n-1]$. Memoria alocată unui tablou 1-dimensional este o secvență contiguă de locații, câte o locație pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea indicilor. Tablourile 1-dimensionale sunt reprezentate grafic ca în figura 1.5. Operațiile asupra tablourilor se realizează prin intermediul componentelor. Prezentăm ca exemplu inițializarea tuturor componentelor cu 0:

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

Un *tablou 2-dimensional (bidimensional)* este un tablou în care componente sunt identificate cu ajutorul a doi indici. De exemplu, dacă numele variabilei tablou este *a* și mulțimea valorilor pentru primul indice este $\{0, 1, \dots, m - 1\}$, iar mulțimea valorilor pentru cel de-al doilea indice este $\{0, 1, \dots, n - 1\}$ atunci variabilele componente sunt $a[0,0], a[0,1], \dots, a[0,n-1], \dots, a[m-1,0], a[m-1,1], \dots, a[m-1,n-1]$. Ca și în cazul tablourilor 1-dimensionale, memoria alocată unui tablou

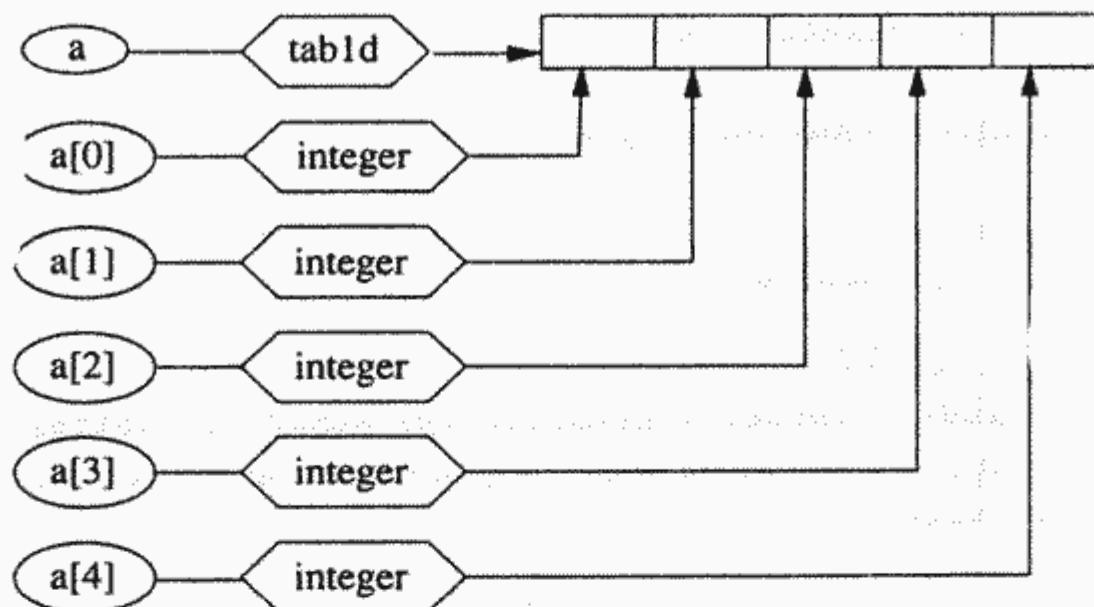


Figura 1.5: Tablou 1-dimenzional

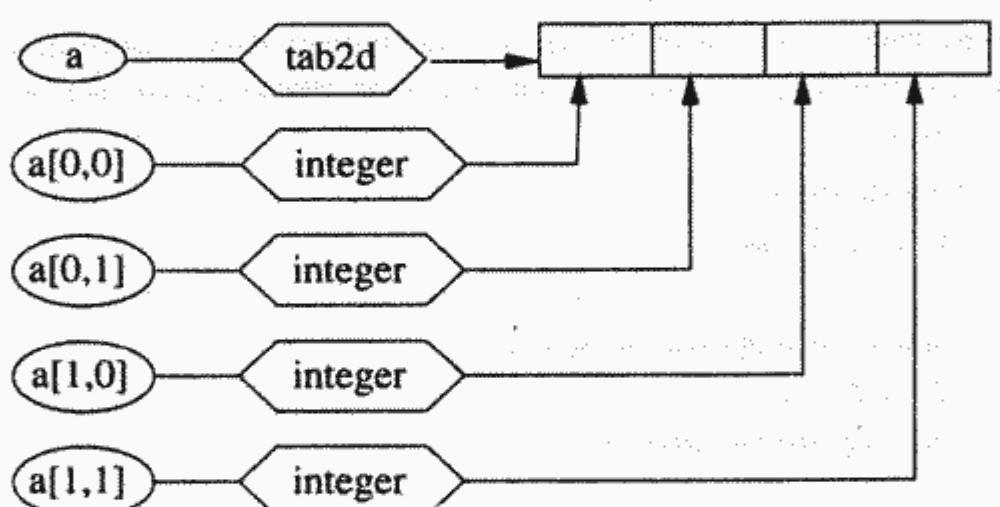


Figura 1.6: Tablou 2-dimenzional

2-dimenzional este o secvență contiguă de locații, câte o locație pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea lexicografică definită peste indici ($[i, j] < [k, l]$ dacă $i < k$ sau $i = k$ și $j < l$). Tablourile 2-dimensionale sunt reprezentate grafic ca în figura 1.6. Așa cum o matrice poate fi văzută ca un vector de linii, la fel un tablou 2-dimenzional poate fi văzut ca un tablou 1-dimenzional de tablouri 1-dimensionale. Din acest motiv, componentele unui tablou 2-dimenzional mai pot fi notate prin expresii de forma $a[0][0]$, $a[0][1]$ etc. (a se vedea și figura 1.7).

1.1.6.2 Siruri de caractere

Sirurile de caractere sunt secvențe de caractere. Pot fi asociate cu tablourile unidimensionale ale căror elemente sunt caractere. O constantă sir de caractere este notată utilizând convenția din limbajul C: 'exemplu de sir'. Peste siruri sunt definite următoarele operații:

- concatenarea, notată cu `+`: 'unu'+'doi' are ca rezultat 'unudois'.

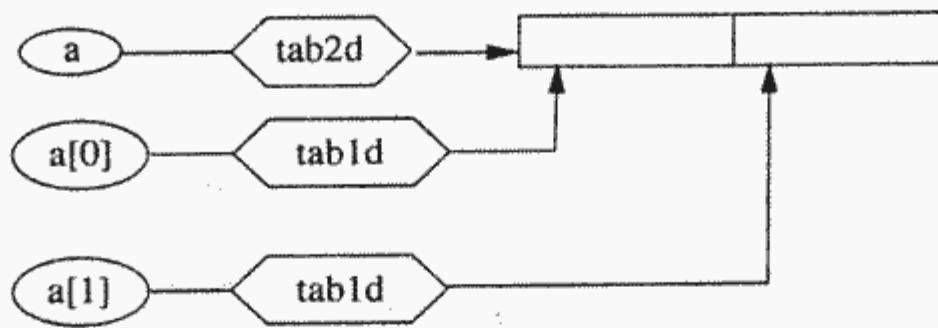


Figura 1.7: Tablou 2-dimENSIONAL văzut ca tablou de tablouri 1-dimensionale

- `strcmp(sir1, sir2)` întoarce rezultatul comparării lexicografice a celor două şiruri: -1 dacă `sir1 < sir2`, 0 dacă `sir1 = sir2`, și +1 dacă `sir1 > sir2`;
- `strlen(sir)` întoarce lungimea şirului dat ca parametru;
- `strcpy(sir1, sir2)` realizează copierea şirului `sir2` în `sir1`

1.1.6.3 Structuri statice

O *structură statică* este un ansamblu eterogen de variabile, numite *câmpuri*, în care fiecare câmp are propriul său nume și propriul său tip. Numele complet al unui câmp se obține din numele structurii urmat de caracterul '.' și numele câmpului. Memoria alocată unei structuri statice este o secvență contiguă de locații, căte o locație pentru fiecare câmp. Componența unei structuri statice nu se schimbă în timp, adică nu pot fi adăugate sau șterse câmpuri. 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 poziția figurii, și `f.vizibil` – valoare booleană care precizează dacă figura este desenată sau nu. La rândul său, punctul poate fi văzut ca o structură statică cu două câmpuri – căte unul pentru fiecare coordonată (considerăm numai puncte în plan având coordonate întregi). Structurile statice sunt reprezentate grafic ca în figura 1.8. Pentru identificarea câmpurilor unei structuri statice referite indirect prin intermediul unui pointer, vom utiliza o notație similară celei din limbajul C (a se vedea și figura 1.9).

1.1.6.4 Structuri dinamice înlănuite

O *structură dinamică înlănuită* este o înlănuire de structuri statice, numite *noduri*, în care pot fi inserate sau eliminate noduri, cu condiția să fie păstrată coerenta înlănuirii.

O *structură dinamică simplu înlănuită* este o înlănuire de structuri statice (noduri), în care fiecare nod „cunoaște” adresa nodului de după el (nodul succesor). Poate face excepție de la această regulă ultimul nod. În forma sa cea mai simplă, un nod `v` este o structură statică cu două câmpuri: un câmp `v->elt` pentru memorarea informației și un câmp `v->succ` care memorează adresa nodului succesor. Se presupune că se cunosc adresele primului și respectiv ultimului nod din structură. O structură simplu înlănuită este reprezentată grafic ca în figura 1.10. Structura

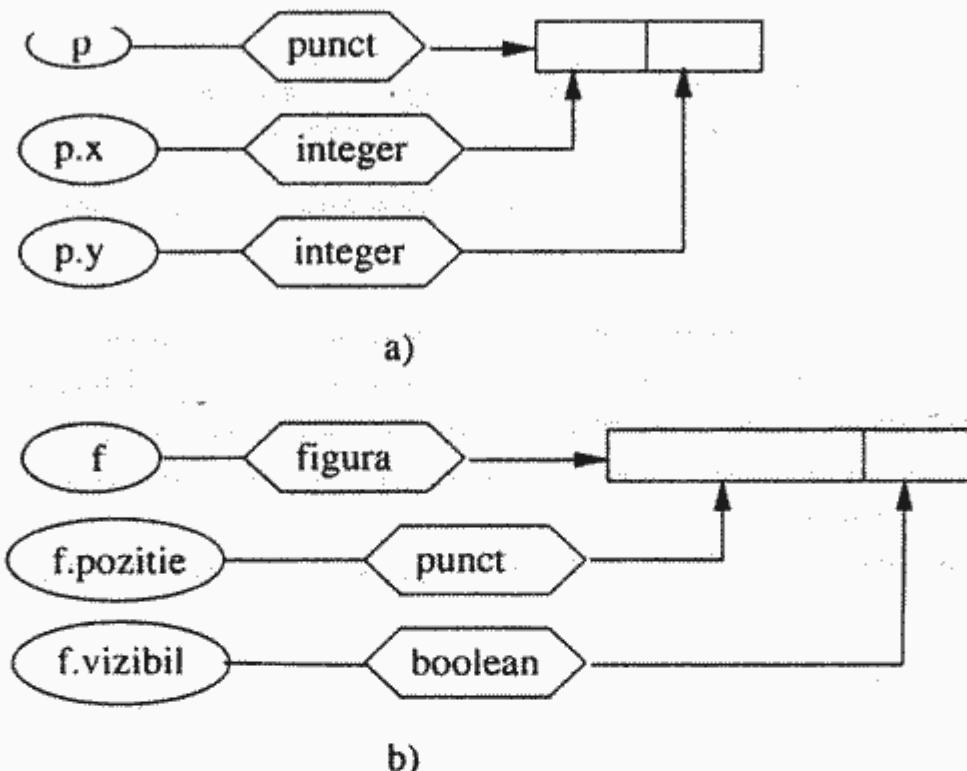


Figura 1.8: Structuri statice.

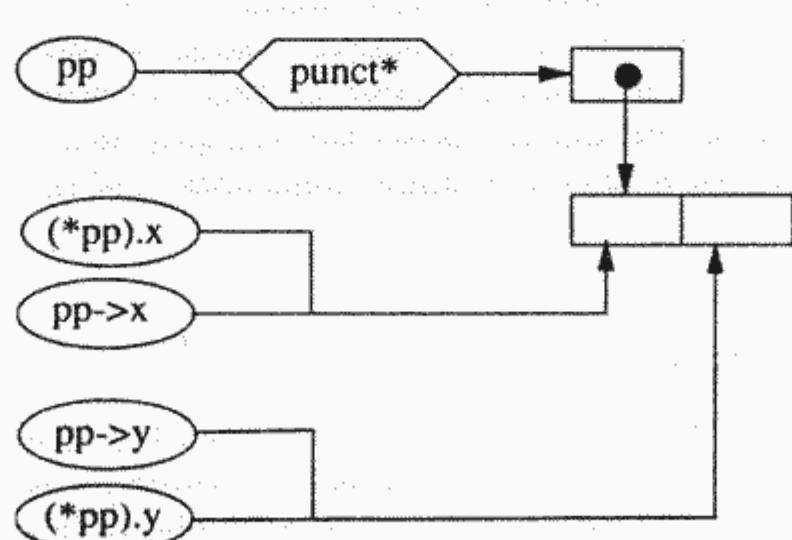


Figura 1.9: Structuri și pointeri

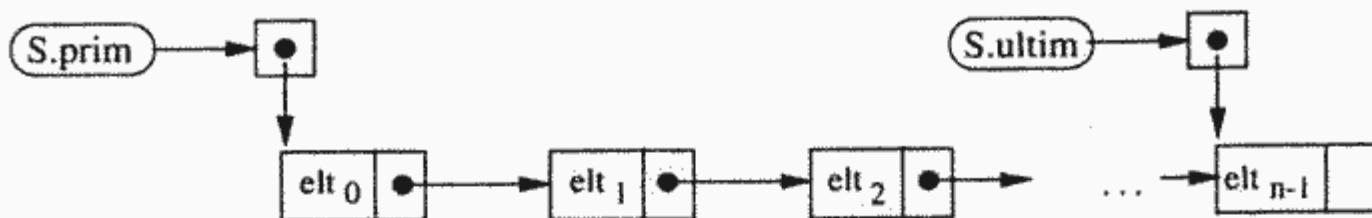


Figura 1.10: Structură dinamică simplu înlățuită

simplu înlățuită este o structură de date dinamică, în sensul că pot fi inserate sau eliminate noduri, cu condiția să fie păstrată proprietatea de înlățuire liniară.

Operațiile de bază ce se pot efectua asupra unei structuri simplu înlățuite sunt adăugarea sau ștergerea unui nod la începutul, în interiorul sau la sfârșitul structurii, parcurgerea structurii.

Prezentăm ca exemplu adăugarea unui nod la începutul structurii (a se vedea figura 1.11):

```

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

```

Structurile dinamice dublu înlățuite sunt asemănătoare celor simplu înlățuite cu deosebirea că, în plus, fiecare nod „cunoaște” adresa nodului din fața sa (nodul predecesor). Astfel, un nod *v* are un câmp în plus, *v->pred*, care memorează adresa nodului predecesor. Primul nod poate face excepție de la regula „cunoașterii” predecesorului. O structură dinamică dublu înlățuită este reprezentată grafic ca în figura 1.12.

1.1.7 Execuția programelor

Intuitiv, o execuție a unui program constă în succesiunea de pași elementari determinați de execuțiile instrucțiunilor ce compun programul. Convenim să utilizăm noțiunea de *calcul* pentru execuția unui program. 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 reprezentat de următorul tabel:

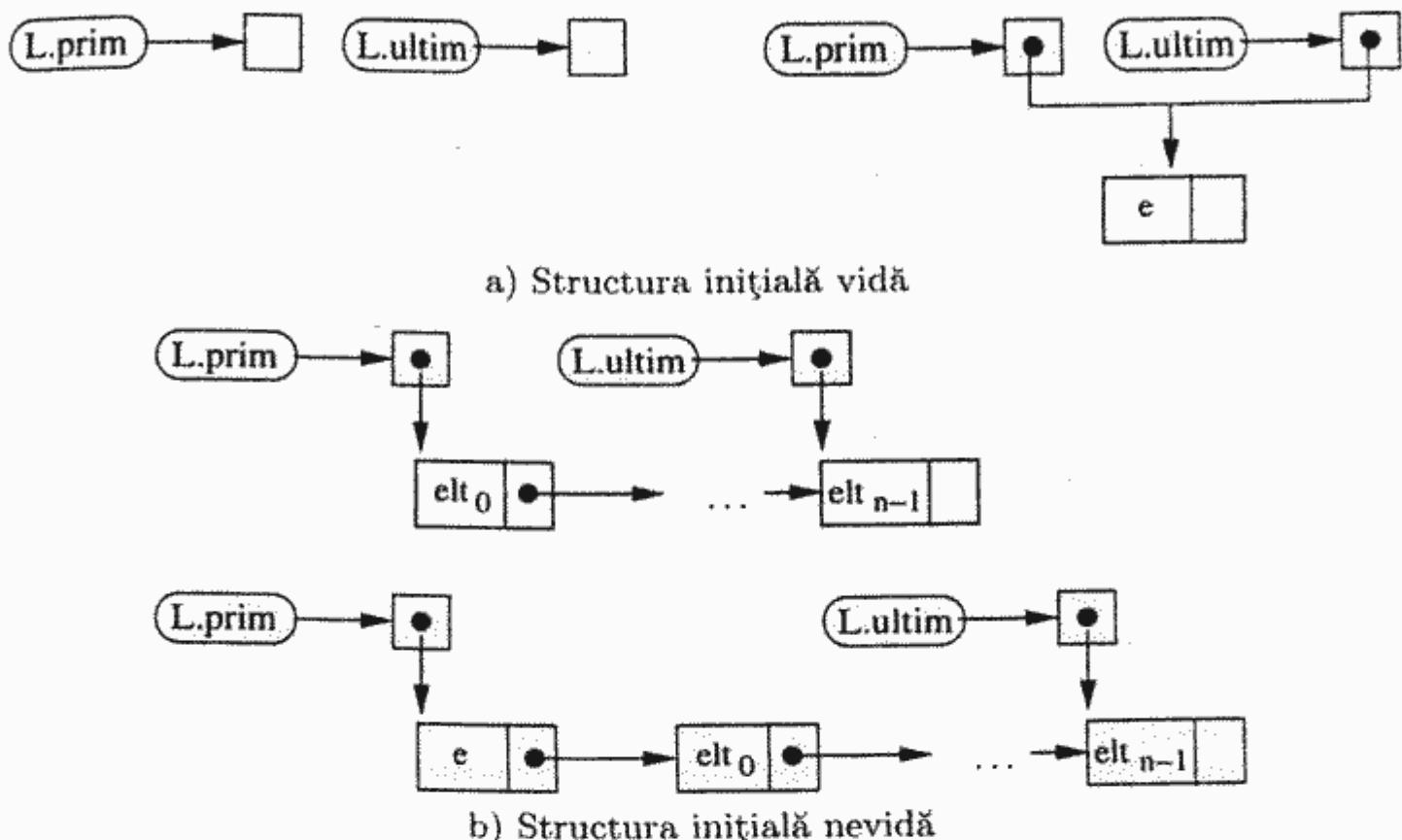


Figura 1.11:

Adăugarea unui nod la începutul unei structuri dinamice simplu înlăntuite

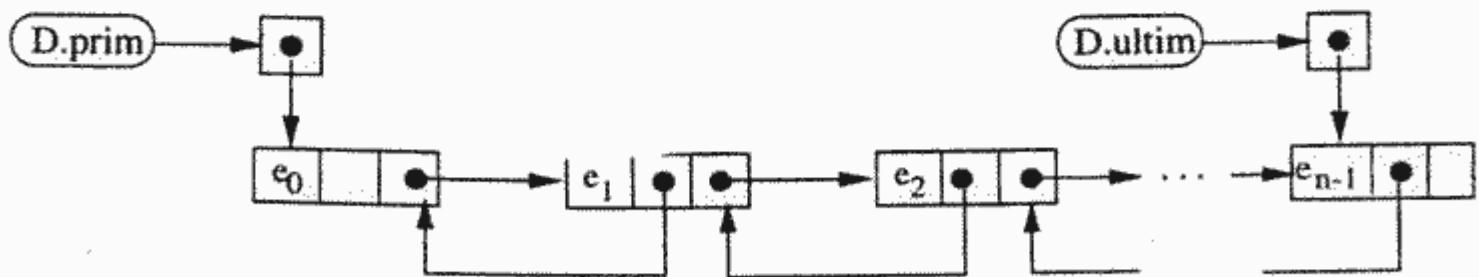


Figura 1.12: Structură dinamică dublu înlăntuită

Pasul	Instrucțiunea	i	x
0	$x \leftarrow 0$	-	-
1	$i \leftarrow 1$	-	0
2	$1 < 10$	1	0
3	$x \leftarrow x * 10 + i$	1	0
4	$i \leftarrow i + 2$	1	1
5	$3 < 10$	3	1
6	$x \leftarrow x * 10 + i$	3	1
7	$i \leftarrow i + 2$	3	13
8	$5 < 10$	5	13
9	$x \leftarrow x * 10 + i$	5	13
10	$i \leftarrow i + 2$	5	135
11	$7 < 10$	7	135
12	$x \leftarrow x * 10 + i$	7	135
13	$i \leftarrow i + 2$	7	1357
14	$9 < 10$	9	1357
15	$x \leftarrow x * 10 + i$	9	1357
16	$i \leftarrow i + 2$	9	13579
17	$11 < 10$	11	13579
18		11	13579

Acest calcul este notat formal printr-o secvență $c_0 \vdash c_1 \vdash \dots \vdash c_{18}$. Prin c_i am notat configurațiile ce intervin în calcul. O configurație include instrucțiunea curentă (starea programului) și starea memoriei (valorile curente ale variabilelor din program). În exemplul de mai sus, o configurație este reprezentată de o linie în tabel. Relația $c_{i-1} \vdash c_i$ are următoarea semnificație: prin execuția instrucțiunii din c_{i-1} , se transformă c_{i-1} în c_i . c_0 se numește configurație inițială, iar c_{18} configurație finală. Notăm că pot exista și calcule infinite. De exemplu, instrucțiunea:

```
while (true) do
    i ← i+1
```

generează un calcul infinit.

1.1.8 Exerciții

Exercițiul 1.1.1. O secțiune a tabloului **a**, notată **a[i..j]**, este formată din elementele $a[i], a[i+1], \dots, a[j]$, $i \leq j$. Suma unei secțiuni este suma elementelor sale. Să se scrie un program care determină secțiunea de sumă maximă.

Exercițiul 1.1.2. Să se scrie o funcție care, pentru un tablou **b**, determină valoarea predicatorului:

$$P \equiv (\forall i, j) 10 \leq i < j \leq n - 1 \Rightarrow b[i] \leq b[j]$$

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

Exercițiul 1.1.3. Să se scrie un subprogram de tip funcție care, pentru un tablou de valori booleene **b**, determină valoarea predicatorului:

$$P \equiv (\forall i, j) 0 \leq i \leq j \leq n - 1 \Rightarrow (b[i] \Rightarrow b[j]))$$

Exercițiul 1.1.4. Se consideră tablou a oraonat crescător și tabloul b ordonat descrescător. Să se scrie un program care determină cel mai mic x , când există, ce apare în ambele tablouri.

Exercițiul 1.1.5. Să se scrie un program care să pună elementele unei matrice pătratice într-o listă simplu înlanțuită, în ordinea parcurgerii acesteia în spirală,

începând cu colțul stânga-jos. De exemplu, pentru matricea $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ ordinea parcurgerii este $a_{31}, a_{32}, a_{33}, a_{23}, a_{13}, a_{12}, a_{11}, a_{21}, a_{22}$. Matricea va fi reprezentată ca tablou 2-dimensional.

Exercițiul 1.1.6. Să se proiecteze o structură statică pentru reprezentarea numerelor complexe. Să se scrie subprograme care realizează operații din algebra numerelor complexe.

Exercițiul 1.1.7. Presupunem că numerele complexe sunt reprezentate ca în soluția exercițiului 1.1.6. Un polinom cu coeficienți complecsi poate fi reprezentat ca un tablou de structuri statice. Să se scrie un subprogram care, pentru un polinom cu coeficienți complecsi P și un număr complex z date, calculează $P(z)$.

Exercițiul 1.1.8. Să se proiecteze o structură statică pentru reprezentarea datei calendaristice. Să se scrie subprograme care realizează următoarele:

- Decide dacă valoarea unei variabile conține o dată validă.
- Având la intrare o dată calendaristică oferă la ieșire data calendaristică următoare.
- Având la intrare o dată calendaristică oferă la ieșire data calendaristică precedentă.

Exercițiul 1.1.9. Dat fiind un sir de numere întregi, să se scrie subprograme pentru următoarele operații:

- Plasarea elementelor sirului într-o structură liniară simplu înlanțuită, prin inserări repetitive în fața structurii.
- Parcurgerea structurii create.

Exercițiul 1.1.10. Submulțimile finite de numere naturale $M \subset Q$ pot fi reprezentate prin tablouri. Să se scrie subprograme pentru operațiile $\in, \subset, \cup, \cap, \setminus$.

Exercițiul 1.1.11. Să se definească o implementare cu structuri dinamice simplu înlanțuite, pentru mulțimile M din exercițiul anterior și să se scrie subprograme pentru operațiile specifice mulțimilor.

1.2 Probleme, algoritmi și programe

Noțiunile de algoritmi, programe și probleme sunt diferite și trebuie înțelese bine care sunt relațiile dintre ele. Un algoritm exprimă o modalitate de a obține soluția unei probleme specifice. Descrierea algoritmului într-un limbaj algoritic se face prin intermediul unui program. Toate noțiunile definite pentru program sunt extinse în mod natural la algoritm. Astfel, putem vorbi despre execuția (calculul) unui algoritm, configurație inițială, etc. În această secțiune vom preciza ce înseamnă că un algoritm exprimă o modalitate de a obține soluția pentru o problemă.

1.2.1 Noțiunea de problemă

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

Intrare/Ieșire. Putem formaliza problema rezolvată de algoritm ca o pereche (*Intrare, Ieșire*). Componenta *Intrare* descrie datele de intrare, iar componenta *Ieșire* descrie datele de ieșire. Un exemplu simplu de problemă reprezentată astfel este următorul:

Intrare: Un număr întreg pozitiv x .

Ieșire: Cel mai mare număr prim mai mic decât sau egal cu x .

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

Instanță: Un număr întreg pozitiv x .

Întrebare: Este x număr prim?

Problemele de decizie sunt preferate atât în teoria complexității, cât și în teoria calculabilității datorită reprezentării foarte simple a ieșirilor. Facem observația că orice problemă admite o reprezentare sub formă de problemă de decizie, indiferent de reprezentarea sa inițială. Un exemplu de reprezentare a unei probleme de optim ca problemă de decizie este dat în secțiunea 13.1.

De obicei, pentru reprezentarea problemelor de decizie se consideră o mulțime \mathcal{U} , iar o instanță este de forma $\mathcal{R} \subseteq \mathcal{U}$, $x \in \mathcal{U}$ și întrebarea de forma $x \in \mathcal{R}?$. Problema din exemplul anterior poate fi reprezentată astfel:

Instanță: \mathcal{P} = mulțimea numerelor prime ($\mathcal{P} \subset \mathbb{Z}$), $x \in \mathbb{Z}^+$.

Întrebare: $x \in \mathcal{P}?$

1.2.2 Problemă rezolvată de un algoritm

Convenim să considerăm intotdeauna intrările p ale unei probleme P ca fiind instanțe și, prin abuz de notație, scriem $p \in P$.

Definiția 1.1. Fie A un algoritm și P o problemă. Spunem că o configurație inițială c_0 a lui A include instanța $p \in P$ dacă există o structură de date inp , definită în A , astfel încât valoarea lui inp din c_0 constituie o reprezentare a lui p . Analog, spunem că o configurație finală c_n a unui algoritm A include ieșirea $P(p)$ dacă există o structură de date out , definită în A , astfel încât valoarea lui out din c_n constituie o reprezentare a lui $P(p)$.

Definiția 1.2. (*Problemă rezolvată de un algoritm*)

1. Un algoritm A rezolvă o problemă P în sensul corectitudinii totale dacă pentru orice instanță p , calculul unic determinat de configurația inițială ce include p este finit și configurația finală include ieșirea $P(p)$.
2. Un algoritm A rezolvă o problemă P în sensul corectitudinii parțiale dacă pentru orice instanță p pentru care calculul unic determinat de configurația inițială ce include p este finit, configurația finală include ieșirea $P(p)$.

Ori de câte ori spunem că un algoritm A rezolvă o problemă P vom înțelege de fapt că A rezolvă o problemă P în sensul corectitudinii totale.

Definiția 1.3. (*Problemă rezolvabilă/nerezolvabilă*)

1. O problemă P este *rezolvabilă* dacă există un algoritm 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 *parțial decidabilă* dacă există un algoritm A care rezolvă P în sensul corectitudinii parțiale astfel încât calculul lui A este finit pentru orice instanță p pentru care răspunsul la întrebare este 'DA'.
3. O problemă P este *nerezolvabilă* dacă nu este rezolvabilă, adică nu există un algoritm care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este *nedecidabilă*.

1.2.3 Un exemplu de problemă nedecidabilă

Pentru a arăta că o problemă este decidabilă este suficient să găsim un algoritm 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 nedecidabilă este cea cunoscută sub numele de *problema opririi*. Notăm cu \mathcal{U} mulțimea perechilor de forma $\langle A, \bar{x} \rangle$ unde A este un algoritm și \bar{x} este o intrare pentru A , iar \mathcal{R} este submulțimea formată din acele perechi $\langle A, \bar{x} \rangle$ pentru care calculul lui A pentru intrarea \bar{x} este finit. Dacă notăm prin $A(\bar{x})$ (a se citi $A(\bar{x}) = \text{true}$) faptul că $\langle A, \bar{x} \rangle \in \mathcal{R}$, atunci problema opririi poate fi scrisă astfel:

Problema opririi

Instanță: Un algoritm A , $\bar{x} \in \mathbb{Z}^*$.

Întrebare: $A(\bar{x})?$

Teorema 1.1. *Problema opririi nu este decidabilă.*

Demonstrație. Un algoritm AO , care rezolvă problema opririi, are ca intrare o pereche $\langle A, \bar{x} \rangle$ și se oprește întotdeauna cu răspunsul 'DA', dacă A se oprește pentru intrarea \bar{x} , sau cu răspunsul 'NU', dacă A nu se oprește pentru intrarea \bar{x} . Fie T următorul algoritm:

```
if (AO( $\bar{x}, \bar{x}$ ) = 'DA')
    then while (true) do /*nimic*/
```

Algoritmul T nu se oprește (bucătă while se execută la infinit), dacă $AO(\bar{x}, \bar{x})$ înțoarce 'DA'. Reamintim că $AO(\bar{x}, \bar{x}) = 'DA'$ înseamnă că algoritmul reprezentat de \bar{x} se oprește pentru intrarea \bar{x} , adică propria sa codificare. Presupunem acum că \bar{x} este codificarea lui T . Există două posibilități.

1. T se oprește pentru intrarea \bar{x} . Rezultă $AO(\bar{x}, \bar{x}) = 'NU'$, adică algoritmul reprezentat de \bar{x} nu se oprește pentru intrarea \bar{x} . Dar algoritmul reprezentat de \bar{x} este chiar T . Contradicție.
2. T nu se oprește pentru intrarea \bar{x} . Rezultă $AO(\bar{x}, \bar{x}) = 'DA'$, adică algoritmul reprezentat de \bar{x} se oprește pentru intrarea \bar{x} . Contradicție.

Așadar, nu există un algoritm AO care să rezolve problema opririi.

sfdem

Observație. Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. „Există un oraș cu un bărbier care bărbierește pe oricine ce nu se bărbierește singur. Cine îl bărbierește pe bărbier?”

sfobs

1.3 Măsurarea performanțelor unui algoritm

Evaluarea algoritmilor din punctul de vedere al performanțelor obținute de aceștia în rezolvarea problemelor este o etapă esențială în procesul de decizie a utilizării acestora în aplicații. La evaluarea (estimarea) algoritmilor se pune în evidență consumul celor două resurse fundamentale: timpul de execuție și spațiul de memorare a datelor. În funcție de prioritățile alese, se aleg limite pentru resursele timp și spațiu. Algoritmul este considerat eligibil dacă consumul celor două resurse se încadrează în limitele stabilite.

Fie P o problemă și A un algoritm pentru P . Fie $c_0 \vdash_A c_1 \vdash_A \dots \vdash_A c_n$ un calcul finit al algoritmului A . Notăm cu $t_A(c_i)$ timpul necesar obținerii configurației c_i din c_{i-1} , $1 \leq i \leq n$, și cu $s_A(c_i)$ spațiul de memorie ocupat în configurația c_i , $0 \leq i \leq n$.

Definiția 1.4. Fie A un algoritm pentru problema P , $p \in P$ o instanță a problemei P și $c_0 \vdash c_1 \vdash \dots \vdash c_n$ calculul lui A corespunzător instanței p . Timpul necesar algoritmului A pentru rezolvarea instanței p este:

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

Spațiul (de memorie) necesar algoritmului A pentru rezolvarea instanței p este:

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

În general sunt dificil de calculat cele două valori în funcție de instanțe. Această dificultate poate fi simplificată astfel. Asociem unei instanțe $p \in P$ o mărime $g(p)$, care este un număr natural, pe care o numim *mărimea* instanței p . De exemplu, $g(p)$ poate fi suma lungimilor reprezentărilor corespunzând datelor din instanța p . Dacă reprezentările datelor din p au aceeași lungime, atunci se poate considera $g(p)$

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 considera $g(p)$ ca fiind gradul polinomului ($=$ numărul coeficienților minus 1); dacă p este un graf se poate lua $g(p)$ ca fiind numărul de vârfuri, numărul de muchii sau numărul de vârfuri + numărul de muchii etc.

Definiția 1.5. 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 spațiul $S_A^{fav}(n)$ dacă:

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

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

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

Funcția T_A^{fav} (S_A^{fav}) se numește *timpul de execuție al algoritmului (spațiul utilizat de algoritmul) A pentru cazul cel mai favorabil*, iar funcția T_A (S_A) se numește *timpul de execuție al algoritmului (spațiul utilizat de algoritmul) A pentru cazul cel mai nefavorabil*.

Exemplu.

Considerăm problema căutării unui element într-o secvență de numere întregi.

Problema P_1

Intrare: $n, (a_0, \dots, a_{n-1}), z$ numere întregi.

Ieșire: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Presupunem că secvența (a_0, \dots, a_{n-1}) este memorată în tabloul $(a[i] \mid 0 \leq i \leq n-1)$. Algoritmul A_1 descris de următorul program rezolvă P_1 :

```
/* algoritmul A1 */
i ← 0
while (a[i] ≠ z) and (i < n-1) do
    i ← i+1
if (a[i] = z)
    then poz ← i
    else poz ← -1
```

Considerăm ca dimensiune a problemei P_1 numărul n al elementelor din secvență în care se caută. Deoarece suntem în cazul când toate datele sunt memorate pe căte un cuvânt de memorie, vom presupune că toate operațiile necesită o unitate de timp. Calculăm timpii de execuție. Cazul cel mai favorabil este obținut când $a_0 = z$ și se

efectuează trei comparații și două atribuiri. Rezultă $T_{A_1}^{fav}(n) = 3 + 2 = 5$. Cazul cel mai nefavorabil se obține când $z \notin \{a_0, \dots, a_{n-1}\}$ sau $z = a[n - 1]$, în acest caz fiind executate $2n + 1$ comparații și $1 + (n - 1) + 1 = n + 1$ atribuiri. Rezultă $T_{A_1}(n) = 3n + 2$. Pentru simplitatea prezentării, nu au mai fost luate în considerare operațiile and și operațiile de adunare și scădere. Spațiul utilizat de algoritm, pentru ambele cazuri, este $n + 7$ (tabloul a, constantele 0, 1 și -1, variabilele i, poz, n și z).

sfex

Exemplu. Considerăm acum problema determinării elementului de valoare maximă dintr-o secvență de numere întregi:

Problema P_2

Intrare: $n, (a_0, \dots, a_{n-1})$ numere întregi.

Ieșire: $\max = \max\{a_i \mid 0 \leq i \leq n - 1\}$.

Și aici presupunem că secvența (a_0, \dots, a_{n-1}) este memorată în tabloul $(a[i] \mid 0 \leq i \leq n - 1)$. Algoritmul A_2 descris de următorul program rezolvă P_2 :

```
/* algoritmul A2 */
max ← a[0]
for i ← 1 to n-1 do
    if (a[i]>max)
        then max ← a[i]
```

Dimensiunea problemei P_2 este n , numărul elementelor din secvență în care se caută maximul. Vom presupune și în acest caz că toate operațiile necesită o unitate de timp. Cazul cel mai favorabil este obținut când a_0 este elementul de valoare maximă. Se efectuează $n + n - 1 = 2n - 1$ comparații și $1 + n$ atribuiri. Rezultă $T_{A_1}^{fav}(n) = 2n - 1 + 1 + n = 3n$. Cazul cel mai defavorabil se obține în situația în care tabloul este ordonat crescător (pentru că de fiecare dată instrucțiunea if se execută pe ramura then, adică se face și comparație și atribuire). În acest caz numărul comparațiilor este $2n - 1$ și cel al atribuirilor este $2n$. Rezultă $T_{A_2}(n) = 2n - 1 + 2n = 4n - 1$. Dimensiunea memoriei utilizate de algoritm este $n + 5$ (tabloul a, constanta 1, variabilele i, n și z), în ambele cazuri.

sfex

Exemplu. Sortarea prin inserare.

Problema P_3

Intrare: $n, (a_0, \dots, a_{n-1})$ numere întregi.

Ieșire: $(a_{i_0}, \dots, a_{i_{n-1}})$ unde (i_0, \dots, i_{n-1}) este o permutare a sirului $(0, \dots, n - 1)$ și $a_{i_j} \leq a_{i_{j+1}}, \forall j \in \{0, \dots, n - 2\}$.

Presupunem din nou că secvența (a_0, \dots, a_{n-1}) este memorată în tabloul $(a[i] \mid 0 \leq i \leq n - 1)$. Algoritmul sortării prin inserare consideră că în pasul k , elementele $a[0..k-1]$ sunt sortate crescător, iar elementul $a[k]$ va fi inserat, astfel încât, după această inserare, primele elemente $a[0..k]$ să fie sortate crescător. Inserarea elementului $a[k]$ în secvența $a[0..k-1]$ presupune:

1. memorarea elementului într-o variabilă temporară;
2. deplasarea tuturor elementelor din vectorul $a[0..k-1]$ care sunt mai mari decât $a[k]$, cu o poziție la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);

3. plasarea lui $a[k]$ în locul ultimului element deplasat.

Algoritmul A_3 care rezolvă P_3 este descris de următorul program:

```
/* algoritmul A3 */
for k ← 1 to n-1 do
    temp ← a[k]
    i ← k-1
    while (i ≥ 0 and a[i] > temp) do
        a[i+1] ← a[i]
        i ← i-1
    a[i+1] ← temp
```

Dimensiunea problemei P_3 este dată de numărul n al elementelor din secvența ce urmează a fi sortată. Presupunem și aici că toate operațiile necesită o unitate de timp. Cazul cel mai favorabil este obținut când elementele secvenței sunt sortate crescător. În această situație se efectuează $n + 2(n - 1)$ comparații și $n + 3(n - 1)$ atribuiri. Rezultă $T_{A_3}^{fav}(n) = 3n - 2 + 4n - 3 = 6n - 5$. Cazul cel mai defavorabil este dat de situația în care deplasarea (la dreapta cu o poziție în vederea inserării) se face de la începutul tabloului, adică sirul este ordonat descrescător. Estimarea timpului de execuție pentru cazul cel mai nefavorabil este următoarea: numărul de comparații este $n + 2(2 + 3 + \dots + n) = n + n(n + 1) - 2 = n^2 + 2n - 2$, iar cel al atribuirilor este $n + 3(n - 1) + 2(1 + 2 + \dots + n - 1) = 4n - 3 + (n - 1)n = n^2 + 3n - 3$. Adunând, avem $T_{A_3}(n) = (n^2 + 2n - 2) + (n^2 + 3n - 3) = 2n^2 + 5n - 5$. Spațiul este $n + 5$ (tabloul a , constantele 0 și 1, variabilele i , temp , n și z). sfex

Exemplu. Căutarea binară. Reconsiderăm problema căutării unui element într-un tablou, dar cu presupunerea suplimentară că tabloul este ordonat.

Problema P_4

Intrare: $n, (a_0, \dots, a_{n-1}), z$ numere întregi;
secvența (a_0, \dots, a_{n-1}) este sortată crescător,
adică $a_i \leq a_{i+1}, i \in \{0, \dots, n-2\}$.

Ieșire: $poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Esența căutării binare constă în compararea elementului căutat cu elementul din mijlocul zonei de căutare și în cazul în care elementul căutat nu este egal cu acesta, se restrânge căutarea la subzona din stânga sau din dreapta, în funcție de rezultatul comparării. Astfel, dacă elementul căutat este mai mic decât cel din mijlocul zonei de căutare, se alege subzona din stânga, altfel subzona din dreapta. Inițial, zona de căutare este tabloul a . Convenim să notăm cu i_{stg} indicele elementului din stânga zonei de căutare în tablou, i_{dr} indicele elementului din dreapta zonei de căutare în tablou. Indicele elementului din mijlocul zonei de căutare în tablou va fi notat i_{med} .

Algoritmul căutării binare (A_4) este descris de subprogramul următor:

```
/* algoritmul A4 */
function cautareBinara(a,n,z)
    istg ← 0
```

```

 $i_{dr} \leftarrow n-1$ 
while ( $i_{stg} \leq i_{dr}$ ) do
     $i_{med} \leftarrow \lfloor i_{stg} + i_{dr} \rfloor / 2$ 
    if ( $a[i_{med}] = z$ )
        then return  $i_{med}$ 
    else if ( $a[i_{med}] > z$ )
        then  $i_{dr} \leftarrow i_{med} - 1$  /* se cauta in stanga */
    else  $i_{stg} \leftarrow i_{med} + 1$  /* se cauta in dreapta */
    return -1
end

```

Dimensiunea problemei P_4 este dată de dimensiunea n a secvenței în care se face căutarea. Și de această dată presupunem că toate operațiile necesită o unitate de timp. Calculul timpului de execuție al algoritmului constă în determinarea numărului de execuții ale blocului de instrucțiuni asociat cu instrucțiunea while. Se observă că, după fiecare iterare a buclei while, dimensiunea zonei de căutare se înjumătășește. Cazul cel mai favorabil este obținut când $a[\frac{n-1}{2}] = z$ și se efectuează două comparații și trei atribuiri. Rezultă $T_{A_4}^{fav}(n) = 2 + 3 = 5$. Ca și la căutarea secvențială, cazul cel mai nefavorabil este în situația în care vectorul a nu conține valoarea căutată. Pentru simplitate, se consideră $n = 2^k$, unde k este numărul de înjumătățiri. Rezultă $k = \log_2 n$ și printr-o majorare, $T_{A_4}(n) \leq c \log_2 n + 1$, unde c este o constantă, $c \geq 1$. Spațiul necesar execuției algoritmului A_4 este $n+7$ (tabloul a, constantele 0 și -1, variabilele i_{stg} , i_{dr} , i_{med} , n și z). sfex

Observație. Timpii de execuție pentru cazul cel mai favorabil nu oferă informații relevante despre eficiența algoritmului. Mult mai semnificative sunt informațiile oferite de timpii de execuție în cazul cel mai nefavorabil: în toate celelalte cazuri algoritmul va avea performanțe mai bune sau cel puțin la fel de bune.

Pentru evaluarea timpului de execuție nu este necesar întotdeauna să numărăm toate operațiile. Pentru exemplul de mai sus, observăm că operațiile de atribuire (fără cea inițială) sunt precedate de comparații. Astfel, putem număra numai comparațiile, pentru că numărul acestora determină numărul atribuirilor. Putem să mergem chiar mai departe și să numărăm numai comparațiile între z și componentele tabloului. sfobs

Uneori, numărul instanțelor p cu $g(p) = n$ pentru care $T_A(p) = T_A(n)$ sau $T_A(p)$ are o valoare foarte apropiată de $T_A(n)$ este foarte mic. 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 experiență = execuția algoritmului pentru o instanță p , valoarea experienței = durata execuției algoritmului pentru instanța p) și să precizăm legea de repartiție a acestei variabile aleatoare. Apoi, *comportarea în medie* se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul timpului de execuție):

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

Dacă mulțimea valorilor variabilei aleatoare $T_A(p) = \{x_1, \dots\}$ este finită sau numărabilă ($T_A(p) = \{x_1, \dots, x_i, \dots\}$) și probabilitatea ca $T_A(p) = x_i$ este p_i , atunci

nedia variabilei aleatoare T_A (timpul mediu de execuție) este:

$$T_A^{med}(n) = \sum_i x_i \cdot p_i$$

Exemplu. Considerăm problema P_1 definită anterior. Multimea valorilor variabilei aleatoare $T_A(p)$ este $\{3i + 2 \mid 1 \leq i \leq n\}$. În continuare trebuie să stabilim legea de repartiție. Facem următoarele presupuneri: probabilitatea ca $z \in \{a_0, \dots, a_{n-1}\}$ este q și probabilitatea ca z să apară prima dată pe poziția $i - 1$ este $\frac{q}{n}$ (indicii i candidează cu aceeași probabilitate pentru prima apariție a lui z). Rezultă că probabilitatea ca $z \notin \{a_0, \dots, a_{n-1}\}$ este $1 - q$. Acum, probabilitatea ca $T_A(p) = 3i + 2$ ($poz = i - 1$) 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 $poz = n - 1$ sau ca $z \notin \{a_0, \dots, a_{n-1}\}$). Timpul mediu de execuție 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 secvență) avem $T_A^{med}(n) = \frac{3n}{2} + \frac{7}{2}$ și pentru $q = \frac{1}{2}$ avem $T_A^{med}(n) = \frac{9n}{4} + \frac{11}{4}$. sfex

1.3.1 Calcul asimptotic

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

$$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$$

$$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$$

$$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0)c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$$

Cu notațiile O , Ω și Θ se pot forma expresii și ecuații. 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. și notează mulțimile:

$$\{g(n) \mid (\exists g_1(n), g_2(n), c > 0, n_0 \geq 0)$$

$$[(\forall n)g(n) = g_1(n) \text{ op } g_2(n)] \wedge [(\forall n \geq n_0)g_1(n) \leq cf_1(n) \wedge g_2(n) \leq cf_2(n)]\}$$

De exemplu:

$$O(n) + O(n^2) = \{g(n) = g_1(n) + g_2(n) \mid (\forall n \geq n_0) g_1(n) \leq cn \wedge g_2(n) \leq cn^2\}$$

Utilizând regulile de asociere și prioritate, se obțin expresii de orice lungime:

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

Orice funcție $f(n)$ poate fi gândită ca o notație pentru mulțimea cu un singur element $f(n)$ și deci putem alcătui expresii de forma:

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

ca desemnând mulțimea:

$$\begin{aligned} & \{f_1(n) + g(n) \mid g(n) \in O(f_2(n))\} = \\ & \{f_1(n) + g(n) \mid (\exists c > 0, n_0 > 1)(\forall n \geq n_0) g(n) \leq c \cdot f_2(n)\} \end{aligned}$$

Peste expresii considerăm formule de forma:

$$\text{expr1} = \text{expr2}$$

cu semnificația că mulțimea desemnată de expr1 este inclusă în mulțimea desemnată de expr2 . De exemplu, avem:

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

pentru că $(\exists c_1 > 0, n_1 > 1)(\forall n \geq n_1) n \log n \leq c_1 n^2$, $g_1(n) \in O(n^2)$ implică $(\exists c_2 > 0, n_2 > 1)(\forall n \geq n_2) g_1(n) \leq c_2 n^2$ și 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 ecuațiilor: părțile stânga și cea din dreapta joacă roluri distinctive. Ca un caz particular, notația $g(n) = O(f(n))$ semnifică, de fapt, $g(n) \in O(f(n))$.

1.3.2 Calculul timpului de execuție pentru cazul cel mai nefavorabil

Un algoritm poate avea o descriere complexă și deci evaluarea sa poate pune unele probleme. De aceea prezentăm câteva strategii ce sunt aplicate în determinarea timpului de execuție pentru cazul cel mai nefavorabil. Deoarece orice algoritm este descris de un program, în continuare considerăm A o secvență de program. Regulile prin care se calculează timpul de execuție sunt date în funcție de structura lui A :

1. A este o instrucțiune de atribuire. Timpul de execuție a lui A este egal cu timpul evaluării expresiei din partea dreaptă.
2. A este forma:

A_1

A_2

Timpul de execuție al lui A este egal cu suma timpilor de execuție ai algoritmilor A_1 și A_2 .

3. A este de forma `if e then A_1 else A_2` . Timpul de execuție al lui A este egal cu maximul dintre timpii de execuție ai algoritmilor A_1 și A_2 la care se adună timpul necesar evaluării expresiei e .
4. A este de forma `while e do A_1` . Se determină cazul când se execută numărul maxim de iterații ale buclei `while` și se face suma timpilor calculați pentru fiecare iterație. Dacă nu este posibilă determinarea timpilor pentru fiecare iterație, atunci timpul de execuție al lui A este egal cu produsul dintre timpul maxim de execuție al algoritmului A_1 și numărul maxim de execuții ale buclei A_1 .

Plecând de la aceste reguli de bază, se pot obține în mod natural reguli de calcul a timpului de execuție pentru toate instrucțiunile.

Teorema 1.2. *Dacă g este o funcție polinomială de grad k , atunci $g = O(n^k)$.*

Demonstrație. Presupunem $g(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \dots + a_1 \cdot n + a_0$. Efectuând majorări în membrul drept, obținem:

$$g(n) \leq |a_k| \cdot n^k + |a_{k-1}| \cdot n^{k-1} + \dots + |a_1| \cdot n + |a_0| < n^k \cdot \underbrace{(|a_k| + |a_{k-1}| + \dots + |a_0|)}_c < n^k \cdot c$$

pentru $\forall n > 1 \Rightarrow g(n) < c \cdot n^k$, cu $n_0 = 1$. Deci $g = O(n^k)$. sfdem

Notățiile O , Ω și Θ oferă informații cu care putem aproxima comportarea unei funcții. Pentru ca această aproximare să aibă totuși un grad de precizie cât mai mare, este necesar ca mulțimea desemnată de partea dreaptă a ecuației să fie cât mai mică. De exemplu, avem atât $3n = O(n)$, cât și $3n = O(n^2)$. Prin incluziunea $O(n) = O(n^2)$ rezultă că prima ecuație oferă o aproximare mai bună. De fiecare dată vom căuta mulțimi care aproximează cel mai bine comportarea unei funcții.

Următoarele incluziuni sunt valabile în cazul notăției O :

$$O(1) \subset O(\log n) \subset O(\log^k n) \subset O(n) \subset O(n^2) \subset \dots \subset O(n^{k+1}) \subset O(2^n)$$

Notăția O este utilizată și pentru clasificarea algoritmilor. Cele mai cunoscute clase sunt:

$\{A \mid T_A(n) = O(1)\}$	= clasa algoritmilor constanți;
$\{A \mid T_A(n) = O(\log n)\}$	= clasa algoritmilor logaritmici;
$\{A \mid T_A(n) = O(\log^k n)\}$	= clasa algoritmilor polilogaritmici;
$\{A \mid T_A(n) = O(n)\}$	= clasa algoritmilor liniari;
$\{A \mid T_A(n) = O(n^2)\}$	= clasa algoritmilor pătratici;
$\{A \mid T_A(n) = O(n^k)\}$	= clasa algoritmilor polinomiali;
$\{A \mid T_A(n) = O(2^n)\}$	= clasa algoritmilor exponențiali.

Cu notățiile de mai sus, doi algoritmi, care rezolvă aceeași problemă, pot fi comparați numai dacă au timpii de execuție în clase de funcții (corespunzătoare notățiilor O , Ω și Θ) 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 timpii de execuție în aceeași clasă, atunci compararea lor devine mai dificilă pentru că trebuie determinate și constantele cu care se înmulțesc reprezentanții clasei.

Putem acum încadra algoritmii prezenți în exemplele anterioare în clasele lor.

Astfel, pentru algoritmul ce rezolva problema P_1 (căutarea unui element într-o secvență de numere), $T_{A_1}(n) = 3n + 2$. Deci, timpul de execuție al acestui algoritm pentru cazul cel mai nefavorabil este în clasa $O(n)$.

Pentru problema P_2 (căutarea într-un sir a elementului de valoare maximă), $T_{A_2}(n) = 1 + 2(n - 1)$. Timpul de execuție a algoritmului ce rezolvă această problemă pentru cazul cel mai nefavorabil este în clasa $O(n)$.

În cazul problemei P_3 (sortarea prin inserare), $T_{A_3}(n) = 3(n - 1) + 3n(n - 1)/2$. Rezultă că timpul de execuție al algoritmului respectiv este în clasa $O(n^2)$.

Pentru problema P_4 (căutarea binară) nu se poate aplica teorema 1.2. Deoarece $T_{A_4}(n) \leq \log_2 n + 1$, timpul de execuție al acestui algoritm este în clasa $O(\log_2 n)$. Baza logaritmului se poate ignora deoarece: $\log_a x = \log_a b \log_b x$ și $\log_a b$ este o constantă, deci rămâne $O(\log n)$, adică o funcție logaritmică. O demonstrație riguroasă pentru evaluarea timpului de execuție al algoritmului căutării binare este dată în capitolul 5, lema 5.1, corolarul 5.1.

1.3.3 Exerciții

Exercițiu 1.3.1. Se consideră programul:

```
x ← 0
y ← 3
while (y > 0) do
    x ← x + 4
    y ← y - 1
```

a) Să se descrie calculul programului.

b) Presupunând că:

- o comparație $x > y$, respectiv o adunare $x + y$, se realizează în timpul $\lfloor \log_2 |x| \rfloor + 1 + \lfloor \log_2 |y| \rfloor + 1$, dacă $x \cdot y \neq 0$, și timpul 1, dacă $x \cdot y = 0$,
- o atribuire $x \leftarrow a$ se realizează în timpul $\lfloor \log_2 |a| \rfloor + 1$, dacă $a \neq 0$, și timpul 1, dacă $a = 0$,
- o decrementare $x--$ se realizează în timpul $\lfloor \log_2 |x| \rfloor + 1$, dacă $x \neq 0$, și timpul 1, dacă $x = 0$,

să se determine timpul necesar executării programului.

c) Ce se poate spune despre timpul de execuție al algoritmului de înmulțire a două numere prin adunări repetate?

Exercițiu 1.3.2. Se consideră programul:

```
max ← -∞
for i ← 1 to n do
    if a[i] > max
        then max ← a[i]
```

Se presupune că numerele din a sunt alese arbitrar din $[0, 1]$.

- Dacă un număr x este ales arbitrar dintr-o mulțime cu n elemente, atunci care este probabilitatea ca x să fie cel mai mare număr din mulțime?
- Care este probabilitatea ca linia 4 să fie executată o singură dată?
- Care este probabilitatea ca linia 4 să fie executată de două ori?

Exercițiul 1.3.3. Să se arate că:

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

Exercițiul 1.3.4. Să se determine timpul de execuție, pentru cazul cel mai nefavorabil, al 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

```

Se presupune $a, b \geq 0$. Se vor considera două cazuri:

1. $g(p)=b$,
2. $g(p) = \lfloor \log n \rfloor$.

Indicație. Se va ține cont de faptul că programul calculează $f(a, b) = a^b$ după formula

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

Exercițiul 1.3.5. Să se determine timpul de execuție al 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

```

Exercițiul 1.3.6. Un tablou $a[1..n]$ conține toți întregii de la 0 la n cu excepția unuia. Acesta poate fi determinat în timpul $O(n)$ utilizând un tablou suplimentar $b[0..n]$ care memorează numerele ce apar în a . Presupunem că singura operație care permite accesul la elementele tabloului este „extrage bitul j din $a[i]$ ” care durează o unitate de timp. Să se arate că, utilizând numai această operație, este încă posibil să determinăm numărul lipsă în $O(n)$.

Exercițiul 1.3.7. Să se scrie un program care pentru un tablou unidimensional dat, determină dacă toate elementele tabloului sunt egale sau nu. Să se determine timpii de execuție ai algoritmului descris de program, pentru cazul cel mai nefavorabil și mediu.

Exercițiul 1.3.8. Se consideră polinomul cu coeficienți reali $p = a_0 + a_1x + \dots + a_nx^n$ și 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 îmbunătățească programul de mai sus, utilizând relația $x_0^{i+1} = x_0^i \cdot x_0$
- c) Să se scrie un program care să calculeze valoarea polinomului p în x_0 , utilizând formula (schema lui Horner):

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

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

Exercițiul 1.3.9. Se consideră programul:

```

k ← rand(n-2) + 1;
for i ← 1 to k do
    a[i] ← rand(n)
max ← -1
while ( i ≤ n and max = -1) do
    a[i] ← rand(n)
    ok ← true
    for j ← 1 to i-1 do
        if a[i] < a[j]
            then ok ← false
        if (ok) then max ← a[i]
    if (max = -1) then max ← a[n]

```

- a) Când sunt şansele maxime ca `max` să memoreze după execuția programului cea mai mare valoare din tabloul `a`?
- b) Care este timpul mediu de execuție dacă se numără numai atribuirile de pe penultima linie?

1.4 Referințe bibliografice

Temele abordate în acest capitol se regăsesc în toate cărțile dedicate studierii algoritmilor [Sed88, CLR93, CLR00, Baa78, BG00, Mel84a, AHU74, HS84, MB00, MS91, LG86].

Capitolul 2

Tipuri de date de nivel înalt

După cum am precizat în capitolul 1, tipurile de date de nivel înalt pot fi descrise într-o manieră independentă de limbaj. În aceasta categorie am inclus: liste liniare, stivele, cozile, listele generalizate, arborii binari, cozile cu priorități, grafurile, *union-find*.

2.1 Liste liniare

2.1.1 Tipul de date abstract LLIN

2.1.1.1 Descrierea obiectelor

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

2.1.1.2 Operații

ListaVidă. Întoarce lista vidă.

- Intrare:* – nimic;
Ieșire: – lista vidă () .

TesteVidă. Testează dacă o listă dată este vidă.

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

Lung. Întoarce lungimea unei listei date.

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

Copie. Întoarce o copie distinctă a unei liste date.

- Intrare:* – o listă liniară L ;
Ieșire: – o copie L' distinctă a lui L .

Egal. Testează dacă două liste liniare sunt egale.

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

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

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

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

- Intrare:* – o listă liniară L și o procedură *Vizitează*(e);
Ieșire: – lista L , dar ale cărei componente au fost prelucrate de procedura *Vizitează*.

Citește. Întoarce elementul de la o poziție dată dintr-o listă dată.

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

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

- Intrare:* – o listă liniară $L = (e_0, \dots, e_{n-1})$, un element e din Elt și o poziție $k \geq 0$;
Ieșire: – $L = (e_0, \dots, e_{k-1}, e, e_k, \dots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \dots, e_{n-1}, e)$ dacă $k \geq n$.

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

- Intrare:* – o listă liniară $L = (e_0, \dots, e_{n-1})$ și o poziție $k \geq 0$;
Ieșire: – $L = (e_0, \dots, e_{k-1}, e_{k+1}, \dots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \dots, e_{n-1})$ dacă $k \geq n$.

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

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

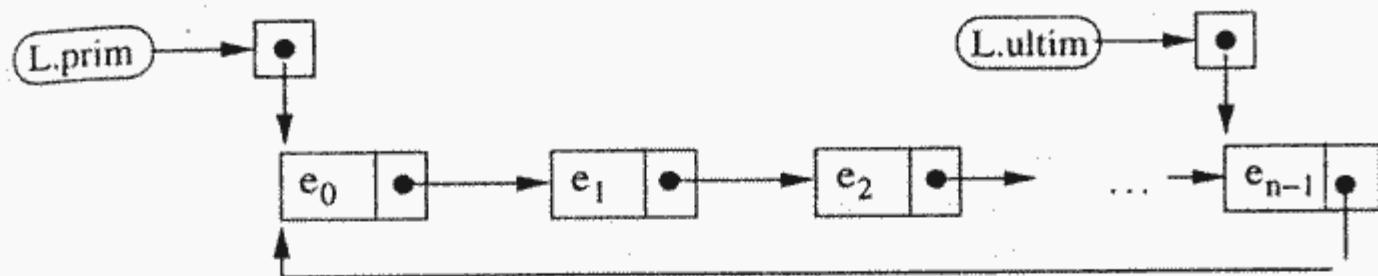


Figura 2.1: Reprezentarea liste liniare cu structură simplu înlățuită

2.1.2 Implementarea cu structuri dinamice simplu înlățuite

2.1.2.1 Reprezentarea obiectelor

O listă liniară $L = (e_0, \dots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică simplu înlățuită (figura 2.1). Fiecare componentă a liste liniare este memorată într-un nod al structurii dinamice simplu înlățuite. Există doi pointeri $L.\text{prim}$ și $L.\text{ultim}$ care fac referire la primul și respectiv ultimul nod.

2.1.2.2 Implementarea operațiilor

ListaVidă. Lista este reprezentată de structura simplu înlățuită fără nici un nod. Funcția **ListaVidă** întoarce $L.\text{prim} = L.\text{ultim} = \text{NULL}$.

EsteVidă. Se testează dacă pointerul $L.\text{prim}$ este egal cu **NULL**.

Lung. Parcurge structura simplu înlățuită și 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 operațiile care modifică lungimea listei să actualizeze corespunzător variabila responsabilă cu memorarea lungimii.

```

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

Copie. Parcurge structura simplu înlățuită sursă și adaugă la structura simplu înlățuită destinație o copie a fiecărui nod parcurs. Inițial lista destinație este vidă. Dacă presupunem că operația de copiere a unui nod se face în timpul $O(t)$, atunci copierea intregii liste se realizează în timpul $O(n \cdot t)$.

```

procedure copie (L, Lcopie)
    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 structuri simplu înlățuite și compară perechile de noduri corespunzătoare. Dacă presupunem că operația 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)
    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 secvențiale: se pleacă din primul nod și se interoghează nod cu nod până când este găsit primul care îl 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 operația de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```

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

```

Parcurge. Se parcurge structura simplu înlățuită nod cu nod și se aplică procedura Vizitează. Se obține un grad mai mare de aplicabilitate a operației dacă procedurile de tip Vizitează au ca parametru adresa unui nod și nu informația memorată în nod. Dacă t este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci timpul de execuție al operației de parcurgere este în clasa $O(n \cdot t)$.

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

Citește. Pentru a putea citi informația din cel de-al k -lea nod, este necesară parcurgerea secvențială a primelor k -noduri. Deoarece $k < n$, rezultă că timpul de execuție în cazul cel mai nefavorabil este în clasa $O(n)$.

```
function citește(L, k)
    if (k<0) then throw 'eroare'
    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 și în cazul operației de citire, este necesară parcurgerea secvențială a primelor $k - 1$ noduri (noul nod va fi al k -lea). Dacă avem $k = 0$ sau $k \geq \text{Lung}(L)$, atunci pointerii prim și respectiv ultim se actualizează corespunzător. Deoarece operația de adăugare a unui nod după o adresă dată se realizează în timpul $O(1)$, rezultă că operația de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure insereaza(L, k, e)
    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

```

EliminăDeLaK. Se realizează într-o manieră asemănătoare cu cea a operației de inserare.

```

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

```

Elimină. Se parurge structura simplu înlățuită secvențial și fiecare nod care memorează un element egal cu e este eliminat. Eliminarea unui nod presupune cunoașterea adresei nodului anterior; aceasta este determinată în timpul parcurgerii secvențiale. Dacă se elimină primul sau ultimul nod atunci pointerii $L.\text{prim}$ și respectiv $L.\text{ultim}$ se actualizează corespunzător. Operația de eliminare a unui nod se realizează în timpul $O(1)$. Dacă operația de comparare se realizează în timpul $O(1)$ atunci operația de eliminare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```

        delete(q)
        p ← p->succ
    end

```

2.1.3 Implementarea cu tablouri

2.1.3.1 Reprezentarea obiectelor

În afara reprezentării prin structuri simplu înlăncuite, o listă liniară $L = (e_0, \dots, e_{n-1})$ mai poate fi reprezentată și printr-un tablou $L.tab$ și o variabilă de tip indice $L.ultim$ ca în figura 2.2. Cele n componente ale 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 $\leq MAX$. Variabila $L.ultim$ memorează indicele (adresa) în tablou a ultimului element din listă.

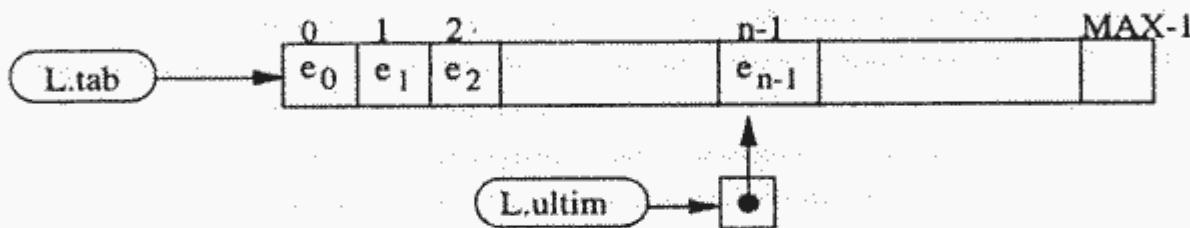


Figura 2.2: Reprezentarea listei liniare cu tablouri

2.1.3.2 Implementarea operațiilor

ListaVidă. Dacă valoarea variabilei $L.ultim$ este -1 , atunci lista este considerată ca fiind vidă.

EsteVidă. Se testează dacă valoarea variabilei $L.ultim$ este egală cu -1 .

Lung. Întoarce valoarea variabilei $L.ultim$ adunată cu 1 . Deci operația este realizată în timpul $O(1)$:

```

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

```

Copie. Parcurge tabloul sursă și copie fiecare componentă parcursă. Dacă presupunem că operația 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)
    Lcopie.ultim ← L.ultim
    for i ← 0 to L.ultim do
        Lcopie.tab[i] ← L.tab[i]
    end

```

Egal. Parcurge simultan cele două tablouri și compară perechile de componente corespunzătoare. Dacă presupunem că operația 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)
    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 secvențiale: se pleacă din prima componentă a tabloului și se interoghează componentă cu componentă până când este găsită prima care îl memorează pe x sau au fost epuizate toate componentele de indice $\leq L.ultim$. Plecând de la ipoteza că interogarea unei componente se face în timpul $O(1)$ rezultă că operația de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
function poz(L, x)
    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 și se aplică procedura Vizitează. Ca și în cazul implementării cu structuri dinamice simplu înlántuite, se obține un grad mai mare de aplicabilitate a operației dacă procedurile de tip Vizitează au ca parametru adresa în tablou. Dacă t este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci timpul de execuție al operației de parcurgere este $O(n \cdot t)$.

```
procedure parcurge(L, viziteaza())
    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)
    if ((k > L.ultim) or (k < 0)) then throw 'eroare'
    return L.tab[k]
end
```

Inserează. Elementele memorate în componentele $k, k+1, \dots, \text{ultim}$ trebuie deplasate la dreapta cu o poziție pentru a elibera componenta k . Valoarea variabilei $L.\text{ultim}$ este incrementată cu o unitate. Cum k poate fi și 0, rezultă că operația de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```

procedure insereaza(L, k, e)
    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

```

EliminăDeLaK. Elementele memorate în componentele $k, k+1, \dots, \text{ultim}$ trebuie deplasate la stânga cu o poziție. Valoarea variabilei $L.\text{ultim}$ este decrementată cu o unitate. Timpul de execuție în cazul cel mai nefavorabil este $O(n)$.

```

procedure eliminaDeLaK(L, k)
    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 parurge tabloul secvențial și fiecare componentă care memorează un element egal cu e este eliminată prin deplasarea la stânga a elementelor memorate după ea. La fiecare eliminare, variabila $L.\text{ultim}$ se actualizează corespunzător. Operația de eliminare a unui nod se realizează în timpul $O(n - k)$, unde k este indicele în tablou al componentei eliminate. Dacă operația de comparare se realizează în timpul $O(1)$, atunci operația de eliminare necesită timpul $O(n^2)$ în cazul cel mai nefavorabil (când toate elementele sunt egale cu e):

```

procedure elimina(L, e)
    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

```

2.1.4 Exerciții

Exercițiul 2.1.1. Să se proiecteze o structură de date de tip listă uniară pentru reprezentarea polinoamelor rare cu o singură variabilă și cu coeficienți întregi (polinoame de grade mari cu mulți coeficienți egali cu zero). Lista va memora numai coeficienții $\neq 0$. Să se scrie proceduri care, utilizând această structură, realizează operațiile algebrice cu polinoame și operațiile de citire și scriere de polinoame.

Exercițiul 2.1.2. 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 operații aritmetice cu întregi reprezentați astfel.

Exercițiul 2.1.3. Fie A un alfabet. Dacă $w = a_1 \dots a_n$ este un sir din A^* , atunci prin \bar{w} notăm oglinditul lui w , $\bar{w} = a_n \dots a_1$. Presupunem că sirurile din A^* sunt reprezentate prin liste liniare simplu înlanțuite. Să se scrie subprograme care să realizeze:

- fiind dat un sir w , determină oglinditul său \bar{w} ;
- fiind dat un sir w , decide dacă w este de forma $u\bar{u}$;
- fiind date două siruri w și w' , decide dacă $w \leq w'$ (se consideră ordinea lexicografică).

Exercițiul 2.1.4. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea mulțimilor finite de numere complexe. Să se scrie proceduri care să realizeze operații cu mulțimi de numere complexe reprezentate astfel.

Exercițiul 2.1.5. Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea numerelor raționale mari și cu multe zecimale. Să se scrie proceduri care să realizeze operații aritmetice cu numere raționale reprezentate astfel.

Exercițiul 2.1.6. Se consideră un sir de numere întregi memorat într-un tablou unidimensional a :

- Să se plaseze numerele din tabloul unidimensional a într-o listă liniară liniară implementată cu structuri dinamice simplu înlanțuite, prin inserări repetate în fața listei;
- Se dă o valoare intreagă x . Să se determine dacă x se află printre elementele listei create;
- Să se insereze în listă pe o poziție dată, un element care să conțină valoarea x ;
- Să se determine elementul situat pe poziția k , numărată de la sfârșitul la începutul listei, fără a parurge lista mai mult de o dată.

Exercițiul 2.1.7. Fie $X = (x_0, x_1, \dots, x_{n-1})$ și $Y = (y_0, y_1, \dots, y_{m-1})$ două liste liniare implementate cu structuri dinamice simplu înlanțuite. Să se scrie un program care să realizeze:

- concatenarea celor două liste: $Z = (x_0, x_1, \dots, x_{n-1}, y_0, y_1, \dots, y_{m-1})$;
- interclasarea celor două liste astfel:
 - $Z = (x_0, y_0, x_1, y_1, \dots, x_{m-1}, y_{m-1}, x_m, \dots, x_{n-1})$ dacă $m < n$,
 - $Z = (x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1}, y_n, \dots, y_{m-1})$ dacă $n \leq m$.

Exercițiul 2.1.8. Se consideră o listă liniară implementată cu structură dinamică simplu înlanțuită. Să se scrie subprograme care implementează următoarele operații:

- parcurgerea listei în ambele sensuri (dus-intors) utilizând $O(1)$ memorie suplimentară;
- testarea situației în care o listă are bucle;
- determinarea mijlocului listei (poziția elementului situat în mijlocul listei).

2.2 Liste liniare ordonate

2.2.1 Tipul de date abstract LLINORD

2.2.1.1 Descrierea obiectelor

O listă liniară ordonată este o listă liniară (e_0, \dots, e_{n-1}) , în care elementele e_i aparțin unui tip de date total ordonat Elt și succesiunea elementelor în secvență este în ordine crescătoare: $e_0 < \dots < e_{n-1}$.

2.2.1.2 Operații

Operațiile ListaVidă, EsteVidă, Lung, Egal, Copie, Parcurge și Poz sunt aceleași ca la lista liniară.

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

Intrare: – o listă liniară ordonată $L = (e_0 < \dots < e_{n-1})$ și un element e din Elt;

Ieșire: – $L = (e_0 < \dots < e_{k-1} < e < e_k < \dots < e_{n-1})$.

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

Intrare: – o listă liniară ordonată $L = (e_0 < \dots < e_{n-1})$ și un element e din Elt;

Ieșire: – $L = (e_0 < \dots < e_{k-1} < e_{k+1} < \dots < e_{n-1})$ dacă e apare în L pe locul k ($e = e_k$),
– lista L neschimbată dacă e nu apare în L .

2.2.2 Implementarea cu structuri dinamice simplu înlăntuite

2.2.2.1 Reprezentarea obiectelor

Similară cu cea a listelor liniare.

2.2.2.2 Implementarea operațiilor

Operațiile ListaVidă, EsteVidă, Lung, Egal, Copie și Parcurge au aceleași implementări precum cele de la lista liniară.

Poz. Algoritmul de căutare secvențială poate fi îmbunătățit în sensul următor: dacă s-a întâlnit un element $e_k > x$ atunci toate elementele care urmează sunt mai mari decât x și procesul de căutare se termină fără succes (întoarce valoarea -1). Timpul de execuție 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 $L.prim$ și $L.ultim$ vor referi în continuare acest nod.
2. $e < L.prim->elt$. Se adaugă un nod la începutul listei și 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 sfârșitul listei și 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 poziția pe care trebuie memorat e . Dacă în timpul căutării este întâlnit 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 poziție k cu proprietatea $e > e_k$, atunci inserează un nod în fața celui pe care s-a terminat procesul de căutare (acesta memorează cel mai mic element mai mare decât e) și memorează e în acest nou nod.

Timpul de execuție în cazul cel mai nefavorabil al algoritmului este $O(n)$.

```
procedure insereaza(L, e)
new(q)
q->elt ← e
switch
    case (L.prim = NULL)
        q->succ ← NULL /* cazul 1 */
        L.prim ← q
        L.ultim ← q
    case (e < L.prim->elt)
        q->succ ← L.prim /* cazul 2 */
        L.prim ← q
    case (e > L.ultim->elt)
        q->succ ← NULL /* cazul 3 */
        L.ultim->succ ← q
        L.ultim ← q
```

```

        otherwise /* (L.prim->elt <= e <= L.ultim->elt) */
            p ← L.prim /* cazul 4 */
            while (p->elt < e) do
                p ← p->succ
            if (p->elt ≠ e)
                then q->succ ← p->succ /* subcazul 4.2 */
                    p->succ ← q
                    q->elt ← p->elt
                    p->elt ← e
            else delete(q)
        end
    *

```

Elimină. Caută nodul care memorează e ca la implementarea operației Poz. În timpul procesului de parcurgere și căutare se memorează într-o variabilă pointer adresa nodului ce precedă 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 poziție, atunci trebuie actualizați pointerii $L.prim$ și respectiv $L.ultim$. Timpul de execuție în cazul cel mai nefavorabil al 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

```

2.2.3 Implementarea cu tablouri

2.2.3.1 Reprezentarea obiectelor

Similară cu cea a listelor liniare.

2.2.3.2 Implementarea operațiilor

Operațiile ListaVidă, EsteVidă, Lung, Egal, Copie și Parurge au aceleași implementări ca de la lista liniară.

Poz. (Căutarea binară). Algoritmul este prezentat la pagina 33. Cititorul este invitat să rescrie acel algoritm în termenii listei liniare ordonate. Timpul de execuție în cazul cel mai nefavorabil al algoritmului este $O(\log_2 n)$. Demonstrația acestui rezultat este prezentată și în secțiunea 5.2.

Inserează. Se realizează în doi pași:

1. Se caută binar poziția k unde urmează să fie memorat e .
2. Dacă în timpul căutării este întâlnită o componentă ce conține e (e este deja în listă), atunci operația se termină fără a modifica lista. Altfel, se translatează la dreapta toate elementele de la poziția k la poziția $L.ultim$ și se memorează e pe poziția k .

Etapa de căutare necesită $O(\log_2 n)$, timp în cazul cel mai nefavorabil, dar operația de translatare necesită $O(n)$. Astfel, algoritmul care realizează operația de inserare are timpul de execuție în cazul cel mai nefavorabil $O(n)$.

```

procedure insereaza(L, e)
    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 insuficientă'
            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 pași:

- 1 Se caută binar poziția k unde este memorat e . Dacă Poz e nu apare în listă și operația se termină fără a modifica lista.
2. Se translatează la stânga toate elementele de la poziția $k+1$ la poziția $L.ultim$. Ca și în cazul inserării, algoritmul care realizează operația de eliminare are timpul de execuție în cazul cel mai nefavorabil $O(n)$.

```

procedure elimina(L, e)
    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 ← ⌊  $\frac{p+q}{2}$  ⌋
    if (e = L.tab[k])
        then L.ultim ← L.ultim-1
            for i ← k to L.ultim
                L.tab[i] ← L.tab[i+1]
    end

```

2.2.4 Exerciții

Exercițiul 2.2.1. Se consideră un sir de numere întregi memorat în -un tablou unidimensional a. Să se plaseze valorile din a într-o listă liniară ordonată.

Exercițiul 2.2.2. 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 decât un n dat. (În legătură cu problema lui Hamming [Luc93].)

Exercițiul 2.2.3. Fie $X = (x_0, x_1, \dots, x_{n-1})$ și $Y = (y_0, y_1, \dots, y_{m-1})$, două liste liniare ordonate crescător. Să se scrie un subprogram care realizează interclasarea celor două liste astfel încât lista obținută să fie ordonată crescător.

2.3 Liste circulare

2.3.1 Tipul de date abstract LCIRC

2.3.1.1 Descrierea obiectelor

O listă circulară este o secvență finită (e_0, \dots, e_{n-1}) în care elementele e_i aparțin unui tip dat și iar ordinea indicilor $0, \dots, n-1$ este dată de regula $i < i+1 \pmod n$, $i = 0, \dots, n-1$. Altfel spus, o listă circulară este o listă liniară modificată astfel încât după e_{n-1} urmează e_0 .

2.3.1.2 Operații

Operațiile sunt aceleași ca la lista liniară.

2.3.2 Implementarea cu structuri dinamice simplu înlántuite

2.3.2.1 Reprezentarea obiectelor

O listă circulară $C = (e_0, \dots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică simplu înlántuită în care succesorul ultimului element este primul element. (figura 2.3). Fiecare componentă a listei circulare este memorată într-un nod al structurii dinamice simplu înlántuite. Ultimul nod al structurii „cunoaște” adresa primului nod. Există doi pointeri $L.\text{prim}$ și $L.\text{ultim}$ care fac referire la primul și respectiv ultimul nod. Se poate renunța la pointerul $L.\text{prim}$, deoarece $L.\text{prim} =$

`L.ultim->succ`. Acesta este de fapt unul dintre motivele introducerii structurii de listă circulară.

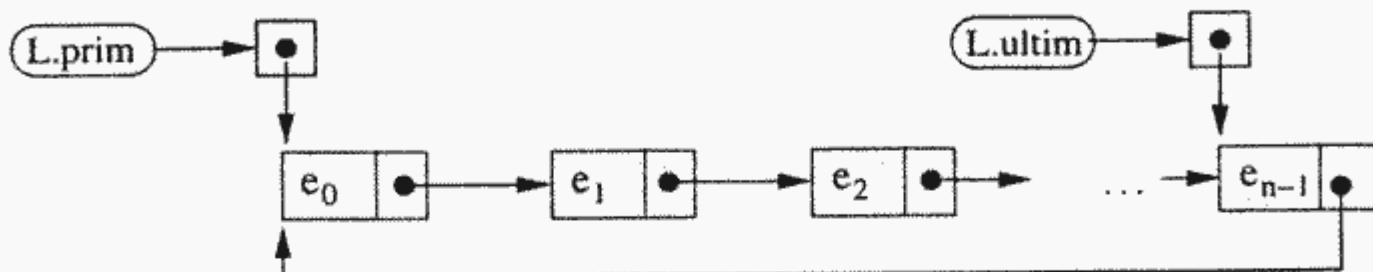


Figura 2.3: Reprezentarea listei circulare cu structură simplu înăntuită

2.3.2.2 Implementarea operațiilor

Invităm cititorul să modifice implementările de la liste liniare pentru a obține algoritmi corespunzători operațiilor cu liste circulare implementate cu structuri dinamice simplu înăntuite.

2.3.3 Exerciții

Exercițiul 2.3.1. Se consideră un sir de numere întregi memorat într-un tablou unidimensional a .

- Să se scrie un subprogram care să plaseze valorile din a într-o listă circulară, prin inserări repetitive în fața listei.
- Se dă o valoare x . Să se scrie un subprogram care să determine dacă aceasta se află printre elementele listei create.
- Să se scrie un subprogram care să insereze un element ce conțină valoarea x , într-o poziție dată în listă.
- Să se scrie un subprogram care să steargă un element din listă, poziția de stergere fiind dată.

Exercițiul 2.3.2. Să se scrie un subprogram care inversează legăturile într-o listă circulară implementată cu structură dinamică simplu înăntuită.

Exercițiul 2.3.3. Fie $X = (x_0, x_1, \dots, x_{n-1})$ și $Y = (y_0, y_1, \dots, y_{m-1})$ două liste circulare. Să se scrie un program care realizează:

- concatenarea celor două liste: $Z = (x_0, x_1, \dots, x_{n-1}, y_0, y_1, \dots, y_{m-1})$;
- interclasarea celor două liste astfel:
 - $Z = (x_0, y_0, x_1, y_1, \dots, x_{m-1}, y_{m-1}, x_m, \dots, x_{n-1})$ dacă $m < n$,
 - $Z = (x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1}, y_n, \dots, y_{m-1})$ dacă $n \leq m$.

Exercițiul 2.3.4. (Problema lui Joseph) Se consideră n persoane aranjate în cerc. Începând cu o poziție particulară p_0 se efectuează numărarea persoanelor și, în mod brutal, se execută fiecare a m -a persoană; cercul se închide după eliminarea unei persoane. Execuția se termină când rămâne o singură persoană; aceasta va rămâne în viață.

- Să se scrie un subprogram care creează o listă cu persoanele în ordinea în care au fost executate;
- Presupunând că se dorește salvarea unei anumite persoane, să se scrie un subprogram care să determine poziția p_0 astfel încât, dacă număratoarea începe de pe această poziție, atunci persoana salvată să fie cea dorită.

2.4 Stive

2.4.1 Tipul de date abstract STIVA

2.4.1.1 Descrierea obiectelor

Stivele sunt colecții de date cu proprietatea că se cunoaște „vechimea” fiecarui element, i.e., se cunoaște ordinea introducerii elementelor în colecție. La orice moment, ultimul element introdus în colecția de date este primul candidat la operațiile de citire și eliminare. Din acest motiv se mai numesc și tipuri de date LIFO (Last In, First Out). Presupunem că elementele unei stive aparțin tipului abstract Elt.

2.4.1.2 Operații

StivaVidă. Întoarce stiva vidă.

- Intrare:* – nimic;
Ieșire: – stiva fără nici un element.

EsteVidă. Testează dacă o stivă este vidă.

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

Push. Scrie un element în stivă.

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

Pop. Elimină ultimul element introdus în stivă.

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

Top. Citește ultimul element introdus într-o stivă.

Intrare: – o stivă S ;

Ieșire: – ultimul element introdus în S dacă S nu este stiva vidă,

– un mesaj de eroare dacă S este stiva vidă.

Observație. Au fost folosite denumirile Push, Pop și Top pentru că traducerea lor ar fi derutat cititorul.

sfobs

Observație. Tipul abstract STIVA poate fi „implementat” și de către tipul abstract LLIN în sensul că o stivă poate fi privită ca o listă liniară și că operațiile stivei pot fi exprimate cu ajutorul celor de la lista liniară:

STIVA.Push(S, e) = LLIN.Inserează($S, \text{Lung}(S), e$)

STIVA.Top(S) = LLIN.Citește($S, \text{Lung}(S)$)

STIVA.Pop(S) = LLIN.Elimină($S, \text{Lung}(S)$)

Astfel, o stivă poate fi reprezentată printr-o secvență (e_0, \dots, e_{n-1}) unde e_{n-1} este elementul din vârful stivei (cu vechimea cea mai mică). Ca o consecință, obținem următoarea interpretare pentru operații:

StivaVidă : $\mapsto ()$

Push : $((e_0, \dots, e_{n-1}), e) \mapsto (e_0, \dots, e_{n-1}, e)$

Pop : $(e_0, \dots, e_{n-1}) \mapsto (e_0, \dots, e_{n-2})$

Pop : $() \mapsto \text{eroare}$

Top : $(e_0, \dots, e_{n-1}) \mapsto e_{n-1}$

Top : $() \mapsto \text{eroare}$

sfobs

2.4.2 Implementarea cu structuri dinamice simplu înlățuită

2.4.2.1 Reprezentarea obiectelor

O stivă este o pereche formată dintr-o listă simplu înlățuită, care conține elementele memorate în stivă la un moment dat, și o variabilă referință care indică vârful stivei, adică ultimul element introdus (figura 2.4).

2.4.2.2 Implementarea operațiilor

StivaVidă. Constanță în crearea listei simplu înlățuite vide.

EsteVidă. Testează dacă vârful S este egal cu NULL.

Push. (Inserează) Constanță în adăugarea unui nod la începutul listei și „mutarea” vârfului S la nodul adăugat. Implementarea operației necesită timpul $O(1)$.

```

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

```

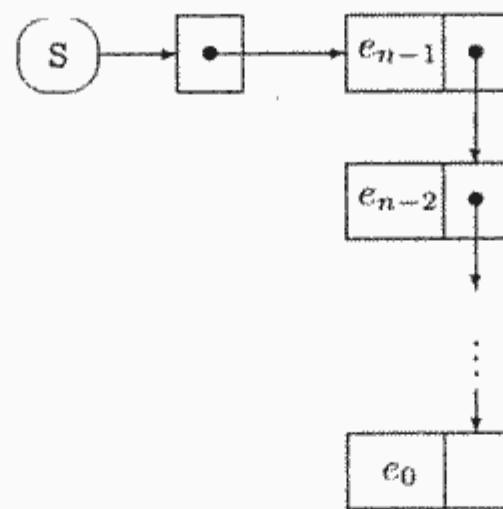


Figura 2.4: Stiva reprezentată ca listă înlățuită

pop. (Elimină) Consta în eliminarea nodului referit de pointerul S și „mutarea” vârfului a următorul nod. Ca și implementarea operației Push, necesită timpul $O(1)$.

```

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

Top. (Citește) Dacă stiva nu este vidă, furnizează informația memorată în nodul referit de vârful S .

```

function top(S)
  if (S = NULL) then throw 'eroare'
  return S->elt
end
  
```

2.4.3 Implementarea cu tablouri

2.4.3.1 Reprezentarea obiectelor

O stivă S este o pereche formată dintr-un tablou $S.tab$, care memorează elementele stivei, și o variabilă indice (vârful) $S.varf$ care indică poziția ultimului element introdus în stivă (figura 2.5). Aceasta poate fi utilizat și pentru determinarea numărului de elemente din stivă (operațiile standard nu cer această informație).

2.4.3.2 Implementarea operațiilor

StivaVidă. Vârful, care memorează adresa ultimului element introdus în stivă, este poziționat pe -1 .

EsteVidă. Testează dacă vârful este -1 .

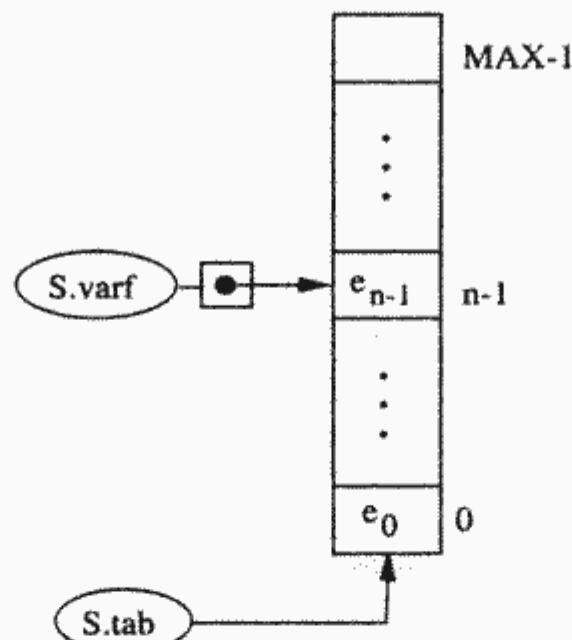


Figura 2.5: Reprezentarea stivei printr-un tablou

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

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

Pop. Dacă vârful este ≥ 0 , atunci acesta este decrementat. Evident, realizarea operației se face în timpul $O(1)$.

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

Top. Dacă vârful este ≥ 0 , atunci întoarce elementul memorat în tablou la adresa dată de vârf.

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

2.4.4 Exerciții

Exercițiul 2.4.1. [Knu74] Se consideră o mașină compusă din trei memorii: o secțiune de intrare și o secțiune de ieșire constând dintr-o secvență de n celule fiecare (o celulă poate memora un număr întreg) și o stivă cu memorie infinită. Controlul finit al mașinii este format dintr-un program construit cu următoarele instrucțiuni:

Read, introduce valoarea din prima celulă din secțiunea de intrare în stivă apoi deplasează conținutul secțiunii de intrare astfel încât conținutul din celula $k + 1$ trece în celula k . Conținutul din prima celulă se pierde, iar cel din ultima celulă devine nedefinit;

Write, deplasează conținutul secțiunii de ieșire astfel încât conținutul din celula k trece în celula $k + 1$. Conținutul din ultima celulă se pierde apoi, extrage un element din stivă și-l memorează în prima celulă din secțiunea de ieșire.

- Presupunând că secțiunea de intrare conține n valori distincte, să se determine numărul a_n al permutărilor ce pot fi obținute în secțiunea de ieșire.
- Să se scrie un program care enumera toate programele mașinilor ce realizează permutări.

Exercițiul 2.4.2. Forma poloneză (notația posfixată) a unei expresii aritmetice se caracterizează prin faptul că operatorii apar în ordinea în care se execută operațiile la evaluarea expresiei. De aceea, evaluarea unei expresii în forma poloneză se face parcurgând într-un singur sens expresia și executând operațiile ținând seama de paritatea lor.

Definirea recursivă a expresiilor în forma poloneză este următoarea:

1. Orice operand (constantă sau variabilă) E este în formă poloneză;
2. Dacă E este o expresie de forma $E_1 op E_2$, forma poloneză a expresiei este $E'_1 E'_2 op$, unde E'_1 și E'_2 sunt respectiv f.p. ale expresiilor E_1 și E_2
3. Dacă E este o expresie de forma (E) forma poloneză a expresiei E este de asemenea f.p. a expresiei E

Exemple: $(9 - 5) + 2$ are f.p. $9\ 5\ -\ 2\ +$, iar $9 - (5 + 2)$ are f.p. $9\ 5\ 2\ +\ -$.

Observație: Expresiile în forma poloneză nu conțin paranteze.

Un algoritm simplu de evaluare a expresiilor în forma poloneză este următorul:

```
/* Expresia se află într-un tablou s cu elemente
simboluri de tip operand sau operator binar */
i ← 0
while (nu s-a terminat sirul s)
    if (s[i] este operand)
        then depune s[i] în stiva
    else if (s[i] este operator)
        then extrage din stiva două simboluri t1 și t2
            executa operatia s[i] asupra lui t1 și t2
            depune rezultatul în stiva
        else semnalizeaza eroare
    i ← i+1
```

Observație. Algoritmul presupune că expresiile conțin doar operatori binari.

Să se scrie un program care să rezolve problema evaluării unei expresii date în forma poloneză.

Exercițiul 2.4.3. O altă formă fără paranteze a unei expresii aritmetice este forma prefixată.

Exemplu: $(9 - 5) + 2$ are forma prefixată $+ - 952$

- Să se formuleze o definiție recursivă pentru forma prefixată a unei expresii aritmetice.
- Să se scrie un program de trecere a unei expresii din forma postfixată în forma prefixată.
- Să se scrie un program care să rezolve problema evaluării unei expresii date în forma prefixată.

2.5 Cozi

2.5.1 Tipul de date abstract COADA

2.5.1.1 Descrierea obiectelor

Ca și stivele, *cozile* sunt colecții de date cu proprietatea că se cunoaște vechimea fiecărui element. La orice moment, primul element introdus în colecția de date este primul candidat la operațiile de citire și eliminare. Din acest motiv se mai numesc și tipuri de date FIFO (First In, First Out). Presupunem că elementele unei stive aparțin tipului abstract Elt.

2.5.1.2 Operații

CoadaVidă. Întoarce coada vidă.

- Intrare:* – nimic;
Ieșire: – coada fără nici un element.

EsteVidă. Testează dacă o coadă este vidă.

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

Inserează. Scrie un element în coadă.

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

Elimină. Elimină elementul cel mai vechi din coadă.

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

Citește. Citește elementul cel mai vechi din coadă.

Intrare: – o coadă C ;

Ieșire: – elementul cel mai vechi din coada C dacă C nu este coada vidă,
– un mesaj de eroare dacă C este coada vidă.

Observație. Ca și în cazul stivei, tipul abstract COADA poate „implementat” și de către tipul abstract LLIN în sensul că o coadă este privită ca o listă liniară și că operațiile cozii pot fi exprimate cu ajutorul celor de la lista liniară:

$\text{COADA.Inserează}(C, e) = \text{LLIN.Inserează}(C, \text{Lung}(C), e)$

$\text{COADA.Citește}(C) = \text{LLIN.Citește}(C, 0)$

$\text{COADA.Elimină}(C) = \text{LLIN.Elimină}(S, 0)$

Astfel, o coadă poate fi reprezentată printr-o secvență (e_0, \dots, e_{n-1}) , unde e_0 reprezintă capul cozii (elementul cu vechimea cea mai mare), iar e_{n-1} este elementul de la sfârșitul cozii (cu vechimea cea mai mică). Utilizând reprezentarea prin secvențe, operațiile se rescriu astfel:

$\text{CoadaVidă} : \rightarrow ()$

$\text{Inserează} : ((e_0, \dots, e_{n-1}), e) \rightarrow (e_0, \dots, e_{n-1}, e)$

$\text{Citește} : (e_0, \dots, e_{n-1}) \rightarrow e_0$

$\text{Citește} : () \rightarrow \text{eroare}$

$\text{Elimină} : (e_0, \dots, e_{n-1}) \rightarrow (e_1, \dots, e_{n-1})$

$\text{Elimină} : () \rightarrow \text{eroare}$

sfobs

2.5.2 Implementarea cu structuri dinamice simplu înlățuite

2.5.2.1 Reprezentarea obiectelor

O coadă este formată dintr-o listă simplu înlățuită C , care conține elementele memorate la un moment dat în coadă, o variabilă referință $C.\text{prim}$ care face referire la nodul cu vechimea cea mai mare (candidatul la ștergere și la interogare) și o variabilă referință $C.\text{ultim}$ care face referire la ultimul nod introdus în coadă (nodul cu vechimea cea mai mică) (figura 2.6).

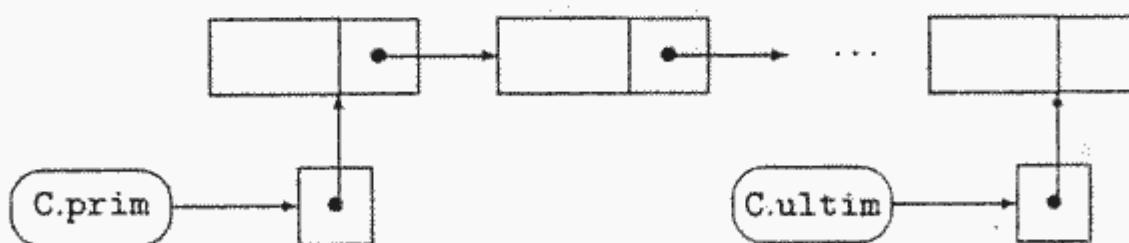


Figura 2.6: Coada reprezentată ca listă simplu înlățuită

2.5.2.2 Implementarea operațiilor

CoadaVidă. Creează lista simplu înlățuită vidă.

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

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

```
procedure insereaza(C, e)
    new(q)
    q->elt ← e
    q->succ ← NULL
    if (C.ultim ≠ NULL)
        then C.ultim->succ ← 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) și mută referința C.prim la nodul următor. Dacă lista avea un singur nod, atunci, după eliminare, va deveni vidă și cei doi pointeri vor fi actualizați corespunzător.

```
procedure elimina(C)
    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
```

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

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

2.5.3 Implementarea cu tablouri

2.5.3.1 Reprezentarea obiectelor

Elementele unei cozi sunt memorate într-un tablou. Câmpul C.prim, care acum este adresă în tablou, referă cel mai vechi element în coadă, iar C.ultim cel mai nou element în coadă (figura 2.7). Pentru o mai bună utilizare a spațiului de memorie vom conveni ca, atunci când cel mai nou element este memorat pe poziția ultimă MAX-1 din tablou, următorul element să fie memorat pe prima poziție din tablou, dacă aceasta este liberă (figura 2.7.b). Aceasta conduce la utilizarea unei aritmetici modulare pentru câmpurile C.prim și C.ultim. Mai rămâne de rezolvat problema

reprezentării cozii vide. Aceasta devine foarte simplă dacă vom considera o variabilă contor suplimentară $C.\text{nrElem}$ care să memoreze numărul elementelor din coadă. În plus, pentru că valoarea lui $C.\text{ultim}$ poate fi dedusă din valorile $C.\text{prim}$ și $C.\text{nrElem}$, utilizarea câmpului $C.\text{ultim}$ devine acum opțională.

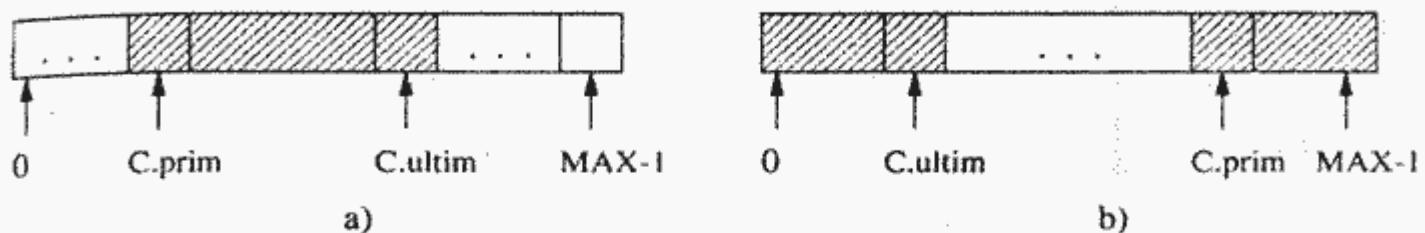


Figura 2.7: Reprezentarea cozii printr-un tablou

2.5.3.2 Implementarea operațiilor

CoadaVidă. Variabila $C.\text{nrElem}$ este pusă pe zero. Este convenabil ca $C.\text{ultim}$ să fie făcut egal cu 0, iar $C.\text{prim}$ să fie făcut egal -1.

EsteVidă. Testează dacă variabila $C.\text{nrElem}$ este zero.

Inserează. Dacă mai este loc în tablou, $C.\text{nrElem} < \text{MAX}$, atunci incrementează $C.\text{ultim}$ modulo MAX și memorează noul element e la noua adresă dată de $C.\text{ultim}$. Variabila contor $C.\text{nrElem}$ este incrementată.

```
procedure insereaza(C, e)
    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 $C.\text{prim}$ modulo MAX și decrementează variabila contor $C.\text{nrElem}$.

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

Citește. Dacă coada nu este vidă, furnizează informația de la adresa din tablou dată de pointerul $C.\text{prim}$.

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

2.5.4 Exerciții

Exercițiu 2.5.1. O coadă completă cu restricție la intrare este o listă liniară abstractă cu proprietatea că elementele se pot insera numai la unul din capete, iar ștergerile/citirile se pot efectua la oricare din capete.

- Să se implementeze această structură cu ajutorul listelor înlățuite.
- Să se implementeze această structură cu ajutorul tablourilor.

Exercițiu 2.5.2. O coadă completă cu restricție la ieșire este o listă liniară abstractă cu proprietatea că elementele se pot insera la oricare din capete iar ștergerile/citirile se pot efectua numai la unul din capete.

- Să se implementeze această structură cu ajutorul listelor înlățuite.
- Să se implementeze această structură cu ajutorul tablourilor.

Exercițiu 2.5.3. O coadă completă (deque) este o listă liniară abstractă cu proprietatea că elementele se pot inseră, șterge și citi la ambele capete.

- Să se implementeze această structură cu ajutorul listelor înlățuite.
- Să se implementeze această structură cu ajutorul tablourilor.

Exercițiu 2.5.4. Se consideră un tablou unidimensional cu 10.000 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.

2.6 Liste generalizate

2.6.1 Tipul de date abstract LGEN

2.6.1.1 Descrierea obiectelor

O listă generalizată este o secvență finită (e_0, \dots, e_{n-1}) în care elementele e_i sunt fie atomice, fie liste generalizate. Elementele atomice aparțin unui tip dat Elt. Componentele unei liste generalizate nu trebuie să fie neapărat distinse; adică, putem avea $i \neq j$ și $e_i = e_j$.

Listele generalizate sunt structuri definite recursiv. Elementele de tip listă ale unei liste generalizate se numesc subliste. În exemplul următor, lista generalizată A este formată din elementul atomic x și sublistele B , C , (a, x, c) , $()$, unde $()$ semnifică lista vidă.

atom listă
 ↓ ↓
 $A = (x, B, C, (a, x, c), ())$
 ↑ ↑
 listă lista vidă

2.6.1.2 Operații

Operațiile ListaVidă, EsteVidă, Copie și Egal au definiții identice cu cele de la lista inițială.

Head. Întoarce primul element dintr-o listă generalizată.

- Intrare:* – o listă generalizată $L = (e_0, \dots, e_{n-1})$;
Ieșire: – e_0 (primul element al listei L), dacă $L \neq ()$.
– eroare, dacă $L = ()$.

De notat că $\text{Head}(L)$ poate fi atom sau sublistă.

Tail. Întoarce lista rezultată prin eliminarea dintr-o listă generalizată a primului element.

- Intrare:* – o listă generalizată $L = (e_0, e_1, \dots, e_{n-1})$;
Ieșire: – $L' = (e_1, \dots, e_{n-1})$ (lista rezultată din L prin eliminarea din L a primului element), dacă $L \neq ()$
– eroare, dacă $L = ()$.

În continuare L și L' partajează elementele (e_0, \dots, e_{n-1}) .

Lung. Întoarce lungimea unei liste generalizate date exprimată în număr de atomi.

- Intrare:* – o listă generalizată $L = (e_0, \dots, e_{n-1})$;
Ieșire: – $\text{Lung}(L) = \text{numărul } m \text{ al componentelor atomice ale listei}$:

$$\text{Lung}(L) = \begin{cases} 0, & \text{dacă } L \text{ este vidă;} \\ 1 + \text{Lung}(\text{Tail}(L)), & \text{dacă } e_0 = \text{Head}(L) \text{ este atom;} \\ \text{Lung}(\text{Head}(L)) + \text{Lung}(\text{Tail}(L)), & \text{altfel.} \end{cases}$$

Pentru exemplul de la începutul secțiunii avem $\text{Lung}((a, x, c)) = 1 + \text{Lung}((x, c))$
 $= 2 + \text{Lung}((c)) = 3 + \text{Lung}(())) = 3 + 0 = 3$ și $\text{Lung}(A) = 1 + \text{Lung}((B, C, (a, x, c), ()))$
 $= 1 + \text{Lung}(B) + \text{Lung}((C, (a, x, c), ())) = \dots = 4 + \text{Lung}(B) + \text{Lung}(C)$.

Parurge. Traversarea listelor generalizate presupune vizitarea sistematică, fără repetare, a elementelor atomice ale listei.

- Intrare:* – o listă generalizată $L = (e_0, \dots, e_{n-1})$ și o procedură Vizitează(atom);
Ieșire: – lista L cu componentele atomice prelucrate de procedura Vizitează.

Observație. Ieșirea operației Parurge poate fi definită recursiv, cu ajutorul operațiilor Head și Tail.

$$\text{Parurge}(L) = \begin{cases} (), & \text{dacă } L \text{ este lista vidă;} \\ (\text{Vizitează}(e_0), \text{Parurge}(\text{Tail}(L))), & \text{dacă } e_0 \text{ este atom;} \\ (\text{Parurge}(\text{Head}(L)), \text{Parurge}(\text{Tail}(L))), & \text{altfel.} \end{cases}$$

Inserează. Adaugă la început un element dat (atom sau listă generalizată) într-o listă generalizată dată.

- Intrare:* – o listă generalizată $L = (e_0, \dots, e_{n-1})$ (posibil vidă) și x un atom
– sau o listă generalizată;
Ieșire: – $L = (x, e_0, \dots, e_{n-1})$.

Elimină. Elimină un element dat (atom sau listă generalizată) dintr-o listă generalizată dată.

Intrare: – o listă generalizată $L = (e_0, \dots, e_{n-1})$ (posibil vidă) și x un atom sau o listă generalizată;

Ieșire: – lista L din care s-au eliminat toate aparițiile lui x .

2.6.2 Implementarea cu structuri dinamice înlățuite

2.6.2.1 Reprezentarea obiectelor

O listă generalizată $L = (e_0, \dots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică înlățuită. Notăm cu L primul pointer care face referire la primul element. Un element de listă generalizată va fi o structură statică (`tip`, `data`, `succ`), unde câmpul `tip` conține o valoare $tip \in \{Atom, Lista\}$. Dacă $tip = Atom$, atunci elementul va corespunde unui obiect atomic. În consecință, câmpul `data` va conține valoarea atomului respectiv. Dacă $tip = Lista$, atunci elementul va corespunde unei subliste și câmpul `data` va semnifica legătura la primul element al sublistei. Câmpul `succ` memorează informația de înlățuire relativă la următorul element din listă.

Considerăm ca exemplu următoarele liste:

$A = (x, (y, z))$;

$B = (A, A, (), w)$.

O reprezentare a listelor generalizate A și B este descrisă în figura 2.8.

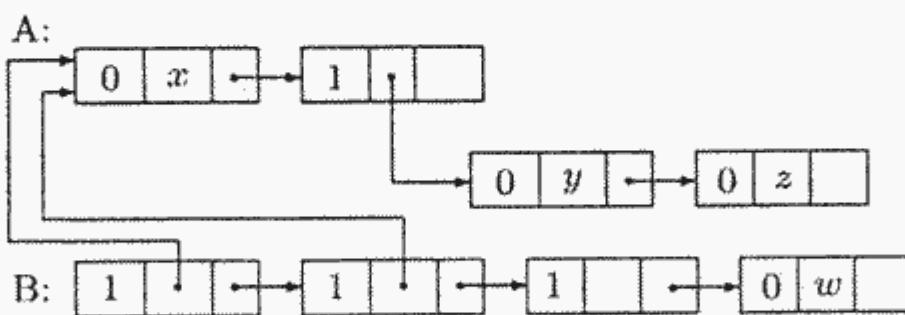


Figura 2.8: Reprezentarea listei generalizate cu structuri înlățuite

Pentru a facilita referirea sublistelor, convenim să dăm nume sublistelor. Astfel, dacă $tip = Lista$, atunci `data` va fi adresa unei structuri statice ce conține (*nume sublista*, *pointer la primul element*). Obținem reprezentarea din figura 2.9.

2.6.2.2 Implementarea operațiilor

Operațiile `ListaVidă`, `EsteVidă` au implementări similare cu cele de la lista liniară.

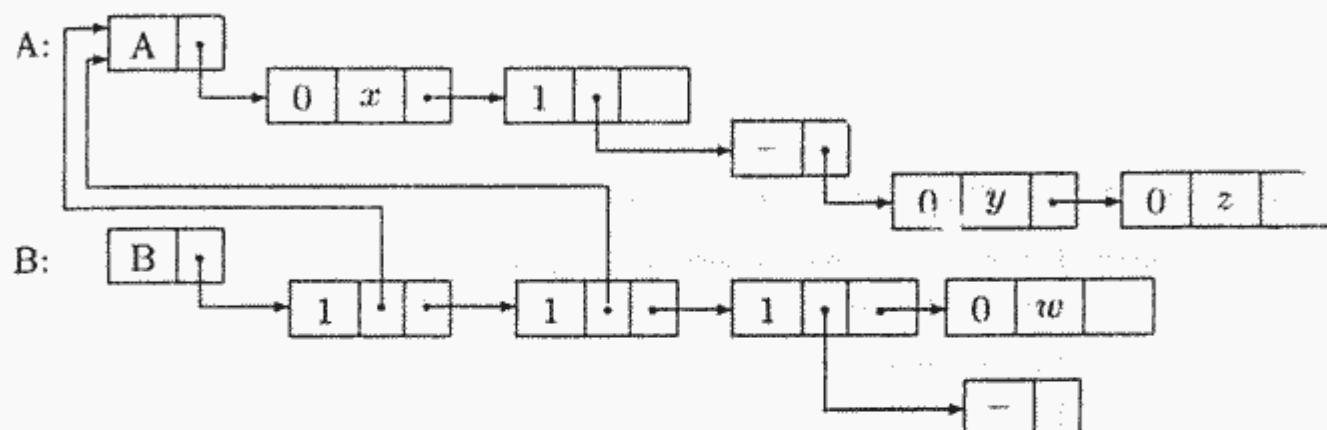


Figura 2.9:

Reprezentarea listei generalizate cu structuri înlățuite unde sublistele au nume

parcurge. Definiția unei liste generalizate este recursivă. În consecință, operația de parcurgere a unei liste generalizate va fi realizată prin algoritmi recursivi.

```

procedure parcurge(L, viziteaza())
    p ← L.prim
    while (p ≠ NULL) do
        if ((tip(p) = ATOM)
            then viziteaza(p->elt)
            else parcurge(p->elt, viziteaza())
        p ← p->succ;
    end
  
```

Se observă că dacă se face abstracție de procesarea diferențiată a elementelor în funcție de tipul acestora (*Atom* sau *Lista*) se obține algoritmul parcurgerii unei liste simplu înlățuite. Procesarea sublistelor se face prin apeluri recursive.

Operația de parcurgere a unei liste generalizate are timpul de execuție, pe cazul cel mai nefavorabil, $O(m)$, unde m este numărul elementelor de tip atom.

Copie. Copierea unei liste generalizate L are ca efect crearea unei dubluri a acesteia.

```
procedure Copie(L)
```

```

if (L = NULL)
    then return NULL
    else p ← new Elt
        if p->tip=Atom
            then p->data ← L->data
            else p->data ← Copie(L->data)
        p->succ ← Copie(L->succ)
        return p
end

```

Implementarea celorlalte operații o lasăm ca exercițiu. Inserarea unui element într-o listă generalizată este asemănătoare cu cea de la liste liniare. Elementul care se inserează poate fi un atom sau o sublistă. Operația de eliminare a unui element trebuie să țină seama de existența unor subliste comune.

2.6.3 Exerciții

Exercițiul 2.6.1. Să se definească recursiv operația Elimină.

Exercițiul 2.6.2. Se definește *Liniarizare*(*L*) ca fiind lista liniară (a_0, \dots, a_{k-1} , a_k, \dots, a_{m-1}) a atomilor listei generalizate *L*, scriși în ordinea dată de operația Parcurge. Să se definească cu ajutorul noțiunii *Liniarizare*(*L*) următoarele operații: Poz. Întoarce poziția unui atom dat dintr-o listă generalizată dată. Dacă atomul nu se află în listă, întoarce -1.

CiteșteAtom. Întoarce atomul de la o poziție dată dintr-o listă generalizată dată. Poziția este dată de numărul de ordine rezultat din parcurgerea listei generalizate (numărul de ordine în lista *Liniarizare*(*L*)). Dacă atomul nu se află în listă, întoarce 'eroare'.

ÎnsereazăAtom. Adaugă un atom dat într-o listă generalizată dată la o poziție dată. ÎnsereazăSublistă. Adaugă o sublistă dată într-o listă generalizată dată la o poziție dată.

Exercițiul 2.6.3. Să se definească recursiv operațiile Poz, CiteșteAtom, ÎnsereazăAtom și ÎnsereazăSublistă.

Exercițiul 2.6.4. Să se scrie subprograme care implementează operațiile Egal, Lung, Poz, CiteșteAtom, ÎnsereazăAtom și ÎnsereazăSublistă.

Exercițiul 2.6.5. Două liste generalizate se numesc echivalente dacă mulțimile atomilor sunt egale, iar sublistele sunt echivalente două câte două (nu contează ordinea). Mai precis, avem:

1. doi atomi sunt echivalenți dacă sunt egali;
2. două liste $L = \{e_0, \dots, e_{n-1}\}$ și $L' = \{e'_0, \dots, e'_{n-1}\}$ sunt echivalente dacă $n = n'$ și pentru orice $i \in \{0, \dots, n-1\}$ există un unic j a.i. e_i este echivalent cu e'_j .

De exemplu, $(A, (B, C), (D)) \equiv ((D), A, (C, B))$.

Să se scrie un subprogram care să determine dacă două liste generalizate sunt echivalente.

2.7 Arbore binari

2.7.1 Tipul de date abstract ABIN

2.7.1.1 Descrierea obiectelor

Multimea *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, și din doi arbori binari t_1 și t_2 , numiți *subarborele stâng* și respectiv *subarborele drept* (ai rădăcinii), este un arbore binar; notăm acest arbore prin $[e](t_1, t_2)$.

Convenim ca arboarele cu un singur nod $[e]([], [])$ să mai fie notat și prin notația mai simplă $[e]$. Fie v un nod într-un arbore binar t și v_1 și v_2 rădăcinile subarborelor stâng și respectiv drept ai nodului v . Atunci spunem că v este *nod tată* pentru v_1 și v_2 , v_1 este *fiul stâng* al lui v , iar v_2 este *fiul drept* al lui v . Un nod pentru care ambii subarbore sunt vizi este numit *frunză*. Totalitatea nodurilor frunză formează *frontiera* arborelui. Un *drum* într-un arbore este o succesiune de noduri v_1, \dots, v_n cu proprietatea că v_i este tatăl lui v_{i+1} pentru $i = 1, \dots, n-1$. *Lungimea* unui drum v_1, \dots, v_n este $n - 1$. *Dimensiunea* unui arbore t este egală cu numărul de noduri din t . *Înălțimea* unui arbore t este lungimea celui mai lung drum de la rădăcina lui t la un nod frunză. Prin definiție, înălțimea 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 definiției obținem succesiv arborii 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 figura 2.10. De exemplu, e este tatăl nodurilor b și g , b este fiul stâng al lui e iar g este fiul drept al lui e . Înălțimea arborelui este 3, iar dimensiunea sa este 9.

sfex

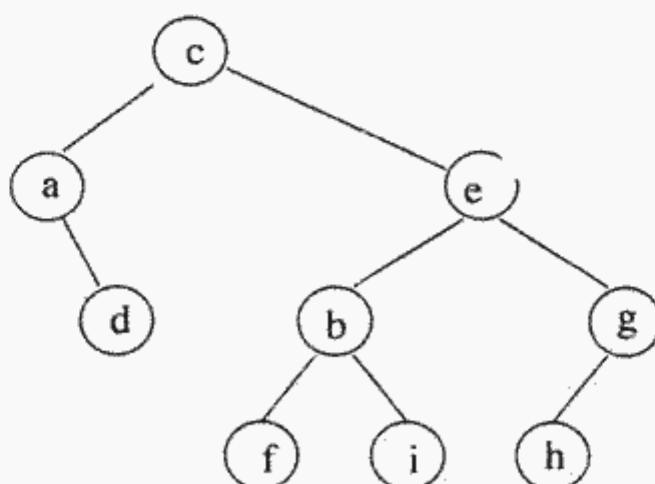


Figura 2.10: Exemplu de arbore binar

2.7.1.2 Operații

ArbBinVid.

- Intrare:* – nimic;
Ieșire: – arborele binar vid.

EsteVid.

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

Dimensiune.

- Intrare:* – un arbore binar t ;
Ieșire: – dimensiunea lui t .

Înălțime.

- Intrare:* – un arbore binar t ;
Ieșire: – înălțimea arborelui t .

Copie.

- Intrare:* – un arbore binar t ;
Ieșire: – o copie distinctă a lui t .

Egal.

- Intrare:* – doi arbori binari t_1 și t_2 ;
Ieșire: – *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 și subarborele stâng al lui t_1 este egal cu subarborele stâng al lui t_2 și subarborele drept al lui t_1 este egal cu subarborele drept al lui t_2 . Doi arbori vizi sunt egali prin definiție. Formal, $t_1 = t_2$ dacă și numai dacă:
• $t_1 = [] = t_2$ sau
• $t_1 = [a](t'_1, t''_1)$, $t_2 = [a](t'_2, t''_2)$ și $t'_1 = t'_2$, $t''_1 = t''_2$
– *false* în caz contrar.

ParurgePreordine.

- Intrare:* – un arbore binar t și o procedură *Vizitează(adr_nod)*;
Ieșire: – arborele t , dar în care nodurile au fost procesate în ordinea dată de parurgerea preordine a lui t . Parurgerea *preordine* a unui arbore t este obținută aplicând următoarea secvență de operații:
• vizitează rădăcina;
• parurge preordine subarborele stâng;
• parurge preordine subarborele drept.

De exemplu, parurgerea preordine a arborelui din figura 2.10 produce lista: $c, a, d, e, b, f, i, g, h$.

Parcurge în ordine.

- Intrare:* – un arbore binar t și o procedură **Vizitează(adr_nod)**;
Ieșire: – 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 obținută aplicând următoarea secvență de operații:
- parcurge inordine subarborele stâng;
 - vizitează rădăcina;
 - parcurge inordine subarborele drept.

De exemplu, parcurgerea inordine a arborelui din figura 2.10 produce lista: $a, d, c, f, b, i, e, h, g$.

Parcurge postordine.

- Intrare:* – un arbore binar t și o procedură **Vizitează(adr_nod)**;
Ieșire: – 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 obținută aplicând următoarea secvență de operații:
- parcurge postordine subarborele stâng;
 - parcurge postordine subarborele drept;
 - vizitează rădăcina.

De exemplu, parcurgerea postordine a arborelui din figura 2.10 produce lista $d, a, f, i, b, h, g, e, c$.

Parcurge BFS.

- Intrare:* – un arbore binar t și o procedură **Vizitează(adr_nod)**;
Ieșire: – 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 distanță 1 față de rădăcină), apoi a nodurilor aflate la distanță 2 față de rădăcină și.a.m.d. De exemplu, parcurgerea BFS a arborelui din figura 2.10 produce lista: $c, a, e, d, b, g, f, i, h$.

2.7.2 Implementarea cu structuri dinamice înlățuite

2.7.2.1 Descrierea obiectelor

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 informației din nod, un câmp **stg** pentru memorarea adresei rădăcinii subarborelui stâng și un câmp **drp** pentru memorarea adresei rădăcinii subarborelui drept. Accesul la întreaga structură de date este realizat prin intermediul rădăcinii. Figura 2.11 descrie reprezentarea dinamică a arborelui binar din figura 2.10.

2.7.2.2 Implementarea operațiilor

ArbBinVid. Constanță în crearea structurii vide ($t \leftarrow \text{NULL}$).

EsteVid. Testează dacă rădăcina face referire la vreun nod.

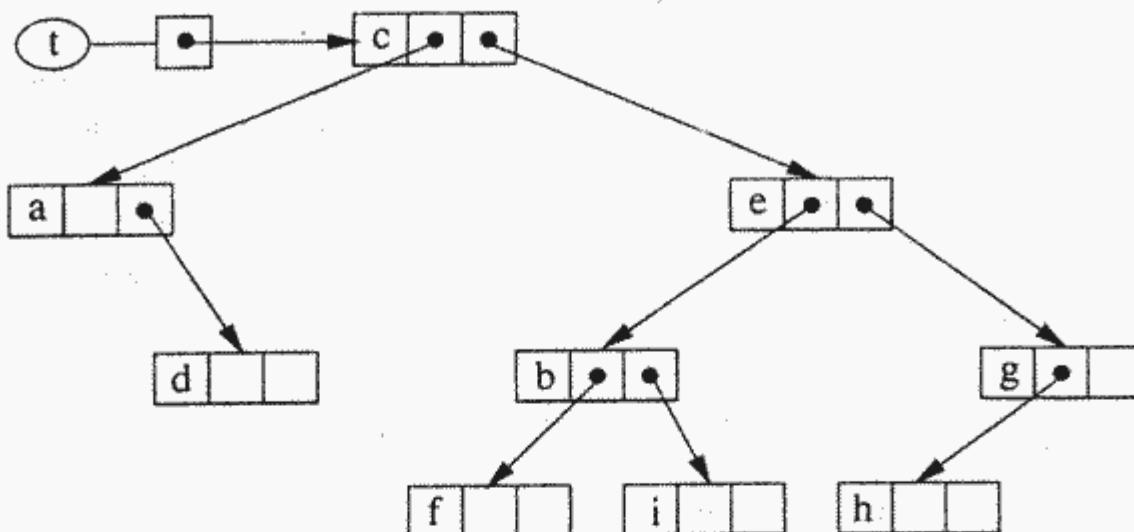


Figura 2.11: Reprezentarea arborelui binar cu structuri dinamice înlățuite

Dimensiune. Descrierea funcției 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 și dimensiunea subarborelui drept la care se adaugă 1 (rădăcina). Timpul de execuție este $O(n)$, unde n este chiar dimensiunea arborelui.

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

Înălțime. Funcția care calculează înălțimea are o descriere asemănătoare celei care calculează dimensiunea. Timpul de execuție este $O(n)$.

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

Copie. și această funcție are o descriere recursivă: se realizează o copie a rădăcinii după care se realizează prin apeluri recursive copii ale subarborelor rădăcinii. Lanțul apelurilor recursive se termină atunci când este întâlnit subarborele vid. Timpul de execuție pentru cazul cel mai nefavorabil este $O(n)$.

```
function copie(t)
    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 funcției `egal` este foarte simplă, omitem scrierea ei.

ParcurgePreordine. Din nou, cea mai simplă descriere este cea recursivă:

```

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

```

ParcurgeInordine. Este similară parcurgerii preordine.

ParcurgePostordine. Este similară parcurgerii preordine.

ParcurgeBFS. Pentru implementarea acestei operații se utilizează o coadă C cu următoarele proprietăți:

1. inițial coada conține numai rădăcina subarborelui;
2. la momentul curent se vizitează nodul extras din C ;
3. atunci când un nod este vizitat, fiile acestuia sunt adăugați în C .

Timpul de execuție pentru cazul cel mai nefavorabil este $O(n)$.

```

procedure parcurgeBFS(t, viziteaza())
    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

```

2.7.3 Implementarea cu tablouri

Nodurile arborelui binar sunt memorate într-un tablou de structuri. Valorile câmpurilor de legătură nu mai sunt adrese de variabile ci indici în tablou. O reprezentare statică a arborelui din figura 2.10 este dată în figura 2.12. Valoarea -1 joacă rolul pointerului NULL. Descrierile operațiilor sunt asemănătoare cu cele de la implementarea dinamică. Prezentăm, ca exemplu, parcurgerea inordine:

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

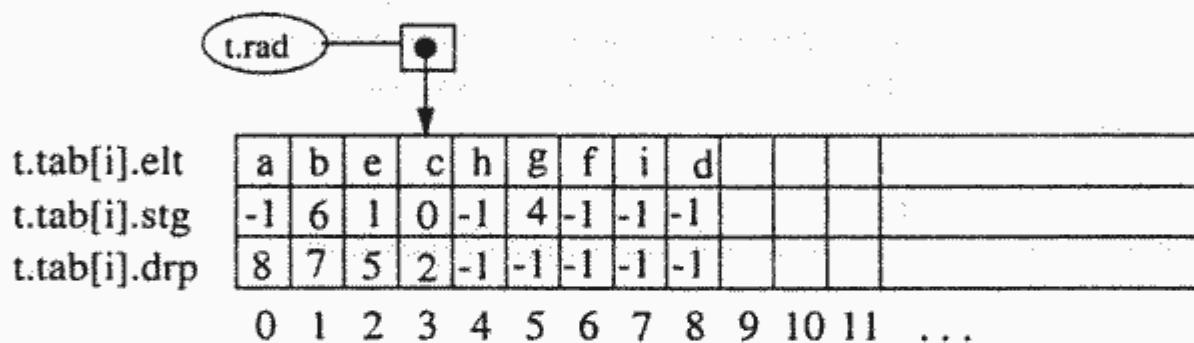


Figura 2.12: Reprezentarea cu tablouri a arborelui binar

2.7.4 Exerciții

Exercițiul 2.7.1. Să se scrie proceduri care implementează operațiile pentru arborii binari reprezentați cu tablouri.

Exercițiul 2.7.2. Asociem vârfurilor unui arbore binar adrese simbolice, ce sunt secvențe de 0 și 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 $\rho 1$.

- Să se scrie un program care, pentru un arbore dat, enumerează adresele simbolice ale tuturor vârfurilor în preordine.
- Să se scrie un program care, având la intrare adresele simbolice ale tuturor vârfurilor, construiește o listă înălțuită care reprezintă arborele.

Exercițiul 2.7.3. Să se scrie un program care, având la intrare liste liniare ale nodurilor unui arbore binar în preordine, respectiv în inordine, construiește lista înălțuită care reprezintă arborele. Mai este posibil același lucru dacă se consideră la intrare oricare altă pereche de liste (preordine și postordine sau inordine și postordine)?

Exercițiul 2.7.4. Fie t un arbore binar cu n noduri, v_1, \dots, v_n lista nodurilor în preordine și v_{p_1}, \dots, v_{p_n} lista nodurilor în inordine. Să se arate că permutarea p_1, \dots, p_n se poate obține cu o mașină de la exercițiul 2.4.1 și reciproc, orice permutare obținută cu o mașină de la exercițiul 2.4.1 corespunde unui arbore binar.

Exercițiul 2.7.5. Presupunem că Elt este total ordonat. Peste arborii binari se definește relația \prec astfel: $t_1 \prec t_2$ dacă și numai dacă:

$$\begin{aligned} t_1 &= [] \text{ sau} \\ t_1 &= [a_1](t'_1, t''_1), t_2 = [a_2](t'_2, t''_2) \text{ și} \\ a_1 &\prec a_2 \text{ sau} \\ a_1 &= a_2 \text{ și } t'_1 \prec t'_2 \text{ sau} \\ a_1 &= a_2 \text{ și } t'_1 = t'_2 \text{ și } t''_1 \prec t''_2. \end{aligned}$$

- Să se arate că, pentru orice doi arbori binari t_1 și t_2 , are loc $t_1 \prec t_2$, $t_1 = t_2$ sau $t_2 \prec t_1$.
- Să se scrie un program care, având la intrare doi arbori t_1 și t_2 , decide dacă $t_1 \prec t_2$ sau $t_2 \prec t_1$ sau $t_1 = t_2$.

Exercițiul 2.7.6. (*Arborei binari însăilați la dreapta*) Presupunem că structura unui nod cuprinde un câmp suplimentar **tdrp** cu următoarea semnificație:

- $v \rightarrow \text{tdrp} = \text{false}$ semnifică faptul că v nu are fiu dreapta, iar $v \rightarrow \text{drp}$ va conține adresa succesorului lui v din lista inordine. Un astfel de pointer se numește *însăilare*;
- $v \rightarrow \text{tdrp} = \text{true}$ semnifică faptul că v are fiu dreapta.

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

- Să se scrie un subprogram **sucInord(p, t)** care determină succesorul-inordine al unui nod arbitrar p în arboarele t . Procedura va utiliza $O(1)$ spațiu suplimentar. Care este timpul de execuție a algoritmului descris de subprogram pentru cazul cel mai nefavorabil?
- Este posibil să se scrie o procedură **sucInord** pentru arborii binari neînsăilați la dreapta? Dacă da, să se arate cum, iar dacă nu să se spună de ce.
- Să se descrie o procedură de parcursere a arborilor binari însăilați la dreapta, utilizând procedura **sucInord**. Să se arate că algoritmul de parcursere descris necesită $O(n)$ timp (n este numărul de noduri din arbore).

Exercițiul 2.7.7. Se consideră următoarea modificare a structurii de arbore binar:

- zona de legături a fiecărui nod conține două câmpuri suplimentare: **tstg** și **tdrp** cu semnificațiile:
 - $v \rightarrow \text{tstg} = \text{true}$, dacă v are subarbore stânga,
 - $v \rightarrow \text{tstg} = \text{false}$, dacă v nu are subarbore stânga și în acest caz $v \rightarrow \text{stg}$ este adresa predecesorului lui v în inordine,
 - $v \rightarrow \text{tdrp} = \text{true}$, dacă v are subarbore dreapta,
 - $v \rightarrow \text{tdrp} = \text{false}$, dacă v nu are subarbore dreapta și în acest caz $v \rightarrow \text{drp}$ este adresa succesorului lui v în inordine.

Un asemenea arbore se numește *însăilat*.

- Să se scrie o procedură de parcursere în inordine a arborilor însăilați.

- (ii) Să se scrie o procedură care transformă un arbore binar obișnuit într-un arbore însălat.

Exercițiul 2.7.8. Să se scrie o procedură care să numere nodurile de pe frontieră unui arbore binar.

Exercițiul 2.7.9. Se consideră arbori binari care au ca informație în noduri numere întregi. *Lungimea externă ponderată* a arborelui t este

$$\sum \{v \rightarrow \inf * \text{lung}(v) \mid v \text{ este nod pe frontieră lui } t\},$$

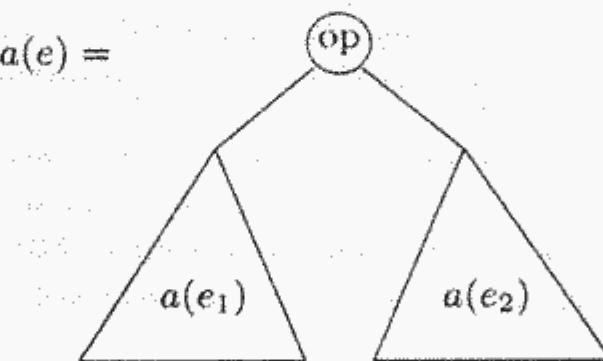
unde $\text{lung}(v)$ este lungimea drumului de la rădăcină la nodul v . Să se scrie un subprogram care determină lungimea externă a unui astfel de arbore.

Exercițiul 2.7.10. (*Reprezentarea expresiilor aritmetice*) Considerăm expresii formate numai din numere întregi și operatorii binari $+$, $-$, $/$, $*$, $\%$. Unei expresii e se poate asocia un arbore binar $a(e)$ astfel:

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

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

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



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 și oferă la ieșire arborele binar asociat expresiei. Arborele asociat este unic? Să se justifice cum rezolvă programul problema unicității.

Exercițiul 2.7.11. Să se arate că notația postfixată se obține prin parcurgerea postordine.

Exercițiul 2.7.12. Să se arate că notația prefixată se obține prin parcurgerea preordine.

Exercițiul 2.7.13. Fie T un arbore binar reprezentat prin structuri dinamice înlanțuite. Să se scrie un subprogram care să numere frunzele lui T care au frate.

Exercițiul 2.7.14. Fie T un arbore binar reprezentat prin structuri dinamice înlanțuite. Să se scrie un subprogram care să numere nodurile de pe frontieră lui T care au nivelul egal cu înălțimea arborelui.

2.8 Coada cu priorități

2.8.1 Tipul de date abstract coadă cu priorități

2.8.1.1 Obiectele

O coadă cu priorități este o structură de date în care elementele sunt numite *atomi*, iar fiecare atom conține un câmp-cheie care ia valori dintr-o mulțime total ordonată. Valoarea acestui câmp-cheie se numește *prioritate*. Operațiile de citire/eliminare se referă întotdeauna la atomul cu prioritatea cea mai mare. Interpretarea noțiunii de *prioritate* poate dифeри de la caz la caz. Există situații când atomii cei mai prioritari sunt cei cu cheile valorilor mai mici și există situații când atomii cei mai prioritari sunt cei cu cheile valorilor mai mari. Noi considerăm aici ultimul caz.

2.8.1.2 Operații

CoadaVidă.

- Intrare:* – nimic;
- Ieșire:* – coada cu priorități vidă.

Elimină.

- Intrare:* – o coadă cu priorități Q ;
- Ieșire:* – Q din care s-a eliminat atomul cu cheia cea mai mare (dacă există).

Înserează.

- Intrare:* – o coadă cu priorități Q și un atom a ;
- Ieșire:* – Q la care s-a adăugat a .

Citește.

- Intrare:* – o coadă cu priorități Q ;
- Ieșire:* – atomul cu cheia cea mai mare din coada Q .

2.8.2 Implementarea cu max-heap-uri

2.8.2.1 Descrierea max-heap-urilor

Definiția 2.1. Un max-heap este un arbore binar complet cu proprietățile:

1. informațiile din noduri sunt valori dintr-o mulțime total ordonată numite chei;
2. pentru orice nod intern v , cheia memorată în v este mai mare sau egală cu cheia oricărui fiu.

În continuare vom arăta cum max-heap-urile sunt reprezentate prin tablouri 1-dimensionale.

Definiția 2.2. Pentru un $n \in N^*$, arborele binar complet atașat lui n este definit prin

$$H_n = (\{0, \dots, n-1\}, A)$$

unde A este mulțimea de arce $\{(\left[\frac{i-1}{2}\right], i) \mid i = 1, \dots, n-1\}$.

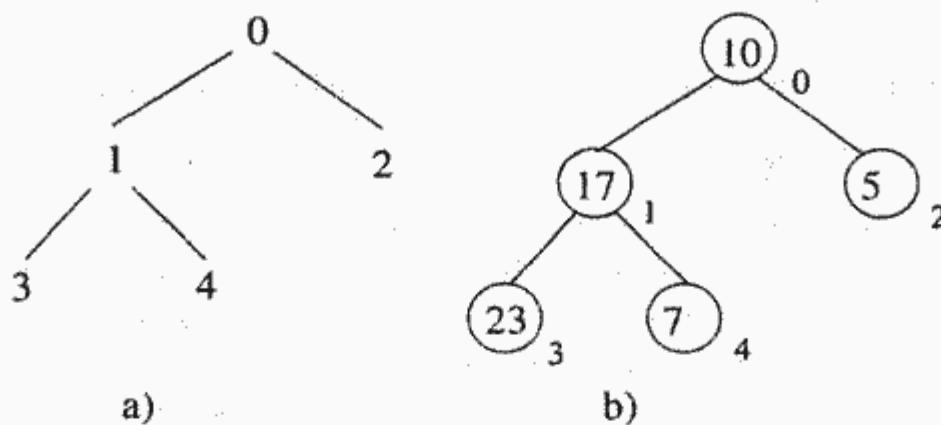


Figura 2.13: Arbore binar complet

Exemplu. Pentru $n = 5$, arborele H_5 este reprezentat în figura 2.13a. Multimea arcelor este $A = \{(0, 1), (0, 2), (1, 3), (1, 4)\}$. sfex

Lema 2.1. Au loc următoarele proprietăți ale arborelui H_n :

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

Definiția 2.3. Fie $a = (a_0, \dots, a_{n-1})$. Prin $H_n(a)$ notăm arborele obținut din H_n prin etichetarea vârfurilor i cu elementele a_i în mod corespunzător.

Exemplu. Dacă $a = (10, 17, 5, 23, 7)$, atunci arborele $H_5(a)$ este reprezentat în figura 2.13b.

Definiția 2.4. a) Tabloul $(a[i] \mid i = 0, \dots, n - 1)$ are proprietatea MAX-HEAP, dacă $(\forall k)(1 \leq k < n \Rightarrow a[\left\lfloor \frac{k-1}{2} \right\rfloor] \geq a[k])$. Notăm această proprietate prin MAX-HEAP(a).

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

Vom da câteva proprietăți ale predicatelor $\text{MAX-HEAP}(a)$ și $\text{MAX-HEAP}(a, \ell)$.

Lema 2.2. 1. Tabloul $(a[i] \mid i = 0, \dots, n - 1)$ are proprietatea MAX-HEAP (adică $\text{MAX-HEAP}(a) = \text{true}$) dacă și 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 $(a[i] \mid i = 0, \dots, n - 1)$ are loc MAX-HEAP(a , $\left\lfloor \frac{n}{2} \right\rfloor$).
 3. Dacă MAX-HEAP(a , ℓ), atunci pentru orice $j > \ell$ are loc MAX-HEAP(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 figura 2.14 este prezentat un exemplu de max-heap însotit de tabloul cu reprezentarea sa.

Teorema 2.1. Fie $a = (a_0, \dots, a_{n-1})$. Atunci $H_n(a)$ este max-heap dacă și numai dacă $\text{MAX-HEAP}(a)$.

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

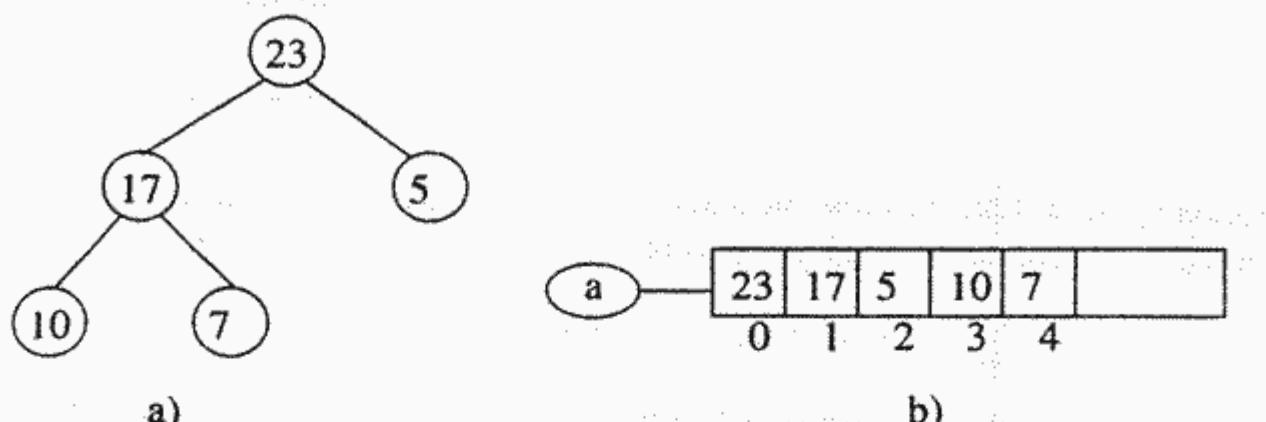


Figura 2.14: Exemplu de max-heap

2.8.2.2 Implementarea operatiilor

CoadaVidă. Este reprezentată de tabloul vid.

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

1. Presupunem că vârful curent este j . Inițial avem $j = 0$.
 2. Dacă $2 * j + 1 \leq n - 1$, atunci determină vârful cu valoarea maximă dintre fiii lui j ; fie acesta k . Altfel, refacerea proprietății este terminată.
 3. Dacă $a[j] \geq a[k]$, atunci refacerea proprietății este terminată.
 4. Dacă $a[j] < a[k]$, atunci
 - (a) Interschimbă $a[j]$ cu $a[k]$.
 - (b) Repetă pasul 2 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de eliminare este:

```

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

```

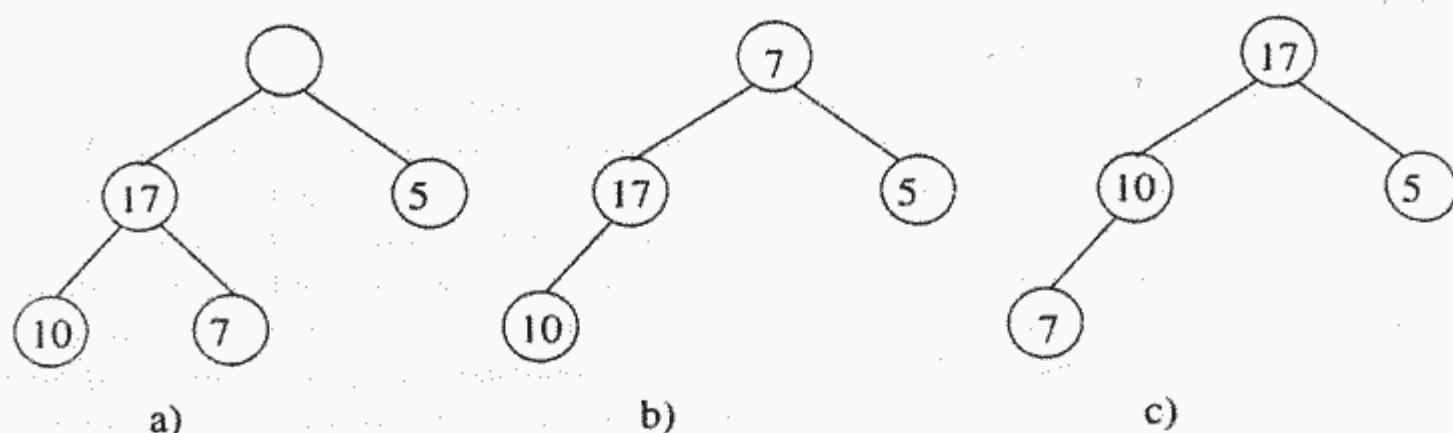


Figura 2.15: Eliminarea din max-heap-ul din figura 2.14

Timpul de execuție în cazul cel mai nefavorabil este $O(\log_2 n)$. În figura 2.15 este arătat modul în care este eliminat atomul cu cheia cea mai mare din max-heap-ul din figura 2.14.

Înserează. Presupunând că max-heap-ul are n elemente, se memorează noul atom pe poziția n și se incrementează dimensiunea max-heap-ului. Apoi se refac proprietatea MAX-HEAP parcurgând un drum de la nodul ce memorează noul atom spre rădăcină, conform următorului algoritm:

1. Presupunem că vârful curent este j . Inițial avem $j = n - 1$.
2. Dacă $j > 0$, atunci determină $k =$ poziția nodului-tată. Altfel, refacerea proprietății este terminată.
3. Dacă $a[j] \leq a[k]$, atunci refacerea proprietății este terminată.
4. Dacă $a[j] > a[k]$, atunci
 - (a) Interschimbă $a[j]$ cu $a[k]$.
 - (b) Repetă pasul 2 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de inserare este:

```

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

Timpul de execuție în cazul cel mai nefavorabil este $O(\log_2 n)$. În figura 2.16 este arătată inserarea unui atom cu cheia egală cu 20 în max-heap-ul din figura 2.14.

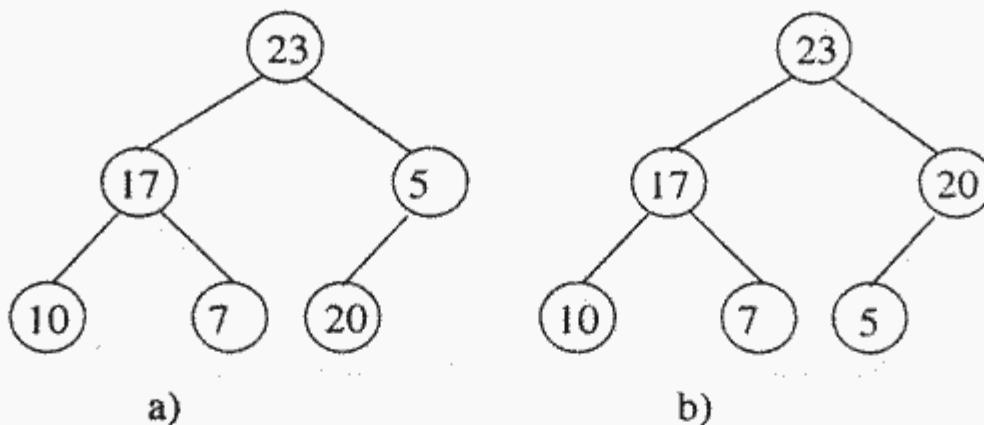


Figura 2.16: Inserarea cheii 20 în max-heap-ul din figura 2.14

Citește. Întoarce primul element din tablou. Timpul de execuție în cazul cel mai nefavorabil este $O(1)$.

2.8.3 Exerciții

Exercițiul 2.8.1. Să se scrie un program care, pentru un tablou a dat, determină dacă a are proprietatea MAX-HEAP.

2.9 Grafuri

2.9.1 Definiții

Prezentarea de aici se bazează pe cea din [Cro92]. Un *graf* este o pereche $G = (V, E)$, unde V este o mulțime ale cărei elemente le numim *vârfuri*, iar E este o mulțime de perechi neordonate $\{u, v\}$ de vârfuri, pe care le numim *muchii*. Aici considerăm numai cazul când V și E sunt finite. Dacă $e = \{u, v\}$ este o muchie, atunci u și v se numesc extremitățile muchiei e ; mai spunem că e este *incidentă* în u și v sau că vârfurile u și v sunt *adiacente* (*vecine*). În general, muchiile și grafurile sunt reprezentate grafic ca în figura 2.17. Dacă G conține muchii de forma $\{u, u\}$, atunci o asemenea muchie se numește *bucă*, iar graful se numește *graf general* sau *pseudograf*. Dacă pot să existe mai multe muchii $\{u, v\}$, atunci G se numește *multigraf*. Denumirea provine de la faptul că, în acest caz, E este o multimulțime. Considerăm numai cazul în care nu există nici bucle și nici muchii multiple. Două grafuri $G = (V, E)$; $G' = (V', E')$ sunt *izomorfe* dacă există o funcție $f : V \rightarrow V'$ astfel încât $\{u, v\}$ este muchie în G , dacă și numai dacă $\{f(u), f(v)\}$ este muchie în G' . Un graf $G' = (V', E')$ este *subgraf* al lui $G = (V, E)$ dacă $V' \subseteq V$, $E' \subseteq E$. Un *subgraf parțial* G' al lui G este un subgraf cu proprietatea $V' = V$. Dacă G este un graf și $X \subseteq V$ o submulțime de vârfuri, atunci *subgraful induș* de X are ca mulțime de vârfuri pe X și mulțimea de muchii formată din toate muchiile lui G incidente numai în vârfuri din X .

Un *mers* de la u la v (de extremități u și v) în graful G este o secvență

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

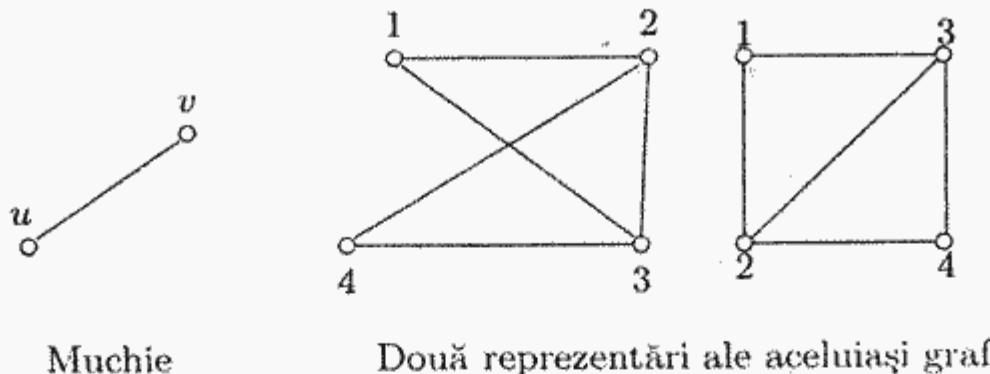


Figura 2.17: Reprezentarea grafurilor

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 figura 2.17, secvența 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 numește *parcurs*, iar dacă toate vârfurile sunt distincte, atunci el se numește *drum*. Mersul din exemplul de mai sus nu este nici parcurs și nici drum. Un mers în care extremitățile coincid se numește *închis* sau *ciclu*. Dacă într-un mers toate vârfurile sunt distincte, cu excepția extremităților, se numește *circuit* (*drum închis*). Un graf G se numește *conex* dacă între oricare două vârfuri ale sale există un drum. Un graf care nu este conex se numește *neconex*. Orice graf se poate exprima ca fiind reuniunea disjunctă de subgrafuri induse, conexe și maximale cu această proprietate; aceste subgrafuri sunt numite *componente conexe*. De fapt, o componentă conexă este subgraful induș de o clasă de echivalență, unde relația de echivalență este dată prin: vârfurile u și v sunt echivalente dacă și numai dacă există un drum de la u la v .

Graful din figura 2.18 are două componente conexe: $G_1 = (\{1, 3, 4, 6\}, \{\{1, 4\}, \{1, 6\}, \{4, 3\}, \{6, 3\}\})$ și $G_2 = (\{2, 5, 7\}, \{\{2, 7\}, \{5, 2\}, \{7, 5\}\})$.

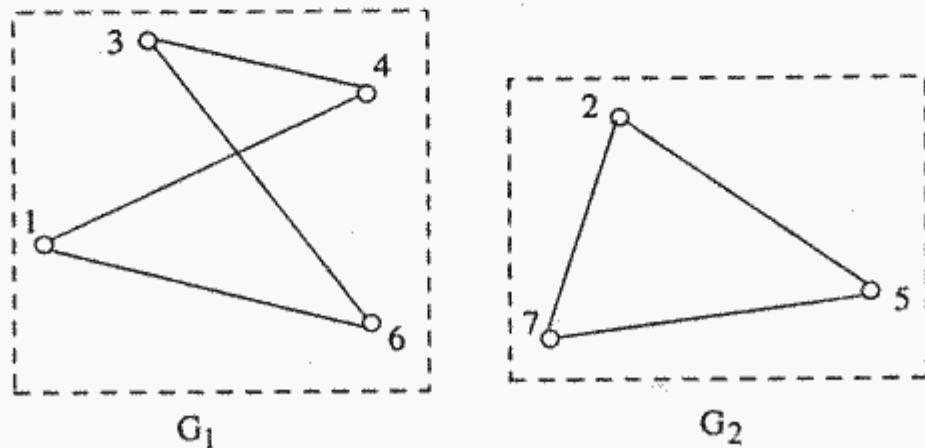


Figura 2.18: Componente conexe

Un *digraf* este o pereche $D = (V, A)$, unde V este o mulțime de vârfuri, iar A este o mulțime de perechi ordonate (u, v) de vârfuri. Considerăm cazul când

V și A sunt finite. Elementele multimii A se numesc *arce*. Dacă $a = (u, v)$ este un arc, atunci spunem că u este *extremitatea inițială (sursa)* a lui a și că v este *extremitatea finală (destinația)* lui a ; mai spunem că u este un *predecesor imediat* al lui v și că v este un *succesor imediat* al lui u . Formulări echivalente: a este *incident din u și incident în (spre) v, sau a este *incident cu v spre interior* și a este *incident cu u spre exterior*, sau a pleacă din u și *sosește în v*. Arcele și digrafurile sunt reprezentate ca în figura 2.19. Orice digraf D definește un graf, numit *graf suport al lui D*, obținut prin înlocuirea oricărui arc (u, v) cu muchia $\{u, v\}$ (dacă mai multe arce definesc aceeași muchie, atunci se consideră o singură muchie). Graful suport al digrafului din figura 2.19 este cel reprezentat în figura 2.17. Mersurile, parcursurile, drumurile, ciclurile și circuitele se definesc ca în cazul grafurilor, dar considerând arce în loc de muchii. Un digraf se numește *tare conex* dacă pentru orice două vârfuri u și v , există un drum de la u la v și un drum de la v la u . O *componentă tare conexă* este subgraful induș de o clasă de echivalență, unde relația de echivalență invocă acum definiția de drum într-un digraf. Digraful din figura 2.20 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.*

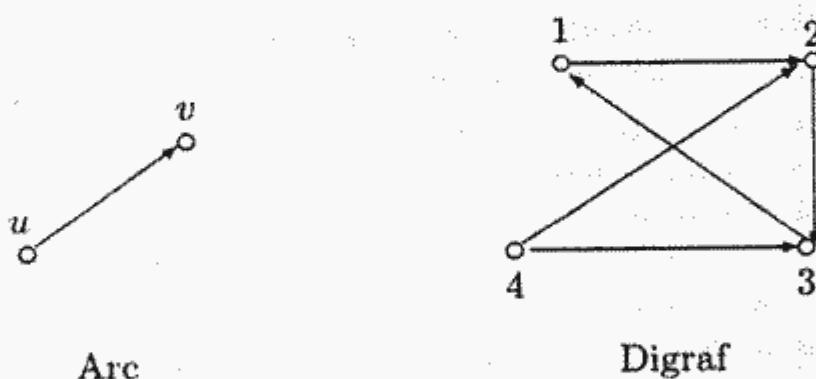


Figura 2.19: Reprezentarea digrafurilor

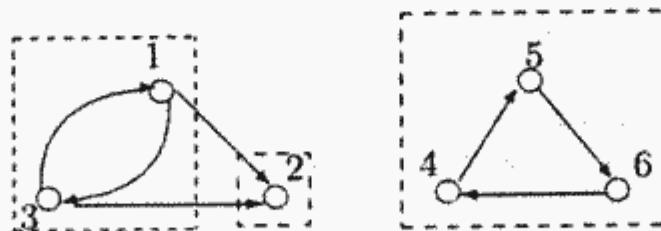


Figura 2.20: Componente tare conexe

Un *(di)graf etichetat pe vârfuri* este un *(di)graf* $G = (V, E)$ împreună cu o mulțime de etichete L și o funcție de etichetare $\ell : V \rightarrow L$. În figura 2.21.a este reprezentat un graf etichetat, în care funcția de etichetare este $\ell(1) = A, \ell(2) = B, \ell(3) = C$. Un *(di)graf etichetat pe muchii (arce)* este un *(di)graf* $G = (V, E)$ împreună cu o mulțime de etichete L și o funcție de etichetare $\ell : E \rightarrow L$. În figura 2.21.b este reprezentat un graf etichetat pe arce, în care funcția de etichetare este $\ell(\{1, 2\}) = A, \ell(\{2, 3\}) = B, \ell(\{3, 1\}) = C$. Dacă mulțimea de etichete este o

submulțime a lui \mathcal{R} (mulțimea numerelor reale), atunci (di)graful se mai numește *ponderat*.

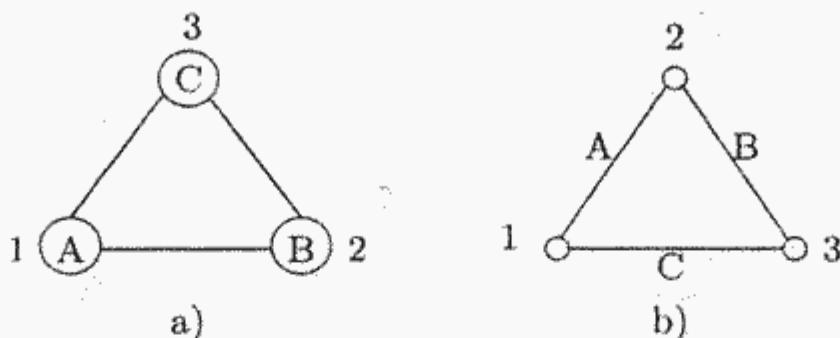


Figura 2.21: Grafuri etichetate

Un *arbore* este un graf conex fără circuite. Un *arbore cu rădăcină* este un digraf fără circuite, în care există un vârf r , numit *rădăcină*, cu proprietatea că, pentru orice alt vârf v , există un singur drum de la r la v . Într-un arbore cu rădăcină, sensurile arcelor sunt unic determinate de drumurile care pleacă din rădăcină și din acest motiv nu vom mai desena săgețile care marchează orientarea. Un *arbore cu rădăcină ordonat* este un arbore cu rădăcină cu proprietatea că orice vârf u , exceptând rădăcina, are exact un predecesor imediat și, pentru orice vârf, mulțimea succesorilor imediați este total ordonată. Într-un arbore cu rădăcină ordonat, succesorii imediați ai vârfului u se numesc și *fii* ai lui u , iar predecesorul imediat al lui u se numește *tatăl* lui u . Un caz particular de arbore ordonat este arborele binar, unde relația de ordine peste mulțimea filor vârfului u este precizată prin (fiul stâng, fiul drept). Exemple de arbori sunt date în figura 2.22.

2.9.2 Tipul de dată abstract Graf

2.9.2.1 Descrierea obiectelor

Obiectele de tip dată sunt grafuri $G = (V, E)$, definite în mod unic până la un izomorfism, unde mulțimea de vârfuri V este o submulțime finită a tipului abstract Vârf, iar mulțimea de muchii E este o submulțime a tipului abstract Muchie. Fără să restrângem generalitatea, presupunem că mulțimile de vârfuri V sunt mulțimi de forma $\{0, 1, \dots, n - 1\}$, unde $n = \#V$, iar mulțimile de muchii sunt submulțimi ale mulțimii $\{\{i, j\} \mid i, j \in \{0, 1, \dots, n - 1\}\}$. Dacă $G = (V, E)$ este oarecare cu $\#V = n$, atunci putem defini o funcție bijectivă $\text{index}_G : V \rightarrow \{0, 1, \dots, n - 1\}$ și graful $G' = (\{0, 1, \dots, n - 1\}, E')$ cu $\{\text{index}_G(u), \text{index}_G(v)\} \in E' \iff \{u, v\} \in E$. Evident, G și G' sunt izomorfe și deci putem lucra cu G' în loc de G . Întoarcerea de la G' la G se poate face via funcția inversă lui index_G , $\text{nume}_G : \{0, 1, \dots, n - 1\} \rightarrow V$, dată prin $\text{nume}_G(i) = v \iff \text{index}_G(v) = i$.

2.9.2.2 Operații

GrafVid.

Intrare: – nimic;

Ieșire: – graful vid $G = (\emptyset, \emptyset)$.

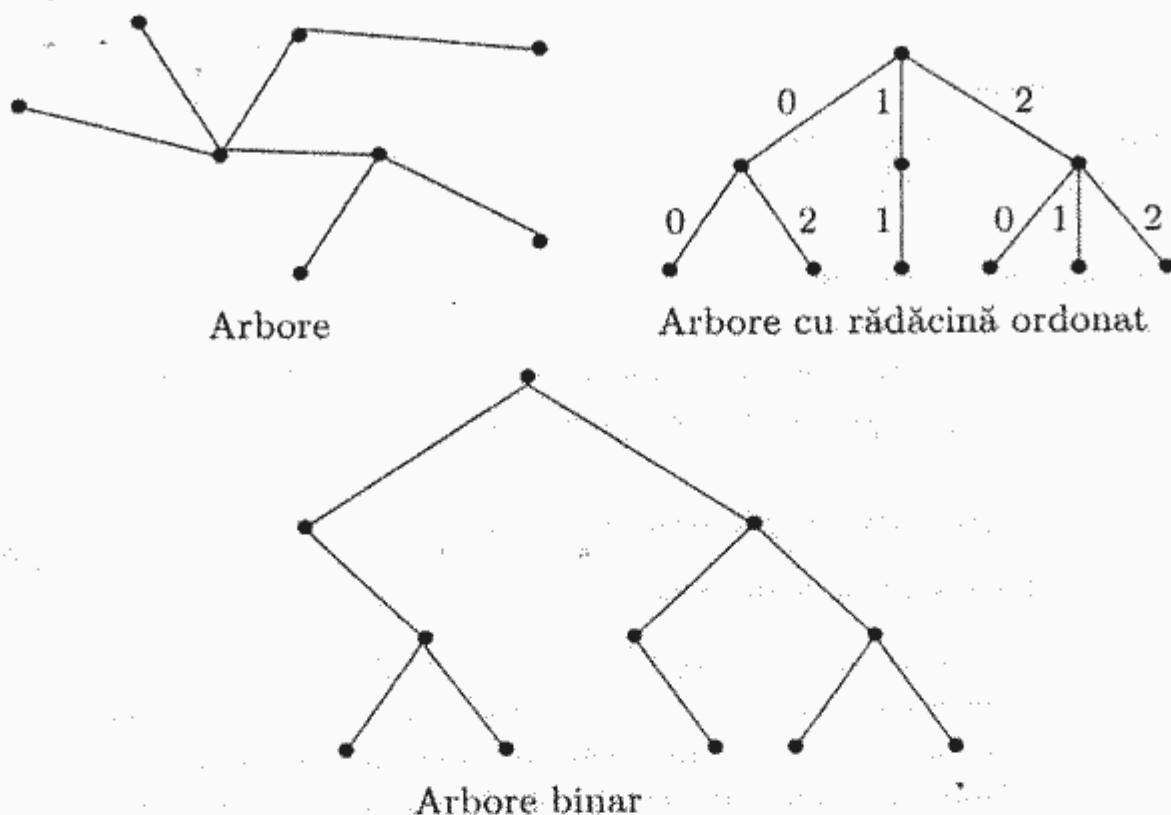


Figura 2.22: Exemple de arbori

EsteGraffiti.

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

InsereazăVârf.

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

InsereazăMuchie.

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

StergeVârf.

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

StergeMuchie.

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

ListaDeAdiacență.

- Intrare:* – un graf $G = (V, E)$ și un vârf $i \in V$;
Ieșire: – mulțimea vârfurilor adiacente cu vârful i .

ListaVârfurilorAccesibile.

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

2.9.3 Tipul de dată abstract Digraf

2.9.3.1 Descrierea obiectelor

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

2.9.3.2 Operații

DigrafVid.

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

EsteDigrafVid.

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

InsereazăVârf.

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

InsereazăArc.

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

ȘtergeVârf.

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

ȘtergeArc.

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

Listă De Adiacență Interioară.

- Intrare:* – un digraf $D = (V, A)$ și un vârf $i \in V$;
Ieșire: – multimea surselor (vârfurilor de plecare ale) arcelor care sosesc în vârful i .

Listă De Adiacență Exterioară.

- Intrare:* – un digraf $D = (V, A)$ și un vârf $i \in V$;
Ieșire: – multimea destinațiilor (vârfurilor de sosire ale) arcelor care pleacă din vârful i .

Listă Vârfurilor Accesibile.

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

2.9.4 Reprezentarea grafurilor ca digrafuri

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

2.9.5 Implementarea cu matrice de adiacență

2.9.5.1 Reprezentarea obiectelor

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

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

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

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

Observație. În loc de valorile 0 și 1 se pot considera valorile booleene *false* și *true*.

sfobs

2.9.5.2 Implementarea operațiilor

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

EsteDigrafVid. Se testează dacă $D.m$ și $D.n$ sunt egale cu zero.

InsereazăVârf. Consta în adăugarea unei linii și a unei coloane la matricea de adiacență.

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

InsereazăArc. Inserarea arcului (i, j) presupune numai actualizarea componentei $D.a[i, j]$, adică $D.a[i, j] \leftarrow 1$.

ȘtergeVârf. Notăm cu a' valoarea matricei $D.a$ înainte de ștergerea vârfului k și cu a' valoarea de după ștergere. Au loc următoarele relații (vezi și figura 2.23):

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 relații deducem următorul algoritm:

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

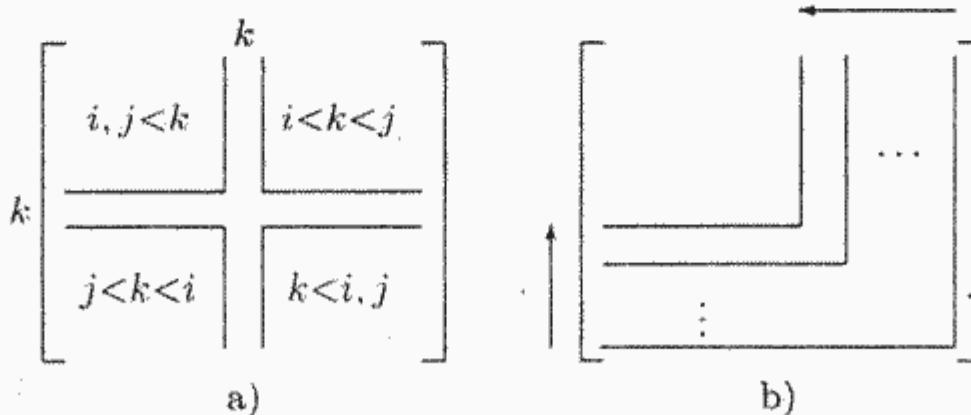


Figura 2.23: Ștergerea unui vârf

ȘtergeArc. Asemănător operației InsereazăArc.

ListaDeAdiacențăInterioară și **ListaDeAdiacențăExterioară**. Lista vârfurilor destinațare 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 obține 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 cu proprietatea că există drum de la i la k și drum de la k la j . De fapt, cele de mai sus sunt echivalente cu afirmația că relația „există drum de la i la j ”, definită peste V , este închiderea reflexivă și tranzitivă a relației „există arc de la i la j ”. Ultima relație este reprezentată de matricea de adiacență, iar prima relație se poate obține din ultima prin următorul algoritm datorat lui Warshall (1962):

```

procedure detInchRefTranz(D, b)
    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 **detInchRefTranz** determină într-adevăr închiderea reflexivă și tranzitivă. Se observă ușor că reflexivitatea este rezolvată de primele două instrucțiuni **for** care realizează și copierea $D.a$ în b . În continuare ne ocupăm de tranzitivitate. Fie i și j două vârfuri cu proprietatea că j este accesibil din i . Există un k și un drum de la i la j cu vârfuri intermedii din mulțimea $X = \{0, \dots, k\}$. Vom arăta că după k iterații avem $b[i,j] = 1$. Procedăm prin inducție 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 intermedii din $\{0, \dots, k-1\}$ și un drum de la k la j cu vârfuri intermedii tot din $\{0, \dots, k-1\}$. Din ipoteza inductivă, rezultă că după $k-1$ execuții ale buclei **for** $k \dots$ avem $b[i,k] = 1$ și $b[k,j] = 1$. După cea de-a k -a execuție al buclei obținem $b[i,j] = 1$, prin execuția ramurii **then** a ultimei instrucțiuni **if**.

2.9.6 Implementarea cu liste de adiacență înlanțuite

2.9.6.1 Reprezentarea obiectelor

Un digraf D este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacență, dar matricea de adiacență este înlocuită cu un tablou unidimensional

de n liste liniare, implementate prin liste simplu înăntărite și notate cu $D.a[i]$ pentru $i = 0, \dots, n - 1$, astfel încât lista $D.a[i]$ conține vârfurile destinate ale arcelor care pleacă din i . (= lista de adiacență exterioară).

Exemplu. Fie G graful reprezentat în figura 2.24a. Tabloul listelor de adiacență corepunzătoare lui G este reprezentat în figura 2.24b. Pentru digrafu D din figura 2.25.a, tabloul listelor de adiacență înăntărite este reprezentat în figura 2.25.b. sfex

De multe ori este util ca, atunci când peste mulțimea vârfurilor este dată o relație de ordine, componentele listelor de adiacență să respecte această ordine.

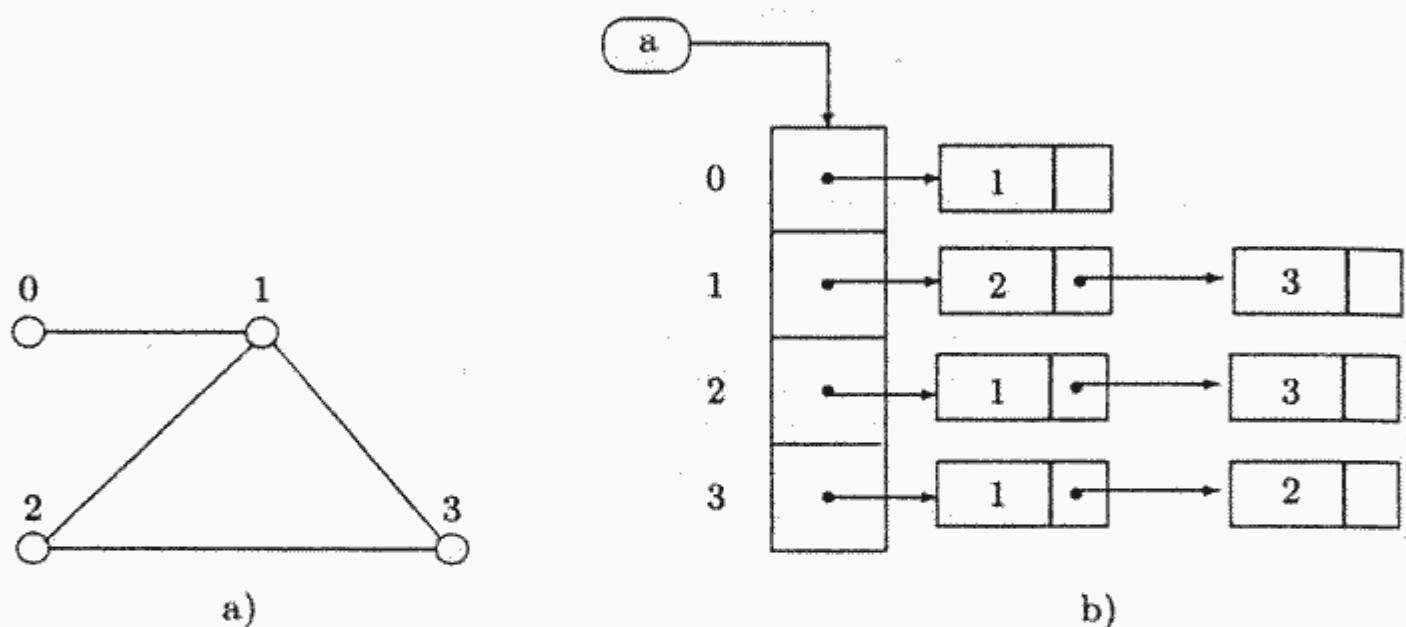


Figura 2.24: Graf reprezentat prin liste de adiacență înăntărite

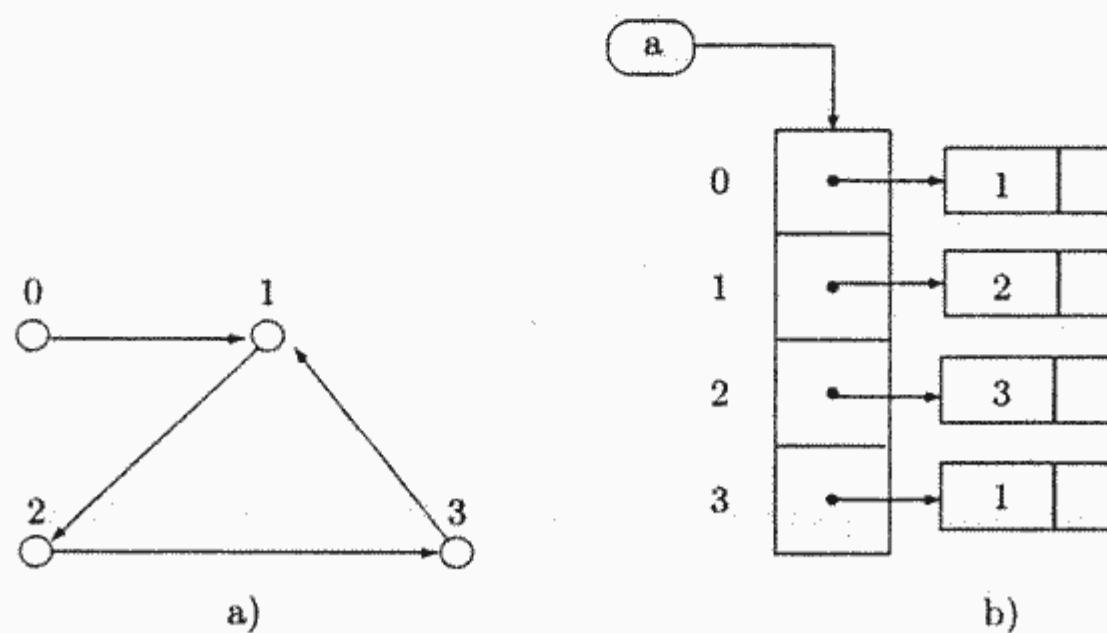


Figura 2.25: Digraf reprezentat prin liste de adiacență înăntărite

2.9.6.2 Implementarea operațiilor

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

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

InsereazăVârf. Se incrementează $D.n$ și se inițializează $D.a[n]$ cu lista vidă:

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

InsereazăArc. 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)
```

Timpul de execuție în cazul cel mai nefavorabil este $O(1)$.

ȘtergeVârf. Ștergerea vârfului i constă în eliminarea listei $D.a[i]$ din tabloul $D.a$ și parcurgerea tuturor listelor pentru a elibera nodurile care memorează i și pentru a redenumi vâfurile $j > i$ cu $j - 1$.

```
procedure stergeVarf(D, k)
    while (D.a[k].prim ≠ NULL) do
        p ← D.a[k].prim
        D.a[k].prim ← p->succ
        delete(p)
        D.a[k].ultim ← NULL
    D.n ← D.n-1
    for i ← k to D.n-1 do
        D.a[i] ← D.a[i+1]
    for i ← 0 to D.n-1 do
        elibera*(D.a[i], k) /* elibera k și redenumește vâfurile */
    end
```

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

ȘtergeArc. Ștergerea arcului (i, j) constă în eliminarea nodului care memorează valoarea j din lista $D.a[i]$.

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

Timpul de execuție în cazul cel mai nefavorabil este $O(n)$.

ListaDeAdiacențăInterioară. Determinarea listei de adiacență interioară pentru vârful i constă în selectarea celorlalte vârfuri j cu proprietatea că i apare în lista $D.a[j]$. presupunem că lista de adiacență interioară este reprezentată de o coadă C :

```

procedure listaAdInt(D, i, C)
    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

```

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

ListaDeAdiacențăExterioară. Lista de adiacență exterioară a vârfului i este $D.a[i]$.

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

- S – mulțimea vârfurilor accesibile din i_0 vizitate până în acel moment;
- SB – o submulțime 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 pointeri ($p[i] | 0 \leq i < n$) cu care ținem evidența primului vârf nevizitat încă din fiecare listă de adiacență. Mai precis, dacă $p[i] \neq \text{NULL}$ și $i \in S$, atunci $i \in SB$. Algoritmul constă în execuția 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 și 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)
    for i ← 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 2.2. Procedura `explorareDigraf` determină vârfurile accesibile din i_0 în timpul $O(\max(n, m_0))$, unde m_0 este numărul muchiilor accesibile din i_0 , și utilizează spațiul $O(\max(n, m))$.

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

S conține o submulțime de noduri accesibile din i_0 vizitate deja, $SB \subseteq S$ și mulțimea vârfurilor accesibile din i_0 nevizitate încă este inclusă în mulțimea vârfurilor accesibile din SB .

Timpul de execuție $O(\max(n, m_0))$ rezultă în ipoteza că operațiile asupra mulțimilor S și SB se realizează în timpul $O(1)$. Se observă imediat că partea de inițializare necesită $O(n)$ timp, iar bucla `while` se repetă de $O(m_0)$ ori. Dimensiunea spațiului de memorie rezultă imediat din declarații. sfdem

În funcție de structura de date utilizată pentru gestionarea mulțimii SB , obținem diferite strategii de explorare a digrafurilor.

Explorarea DFS (Depth First Search). Se obține din `explorareDigraf` prin reprezentarea mulțimii SB printr-o stivă. Presupunem că mulțimea S este reprezentată prin vectorul său caracteristic:

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

Înlocuim în procedura `explorareDigraf` operațiile corespunzătoare lui SB cu operații caracteristice stivei, iar operațiile corespunzătoare lui S cu operații caracteristice tablourilor. Obținem astfel procedura DFS care descrie strategia DFS:

```

procedure DFS(D, i0, viziteaza(), S)
    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 cerința ca operațiile peste mulțimile S și SB să se execute în timpul $O(1)$.

Exemplu. Considerăm digraful din figura 2.27. Făcând $i_0 = 0$. Calculul procedurii DFS este descris în tabelul din figura 2.26. 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 revine la 1 și vizitează al doilea din lista vârfului 1 – adică 4. Analog vârfului 2, pentru că 4 nu are vârfuri adiacente spre exterior se revine la 1 și pentru că lista lui 1 este epuizată se revine la lista lui 0. Al doilea din lista lui 0 este 2, dar acesta a fost deja vizitat și 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, însă acesta a mai fost vizitat. La fel și următorul din lista lui 3 – adică 4. Așadar, lista ordonată dată de explorarea DFS a vârfurilor accesibile din 0 este: (0, 1, 2, 4, 3).

sfex

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, 4}	(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 2.26: Un calcul al procedurii DFS

Metodei ii putem asocia un arbore, numit *arbore parțial DFS*, în modul următor: Notăm cu T mulțimea arcelor (i, j) cu proprietatea că j este vizitat prima dată parcurgând acest arc. Pentru a obține T este suficient să înlocuim în procedura DFS apelul *Viziteaza(j)* cu o instrucțiune de forma adaugă (i, j) la T . Arborele

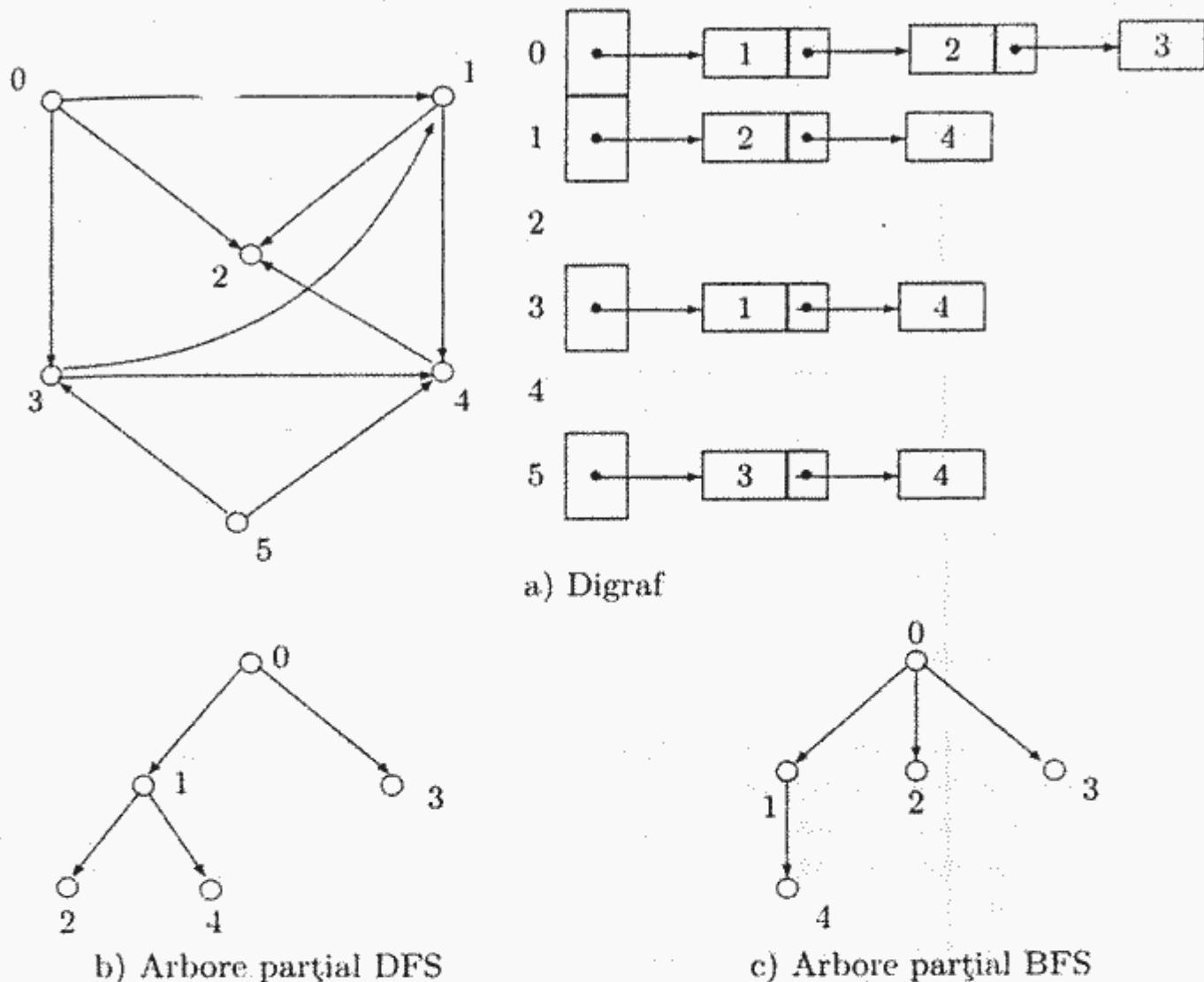


Figura 2.27: Explorarea unui digraf

parțial DFS este $S(D, i_0) = (V_0, T)$, unde V_0 este mulțimea vârfurilor accesibile din i_0 . Definiția este valabilă și pentru cazul când argumentul procedurii DFS este reprezentarea unui graf, doar cu precizarea că se consideră muchii în loc de arce. În figura 2.27.a este reprezentat arborele parțial DFS pentru digraful din figura 2.27.b și vârful 0. Arborele parțial DFS este util în multe aplicații ce necesită parcurgeri de grafuri.

Explorarea BFS (Breadth First Search). Se obține din ExplorareGraf prin reprezentarea mulțimii SB printr-o coadă.

Exercițiul 2.9.1. Să se înlocuiască în procedura ExplorareGraf operațiile corespunzătoare lui SB cu operații caracteristice cozii, iar operațiile corespunzătoare lui S cu operații caracteristice tablourilor. 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 figura 2.28. 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

<i>i</i>	<i>j</i>	<i>S</i>	<i>SB</i>
		{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 2.28: Un calcul al procedurii BFS

2 și așa mai departe. Lista ordonată dată de parcurgerea BFS este $(0, 1, 2, 3, 4)$

Ca și în cazul parcurgerii DFS, metodei i se poate ataşa un arbore, numit *arbore parțial BFS*. În figura 2.27.c este reprezentat arborele parțial BFS din exemplul de mai sus.

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

Exercițiul 2.9.2. Am văzut că, în cazul reprezentării digrafurilor prin matricea de adiacență, liste de adiacență exterioară sunt incluse în liniile matricei. Să se scrie subprogramele DFS și BFS pentru cazul când digraful este reprezentat prin matricea de adiacență.

2.9.7 Implementarea cu liste de adiacență reprezentate cu tablouri

Listele de adiacență ale digrafului $D = \langle V, A \rangle$ sunt reprezentate prin două tablouri $D.a[i]$, $i = 0, \dots, D.n$, și $D.s[j]$, $j = 0, \dots, D.m$, care satisfac proprietățile:

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

Exemplu. În figura 2.29 este dat un exemplu de digraf reprezentat prin liste de acest fel. Din vârful 0 pleacă trei arce cu destinațiile 1, 2 și 3 – ce sunt memorate în primele trei componente ale tabloului $D.s$. Din vârful 1 pleacă două arce cu destinațiile 2 și 3 – ce sunt memorate în $D.s[3]$ și $D.s[4]$. Din vârful 2 nu pleacă nici

un arc și de aceea $D.a[2] = D.a[3]$. Din vârful 3 pleacă un singur arc, memorat în $D.s[5]$. Elementul $D.a[4]$ are rolul de a marca locul unde se termină lista de adiacență exterioară a vârfului 3.

sfex

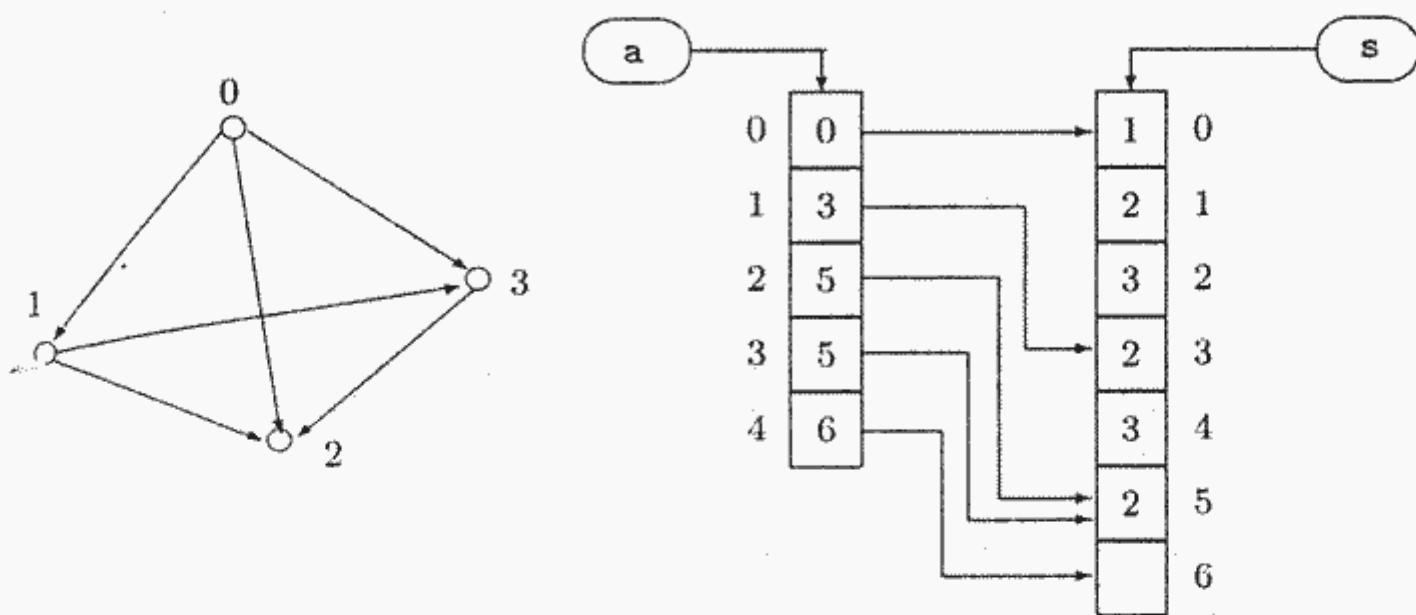


Figura 2.29: Digraf reprezentat prin liste de adiacență tablouri

Exercițiul 2.9.3. Să se descrie implementările operațiilor tipului Digraf pentru cazul când digrafurile sunt implementate cu liste de adiacență reprezentate prin tablouri.

2.9.8 Exerciții

Exercițiul 2.9.4. Dacă G este un graf atunci gradul $\delta(i)$ al unui vârf i este egal cu numărul de vârfuri adiacente cu i . Dacă D este un digraf, atunci gradul extern ($\delta^+(i)$) al unui vârf i este egal cu numărul de vârfuri adiacente cu i spre exterior, iar gradul intern ($\delta^-(i)$) este numărul de vârfuri adiacente cu i spre interior.

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

Exercițiul 2.9.5. O alegere de drumuri care unesc toate perechile de vârfuri conectabile i, j într-un digraf poate fi memorată într-o matrice p cu semnificația: $p[i, j]$ este primul vârf întâlnit după i pe drumul de la i la j .

1. Să se modifice subprogramul `detInchRef1Tranz` astfel încât să determine și o alegere de drumuri care unesc vâfurile conectabile.
2. Să se scrie un subprogram care, având date matricea drumurilor și două vârfuri i, j , determină un drumul de la i la j .

Exercițiul 2.9.6. Un digraf D poate fi reprezentat și prin *listele de adiacență interioară*, unde $D.a[i]$ este lista vâfurilor-sursă ale arcelor care sosesc în i . Să se scrie procedurile care implementează operațiile tipului Digraf pentru cazul în care digrafurile sunt reprezentate prin listele de adiacență interioară.

Exercițiul 2.9.7. Să se scrie o procedură `Conex(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de D este conex sau nu.

Exercițiul 2.9.8. Să se scrie o procedură `Arbore(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de D este arbore sau nu.

Indicație. Se poate utiliza faptul că un graf cu n vârfuri este arbore dacă și numai dacă este conex și are $n - 1$ muchii [Cro92].

Exercițiul 2.9.9. Să se proiecteze o structură de date pentru reprezentarea componentelor conexe ale unui graf și să se scrie o procedură care, având la intrare reprezentarea unui graf, construiește componentele conexe ale acestuia.

Exercițiul 2.9.10. Fie $D = (V, A)$ un digraf cu n vârfuri. Un vârf i se numește groapă (sink) dacă pentru orice alt vârf $j \neq i$ există un arc $(j, i) \in A$ și nu există arc de forma (i, j) . Să se scrie o funcție `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 timpul de execuție $O(n)$. Pot exista mai multe gropi?

Exercițiul 2.9.11. 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 sir $a \in A^*$ dat, determină dacă există un drum în G etichetat cu a .

Exercițiul 2.9.12. Multimea *grafurilor serie-paralel* (G, s, t) , unde G este un multi-graf (graf cu muchii multiple), iar s și t sunt vârfuri în G numite *sursă*, respectiv *destinație*, este definită recursiv astfel:

- Orice muchie $G = \{u, v\}$ definește două grafuri serie-paralel: (G, u, v) și (G, v, u) .
- Dacă (G_1, s_1, t_1) și (G_2, s_2, t_2) sunt două grafuri serie-paralel, atunci:
 - *componerea serie* $G_1 G_2 = (G, s_1, t_2)$, obținută prin reuniunea disjunctă a grafurilor G_1 și G_2 în care vâfurile s_2 și t_1 sunt identificate, este graf serie-paralel;
 - *componerea paralelă* $G_1 \| G_2 = (G, s_1 = s_2, t_1 = t_2)$, obținută prin reuniunea disjunctă a grafurilor G_1 și G_2 în care sursele s_1 și s_2 și respectiv destinațiile t_1 și t_2 sunt identificate, este graf serie-paralel.

Definiția este sugerată grafic în figura 2.30.

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

Exercițiul 2.9.13. Să se scrie un program care, pentru un graf $G = (V, E)$ și $i_0 \in V$ date, enumeră toate drumurile maximale care pleacă din i_0 .

Exercițiul 2.9.14. Se consideră problema din exercițiul 2.9.13. Presupunem că muchiile grafului G sunt etichetate cu numere întregi. Să se modifice programul de la 2.9.13 astfel încât drumurile să fie enumerate în ordine lexicografică.

Exercițiul 2.9.15. Se numește triunghi într-un graf G un ciclu de lungime 3. Să se scrie un program care să determine dacă graful G conține un triunghi.

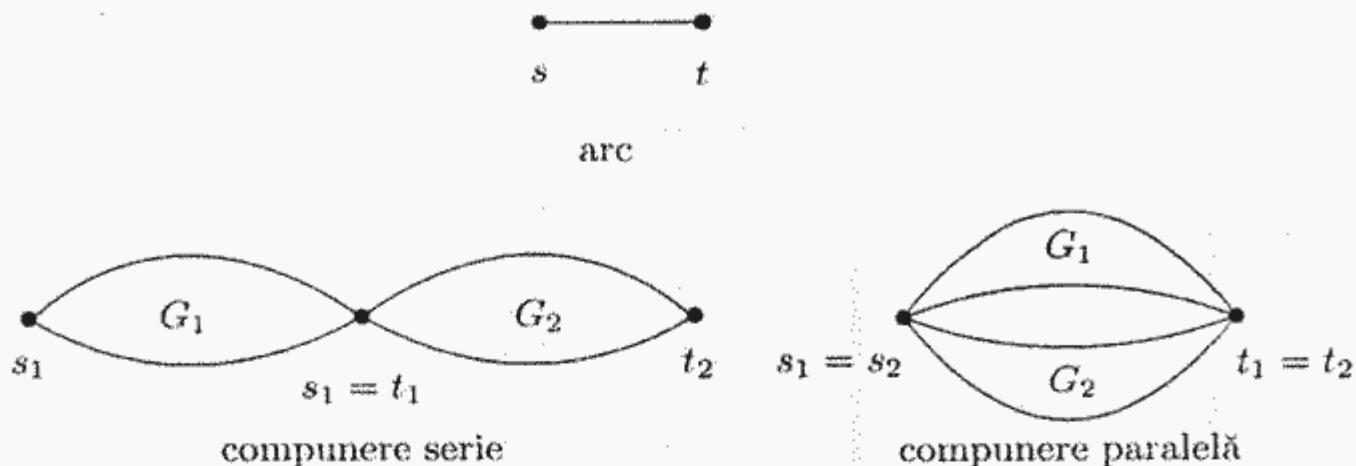


Figura 2.30: Grafuri serie-paralel

2.10 Union-find

Grafurile pot reprezenta colecții de mulțimi disjuncte într-un mod foarte natural: vârfurile corespund obiectelor, iar o muchie are semnificația că extremitățile sale sunt în același mulțime. Această reprezentare permite rezolvarea eficientă a multor probleme legate de mulțimi. De exemplu, a răspunde la întrebarea „Cărei mulțimi îi aparține obiectul x ” (operația *find*) presupune de fapt a preciza cărei componente conexe îi aparține vârful x . Există și un inconvenient al reprezentării; aceeași mulțime poate fi reprezentată prin mai multe grafuri conexe. De aceea, de exemplu, informațiile oferite de parcurgerile sistematice are o utilitate mai redusă.

Un caz aparte se obține când se dă un aspect dinamic problemei: la diferite momente, două mulțimi se pot reuni într-o singură (operația *union*). Formularea unei cereri de acest tip este de formă: „reunește mulțimea la care aparține i cu mulțimea la care aparține j ”. Aceasta operație se poate realiza foarte simplu prin adăugarea unei muchii care să unească un vârf din componenta conexă corespunzătoare primei mulțimi cu un vârf din componenta corespunzătoare celei de-a doua mulțimi. Rămâne de stabilit care vârfuri candidează la momentul respectiv pentru unire printr-o muchie. Această alegere va trebui să permită construirea de algoritmi eficienți pentru ambele operații: *union* și *find*. Având în vedere că topologia componentelor conexe nu este importantă, vom alege structura de tip arbore cu rădăcină pentru reprezentarea unei mulțimi. Colecția de mulțimi este reprezentată de o colecție de arbori (*o pădure*). Pentru reprezentarea unei păduri vom utiliza un tablou *parinte* cu semnificația că *parinte*[i] este predecesorul imediat (părintele) vârfului i . Mulțimea univers, peste care se consideră colecția de mulțimi, este aplicată prin funcția *index* într-un interval de numere întregi $[0, n - 1]$. Astfel, o colecție C va fi reprezentată de numărul întreg n și tabloul *parinte*. Dacă i este rădăcina unui arbore, atunci *parinte*[i] = -1 . Un exemplu de colecție reprezentată astfel este arătat în figura 2.31a.

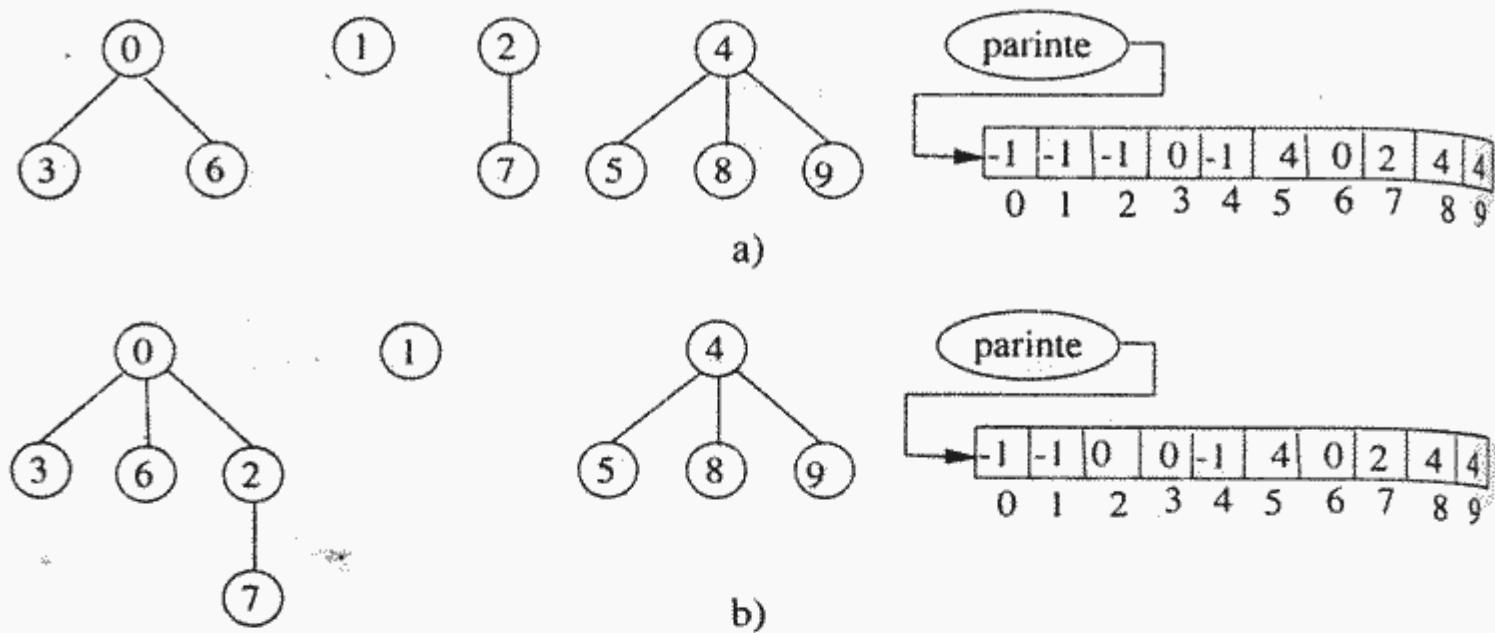


Figura 2.31: Structuri union-find

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

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

Determinarea mulțimii căreia îi aparține i este echivalentă cu determinarea rădăcinii:

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

Reuniunea a două mulțimi înseamnă construirea unui arc de la rădăcina unui arbore la rădăcina celuilalt:

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

Executând funcția $\text{union}(C, 6, 7)$ pentru colecția din figura 2.31.a obținem structura din figura 2.31.b.

Prin execuția repetată a operației union se pot obține arbori dezechilibrați, în care determinarea rădăcinii pentru anumite vârfuri să dureze mult. Structura ar putea fi îmbunătățită, dacă s-ar putea menține o formă cât mai aplatizată a arborilor, adică să nu existe noduri aflate la distanțe mari de rădăcină. Aceasta se poate realiza prin memorarea în rădăcini a numărului de vârfuri din arbore (greutatea

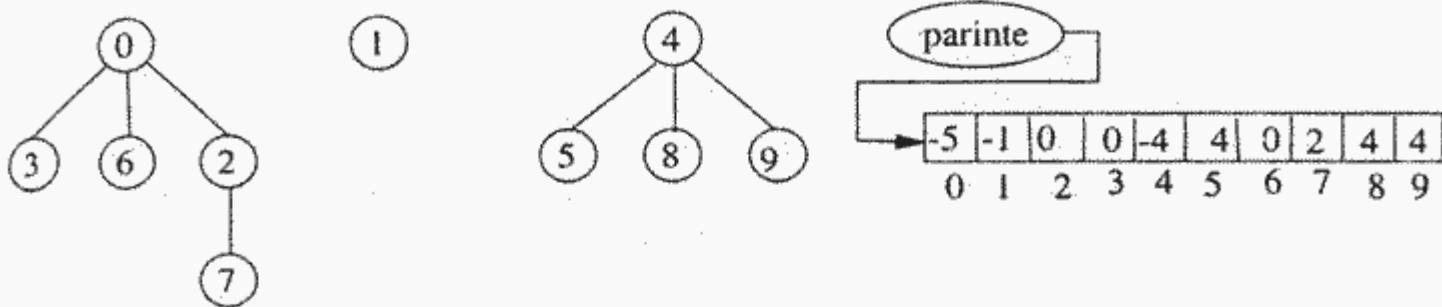


Figura 2.32: Structură “union-find” ponderată

arborelui). Aceasta va fi memorată cu semnul minus pentru a distinge rădăcinile de celelalte noduri. În figura 2.32 este reprezentată o asemenea structură. În plus, procedura `union` va realiza mai întâi o „aplatizare” a arborilor și apoi va construi un arc de la rădăcina arborelui cu greutatea mai mare la rădăcina celui cu greutatea mai mică.

```

union(C, i, j)
    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 soluție alternativă la cea de mai sus este să memorăm înălțimea arborelui în loc de greutate. Operația `union` va ancorea arboarele cu înălțime mai mică de cel cu înălțimea mai mare.

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

Teorema 2.3. *O secvență de m operații singleton, `union` și `find` din care n sunt operații 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 $\log^{(k)} n \leq 1$ și $\log^{(k)}$ desemnează funcția log compusă cu ea însăși de k ori: $\log^{(0)} n = n$, $\log^{(i+1)} n = \log \log^{(i)} n$.*

Demonstrația acestei teoreme se reduce la demonstrarea teoremei 2.4.

Înținând cont de componența operației union, aceasta poate fi înlocuită cu două operații find și o operație de legare (link). Operația link se referă la actul legării printr-un arc a celor două rădăcini ale arborilor care reprezintă mulțimile ce se reunesc. Lema următoare arată că această înlocuire nu modifică comportarea asymptotică a unei secvențe de operații singleton, union și find.

Lema 2.3. *Dată find o secvență S' de m' operații singleton, union și find, se convertește secvența S' într-o secvență S de m operații singleton, link și find prin înlocuirea operației union cu două operații find și o operație link. Dacă secvența S are timpul de execuție $O(m \log^* n)$, atunci secvența S' are timpul de execuție $O(m' \log^* n)$.*

Demonstrație. O operație union din S' corespunde la trei operații din S (două operații find și o operație link). Rezultă $m' \leq m \leq 3m'$. Aceasta înseamnă că $m = O(m')$. În consecință $O(m \log^* n) = O(m \log^* n)$. sfdem

Rămâne să demonstrăm că secvența S are timpul de execuție $O(m \log^* n)$.

Teorema 2.4. *O secvență S de m operații singleton, link și find, din care n operații sunt singleton, pot fi executate în cazul cel mai nefavorabil în timpul $O(m \log^* n)$.*

Demonstrația acestei teoreme are la baza analiză amortizată și poate fi găsită în [CLR93].

2.11 Analiza amortizată

Analiza cazului cel mai nefavorabil poate fi abordată și altfel decât în maniera prezentată în capitolul 1. În locul evaluării izolate a timpului de execuție în cazul cel mai nefavorabil al fiecărei operații asupra unei structuri de date, uneori este benefic să fie evaluat timpul de execuție al unei secvențe de operații sau al tuturor operațiilor asupra structurii de date. Analiza amortizată definește acest proces [CLR00]. Se demonstrează astfel că operațiile compuse din alte operații au cost mediu mic, deși în cazul cel mai nefavorabil, costul este mult mai mare. Aceasta se justifică prin faptul că între operațiile care compun o operație multiplă există interdependențe ce trebuie luate în calcul atunci când se încearcă determinarea unui majorant pentru timpul de execuție al ansamblului (operația multiplă). Analiza amortizată diferă de analiza cazului mediu, deoarece nu sunt implicate probabilități. Analiza amortizată garantează performanța medie a unei operații în cazul cel mai nefavorabil, adică valoarea evaluată a timpului de execuție pentru o secvență de operații asupra unei structuri de date nu este mai mică decât valoarea reală a acestuia.

2.11.1 Metoda agregării

Prin această metodă de analiză amortizată se determină costul mediu al unei operații ca fiind $\frac{T(n)}{n}$, unde $T(n)$ este timpul de execuție în cazul cel mai nefavorabil al unei secvențe de n operații. Acest cost va fi numit costul amortizat și va fi aplicat la fiecare operație din secvență.

Pentru exemplificare vom folosi operația de extragere multiplă dintr-o stivă. La operațiile de bază cu stiva vom adăuga operația MultiPop definită astfel:

MultiPop. Elimină ultimele k elemente introduse în stivă.

- Intrare:** – o stivă S , un număr întreg $k > 0$;
Ieșire: – stiva S din care s-au eliminat ultimele k elemente introduse, dacă S are cel puțin $k + 1$ elemente,
 – stiva vidă, dacă S stiva conține cel mult k elemente.

Dacă stiva este implementată cu structuri dinamice simplu înlăncuite, operația MultiPop este implementată de subprogramul $\text{MultiPop}(S, k)$ de mai jos:

```
procedure MultiPop(S, k)
    while (S ≠ NULL and k ≠ 0) do
        pop(S)
        k ← k-1
    end
```

Ne propunem să determinăm timpul de execuție pentru $\text{MultiPop}(S, k)$ în cazul unei stive cu s elemente. Numărul de iterații ale buclei while este $\min(s, k)$. În fiecare iterație se execută o operație $\text{pop}(S)$ și o atribuire. Rezultă că timpul de execuție pentru $\text{MultiPop}(S, k)$ este $O(\min(s, k))$.

Să analizăm acum o secvență de n operații Push, Pop și MultiPop asupra unei stive S inițial vide. Pentru simplitatea limbajului, în continuare vom identifica operațiile Push, Pop și MultiPop cu subprogramele $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$. Cazul cel mai nefavorabil pentru $\text{MultiPop}(S, k)$ este dat de stiva cu n elemente și $k \geq n$. Numărul maxim de elemente ce pot compune stiva S este n , deoarece numărul maxim de operații $\text{Push}(S)$ este n . Timpul de execuție al unei operații $\text{Push}(S)$ sau $\text{Pop}(S)$ este $O(1)$. Rezultă că, în cazul cel mai nefavorabil, timpul de execuție al operației $\text{MultiPop}(S, k)$ este $O(n)$. În consecință, în acest caz, timpul de execuție a secvenței de n operații $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$ este $O(n^2)$. Deși analiza este corectă, adunând timpii de execuție în cazul cel mai nefavorabil pentru fiecare operație, se obține un majorant al timpului de execuție mult mai mare decât marginea superioară.

Prin analiza amortizată se poate obține un majorant mai bun. Astfel, deși o singură operație $\text{MultiPop}(S, k)$ este costisitoare, orice secvență de n operații $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$ asupra unei stive inițial vide are timpul de execuție cel mult $O(n)$. Aceasta se explică prin faptul că numărul elementelor eliminate din stivă nu poate depăși numărul elementelor introduse în stivă. Menționăm că, inițial, stiva este vidă.

Numărul operațiilor Pop(S), inclusiv al celor apelate din $\text{MultiPop}(S, k)$, este mai mic sau egal cu numărul operațiilor Push(S). Rezultă pentru secvența celor n operații $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$ un timp de execuție de $O(n)$. Costul amortizat pentru o operație va fi $\frac{O(n)}{n} = O(1)$. Suma costurilor amortizate (costul mediu pentru cazul cel mai nefavorabil) este evident egală cu majorantul timpului de execuție obținut prin metoda agregării. Astfel, prin utilizarea costului mediu este garantat faptul că valoarea evaluată a timpului de execuție pentru o secvență de operații asupra unei structuri de date nu este mai mică decât valoarea reală a acestuia.

Am arătat astfel că pentru cazul cel mai nefavorabil, prin metoda agregării, se poate obține un majorant mai bun pentru timpul de execuție decât prin analiza

separată a fiecărei operații.

2.11.2 Metoda conturilor

Prin metoda conturilor, anumite operații sunt supraevaluate, iar altele sunt subevaluate. Valoarea astfel asociată unei operații va fi numită cost amortizat. Dacă acest cost depășește costul real, diferența este memorată drept credit în structura de date. Acest credit va fi folosit mai târziu de operațiile subevaluate, adică de cele cu costul amortizat inferior costului real. Diferența față de metoda agregării constă în faptul că acum operațiile nu mai au același cost amortizat.

Costul amortizat trebuie ales cu atenție. Costul amortizat al unei secvențe de operații trebuie să fie un majorant al costului real al secvenței (timpul real de execuție). Mai mult, aceasta trebuie să se întâmple pentru orice secvență de operații. Rezultă că structura de date trebuie să memoreze în orice moment un credit pozitiv, ceea ce asigură faptul că valoarea costului amortizat total este mai mare decât valoarea costului real total.

Pentru a exemplifica metoda conturilor vom folosi același exemplu ca în cazul metodei agregării. Timpul de execuție al unei operații $\text{Push}(S)$ sau $\text{Pop}(S)$ este $O(1)$. Vom considera acest timp ca unitate de cost. Astfel, costul real pentru fiecare din aceste operații va fi 1. Timpul de execuție pentru $\text{MultiPop}(S, k)$ este $O(\min(s, k))$. Dacă folosim ca unitate de cost timpul de execuție al unei operații $\text{Push}(S)$ sau $\text{Pop}(S)$, atunci costul lui $\text{MultiPop}(S, k)$ este $\min(s, k)$.

Pentru a asigura un credit permanent pozitiv, vom asocia următoarele costuri amortizate: 2 pentru $\text{Push}(S)$, 0 pentru $\text{Pop}(S)$ și 0 pentru $\text{MultiPop}(S, k)$. Se observă că, pentru $\text{MultiPop}(S, k)$, costul este o constantă (0), deși costul real depinde de dimensiunea stivei S și de valoarea k .

Costul real al operației $\text{Push}(S)$ este 1, în timp ce costul amortizat este 2. Astfel, operația $\text{Push}(S)$ aplicată unui element va consuma doar $1/2$ din costul amortizat asociat acesteia. În consecință, fiecare element din stivă va dispune de un credit 1, care va acoperi costul eliminării acestuia din stivă ($\text{Pop}(S)$). Operația $\text{MultiPop}(S, k)$ nu necesită cost amortizat strict pozitiv, deoarece este compusă numai din operații $\text{Pop}(S)$ creditate în momentul aplicării operației $\text{Push}(S)$. Rezultă că orice secvență de operații aplicate unei stive va avea un cost amortizat pozitiv.

Costul amortizat al unei secvențe de operații $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$ va fi întotdeauna un majorant al costului real. Aceasta este justificată de faptul că elementele aflate la un moment dat în stivă dispun de credit neconsumat, iar operațiile efectuate până la acel moment au costurile reale acoperite. Costul amortizat este $O(n)$, iar costul real este cel puțin n . Rezultă un cost real de $O(n)$.

2.11.3 Metoda potențialului

Metoda potențialului diferă de metoda conturilor prin faptul că efortul de calcul estimat a fi consumat ulterior nu mai este memorat ca un credit al unui obiect, ci ca energie potențială a structurii de date, energie care va fi eliberată în momentul execuției anumitor operații asupra structurii. Metoda poate fi descrisă astfel:

Se presupune că structura inițială este S_0 și asupra acesteia urmează a se efectua n operații. Se notează cu c_i costul real al operației cu numărul de ordine i și cu S_i structura rezultată din aplicarea acestei operații asupra structurii S_{i-1} . Peste mulțimea acestor structuri de date se definește o funcție potențial care asociază fiecărei structuri S_i un număr real $\Pi(S_i)$. Costul amortizat al operației cu numărul de ordine i se definește prin relația:

$$\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1})$$

Costul amortizat al unei operații este costul real plus diferența de potențial datorată execuției operației. Din relația de mai sus rezultă că pentru n operații costul este:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Pi(S_i) - \Pi(S_{i-1})) = \sum_{i=1}^n c_i + \Pi(S_n) - \Pi(S_0)$$

Dacă $S_n \geq S_0$, atunci costul total amortizat este un majorant al costului real. Din nefericire, nu se cunoaște întotdeauna numărul operațiilor ce urmează a fi efectuate asupra unei structuri. Din acest motiv, pentru ca proprietatea de majorant să fie asigurată, este nevoie ca $(\forall i) S_i \geq S_0$.

Intuitiv, dacă $S_i - S_{i-1} > 0$, atunci operația cu numărul de ordine i realizează o încărcare a structurii cu o cantitate de energie potențială, iar dacă $S_i - S_{i-1} < 0$, atunci operația cu numărul de ordine i realizează o descărcare din structura de date a unei cantități de energie potențială. Costul amortizat depinde de modul în care este definită funcția potențial. Este de dorit ca funcția potențial să confere costului amortizat calitatea de margine superioară a costului real.

Vom folosi în continuare exemplul stivei pentru a explica metoda potențialului. Vom defini funcția potențial astfel:

$$\Pi : \{S \mid S = \text{stiva}\} \rightarrow \mathbb{Z}, \quad \Pi(S) = \text{numărul elementelor din stivă.}$$

Pentru stiva S_0 vom avea $\Pi(S_0) = 0$. Deoarece numărul elementelor dintr-o stivă nu este negativ, stiva S_i rezultată din aplicarea operației cu numărul de ordine i va avea întotdeauna potențial pozitiv, adică $(\forall i) S_i \geq 0 = S_0$. Astfel, costul amortizat pentru n operații este întotdeauna un majorant al costului real.

Să calculăm acum costul amortizat pentru fiecare dintre operațiile $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$.

Dacă operația cu numărul de ordine i este $\text{Push}(S)$, iar stiva conține s elemente, atunci $\Pi(S_i) - \Pi(S_{i-1}) = (s+1) - s = 1$. Costul amortizat va fi $\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1}) = 1 + 1 = 2$.

Să presupunem acum că operația cu numărul de ordine i este $\text{Pop}(S)$. Diferența de potențial în acest caz va fi $\Pi(S_i) - \Pi(S_{i-1}) = (s-1) - s = -1$, dacă stiva S nu este vidă. Costul amortizat va avea valoare 0, deoarece costul real al operației $\text{Pop}(S)$ este 1. Dacă stiva S este vidă $\Pi(S_i) - \Pi(S_{i-1}) = 0 - 0 = 0$, costul amortizat al operației $\text{Pop}(S)$ va fi tot 0, deoarece costul real în acest caz este 0.

Dacă operația cu numărul de ordine i este $\text{MultiPop}(S, k)$, atunci diferența de potențial va fi $\Pi(S_i) - \Pi(S_{i-1}) = (s - k') - s' = -k'$, unde $k' = \min(k, s)$

(costul real al operației). Astfel, costul amortizat al operației $\text{MultiPop}(S, k)$ va fi $\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1}) = k' - k' = 0$.

Rezultă pentru fiecare dintre operațiile $\text{Push}(S)$, $\text{Pop}(S)$ și $\text{MultiPop}(S, k)$ un cost de $O(1)$, deci un cost amortizat de $O(n)$ pentru n operații. Din definiția funcției potențial rezultă că $(\forall i) S_i - S_0 \geq 0$, ceea ce implică pentru costul amortizat al celor n operații proprietatea de majorant pentru costul real. În cazul cel mai nefavorabil, costul celor n operații este astfel $O(n)$.

2.12 Referințe bibliografice

Tipurile de date de nivel înalt sunt analizate intens în literatura de specialitate [CLR93, CLR00, Sah98, Baa78, BG00, Mel84a, AHU74, HS84, MB00, Wei92, LG86, CB02]. Analiza amortizată este prezentată în detaliu în [CLR93, CLR00, Hei96].

Capitolul 3

Derecursivare

În acest capitol prezentăm câteva strategii prin care soluțiile recursive pot fi descrise prin programe nerecursive. Dintre toate motivele care impun necesitatea cunoașterii unei asemenea strategii, următoarele două ni se par a fi cele mai importante:

- Uneori, eficiența algoritmilor descriși prin programe nerecursive este mult mai bună decât a celor descriși recursiv. Vom vedea cazuri când descrierea recursivă are complexitate exponențială, iar descrierea nerecursivă complexitate liniară.
- Execuția subprogramelor recursive presupune gestionarea (cel puțin a) unei stive. Prin derecursivare, programatorul își gestionează singur această stivă (în cazul programelor recursive aceasta este gestionată de sistem) și poate astfel exploata informația conținută în stivă pentru a crește calitatea sau gradul de utilizare a programului.

Tehnicile descrise în această secțiune se referă numai la recursia directă.

3.1 Parcurgerea în inordine a arborilor binari

Sunt multe cazuri în activitatea de programare când prelucrarea informației dintr-un nod al unui arbore binar este însotită de o procesare a informației aflate pe drumul de la rădăcina arborelui la nod. Vom vedea că, prin derecursivare, acest drum este memorat chiar de stiva cu valorile parametrilor.

Reamintim mai întâi varianta recursivă a algoritmului de parcurgere în inordine a arborilor binari:

```
procedure inordine(v)
  1: if v ≠ NULL
  2:   then inordine(v->stg)
  3:       viziteaza(v)
  4:       inordine(v->drp)
  end
```

Tehnica utilizată este inspirată din descrierea semanticii operaționale a instrucțiunii de apelare a unui subprogram. Vom descrie un program nerecursiv care simulează

activitatea celui recursiv. Programul cu cea mai simplă structură este obținut în urma unui proces de rafinare.

Varianta 1. Prima variantă utilizează cele două stive de la descrierea semanticii operaționale a instrucțiunii de apelare a unui subprogram:

- o stivă pentru parametrii în care se vor memora referințe de noduri, notată **spar**;
- o stivă de protocol în care sunt memorate adresele de întoarcere în subprogram, notată **sadr**. Adresele sunt identificate prin intermediul etichetelor.

În general, fiecare apel recursiv este simulațat de o secvență de instrucțiuni care realizează:

- salvarea stării curente a subprogramului: Aceasta constă din:
 - introducerea valorilor (în cazul apelului prin valoare) sau adreselor (în cazul apelului prin referință) parametrilor din instrucțiunea de apelare în stiva parametrilor; dacă există variabile locale, atunci se salvează în stivă și valorile acestora,
 - introducerea adresei primei instrucțiuni ce urmează apelului în stiva adreselor de return;
- actualizarea valorile parametrilor pentru noul apel recursiv;
- saltul la prima instrucțiune din subprogram.

Astfel, primul apel recursiv de la eticheta 2 se înlocuiește cu:

```

push(spar, v) //introduce radacina subarborelui in
                //          stiva de parametri
push(sadr, 3) //introduce adresa de return in stiva
                //          de protocol
v ← v->lsg; //inainteza in subarborele aflat la
                //          stanga
goto 1        //se reia calculul procedurii pentru
                //          subarborele stang
  
```

Analog, al doilea apel recursiv va fi înlocuit cu:

```

push(spar, v)
push(sadr, 5)
v ← v->lsp
goto 1
  
```

Simularea terminării execuției unui apel necesită următoarele operații:

- Se restabilește starea subprogramului de dinaintea ultimului apel recursiv. Aici, aceasta constă din extragerea din stiva de parametri a valorilor corespunzând rădăcinii subarborelui care tocmai s-a vizitat. În general, sunt extrași toți parametrii corespunzători apelului care tocmai s-a terminat și se actualizează parametrii apelului suspendat (ce urmează a se relua). Dacă există variabile locale, atunci și acestea vor fi supuse unui proces de extragere și actualizare.
- Se reia calculul cu instrucțiunea dată de vârful stivei de protocol; această adresă va fi eliminată din stivă.

Secvența de instrucțiuni care realizează aceste operații este:

```

if not esteVida(spar)
    then v ← top(spar); //radacina arborelui ce s-a terminat
        pop(spar)      //de vizitat este eliminata din stiva
        adret ← top(sadr); //adresa de return de la care se
                            //continua calculul
    pop(sadr);
    switch (adret)      //salt la instructiunea data de
                        //adresa de return
        case 3
            goto 3
        case 5
            goto 5

```

Asamblând secvențele de mai sus obținem următorul subprogram nerecursiv:

```

procedure inordNerec1(t)
// initializari
spar ← stivaVida()
sadr ← stivaVida()
v ← t
1:if v ≠ nil
    then // simuleaza primul apel recursiv
2:    push(spar, v)
        push(sadr, 3)
        v ← v->lstg;
        goto 1
3:viziteaza(v)
    //simuleaza al doilea apel recursiv
4:push(spar,v)
    push(sadr,5)
    v ← v->ladrp
    goto 1
    //simuleaza intoarcerea
5:if not esteVida(spar)
    then v ← Top(spar)
        pop(spar)
        adret ← top(sadr)
        pop(sadr)
    switch (adret)
        case 3
            goto 3
        case 5
            goto 5
end

```

Varianta 2. Se obține prin structurarea primei variante. Un vârf v , după ce a fost vizitat, este introdus în stiva parametrilor prin instrucțiunea de la eticheta 4. Când se execută instrucțiunea de la eticheta 5 și în vârful stivei se află v , adresa de return fiind 5, v este eliminat din stivă fără ca asupra lui să se efectueze vreo operatie. De aici rezultă că introducerea sa în stivă este inutilă. Astfel că putem pune secvențele de la etichetele 3 și 4 la un loc direct în secvența de la 5.

```

procedure inordNerec2(t)
    spar ← stivaVida()
    sadr ← stivaVida()
    v ← t
    repeat
        while v ≠ nil do
            push(spar, v)
            push(sadr, 3)
            v ← v->lstg
            if not esteVida(spar)
                then v ← top(spar)
                    pop(spar)
                    adret ← top(sadr)
                    pop(sadr)
                    switch (adret)
                        case 3
                            viziteaza(v)
                            v ← v->lgrp
            until esteVida(spar) and (v = NULL)
end

```

Varianta 3. Se obține din varianta 2 prin eliminarea stivei de protocol **sadr**, a cărei prezență este acum inutilă.

```

procedure inordNerec3(t)
    spar ← stivaVida()
    v ← t
    repeat
        while v ≠ NULL do
            push(spar, v)
            v ← v->lstg
            if (not esteVida(spar))
                then v ← top(spar)
                    pop(spar)
                    viziteaza(v)
                    v ← v->lgrp
            until esteVida(spar) and (v = NULL)
end

```

3.2 Recursia liniară

O schemă de recursie liniară este o definiție de forma:

$$f(x) = \text{if } p(x) \text{ then } g(x) \text{ else } \varphi(k(x), f(h(x))) \text{ fi}$$

Un exemplu de funcție definită prin recursie liniară este factorialul: luăm $p(x) \equiv (x \leq 1)$, $k(x) = x$, $h(x) = x - 1$ și $\varphi(x, y) = x \cdot y$. Un calcul sintactic al unei funcții definite prin recursie liniară arată ca în figura 3.1. Acest calcul este finit dacă și numai dacă există un n natural cu proprietatea $p(h^n(a)) = \text{true}$; în acest caz, calculul se oprește la cel mai mic număr natural nenul n cu această proprietate.

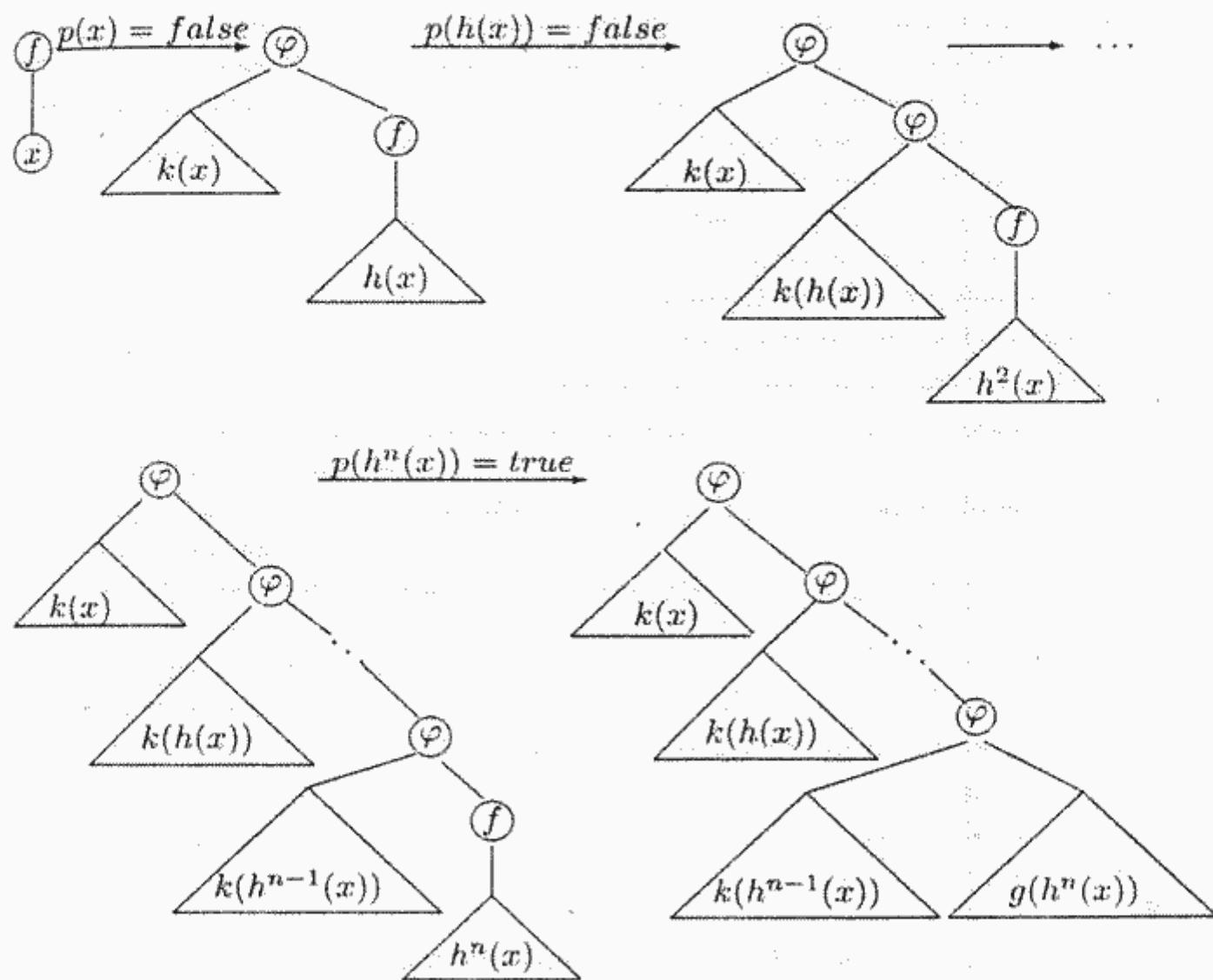


Figura 3.1: Calculul sintactic al recursiei liniare

Programul nerecursiv, care simulează calculul apelurilor recursive, se obține în nodul următor:

- se utilizează o stivă în care se vor memora valorile parametrilor din apelurile recursive: a , $h(a)$, $h^2(a)$, Stiva cu adresele de return nu mai este necesară. Se observă că prin introducerea parametrilor în stivă se simulează, de fapt, construcția arborelui final din calculul sintactic;

- după terminarea unui apel, cu ajutorul valorilor din vârful stivei, se evaluează funcția φ . Procedeul continuă până când stiva devine vidă (după utilizarea sa la calculul funcției φ , elementul corespunzător este eliminat din stivă). În reprezentarea arborescentă, această fază simulează evaluarea arborelui final din calculul sintactic.

Strategia de mai sus este descrisă de următorul program nerecursiv:

```

function f(x)
    // initializare
    u ← x
    s ← stivaVida()
    // constructia arborelui
    while not p(u) do
        push(S, u)
        u ← h(u)
    // evaluarea arborelui
    v ← g(u)
    while not EsteStivaVida(S) do
        u ← top(S); pop(S) // extrage parametrul din stiva
        w ← k(u)           // evaluarea functiei k
        v ← phi(w,v)       // evaluarea functiei phi
    return v
end

```

Rolul variabilelor u, v, w este sugerat grafic în figura 3.2.

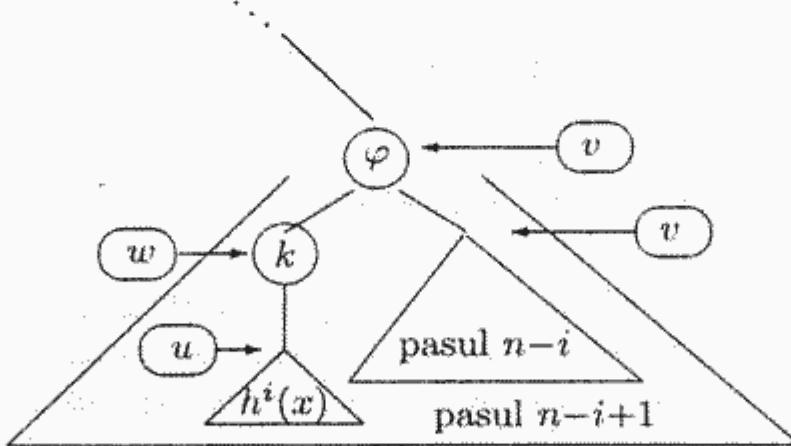


Figura 3.2: Rolul variabilelor u, v, w

Există cazuri când programul de mai sus poate fi simplificat foarte mult. Iată două dintre acestea:

- În cazul când h admite inversă, se poate renunța la utilizarea stivei. Se utilizează un contor și introducerea în stivă se înlocuiește cu incrementarea contorului, iar extragerea din stivă este înlocuită cu instrucțiuni care realizează $u \leftarrow h^{-1}(u)$ și decrementarea contorului. Rolul primei faze, corespunzătoare construcției arborelui și a contorului, este de a-l determina pe cel mai mic n cu $p(h^n(x)) = \text{true}$. De exemplu, pentru factorial, avem $h^{-1}(x) = x + 1$.

2. Dacă există o funcție care determină direct cel mai mic n cu proprietatea $p(h^n(x)) = true$, atunci instrucțiunea `while` poate fi înlocuită cu o instrucțiune `for`. De asemenea, faza corespunzătoare construcției arborelui nu mai este necesară. În cazul funcției factorial, avem $n = x - 1$.

Prezentăm în continuare cele trei variante care conduc la forma simplificată a programului nerecursiv ce calculează factorialul.

Varianta 1. Corespunde schemei generale cu utilizarea stivei.

```
function fact(x)
    //initializare
    u ← x
    s ← stivaVi.'a()
    //constructia arborelui
    while (u > 1) do
        push(s, u)
        u ← u-1
    //evaluarea arborelui
    v ← 1
    while not esteVida(s) do
        u ← top(s)
        pop(s)
        w ← u
        v ← w*v
    return v
end
```

Varianta 2. Această variantă este obținută din prima prin implementarea stivei cu ajutorul unui simplu contor.

```
function fact(x)
    //initializare
    u ← x
    contor ← 0
    //determinarea lui n
    u ← x
    if (u = 0) then u ← 1
    while (u > 1) do
        contor ← contor + 1
        u ← u-1
    v ← 1
    while (contor > 0) do
        contor ← contor - 1
        u ← u+1
        w ← u
        v ← w * v
    return v
```

```
end
```

Varianta 3. Deoarece se cunoaște $n = x-1$, prima fază corespunzătoare construcției arborelui poate fi eliminată.

```
function fact(x)
    //initializare
    v ← 1
    //proces iterativ
    for u ← x downto 1 do
        v ← u * v
    return v
end
```

3.3 Recursia în cascadă

Schema de recursie în cascadă are următoarea definiție:

$$f(x) = \text{if } p(x) \text{ then } g(x) \text{ else } \varphi(f(k(x)), f(h(x))) \text{ fi}$$

Un exemplu de funcție definită prin recursie în cascadă îl constituie funcția Fibonacci. Considerăm mai întâi subprogramul recursiv care calculează o funcție definită prin recursie în cascadă:

```
function f(x)
1: if p(x)
2:     then f ← g(x)
3:     else
4:         u ← k(x)
5:         v1 ← f(u)
6:         u ← h(x)
7:         v2 ← f(u)
8:         return phi(v1, v2)
end;
```

Procedura de mai sus utilizează mai multe variabile locale decât ar fi necesare, dar ele vor face prezentarea mai explicită. Putem descrie un program nerecursiv, care îl simulează pe cel de mai sus, procedând ca la parcurgerea în inordine a arborilor binari. Pentru memorarea parametrilor utilizăm două stive: o stivă *spar* care memorează argumentele pentru apelurile funcției *f* și o stivă *srez*, unde sunt memorate rezultatele evaluărilor parțiale ale funcției *f*. Stiva *spar* poate fi utilizată și pentru salvarea eventualelor variabile locale.

```
function f(x)
    //initializari
    spar ← stivaVida()
    sadr ← stivaVida()
    srez ← stivaVida()
    u ← x
```

```

//proces iterativ
1: if (p(u))
    then v ← g(u)
        push(srez, v)
    else //simuleaza primul apel
        push(spar, u)
        push(sadr, 6)
        u ← k(u)
        goto 1
6:   //simuleaza al doilea apel
    push(spar, u)
    push(sadr, 8)
    u ← h(u)
    goto 1
8:   //evalueaza phi
    v2 ← top(srez); pop(srez)
    v1 ← top(srez); pop(srez)
    v ← phi(v1, v2)
    push(srez,v)
    if (not esteVida(spar))
        then adr ← Top(sadr); pop(sadr)
            u ← top(spar); pop(spar)
            switch (adr)
                case 6
                    goto 6;
                case 8
                    goto 8
    else return top(srez)
end

```

Dacă efectuăm un calcul sintactic (figura 3.3), atunci rezultatul final este un arbore binar în care fiecare nod sau are exact doi subarbori nevizi sau are ambii subarbori vizi. Procedura descrie o strategie de explorare a acestui arbore. Ca și în cazul parcurgerii de arbori, programul poate fi simplificat. De exemplu, prin structurare, stiva cu adresele de return poate fi eliminată. Considerăm că efectuarea acestor modificări constituie un bun exercițiu pentru cititor.

3.3.1 Studiu de caz: funcția Fibonacci

Un subprogram recursiv care calculează funcția Fibonacci este următorul:

```

function fib(n)
    if (n ≤ 1
        return n
    else
        return fib(n-1) + fib(n-2)
end

```

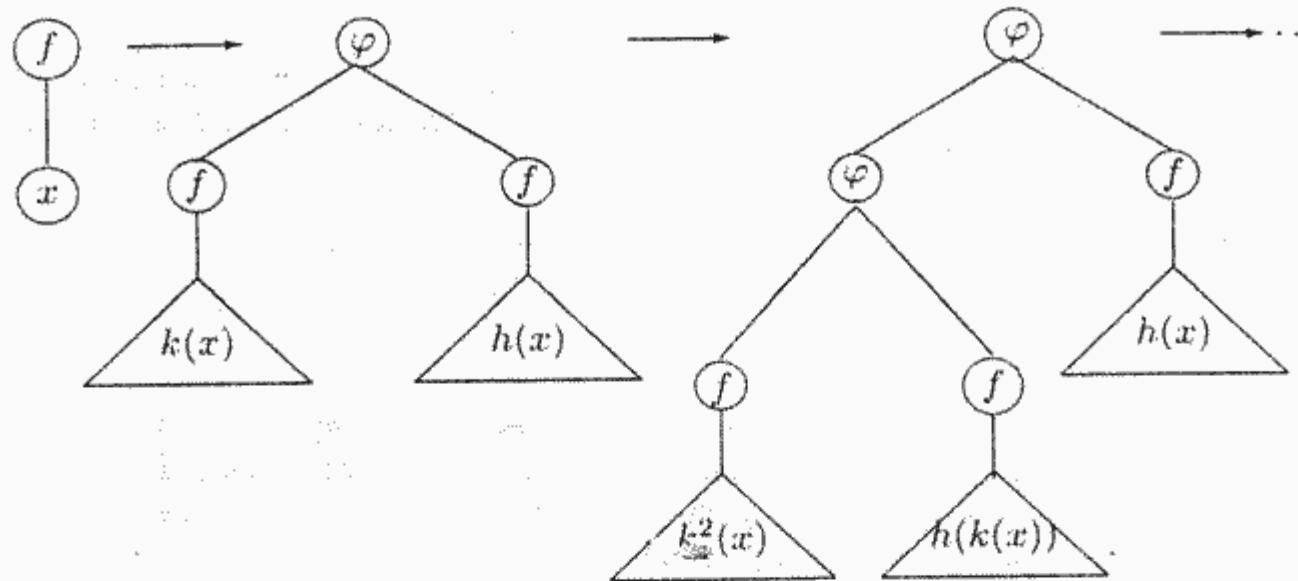


Figura 3.3: Recursie în cascadă

Dacă notăm cu $T(n)$ complexitatea timp a programului de mai sus, rezultă că $T(n)$ satisface relația de recurență:

$$T(n) = \begin{cases} 1 & , \text{ dacă } n \leq 1 \\ T(n-1) + T(n-2) + 1 & , \text{ dacă } n > 1 \end{cases}$$

Avem $T(n) > T(n-1) + T(n-2)$, care implică $T(n) > f_n$, unde f_n este cel de-al n -lea termen din sirul lui Fibonacci ($f_n = fib(n)$). Știm că f_n poate fi calculat și cu formula:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] = O(\phi^n)$$

unde¹ $\phi = \frac{1+\sqrt{5}}{2}$. Deci f_n are o creștere foarte rapidă, exponențială, și de aici rezultă că programul recursiv are complexitatea timp exponențială. Vom arăta că prin derecursivare se poate obține un program cu complexitate liniară. Considerăm primii pași din calculul sintactic al funcției (figura 3.4). Observăm că $fib(n-2)$ este evaluat de două ori. În varianta nerecursivă vom forța ca $fib(n-2)$ să fie evaluat o singură dată. Programul nerecursiv va descrie un proces iterativ, la fiecare iterație determinându-se o valoare parțială a funcției fib (un termen al sirului Fibonacci). Generalizăm, considerând în loc de n o variabilă liberă i . Utilizăm trei variabile, u, v, w , pentru care vor păstra următoarele relații invariante: u memorează $f_{i-1} = fib(i-1)$, v memorează $f_{i-2} = fib(i-2)$ și w memorează $f_i = fib(i)$.

¹ Numărul $\frac{1+\sqrt{5}}{2}$ este cunoscut sub numele de „numărul de aur” (golden ratio), el fiind important atât în diferite ramuri ale matematicii, cât și în lumea artei. El a fost notat cu litera grecească ϕ în memoria lui Phidias, despre care se spune că a utilizat acest număr la realizarea sculpturilor sale.

Menținerea acestor relații invariante se realizează prin următoarele instrucțiuni:

```

 $\{u = f_{i-1} \wedge v = f_{i-2} \wedge w = f_i\}$ 
v ← u;
 $\{u = f_{i-1} \wedge v = f_{i-1} \wedge w = f_i\}$ 
u ← w;
 $\{u = f_i \wedge v = f_{i-1} \wedge w = f_i\}$ 
w ← u + v;
 $\{u = f_i \wedge v = f_{i-1} \wedge w = f_{i+1}\}$ 
i ← i + 1;
 $\{u = f_{i-1} \wedge v = f_{i-2} \wedge w = f_i\}$ 

```

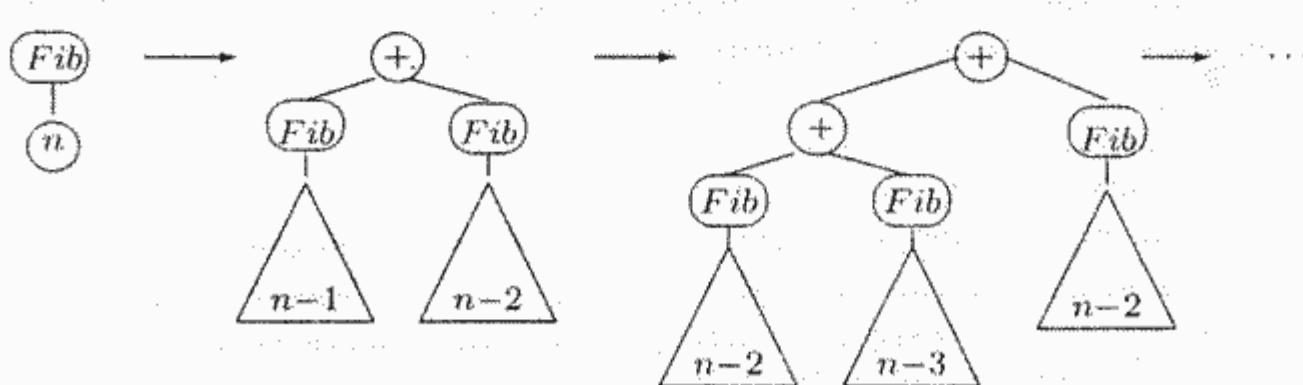
Acum, programul nerecursiv poate fi descris astfel:

```

function fib(n)
    //initializari
    u ← 1;           //u = f_1
    v ← 0;           //v = f_0
    w ← 1;           //w = f_2
    i ← 2;           //i = index
    //proces iterativ
    //invariant: u=f_(i-1), v=f_(i-2), w=f_i
    while (i < n) do
        v ← u
        u ← w
        w ← u + v
        i ← i + 1
    //i = n and w = f_n
    return w
end

```

Se observă imediat că programul nerecursiv are complexitatea timp $O(n)$.



3.4 Memorarea într-un tabel a rezultatelor subproblemelor

În această subsecțiune vom prezenta o tehnică prin care putem extrage algoritmi nerecursivi de complexitate polinomială (uneori chiar liniară) din algoritmi recursivi cu complexitate exponențială. Ideea principală a acestei tehnici este următoarea: *se rezolvă subproblemele de la mic la mare și se memorează rezultatele acestora într-un tabel*. Avantajul acestei strategii constă în faptul că, dacă o subproblemă apare în rezolvarea mai multor subprobleme mai mari, ea va fi rezolvată o singură dată și valoarea soluției sale va fi regăsită în tabel de fiecare dată în timpul $O(1)$. Asupra acestei strategii vom mai reveni în capitolul dedicat programării dinamice. Exemplificăm aplicarea acestei tehnici prin câteva studii de caz.

3.4.1 Funcția Fibonacci

Este exemplul cel mai simplu și mai elocvent. Am văzut deja că algoritmul descris de subprogramul recursiv are complexitate exponențială. Presupunem acum că valorile funcției fib sunt calculate de la mic la mare, i.e., în ordinea $fib(1), fib(2), fib(3), \dots$ și că sunt memorate într-un tabel unidimensional f . Calculul lui $fib(i)$, $i > 2$, se poate face în timpul $O(1)$ printr-o instrucțiune de forma:

$$f[i] \leftarrow f[i-2] + f[i-1]$$

Rezultă că primele n numere Fibonacci pot fi calculate în timpul $O(n)$.

3.4.2 Numărul de expresii cu paranteze corect formate

Considerăm expresii scrise numai cu operatori binari. O astfel de expresie este următoarea $((a * b) + c) + (a - b)$. Prin stergerea operanzilor și operatorilor obținem o expresie formată numai cu paranteze: $((())()$. Expresiile obținute în acest mod constituie expresiile cu paranteze corect formate. Pentru un număr natural n dat, notăm cu $P(n)$ numărul de expresii cu paranteze corect formate de lungime n . Ne propunem găsirea unei relații de recurență pentru $P(n)$ și a unui algoritm eficient de calcul al lui $P(n)$.

Mai întâi vom căuta o definiție recursivă pentru aceste expresii. Notăm cu mulțimea expresiilor. O definiție recursivă pentru E este următoarea:

- i) $() \in E$;
- ii) dacă $e \in E$, atunci $(e) \in E$;
- iii) dacă e_1, e_2 , atunci $e_1 e_2 \in E$;
- iv) E este cea mai mică mulțime care satisfac condițiile de mai sus.

Un prim fapt care se observă imediat este că orice expresie este de lungime pară (se demonstrează prin inducție). De aici rezultă că $P(n) = 0$ pentru orice n impar. Rămâne de determinat $P(n)$, pentru n par. Evident, $P(2) = 1$. Fie $n > 2$. Vom încerca să găsim căte descompuneri ii) sau iii) pot exista pentru o expresie de lungime n . Se observă ușor că există $P(n-2)$ descompuneri ii). Pentru descompunerile iii) vom vedea mai întâi care sunt valorile posibile pentru lungimea lui e_1 (evident, $|e_2| = n - |e_1|$). Nu este greu de observat că aceste valori sunt

$2, 4, \dots, n - 2$. Numărul expresiilor e_1 de lungime i este $P(i)$, iar pentru un e_1 fixat, numărul expresii de forma iii) este $P(n - i)$. De aici, rezultă următoarea formulă recurrentă pentru $P(n)$:

$$P(n) = P(n - 2) + \sum_{i \in I(n)} P(i) \cdot P(n - i)$$

unde $I(n) = \{2, 4, \dots, n - 2\}$. Un subprogram recursiv care calculează $P(n)$ are complexitatea timp exponențială. De exemplu, un apel pentru calculul lui $P(4)$ necesită trei apeluri recursive pentru calculul lui $P(2)$. Calculând valorile $P(n)$ în ordinea $P(2), P(4), \dots$ și memorându-le într-un tabel unidimensional obținem un algoritm de complexitate timp $O(n^2)$.

3.4.3 Numărul de partiții ale unui număr natural

Un număr natural n poate fi scris ca sumă de numere naturale nemulțumite mai multe moduri (nu vom face distincție între două sume în care diferă numai ordinea termenilor). De exemplu, 4 poate fi scris ca: $1+3, 1+1+2, 1+1+1+1, 2+2$. Notăm cu $S(n)$ numărul de posibilități în care n poate fi scris ca sume. Pentru a găsi numărul $S(n)$, considerăm mai întâi câteva cazuri particolare:

$$\begin{aligned} 2 &= 1 + 1 \\ 3 &= 1 + 2 \\ &\quad 1 + 1 + 1 \\ 4 &= 1 + 3 \\ &\quad 1 + 1 + 2 \\ &\quad 1 + 1 + 1 + 1 \\ &\quad 2 + 2 \end{aligned}$$

Privind aceste exemple observăm următoarele:

- O sumă poate fi reprezentată unic de o secvență de forma (n_1, n_2, \dots, n_k) cu proprietățile: $n_1 \leq n_2 \leq \dots \leq n_k$ și $n_1 + n_2 + \dots + n_k = n$.
- Sumele lui 3 (care toate încep cu 1) intră în sumele lui 4 care încep cu 1. Deci este util să considerăm numărul $s(i, j)$ al sumelor lui $i + j$ care încep cu i . Invităm cititorul să mai considere câteva exemple particolare ($n = 4, 5, 6, 7$) pentru a vedea utilitatea acestei notății.

Ne propunem acum să găsim o formulă recurrentă pentru $s(i, j)$. Dacă $j < i$, atunci $s(i, j) = 0$ și dacă $i = j$, atunci $s(i, i) = 1$. În continuare presupunem $i < j$. O sumă a lui $i + j$ care începe cu i este de forma (i, j) sau de forma (i, k, \dots) , unde (k, \dots) este o sumă a lui j care începe cu k . Valoarea minimă pentru k este i , iar cea maximă este $\frac{j}{2}$. De aici rezultă următoarea formulă pentru $s(i, j)$ (cazul $i < j$):

$$s(i, j) = \sum_{i \leq k \leq \frac{j}{2}} (1 + s(k, j - k))$$

Valorile $s(i, j)$ pot fi memorate într-un tabel bidimensional. Elementele din acest tabel pot fi calculate în ordinea parcurgerii tabelului pe coloane. Cunoscând valorile $s(i, j)$, numărul $S(n)$ poate fi calculat prin formula:

$$S(n) = \sum_{1 \leq i \leq \frac{n}{2}} s(i, n-i)$$

Rezultă un algoritm nerecursiv care determină $S(n)$ în timpul $O(n^3)$.

3.5 Referințe bibliografice

Exemple privind proiectarea de algoritmi recursivi pot fi găsite în [Woo84, HE87, HS84]. În [HE87] este explicitată foarte bine evoluția stivei "run-time" în timpul apelurilor recursive. Pentru aspectele teoretice privind recursia se poate consulta [Wan80, Gri81, AA78, Coh90].

Capitolul 4

Sortare internă

Sortarea este una dintre problemele cele mai importante atât din punct de vedere practic, cât și teoretic. Sortarea poate fi formulată în diferite moduri. Cea mai generală formulare și mai des utilizată este următoarea:

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

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

Fie dată o secvență de structuri statice (R_0, \dots, R_{n-1}) , unde fiecare structură R_i are o valoare cheie K_i . Pește mulțimea cheilor K_i este definită o relație de ordine totală. Problema sortării constă în determinarea unei permutări π cu proprietatea $K_{\pi(0)} \leq K_{\pi(1)} \leq \dots \leq K_{\pi(n-1)}$ și în rearanjarea structurilor în ordinea $(R_{\pi(0)}, \dots, R_{\pi(n-1)})$.

Pentru ambele formulări presupunem că secvența dată este reprezentată printr-o listă liniară. Dacă această listă este memorată în memoria internă a calculatorului, atunci avem *sortare internă* și, dacă se găsește într-un fișier memorat pe un suport extern, atunci avem *sortare externă*. În acest capitol ne ocupăm numai de sortarea internă.

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

Intrare: n și tabloul $(a[i] \mid i = 0, \dots, n - 1)$ cu $a[i] = v_i$, $i = 0, \dots, n - 1$.

Ieșire: tabloul a cu proprietățile: $a[i] = w_i$ pentru $i = 0, \dots, n - 1$, $w_0 \leq \dots \leq w_{n-1}$ și (w_0, \dots, w_{n-1}) este o permutare a secvenței (v_0, \dots, v_{n-1}) ; convenim să notăm această proprietate prin $(w_0, \dots, w_{n-1}) = \text{Perm}(v_0, \dots, v_{n-1})$.

Un algoritm de sortare este stabil dacă păstrează ordinea relativă inițială a elementelor egale.

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

4.1 Sortare bazată pe comparații

În această secțiune am grupat metodele de sortare bazate pe următoarea tehnică: determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ și $a[j]$ 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 subsecvență ordonată deja (sortare prin inserție), sau pentru a selecta o valoare ce va fi pusă pe poziția sa finală (sortare prin selecție). Decizia că o anumită metodă aparține uneia dintre subclasele de mai sus are un anumit grad de subiectivitate. De exemplu, selecția naivă ar putea fi foarte bine considerată o metodă bazată pe interschimbare.

4.1.1 Sortarea prin interschimbare

Vom prezenta întâi strategia cunoscută sub numele de *sortare prin metoda bulelor* (*bubble-sort*).

Notăm cu $SORT(a)$ predicatul care ia valoarea *true* dacă și numai dacă tabloul a este sortat. Metoda bubble-sort se bazează pe următoarea definiție a predicatului $SORT(a)$:

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

O pereche (i, j) , cu $i < j$, formează o *inversiune* (*inversare*), dacă $a[i] > a[j]$. Astfel, pe baza definiției de mai sus vom spune că tabloul a este sortat dacă și numai dacă nu există nici o inversiune $(i, i + 1)$. Metoda bubble-sort propune parcurgerea iterativă a tabloului a și, la fiecare parcurs, ori de câte ori se întâlnește o inversiune $(i, i + 1)$ se procedează la interschimbarea $a[i] \leftrightarrow a[i + 1]$. La prima parcurs, elementul cel mai mare din secvență formează inversiuni cu toate elementele aflate după el și, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este și locul său final. La iterația următoare, la fel se va întâmpla cu cel de-al doilea element cel mai mare. În general, dacă subsecvența $a[r + 1..n - 1]$ nu are nici o inversiune la iterația curentă, atunci ea nu va avea inversiuni la nici una din iterațiile următoare. Aceasta permite ca la iterația următoare să fie verificată numai subsecvența $a[0..r]$. Terminarea algoritmului este dată de faptul că la fiecare iterație numărul de interschimbări este micșorat cu cel puțin 1.

Descrierea algoritmului este următoarea:

```

procedure bubbleSort(a, n)
    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 întâlnit atunci când secvența de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operații. Cazul cel mai nefavorabil este obținut când secvența de intrare este ordonată crescător și, în această situație, procedura execută $O(n^2)$ operații. În continuare, ne ocupăm de timpul mediu de execuție. Mai precis, vom arăta că numărul mediu de comparații este

$$O\left(\frac{1}{2}(n^2 - n \log n - O(n)) + O(\sqrt{n})\right)$$

Fără a restrânge generalitatea, presupunem că valoarea inițială a tabloului a este o permutare a mulțimii $\{1, 2, \dots, n\}$: $\langle v_0, \dots, v_{n-1} \rangle = \text{Perm}(0, 1, \dots, n-1)$.

Definiția 4.1. Fie $x = \langle x_0, \dots, x_{n-1} \rangle$ o permutare a mulțimii $\{0, 1, \dots, n-1\}$. Numim cod al permutării x secvența $\text{cod}(x) = (y_0, \dots, y_{n-1})$, unde y_j reprezintă numărul elementelor x_i aflate la stânga lui j și mai mari decât j , i.e.

$$y_j = \#\{x_i \mid i < j \text{ și } x_i > j\},$$

unde k satisface $x_k = j$. Secvența $\text{cod}(x)$ mai este numită și tablou de inversări. Vom prefera totuși denumirea de cod al permutării.

Codurile permutărilor ne vor ajuta la studiul proprietăților algoritmului `BubbleSort`. Să observăm mai întâi că valoarea $\text{cod}(v)(j)$ reprezintă numărul de inversiuni ale căror componentă secundară este j . De exemplu, dacă $v = (4, 2, 3, 1, 0)$, atunci $\text{cod}(v) = (4, 3, 1, 1, 0)$. După o trecere a algoritmului `BubbleSort` (o execuție a buclei `while`) se obține următoarea valoare pentru a : $w = (2, 3, 1, 0, 4)$. Avem $\text{cod}(w) = (3, 2, 0, 0, 0)$. Se observă următoarea proprietate:

$$\text{cod}(v)(i) \neq 0 \Rightarrow \text{cod}(w)(i) = \text{cod}(v)(i) - 1$$

În general are loc:

Teorema 4.1. Presupunem că valoarea tabloului a este $v = (v_0, \dots, v_{n-1}) = \text{Perm}(0, 1, \dots, n-1)$ și că, după o trecere a algoritmului `BubbleSort`, se obține valoarea $w = (w_0, \dots, w_{n-1})$ pentru a. Atunci are loc:

$$(\forall i) \text{cod}(v)(i) \neq 0 \Rightarrow \text{cod}(w)(i) = \text{cod}(v)(i) - 1$$

Teorema de mai sus justifică atât proprietatea de terminare a algoritmului `BubbleSort`, cât și corecta utilizare a variabilei `ultim`. În continuare ne ocupăm de calculul numărului mediu de comparații. Plecăm de la presupunerea că orice permutare v a mulțimii $\{0, 1, \dots, n-1\}$ poate să apară cu aceeași probabilitate, $p = \frac{1}{n!}$, ca intrare a algoritmului `BubbleSort`. Notăm cu u_j valoarea expresiei `ultim-1` (variabilei `n1` la începutul iterăției j). Fie w valoarea tabloului a obținută după cea de-a $(j-1)$ -a iterăție. Dacă $\text{cod}(w)(i) > 0$, atunci există $\text{cod}(w)(i)$ elemente mai mari decât i aflate la stânga lui i ; cel mai mare dintre aceste elemente va avea poziția finală $p \geq \text{cod}(w)(i) + i$. Mai mult, dacă elementul i este cel mai din dreapta care trebuie schimbat la iterăția j , atunci are loc:

$$\text{cod}(w)(i) \geq 1 \text{ și } \text{ultim} = \text{cod}(w)(i) + i$$

De aici rezultă $u_j = \max_i \{ \text{cod}(w)(i) + i - 1 \mid \text{cod}(w)(i) \geq 1 \}$. Aplicând teorema de mai sus de $j - 1$ ori, obținem: $\text{cod}(w)(i) = \text{cod}(v)(i) - (j - 1)$. Relația de mai sus devine $u_j = \max_i \{ \text{cod}(v)(i) + i - j \mid \text{cod}(v)(i) > j - 1 \}$. Pentru un k dat, algoritmul bubbleSort va executa cel mult k iterații pentru acele intrări care satisfac $\forall i : \text{cod}(v)(i) < k$. Numărul de permutări care satisfac proprietatea de mai sus este egal cu $k^{n-k}k!$ (conform [Knu76] p. 108). Rezultă că probabilitatea ca algoritmul BubbleSort să execute cel mult k iterații este $\frac{1}{n!}(k^{n-k}k!)$, iar probabilitatea ca să se execute exact k iterații este $\frac{1}{n!}(k^{n-k}k! - (k-1)^{n-k+1}(k-1)!)$. Pentru calcularea numărului mediu de comparații, vom determina valorile medii pentru u_j . Pentru aceasta, este util să determinăm numărul permutărilor v cu proprietatea

$$(\forall i) \text{cod}(v)(i) < j - 1 \text{ sau } \text{cod}(v)(i) + i - j \leq k$$

(numărul intrărilor pentru care BubbleSort execută cel mult $j+k$ iterații). Conform cu [Knu76] p. 108, acest număr este:

$$f_j(k) = (j+k)!(j-1)^{n-j-k}, \text{ pentru } j \leq j+k \leq n$$

Valoarea medie a lui u_j este:

$$u_j^{\text{med}} = \frac{\sum_{(k|j < k+j \leq n)} k(f_j(k) - f_j(k-1))}{n!}$$

Conform cu [Knu76] p. 108 și pp. 129-134 are loc:

$$C^{\text{med}}(n) = O\left(\frac{1}{2}(n^2 - n \log n - O(n)) + O(\sqrt{n})\right)$$

În cele ce urmează, vom prezenta strategia cunoscută sub numele de *par-impar* (*odd-even-sort*). Esența metodei *par-impar* constă în alternarea secvenței de operații *swap(a[i], a[i+1])*, $i=0, 2, 4, \dots$ cu secvența *swap(a[i], a[i+1])*, $i=1, 3, 5, \dots$. După cel mult $n/2$ execuții a fiecăreia din cele două secvențe de operații, elementele tabloului *a* vor fi sortate. Metoda nu conduce la un algoritm superior algoritmului bubbleSort. Totuși, algoritmul de sortare prin metoda *par-impar* are calitatea de a fi paralelizabil. Aceasta se datorează faptului că perechile care interacționează, atunci când se execută una din cele două secvențe de operații menționate anterior, sunt disjuncte.

```

procedure par-imparSort(a, n)
    for s ← 0 to n-1 do
        if s este par
            then for i ← 0 to n-2 step 2 do
                  swap(a[i], a[i+1])
        if s este impar
            for i ← 1 to n-2 step 2 do
                  swap(a[i], a[i+1])
    end

```

4.1.2 Sortare prin inserție

Una dintre familiile importante de tehnici de sortare se bazează pe metoda „jucătorului de bridge” (atunci când își aranjează cărțile), prin care fiecare element este inserat în locul corespunzător în raport cu elementele sortate anterior.

4.1.2.1 Sortare prin inserție directă

Principiul de bază al algoritmului de sortare prin inserție este următorul: Se presupune că subsecvența $(a[0], \dots, a[j-1])$ este sortată. Se caută în această subsecvență locul i al elementului $a[j]$ și se inserează $a[j]$ pe poziția i . Poziția i este determinată astfel:

- $i = 0$, dacă $a[j] < a[0]$;
- $0 < i < j$ și satisfacă $a[i-1] \leq a[j] < a[i]$;
- $i = j$, dacă $a[j] \geq a[j-1]$.

Determinarea lui i se poate face prin căutare secvențială sau prin căutare binară. Considerăm cazul când poziția i este determinată prin căutare secvențială (de la dreapta la stânga) simultan cu deplasarea elementelor mai mari decât $a[j]$ cu o poziție 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)
    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 poziției i în subsecvența $a[0..j-1]$ necesită $O(j-1)$ timp. Rezultă că timpul total în cazul cel mai nefavorabil este $O(1 + \dots + n-1) = O(n^2)$. Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, timpul de execuție este $O(n)$.

Exercițiul 4.1.1. Timpul de execuție al algoritmului de sortare prin inserție poate fi îmbunătățit considerând secvența de sortat reprezentată printr-o listă simplu înlănită. Să se rescrie algoritmul InsertSort corespunzător acestei reprezentări. Să se precizeze timpul de execuție al nouui algoritm.

Exercițiul 4.1.2. Să se rescrie algoritmul InsertSort, astfel încât căutarea poziției i în subsecvența $a[0..j-1]$ să se facă prin metoda căutării binare. Să se analizeze timpul de execuție al nouui algoritm.

4.1.2.2 Metoda lui Shell

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

Exemplu. Presupunem $n = 9$. Sunt execuțiile următorii pași (figura 4.1):

1. *Prima trecere.* Se împart elementele în grupe de câte trei sau două elemente (valoarea incrementului $h_1 = 4$) care se sortează separat:

$$(a[0], a[4], a[8]), \dots, (a[3], a[7])$$

2. *A două trecere.* Se grupează elementele în două grupe de câte trei elemente (valoarea incrementului $h_2 = 3$): $(a[0], a[3], a[6]) \dots (a[2], a[5], a[8])$ și se sortează separat.
3. *A treia trecere.* Acest pas termină sortarea prin considerarea unei singure grupe care conține toate elementele. În final, cele nouă 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 inserția directă poate fi utilizată cu succes pentru fiecare operație de sortare. Prin aceste inserții intermediare, elementele tind să conveargă rapid spre destinația lor finală. Secvența de incremente $4, 3, 1$ nu este obligatorie; poate fi utilizată orice secvență $h_i > h_{i-1} > \dots > h_0$, cu condiția ca ultimul increment h_0 să fie 1.

Presupunem că numărul de valori de incrementare este memorat de variabila `nincr` și că acestea sunt memorate în tabloul (`kval[h] | 0 ≤ h ≤ 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

```

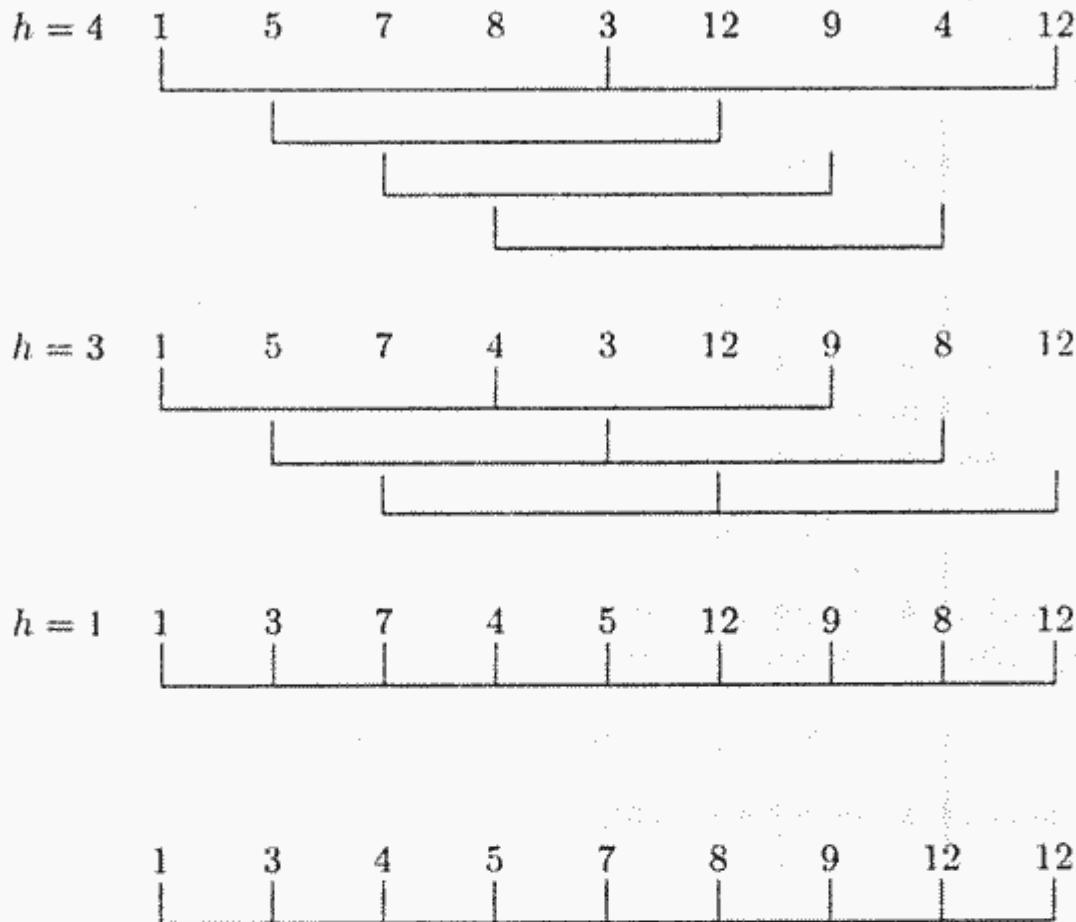


Figura 4.1: Sortarea prin metoda lui Shell

Evaluarea metodei lui Shell. Pentru evaluare vom presupune că elementele din secvență sunt diferite și dispuse aleator. Vom denumi operația de sortare corespunzătoare primei treceri h_1 -sortare, apoi h_{t-1} -sortare etc. O subsecvență 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 inserției directe, și anume cazul când avem numai două valori de incrementare: $h_1 = 2$ și $h_0 = 1$. Cazul cel mai favorabil este obținut când secvența de intrare este ordonată crescător și sunt executate $\frac{n}{2} - 1 + n - 1$ comparații și nici o deplasare. Cazul cel mai nefavorabil, când secvența de intrare este ordonată descrescător, necesită $\frac{1}{4}n(n - 2) + \frac{n}{2}$ comparații și 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 secvență 2-ordonată de elemente $a[0], a[1], \dots, a[n - 1]$. Este ușor de văzut că numărul de permutări $(i_0, i_1, \dots, i_{n-1})$ ale mulțimii $\{0, 1, \dots, n - 1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n - 3$, este $C_n^{[\frac{n}{2}]}$, deoarece obținem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse pe poziții impare 1, 3, Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvență 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 $\{0, 1, \dots, n - 1\}$. Relațiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele șase cazuri 2-ordonate

vom găsi $A_4 = 0 + 1 + 1 + 2 + 3 = 8$. În urma calculelor, care sunt un pic dificile (a se vedea [Knu76], p. 87), se obține 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:

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

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

Teorema 4.2. *Numărul mediu de inversări executate de algoritmul lui Shell pentru secvența de incremente $(2, 1)$ este $O(n^{\frac{3}{2}})$.*

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

Teorema 4.3. *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 secvența de valori de incrementare pentru algoritmul lui Shell este h_{t-1}, \dots, h_0 , se cunosc următoarele rezultate.

Teorema 4.4. *Dacă secvența de incremente h_{t-1}, \dots, h_0 satisface condiția*

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

atunci timpul de execuție 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 interacțiune între elementele de pe pozițiile pare și 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 4.5. *Timpul de execuție în cazul cel mai nefavorabil a algoritmului ShellSort este $O(n^{\frac{3}{2}})$, atunci când $h_s = 2^{s+1} - 1$, $0 \leq s \leq t-1$, $t = [\log_2 n] - 1$.*

4.1.3 Sortarea prin selecție

Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă: la pasul curent se selectează un element din secvență și 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 puțin eficientă. Noi ne vom ocupa doar de două strategii de sortare prin selecție.

4.1.3.1 Selecția naivă

Este o metodă mai puțin eficientă, dar foarte simplă în prezentare. Se bazează pe următoarea caracterizare a predicatului $SORT(a)$:

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

Ordinea în care sunt așezate elementele pe pozițiile lor finale este $n - 1, n - 2, \dots, 0$. O formulare echivalentă este:

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

caz în care ordinea de așezare este $0, 1, \dots, n - 1$.

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

```

procedure naivSort(a, n)
  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

```

Evaluarea algoritmului descris de procedura `naivSort` este simplă și conduce la un timp de execuție $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` ar avea un timp de execuție $\Theta(n^2)$. Este interesant de comparat `BubbleSort` cu `NaivSort`. Cu toate că sortarea prin metoda bulelor face mai puține comparații decât selecția naivă, ea este aproape de două ori mai lentă decât selecția naivă, din cauza faptului că realizează multe schimbări în timp ce selecția naivă implică o mișcare redusă a datelor. În tabelul din figura 4.2 sunt redați timpii de execuție (în sutimi de secunde) pentru cele două metode obținuți în urma a zece teste pentru $n = 1000$.

4.1.3.2 Selecția sistematică

Se bazează pe structura de date de tip `max-heap` 2.8.2. Metoda de sortare prin selecție sistematică constă în parcurgerea a două etape:

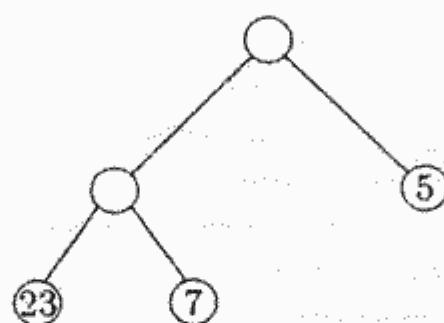
1. construirea pentru secvența curentă a proprietății `MAX-HEAP(a)`;
2. selectarea în mod repetat a elementului maximal din secvența curentă și refacerea proprietății `MAX-HEAP` pentru secvența rămasă.

Etapa 1. Considerăm că tabloul `a` are lungimea n . Inițial, 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 $MAX-HEAP(a, 2)$:

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

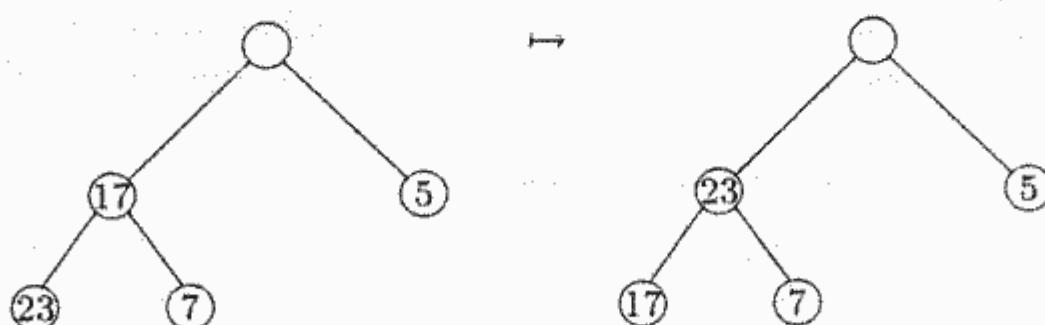
Figura 4.2: Compararea algoritmilor BubbleSort și NaivSort



sfex

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

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



Se obține secvența $(10, 23, 5, 17, 7)$ care are proprietatea MAX-HEAP începând cu 1. Considerăm $\ell = 0$ și introducem $a[0] = 10$ în grămadă $a[1..4]$. Se obține valoarea $(23, 17, 5, 10, 7)$ pentru tabloul a , valoare care verifică proprietatea MAX-HEAP.

sfex

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

```

procedure intrInGr(a, n, ℓ)
  j ← ℓ
  
```

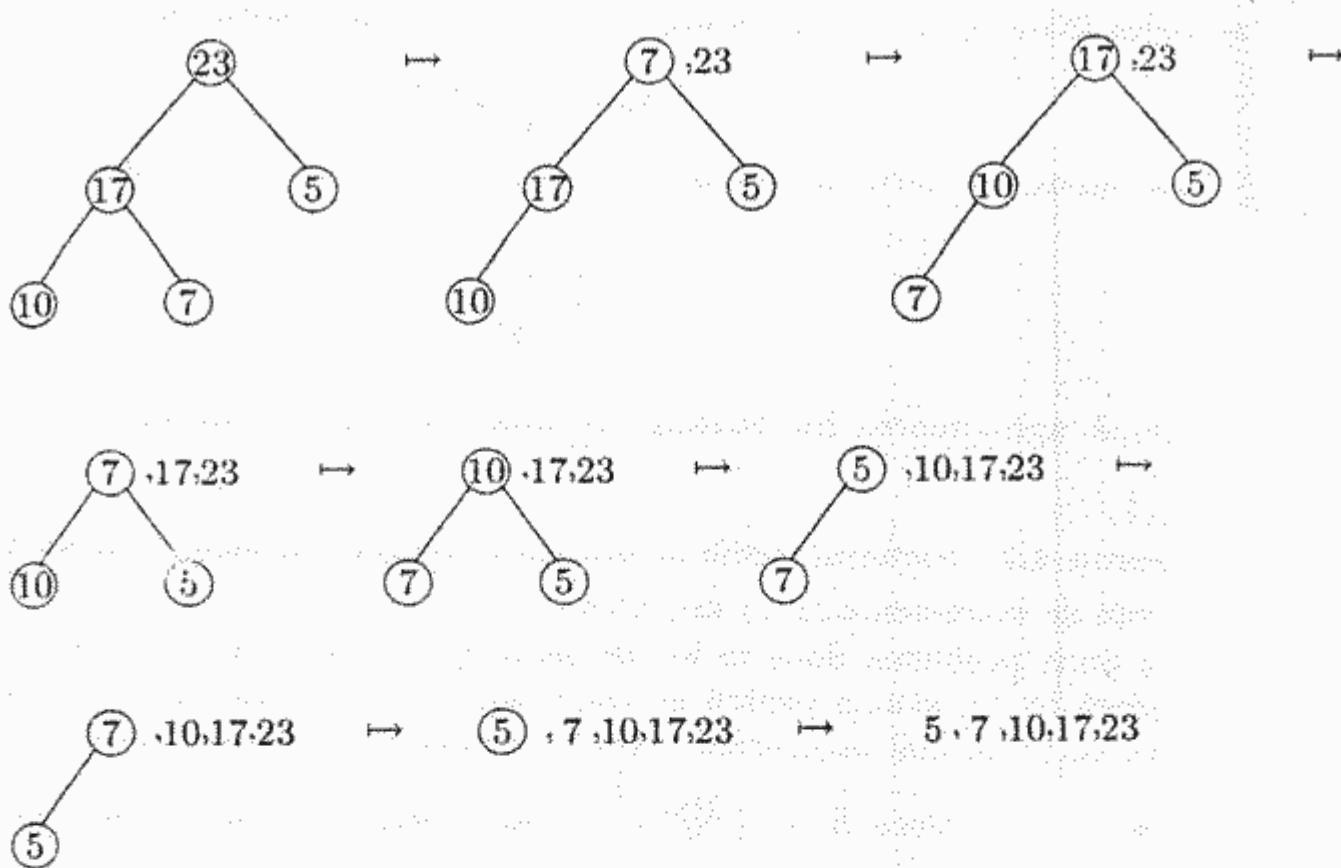


Figura 4.3: Etapa a doua

```

esteHeap ← false
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 2. Deoarece inițial avem MAX-HEAP(a), rezultă că pe primul loc se găsește elementul maximal din $a[0..n - 1]$. Punem acest element la locul său final prin interschimbarea $a[0] \leftrightarrow a[n - 1]$. Acum $a[0..n - 2]$ are proprietatea MAX-HEAP începând cu 1. Refacem MAX-HEAP pentru această secvență prin introducerea lui $a[0]$ în grămadă $a[0..n - 2]$, după care punem pe locul său final cel de-al doilea element cel mai mare din secvența de sortat. Procedeul continuă până când toate elementele ajung pe locurile lor finale.

Exemplu. Etapa a doua pentru exemplul anterior este ilustrată în figura 4.3.

sfx

Algoritmul de sortare prin selecție sistematică este descris de subprogramul **heapSort**:

```
procedure heapSort(a,n)
```

```

n1 ← ⌊ $\frac{n-1}{2}$ ⌋
for  $\ell \leftarrow n1$  downto 0 do
    intrInGr(a, n,  $\ell$ )
    r ← n-1
    while ( $r \geq 1$ ) 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 proprietății MAX-HEAP pentru toată secvența de intrare sunt efectuate următoarele operații:

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

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

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

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

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

Numărul total de comparații 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 comparații este $C(n) = O(n \log_2 n)$.

4.1.4 Sortarea rapidă

A se vedea secțiunea 10.3.

4.1.5 Sortarea prin interclasare

A se vedea secțiunea 10.2.

4.1.6 Numărul minim de comparații pentru sortare

Algoritmii de sortare prezentați până acum se bazează pe executarea a două operații primitive: comparația și interschimbarea a două elemente. Deoarece orice interschimbare este, în general, precedată de o comparație (prin care se decide dacă interschimbarea este necesară) putem spune că operațiile de comparare domină calculul oricărui algoritm prezentat până acum. Ne punem următoarele două întrebări:

- care este numărul minim de comparații executate în cazul cel mai nefavorabil?
- care algoritmi de sortare realizează minimul de comparații, i.e. care algoritmi sunt optimali?

Pentru a putea răspunde la cele două întrebări trebuie mai întâi să precizăm modelul de calcul peste care sunt construși acești algoritmi.

Să analizăm câțiva algoritmi pentru ordonarea secvenței (a_0, a_1, a_2) . Vom încerca să determinăm care elemente se compară pentru obținerea permutării dorite. O primă observație pe care o facem este aceea că dacă se compară elementele a_i și a_j , notat $i ? j$, atunci mulțimea comparațiilor care urmează a fi făcute depinde de răspunsul obținut prin efectuarea acestei comparații. Pentru simplitate vom presupune $a_i \neq a_j$ dacă $i \neq j$. Deoarece răspunsul dat de comparația $i ? j$ are numai două posibilități de alegere, rezultă că putem reprezenta cele două mulțimi de comparații prin intermediul unui arbore binar:

- rădăcina va conține o comparație $i ? j$;
- subarborele din stânga conține comparațiile făcute în cazul $a_i < a_j$;
- subarborele din dreapta conține comparațiile făcute în cazul $a_i > a_j$.

Arborii corespunzători sortării prin selecție naivă și sortării prin inserție sunt reprezentați în figura 4.4.

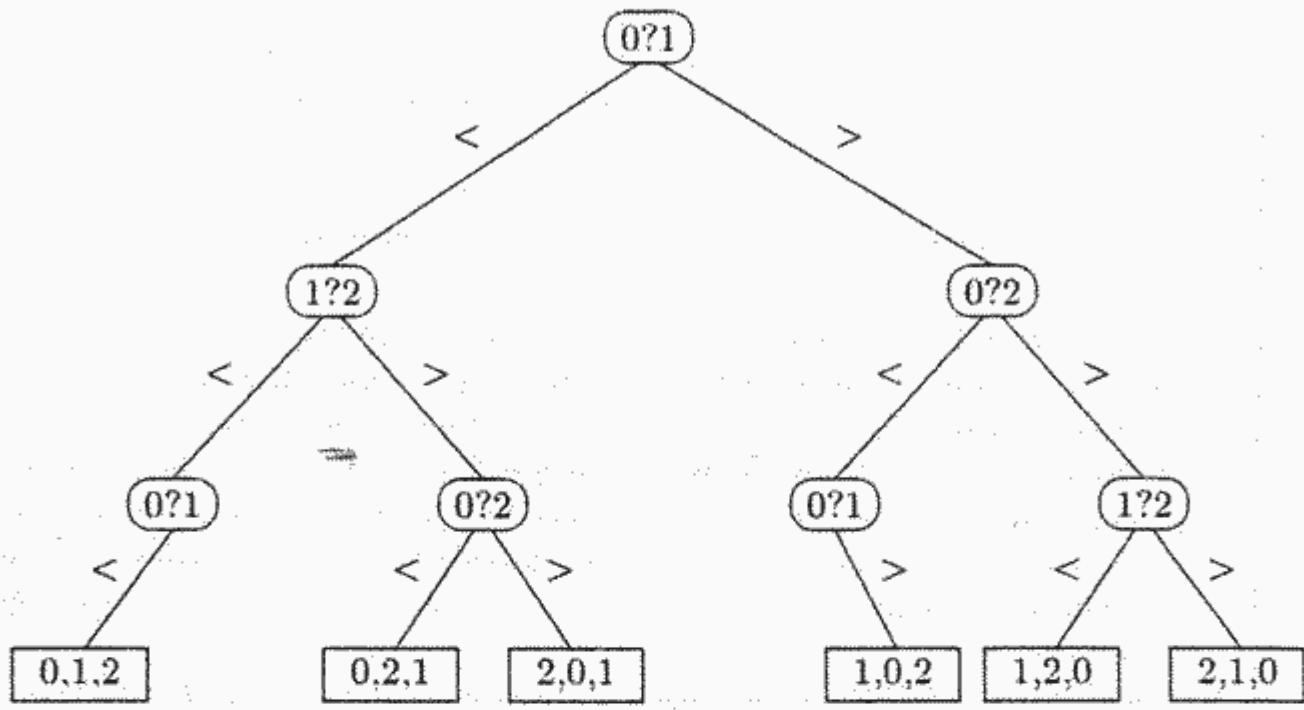
În vîrfurile de pe frontieră am trecut permutările care dău ordinea crescătoare a elementelor. În acest fel putem asocia fiecărui algoritm de sortare bazat pe comparații un arbore binar. Vom utiliza acești arbori pentru a defini modelul de calcul.

Definiția 4.2. Un arbore de decizie pentru n elemente este un arbore binar în care vîrfurile interne sunt etichetate cu perechi de forma $i ? j$, iar vîrfurile de pe frontieră sunt etichetate cu permutări ale mulțimii $\{0, 1, \dots, n - 1\}$.

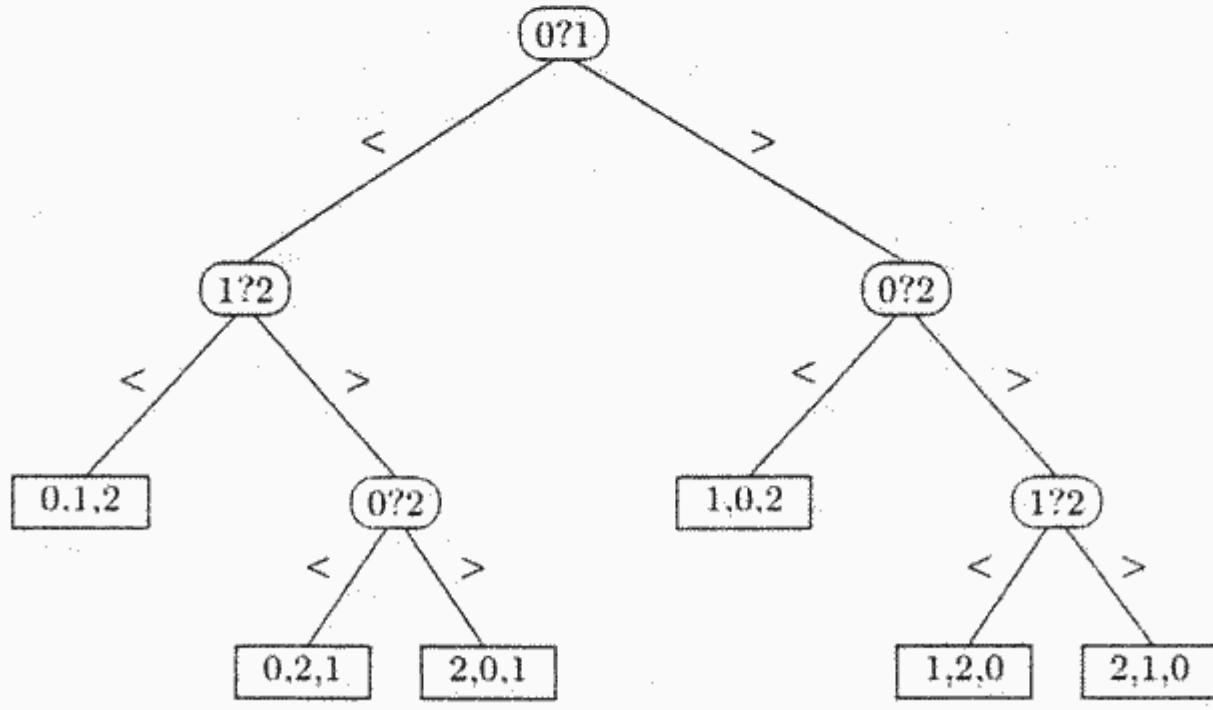
Definiția 4.3. Fie t un arbore de decizie pentru n elemente și secvența $a = (a_0, \dots, a_{n-1})$. Calculul lui t pentru intrarea a constă în parcurgerea unui drum de la rădăcină la un vîrf de pe frontieră definit astfel:

1. Inițial se pleacă din rădăcină.
2. Presupunem ca vîrful curent este $i ? j$. Dacă $a_i < a_j$, atunci fiul din stânga lui $i ? j$ devine vîrf curent; altfel, fiul din dreapta devine vîrf curent.
3. Calculul se oprește, dacă vîrful curent este pe frontieră.

Definiția 4.4. Fie t un arbore de decizie pentru n elemente. Spunem că t rezolvă problema sortării, dacă pentru orice intrare $a = (a_0, \dots, a_{n-1})$, calculul lui t pentru a se termină într-un vîrf etichetat cu permutarea π , care are proprietatea $a_{\pi(0)} < \dots < a_{\pi(n-1)}$. Un arbore de decizie care rezolvă problema sortării va mai fi numit și arbore de decizie pentru sortare, iar modelul de calcul va fi numit modelul arborilor de decizie pentru sortare.



Sortare prin selecție naivă



Sortare prin inserție

Figura 4.4: Arbori de decizie pentru sortare

Putem defini acum timpul de execuție minim pentru cazul cel mai nefavorabil prin expresia:

$$T(n) = \min_t \max_{\pi} \text{Lung}(\pi, t),$$

unde $\text{Lung}(\pi, t)$ reprezintă drumul de la rădăcină la vârful pe frontieră etichetat cu π în arborele de decizie pentru sortare t .

Teorema 4.6. Problema sortării are timpul de execuție pentru cazul cel mai nefavorabil $\Omega(n \log n)$ în modelul arborilor de decizie pentru sortare.

Demonstrație. Un arbore de decizie pentru n elemente, care rezolvă problema sortării, are $n!$ vârfuri pe frontieră. Un arbore de înălțime k are cel mult 2^k vârfuri pe frontieră. De aici rezultă:

$$2^{T(n)} \geq n!$$

care implică $T(n) \geq \log_2(n!) = \Theta(n \log_2 n)$.

sfdem

Corolar 4.1. Algoritmul HeapSort este optimal în modelul arborilor de decizie pentru sortare.

4.1.7 Exerciții

Exercițiul 4.1.3. Se consideră un tablou de structuri statice ($a[i] \mid 0 \leq i < n$) și un al doilea tablou ($b[i] \mid 0 \leq i < n$) care conține o permutare a mulțimii $\{0, 1, \dots, n - 1\}$. Să se scrie un program care rearanjează componentele tabloului a conform permutării date de b .

Exercițiul 4.1.4. Să se modifice ShellSort astfel încât să utilizeze întotdeauna secvență de incremente (h_0, \dots, h_{k-1}) dată prin recurență: $h_0 = 1; h_{i+1} = 3h_i + 1$ și h_{k-1} cel mai mare increment de acest fel mai mic decât $\frac{n}{2}$. Să se încerce apoi găsirea de alte secvențe de incremente care să producă algoritmi de sortare mai eficienți.

Exercițiul 4.1.5. Să se scrie o variantă recursivă a algoritmului de inserare într-un heap intrInGr. Care dintre cele două variante este mai eficientă?

Exercițiul 4.1.6. Care este timpul de execuție al algoritmului heapSort dacă secvența de intrare este ordonată crescător? Dar dacă este ordonată descrescător?

Exercițiul 4.1.7. Să se proiecteze un algoritm care să determine cel de-al doilea cel mai mare element dintr-o listă. Algoritmul va executa $n + \lceil \log n \rceil - 2$ comparații. *Indicație.* Fie (a_0, \dots, a_{n-1}) secvența de intrare. Cel mai mare element din listă se poate face prin $n - 1$ comparații. Se va utiliza următoarea metodă pentru determinarea maximului:

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

Metodei de mai sus i se poate atașa un arbore binar complet de mărime n , fiecare vârf intern reprezentând o comparație, 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.

Exercițiul 4.1.8. (Selecție) Să se generalizeze metoda din exercițiul anterior pentru a determina cel de-al k -lea cel mai mare element dintr-o listă. Care este timpul de execuție al algoritmului? (Se știe că există algoritmi care rezolvă această problemă în timpul $\Theta(n + \min(k, n - k) \log n)$).

Exercițiul 4.1.9. (Sortare prin metoda turneeelor.) Să se utilizeze algoritmul din exercițiul precedent pentru sortarea unei liste. Care este timpul de execuție al algoritmului de sortare?

Exercițiul 4.1.10. 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 clasati relativ la relația de ordine „ a mai bun decât b ”, definită astfel:

- dacă a îl învinge pe b , atunci a este mai bun decât b ;
- dacă a este mai bun decât b și b mai bun decât c , atunci a este mai bun decât c .

4.2 Sortare prin distribuire

În următoarele două secțiuni vom studia algoritmi de sortare pentru cazurile când se cunosc informații suplimentare despre elementele care se sortează. Așa cum reiese și din denumire, algoritmii de sortare prin distribuire presupun cunoașterea de informații privind distribuția acestor elemente. Aceste informații sunt utilizate pentru a distribui elementele secvenței de sortat în „pachete” care vor fi sortate în același mod sau prin altă metodă, după care pachetele se combină pentru a obține lista finală sortată.

4.2.1 Sortarea cuvintelor

Presupunem că avem n fișe, iar fiecare fișă conține un nume ce identifică în mod unic fișa (cheia). Se pune problema sortării manuale a fișelor. Pe baza experienței câștigate se procedează astfel: se împart fișele în pachete, fiecare pachet conținând fișele ale căror cheie începe cu aceeași literă. Apoi se sortează fiecare pachet în aceeași manieră după a doua literă, apoi etc. După sortarea tuturor pachetelor, acestea se combină astfel încât formează o listă liniară sortată.

Vom încerca să formalizăm metoda de mai sus într-un algoritm de sortare a sirurilor de caractere (cuvinte). Presupunem că elementele secvenței de sortat sunt siruri de lungime fixată m definite peste un alfabet cu k litere. Echivalent, se poate presupune că elementele de sortat sunt numere reprezentate în baza k . Din acest motiv, sortarea cuvintelor este denumită în engleză radix-sort (cuvântul radix traducându-se prin bază).

Dacă urmărm̄ ideea din exemplul cu fișele, atunci algoritmul ar putea fi descris recursiv astfel:

1. Se împart cele n cuvinte în k pachete, cuvintele din același pachet având aceeași literă pe poziția i (numărând de la stânga la dreapta).
2. Apoi fiecare pachet este sortat în aceeași manieră după literele de pe pozițiile $i + 1, \dots, m - 1$.

3. Se concatenează cele k pachete în ordinea dată de literele de pe poziția i . Lista obținută este sortată după subcuvintele formate din literele de pe pozițiile $i, i+1, \dots, m-1$.

Inițial se consideră $i = 0$. Apare următoarea problemă. Un grup de k pachete nu va putea fi combinat într-o listă sortată decât dacă cele k pachete au fost sorteate complet pentru subcuvintele corespunzătoare. Este deci necesară ținerea unei evidențe a pachetelor, fapt care conduce la utilizarea de memorie suplimentară și creșterea gradului de complexitate a metodei. O simplificare majoră apare dacă împărțirea cuvintelor în pachete se face parcurgând literele acestora de la dreapta la stânga. Procedând aşa, observăm următorul fapt surprinzător: după ce cuvintele au fost distribuite în k pachete după litera de pe poziția i , cele k pachete pot fi combinate înainte de a le distribui după litera de pe poziția $i-1$.

Exemplu. Presupunem că alfabetul este $\{0 < 1 < 2\}$ și $m = 3$. Cele trei fazuri care cuprind distribuirea elementelor listei în pachete și apoi concatenarea acestora într-o singură listă sunt sugerate grafic în figura 4.5.

sfex

Observație. Atunci când se face distribuirea cuvintelor în pachete, ordinea apariției cuvintelor într-un pachet trebuie să coincidă cu ordinea din lista inițială. Dacă distribuirea se face după cheia i și două chei au pe poziția i aceeași literă, atunci o cheie o va preceda pe celalaltă în pachet dacă și numai dacă o precedă și în secvența inițială (cea supusă sortării). Această proprietate este un caz particular al celei cunoscute sub numele de *stabilitate a algoritmilor de sortare*.

sfobs

Pentru gestionarea pachetelor vom utiliza un tablou de pointeri numit *pachet*, cu semnificația că elementul *pachet[i]* face referire la primul element din lista ce reprezintă pachetul i . Subetapa de distribuire este realizată în modul următor: Inițial, se consideră listele *pachet[i]* vide. Apoi se parcurge secvențial lista supusă distribuirii și fiecare element al acesteia este distribuit în pachetul corespunzător. Subetapa de combinare a pachetelor constă în concatenarea celor k liste *pachet[i]*, $i = 0, \dots, k-1$. Pentru ca operația de inserare a unei noi chei într-un pachet să se realizeze ușor, este util să folosim încă un tablou de pointeri *ultim* cu semnificația că *ultim[i]* face referire la ultimul element din lista corespunzătoare pachetului i . De asemenea, tabloul *ultim* va fi utilizat cu succes și la concatenarea listelor. Descrierea procedurală a algoritmului este următoarea:

```

procedure radixSort(a,n)
begin
    for i ← m-1 downto 0 do
        for j ← 0 to k-1 do
            pachet[j] ← listaVida()
            while (not esteVida(L)) do
                w ← citeste(L, 0)
                elimina(L, 0)
                insereaza(pachet[w[i]], lung(pachet[w[i]]), w)
            for j ← 0 to k-1 do
                concateneaza(L, pachet[j])
    end

```

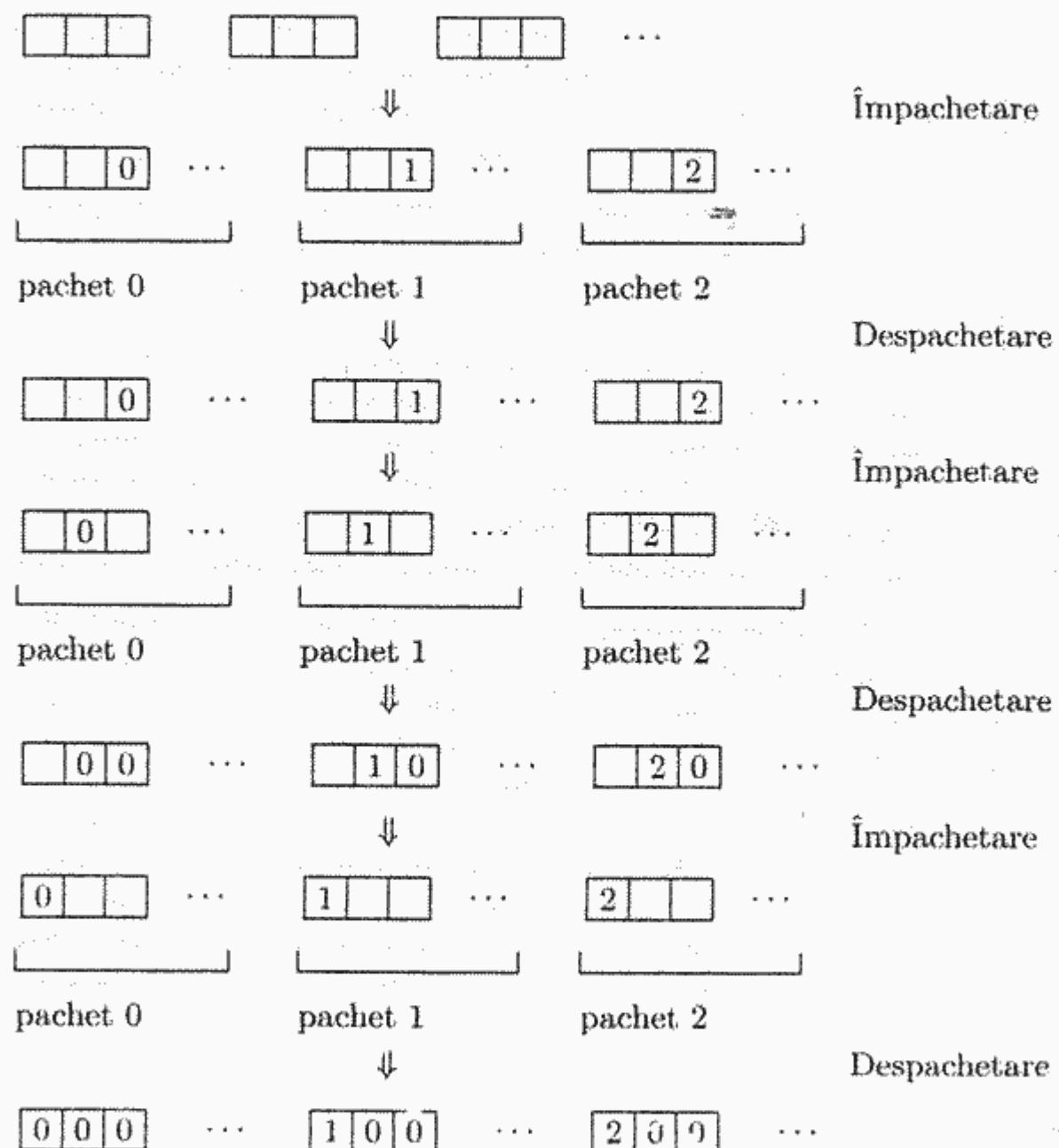


Figura 4.5: Sortare prin distribuire

Procedura concateneaza(L1, L2) concatenează listele L1 și L2 și memorează rezultatul în L1.

Evaluarea algoritmului radixSort. Distribuirea în pachete presupune parcurgerea completă a listei de intrare, iar procesarea fiecărui element al listei necesită $O(1)$ operații. Rezultă că faza de distribuire se face în timpul $O(n)$, unde n este numărul de elemente din listă. Combinarea pachetelor presupune o parcursere a tabloului pachet, iar adăugarea unui pachet se face cu $O(1)$ operații, cu ajutorul tabloului **ultim**. Astfel, faza de combinare a pachetelor necesită $O(k)$ timp. Rezultă că algoritmul RadixSort are un timp de execuție de $O(m \cdot n)$. De remarcat că, atunci când m are valori foarte mici comparativ cu n ($O(\frac{n}{m}) = \Omega(n)$), se obține o limită inferioară a valorii optime pentru modelul arborilor de decizie. Această îmbunătățire este posibilă datorită informației suplimentare despre organizarea cheilor supuse sortării.

4.2.2 Distribuire prin segmentare

Algoritmul de sortare care utilizează distribuirea prin segmentare (în engleză *bucket-sort*) folosește faptul că secvența de intrare este furnizată de un proces aleator care distribuie elementele uniform peste intervalul $[0, 1]$. Așadar, presupunem că elementele de sortat sunt numere din acest interval. Ideea algoritmului este următoarea:

- Se divide intervalul $[0, 1]$ în n subintervale de mărimi egale.
- Se distribuie cele n intrări în n pachete, un pachet conținând numerele ce aparțin unui subinterval. Pachetul căruia îi aparține intrarea a_i este determinat de valoarea expresiei $[n \cdot a_i] + 1$, dacă pachetele sunt numerotate de la 1 la n . Deoarece intrările sunt distribuite uniform peste intervalul $[0, 1]$, este de așteptat ca un pachet să nu conțină prea multe numere.
- Se sortează fiecare pachet utilizând o altă metodă, de exemplu, sortarea prin inserție directă.
- Se combină cele n pachete într-o listă sortată.

Exercițiul 4.2.1. Presupunem că lista de intrare este reprezentată de un tablou. Să se scrie un subprogram BucketSort care descrie algoritmul de sortare de mai sus. Pachetele vor fi gestionate cu liste înlățuite, ca în cazul algoritmului RadixSort. Sortarea unui pachet se va face prin inserție directă.

Vom arăta că algoritmul BucketSort are timpul mediu de execuție $O(n)$. Notăm cu n_i variabila aleatoare care, pentru o intrare dată, dă numărul de elemente aparținând intervalului i . Timpul de execuție al algoritmului de sortare prin inserție directă este $O(n^2)$ și deci timpul mediu de execuție pentru sortarea pachetului i este $M[O(n_i^2)] = O(M[n_i^2])$. Rezultă că algoritmul BucketSort are timpul mediu de execuție $\sum_{i=1}^n O(M[n_i^2]) = O(\sum_{i=1}^n M[n_i^2])$. Valorile posibile pentru variabila aleatoare n_i sunt $0, 1, \dots, n$, iar probabilitatea ca $n_i = k$ este $p_k = C_n^k (\frac{1}{n})^k (1 - \frac{1}{n})^{n-k}$ (distribuție binomială). Rezultă că media variabilei aleatoare n_i este $M[n_i] = \sum_{k=0}^n k \cdot p_k = n \cdot \frac{1}{n} = 1$. Pentru calculul mediei variabilei aleatoare n_i^2 vom utiliza formula $M[n_i^2] = D[n_i] + M^2[n_i]$, unde $D[n_i]$ este dispersia variabilei aleatoare n_i .

Se știe că dispersia unei variabile aleatoare cu distribuție binomială este $np(1 - p)$, de unde $D[n_i] = n \frac{1}{n} (1 - \frac{1}{n}) = 1 - \frac{1}{n}$. Rezultă $M[n_i^2] = 2 - \frac{1}{n} = \Theta(1)$, care implica $O(\sum_{i=1}^n M[n_i^2]) = O(\sum_{i=1}^n \Theta(1)) = O(n)$. De aici timpul mediu de execuție a algoritmului BucketSort este $O(n)$.

4.2.3 Exerciții

Exercițiul 4.2.2. Să se descrie un algoritm care sortează cuvinte, astfel încât distribuirea în pachete să se facă prin parcurgerea cuvintelor de la stânga la dreapta. Se presupune că lista cuvintelor este reprezentată în memorie printr-un tablou. Algoritmul va utiliza un singur tablou suplimentar de lungime n și un singur tablou suplimentar de lungime k și va avea timpul de execuție $O(m \cdot n)$.

Exercițiul 4.2.3. Să se aplique RadixSort la sortarea numerelor întregi, ținând cont de reprezentarea lor în memoria calculatorului.

Indicație. Reprezentarea binară a unui număr este divizată, de exemplu, în patru părți. Astfel un întreg poate fi văzut ca o cheie formată din patru „litere” (numere aparținând unui domeniu restrâns). Să se testeze eficiența nouului algoritm de sortare cu cele ale algoritmilor bazați pe comparații.

Exercițiul 4.2.4. [CLR93] Să se utilizeze inducția pentru a dovedi că RadixSort rezolvă corect problema sortării. Unde este utilizată proprietatea că sortările intermedie sunt stabile?

Exercițiul 4.2.5. Să se scrie o variantă recursivă a algoritmului RadixSort în care ordonarea cuvintelor după literele de pe o anumită poziție se face prin interschimbare: se pleacă cu un indice i din stânga și cu un indice j din dreapta și se înaintează alternativ până când se întâlnește o inversiune. Se rezolvă inversiunea prin interschimbare după care procesul continuă până se parcurge toată secvența (a se vedea și pasul de divizare de la QuickSort din secțiunea 10.3). Mai există diferențe semnificative între cele două moduri de considerare a literelor după care se face ordonarea: de la stânga la dreapta și de la dreapta la stânga?

Exercițiul 4.2.6. [CLR93] Să se proiecteze un algoritm care să sorteze n întregi din intervalul $[0, n^2]$ în timpul $O(n)$.

Indicație. Numerele din intervalul $[0, n^2]$ pot fi scrise ca secvențe de două cifre în baza $\log_2 n$ și se aplică algoritmul RadixSort.

Exercițiul 4.2.7. Să se arate că algoritmul BucketSort rezolvă corect problema sortării.

Exercițiul 4.2.8. Care este timpul de execuție al algoritmului BucketSort pentru cazul cel mai nefavorabil? Poate fi modificat BucketSort pentru a avea un timp de execuție de $O(n \log n)$ pentru cazul cel mai nefavorabil? (Modificarea va trebui să păstreze timpul mediu de execuție liniar).

4.3 Sortare prin numărare

Presupunem că tipul `TElement`, peste care sunt definite secvențele supuse sortării, conține numai două elemente: *alb < negru*. Sortarea unui tablou `a : array[1..n]`

ofTElement este foarte simplă: se determină numărul n_1 de elemente din a egale cu *alb* și numărul n_2 de elemente egale cu *negru*. Aceasta se poate face în timpul $\Theta(n)$. Apoi se pun în tablou n_1 elemente *alb* urmate de n_2 elemente *negru*. Și cea de-a doua etapă necesită $\Theta(n)$ timp. Rezultă un algoritm de sortare cu timpul de execuție $\Theta(n^2)$. Și de această dată, obținerea unui algoritm de sortare liniar a fost posibilă datorită faptului că s-au cunoscut informații suplimentare despre elementele supuse sortării, anume că multimea univers conține numai două elemente. Algoritmul de sortare prin numărare are ca punct de plecare observația de mai sus și se utilizează atunci când multimea univers conține puține elemente. Prin simplificare, presupunem că $U = \{0, 1, \dots, k-1\}$ cu relația de ordine naturală. Algoritmul de la începutul secțiunii are un defect: dacă secvența de sortare este una de structuri statice și sortarea se face după chei, atunci etapa a doua a algoritmului devine inefективă. În plus, algoritmul nu are proprietatea de stabilitate. Vom modifica algoritmul astfel încât elementele tabloului de intrare a să fie transferate direct în tabloul de ieșire b pe locurile lor finale. Cum poate fi utilizată numărarea pentru a determina locurile elementelor în tabloul sortat? Revenim la exemplul de la începutul secțiunii. Observăm că elementul *negru* aflat cel mai la dreapta în a are poziția $n_1 + n_2$ în b , următorul va avea poziția $n_1 + n_2 - 1$ și așa mai departe. Elementul *alb* aflat cel mai la dreapta în a are poziția n_1 în b , următorul poziția $n_1 - 1$ și așa mai departe. Presupunem existența unui vector p cu valoarea inițială $p[\text{alb}] = n_1, p[\text{negru}] = n_1 + n_2$. Etapa a doua a algoritmului va parcurge tabloul a de la dreapta la stânga și va păstra invariantă următoarea proprietate: $p[\text{alb}]$ va fi poziția primului element *alb* ce va fi întâlnit și $p[\text{negru}]$ va fi poziția primului element *negru* ce va fi întâlnit. Această proprietate poate fi păstrată prin decrementarea componentei din p imediat după ce elementul corespunzător din a a fost transferat. Prima etapă a algoritmului va determina vectorul p unde $p[i] =$ numărul de elemente din a mai mici decât sau egale cu i .

```

procedure sortNumarare(a, b, n)
begin
    for i ← 0 to k-1 do
        p[i] ← 0
    for j ← 0 to n-1 do
        p[a[j]] ← p[a[j]]+1
    for i ← 1 to k-1 do
        p[i] ← p[i-1] + p[i]
    for j ← n-1 downto 0 do
        i ← a[j]
        b[p[i]-1] ← i
        p[i] ← p[i]-1
end

```

Determinarea tabloului p necesită $O(k + n)$ timp, iar transferul $O(n)$ timp. Rezultă că algoritmul `sortNumarare` are timpul de execuție $O(n + k)$. În practică, algoritmul este aplicat pentru $k = O(n)$, caz în care rezultă un timp de execuție liniar a pentru `sortNumarare`.

4.3.1 Exerciții

Exercițiu 4.3.1. Să se arate că algoritmul `sortNumarare` este stabil.

Exercițiu 4.3.2. Presupunem că în etapa a doua a algoritmului `sortNumarare` se înlocuiește `for j ← n-1 downto 0 cu for j ← 0 to n-1`. Să se arate că și noul algoritm rezolvă corect problema sortării. Mai este noul algoritm stabil?

Exercițiu 4.3.3. [CLR93] Să se proiecteze un algoritm care, pentru o secvență de n întregi aparținând intervalului $[0, k - 1]$, preprocesează secvența în timpul $O(n + k)$, după care răspunde la întrebări de formă „câte elemente din secvență sunt într-un interval $[a, b]$ ” în timpul $O(1)$.

Exercițiu 4.3.4. Presupunem că despre secvența de intrare se cunoaște numai informația că elementele sunt distințe două câte două. Să se proiecteze un algoritm bazat pe numărare care să sorteze o astfel de secvență. Care este timpul de execuție al algoritmului proiectat? Mai este posibilă proiectarea unui algoritm de sortare liniar utilizând numai această informație suplimentară despre datele de intrare?

4.4 Sortare topologică

Presupunem acum că relația de ordine este parțială. Fie, de exemplu, $a_1 < a_0, a_1 < a_2 < a_3$. Problema constă în a crea o listă liniară care să fie compatibilă cu relația de ordine: dacă $a_i < a_j$, atunci a_i va precede pe a_j în lista finală. Pentru exemplul nostru, lista liniară finală va putea fi (a_1, a_0, a_2, a_3) , sau (a_1, a_2, a_0, a_3) , sau (a_1, a_2, a_3, a_0) .

Definiția 4.5. Fie (S, \leq) o mulțime parțial ordonată finită și $a = (a_0, a_1, \dots, a_{n-1})$ o liniarizare a sa. Spunem că secvența a este sortată topologic, dacă $\forall i, j : a_i < a_j \Rightarrow i < j$.

Teorema 4.7. Orice mulțime parțial ordonată finită (S, \leq) poate fi sortată topologic.

Demonstrație. Vom extinde relația \leq la o relație de ordine totală. Fie elementele distințe $a, b \in S$ ce nu pot fi comparate, i.e., nu are loc nici $a < b$ și nici $b < a$. Extindem relația $<$, considerând $x < y$ pentru orice x cu $x \leq a$ și orice y cu $b \leq y$. Procedeul continuă până când sunt eliminate toate perechile incomparabile. O mulțime total ordonată poate fi sortată, iar secvența sortată respectă ordinea parțială. sfdein

Există o legătură strânsă între mulțimile parțial ordonate finite și digrafurile aciclice (digrafuri fără circuite numite pe scurt dag-uri). Orice mulțime parțial ordonată (S, \leq) definește un dag $D = (S, A)$, unde există arc de la a la b , dacă $a < b$ și nu există $c \in S$ cu proprietatea $a < c < b$. Reciproc, orice dag $D = (V, A)$ definește o relație de ordine parțială \leq peste V , dată prin: $u \leq v$, dacă există un drum de lungime ≥ 0 de la u la v . De fapt, \leq este închiderea reflexivă și tranzitivă a lui A (se mai notează $\leq = A^*$). Sortarea topologică a unui dag constă în crearea unei liste liniare a vîrfurilor cu următoarea proprietate: dacă există arc de la u la v , atunci u precedă pe v în listă, pentru oricare două vîrfuri u și v . Vîrfurile care

candidatează pentru primul loc în lista sortată topologic au proprietatea că nu există arce incidente spre interior (care sosesc în acel vârf) și se numesc *surse*.

Există mai multe metode de a sorta topologic un dag. Prima are ca punct de plecare algoritmul de explorare DFS a unui graf. Reamintim că în timpul explorării DFS, un vârf poate fi întâlnit de mai multe ori. Notăm cu $f[v]$ momentul când vârful v este întâlnit ultima dată (când lista de adiacență este epuizată). Algoritmul este descris schematic astfel:

- apelează algoritmul DFS pentru a determina momentele de terminare $f[v]$ pentru orice vârf v ;
- se fiecare dată când un vârf este terminat este adăugat la începutul unei liste înlățuite;
- lista înlățuită finală va conține o sortare topologică a vâfurilor.

Corectitudinea algoritmului se bazează pe următorul rezultat.

Lema 4.1 ([CLR93]) *Un digraf D este aciclic dacă și numai dacă explorarea DFS nu produce arce înapoi.*

Exercițiul 4.4.1. Să se scrie un program **sortareTopologicaDFS(D, S)** care implementează algoritmul de mai sus. Să se arate că algoritmul are timpul de execuție $\Theta(\#V + \#A)$.

Cea de-a doua metodă pe care o studiem aici este o generalizare a explorării BFS. Presupunem că pentru dag-ul D sunt create atât listele de adiacență interioară, cât și cele de adiacență exterioară. Listele de adiacență interioară vor fi utilizate la determinarea surselor. Ca și BFS, algoritmul de sortare topologică va utiliza o coadă.

- inițializează coada cu vâfurile sursă (cele care au liste de adiacență interioară vide);
- extrage un vârf u din coadă pe care-l adaugă la lista sortată parțial;
- elimină din reprezentarea (acum parțială) a lui D vârful u și toate arcele (u, v) . Dacă pentru un astfel de arc lista de adiacență interioară a vârfului v devine vidă, atunci v va fi adăugat la coadă;
- repetă pașii 2 și 3 până când coada devine vidă.

Extindem structura D , care reprezintă digraful D , cu tabloul $np[1..n]$; $D,np[u]$ va conține numărul succesorilor vârfului u . Subprogramul care implementează metoda de sortare topologică bazată pe explorarea BFS este următorul:

```

function sortareTopologicaBFS(D,np)
    coadaVida(C) //initializeaza coada C
    for u ← 0 to D.n-1 do
        // insereaza in C varfurile fara predecesor
        if D,np[u]=0 then insereaza(C,u)
    for k ← 0 to D.n-1 do
        // afiseaza varfurile in ordine topologica
        if EsteVida(C)
            then return ("Graful contine cicluri")
        u ← elmina(C)
        inserează(L, lung(1), u)
    
```

```

p ← D.a[u]
while p≠NULL do
    v← p->elt //v este un succesor imedial al lui u
    D.np[v] ← D.np[v]-1
    if D.np[v]=0
        then insereaza(C,v)
    p ← p->succ
end

```

4.4.1 Exerciții

Exercițiul 4.4.2. Să se compare eficiența algoritmului `sortareTopologicaBFS` cu cea a algoritmului `sortareTopologicaDFS`. De asemenea, să se compare ordinea vârfurilor în listele determinate de cei doi algoritmi.

Exercițiul 4.4.3. Ce se întâmplă cu cei doi algoritmi dacă digraful de intrare are circuite?

4.5 Referințe bibliografice

Acest capitol cuprinde numai o parte dintre algoritmii de sortare internă. Alți doi algoritmi importanți, sortarea prin interclasare și sortarea rapidă, sunt prezențați în capitolul dedicat metodei *divide-et-impera*. O prezentare aproape completă (dar care trebuie actualizată) este [Knu76]. Algoritmii de sortare, datorită importanței lor, sunt prezențați în toate cărțile dedicate studierii algoritmilor [Sed88, CLR93, CLR00, Baa78, BG00, Mel84a, AHU74, HS84, Hei96, KGGK94, MS91, LG86]. Această lucrare nu include algoritmi de sortare externă. Pentru cititorul interesat recomandăm [Knu76, Sed88, Baa78].

Capitolul 5

Căutare

Alături de sortare, căutarea în diferite structuri de date constituie una dintre operațiile cele mai des utilizate în activitatea de programare. Problema căutării poate fi formulată în diverse moduri:

- dat un tablou ($s[i] \mid 0 \leq i < n$) și un element a , să se decidă dacă există $i \in \{0, \dots, n-1\}$ cu proprietatea $s[i] = a$;
- dată o structură înăntărită (liniară, arbore, etc.) și un element a , să se decidă dacă există un nod în structură a cărui informație este egală cu a ;
- dat un fișier și un element a , să se decidă dacă există o componentă a fișierului care este egală cu a etc.

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

- informațiile din componente sunt distințe două câte două sau nu;
- componentele sunt ordonate în conformitate cu o relație de ordine peste mulțimea informațiilor sau nu;
- căutarea se poate face pentru toată informația memorată într-o componentă a structurii sau numai pentru o parte a sa numită *cheie*:
 - cheile pot fi unice (ele identifică în mod unic componente) 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 operațiilor de inserare/ștergere (aspectul *dinamic*).

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

Instanță: o mulțime univers \mathcal{U} , o submulțime $S \subseteq \mathcal{U}$ și un element a din \mathcal{U} ;
Întrebare: $x \in S$?

Aspectul *static* este dat de cazul când, între oricare două căutări, mulțimea S nu suferă nici o modificare. Aspectul *dinamic* este obținut atunci când între două căutări mulțimea S poate face obiectul următoarelor operații:

- Inserare.

– *Intrare:* $S, x; \dots$
– *Iesire:* $S \cup \{x\}$.

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

Figura 5.1: Timpul de execuție pentru cazul cel mai nefavorabil

- Ștergere.

Intrare: $S, x;$
Ieșire: $S \setminus \{x\}$.

Evident, realizarea eficientă a căutării și, în cazul aspectului dinamic, a operațiilor de inserare și de ștergere, depinde de structura de date aleasă pentru reprezentarea mulțimii S .

5.1 Căutare în liste liniare

Mulțimea S este reprezentată printr-o listă liniară. Dacă mulțimea \mathcal{U} este total ordonată, atunci S poate fi reprezentată de o listă liniară ordonată. Algoritmii corespunzători celor trei operații au fost deja prezențați în secțiunile 2.1, respectiv 2.2. Timpul de execuție pentru cazul cel mai nefavorabil este dependent de implementarea listei. Tabelul 5.1 include un sumar al valorilor acestor timpi de execuție. Facem observația că valorile pentru operațiile de inserare și ștergere nu presupun și componenta de căutare. În mod obișnuit, un element x este adăugat la S numai dacă el nu apare în S ; analog, un element x este șters din S numai dacă el apare în S . Deci ambele operații ar trebui precedate de căutare. În acest caz, la valorile timpilor de execuție pentru inserare și ștergere se adaugă și valoarea corespunzătoare pentru căutare. De exemplu, timpul de execuție 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 aceeași, $O(n)$.

Pentru calculul timpului mediu de execuție vom presupune că $a \in S$ cu probabilitatea q și că a poate apărea în S la adresa adr cu aceeași probabilitate $\frac{q}{n}$. Timpul mediu de execuție al căutărilor cu succes (a este găsit în S) este:

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

iar în cazul general avem:

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

Cazul când se ia în considerare frecvența căutărilor. Presupunem că x_i este căutat cu frecvența f_i . Se poate demonstra că se obține o comportare în medie bună, atunci când $f_1 \geq \dots \geq f_n$. Dacă aceste frecvențe nu se cunosc aprioric, se pot utiliza *tablourile cu autoorganizare*. Într-un tablou cu autoorganizare, ori

de câte ori se caută un $a = s[i]$, acesta este deplasat la începutul tabloului în modul următor: elementele de pe pozițiile $1, \dots, i - 1$ sunt deplasate la dreapta cu o poziție după care se pune a pe prima poziție. Dacă în loc de tablouri se utilizează liste înlăntuite, atunci deplasările la dreapta nu mai sunt necesare. Se poate arăta [Knu76] că pentru tablourile cu autoorganizare timpul mediu de execuție este:

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

5.2 Modelul de calcul al arborilor de decizie pentru căutare

În această secțiune definim o clasă de algoritmi bazați pe paradigma *divide-et-impera* și arătăm că algoritmul de căutare binară este optim în această clasă.

Presupunem că mulțimea S este reprezentată de tabloul $(s[i] \mid 0 \leq i \leq n - 1)$. Mai întâi generalizăm problema presupunând că se caută a în secvența $(s[p], \dots, s[q])$. Reamintim că are loc $s[p] < \dots < s[q]$. Algoritmii de căutare bazați pe paradigma *divide-et-impera* [capitolul 10] au o descriere recursivă definită după următoarea strategie:

- se determină m cu $p \leq m \leq q$;
- dacă $a = s[m]$, atunci căutarea se termină cu succes;
- dacă $a < s[m]$, atunci căutarea continuă cu subsecvența $(s[p], \dots, s[m - 1])$;
- dacă $a > s[m]$, atunci căutarea continuă cu subsecvența $(s[m + 1], \dots, s[q])$.

O schemă procedurală nerecursivă care descrie strategia de mai sus este următoarea:

```

function poz(s, n, a)
    1: p ← 0      q ← n-1
    2: alege m intre p si q
    3: while ((a ≠ s[m]) and (p < q)) do
        4: if (a < s[m])
            then q ← m-1
            else p ← m+1
        5: alege m intre p si q
    6: if (a = s[m])
        then return m
        else return -1
    end

```

În funcție de modul de alegere a valorii m prin instrucțiunile 2 și 5, se disting mai mulți algoritmi de căutare. Cei mai cunoscuți dintre aceștia sunt:

- *căutare liniară* (secvențială). Se alege $m = p$;
- *căutare binară*. Se alege $m = \lceil \frac{p+q}{2} \rceil$;
- *căutare Fibonacci*. Se presupune $q + 1 - p = Fib(k) - 1$, unde $Fib(k)$ este la k -lea număr Fibonacci. Se alege m astfel încât $m - p = Fib(k - 1) - 1$ și $q - m = Fib(k - 2) - 1$.

Numărul de comparații pentru cazul cel mai nefavorabil este:

- căutare liniară: $O(n)$.
- căutare binară: $O(\log_2 n)$.
- căutare Fibonacci: $O(\log_2 n)$

Pentru a studia timpul mediu de execuție și criterii de optimalitate este necesar să precizăm formal modelul de calcul. Acesta corespunde arborilor de decizie pentru căutare definiți de schema procedurală de mai sus.

Definiția 5.1. Arborele de decizie pentru căutare de dimensiune n atașat unui algoritm bazat pe metoda divide-et-impera este definit după cum urmează:

- mai întâi se definește recursiv arborele $T(p, q)$ astfel:
 - dacă $p > q$ atunci $T(p, q)$ este arborele vid;
 - altfel, rădăcina este m calculată de algoritmul utilizat de instrucțiunea 2 sau 5, iar subarborele stâng este $T(p, m - 1)$ și cel drept este $T(m + 1, q)$;
- arborele de decizie pentru căutare de dimensiune n este $T(0, n - 1)$ la care se adaugă vârfurile externe având ca etichete intervalele $(-\infty, X_0), (X_0, X_1), \dots, (X_{n-1}, +\infty)$ în această ordine de la stânga la dreapta, unde X_0, \dots, X_{n-1} sunt n variabile.

Calculul unui arbore de decizie pentru intrarea x_0, \dots, x_{n-1}, a , unde $x_0 < \dots < x_{n-1}$, constă în:

1. etichetarea nodurilor interne cu x_0, \dots, x_{n-1} , astfel încât lista inordine ne dă ordinea crescătoare a etichetelor și
2. parcurgerea unui drum de la rădăcină spre frontieră determinat astfel: dacă vârful curent v este etichetat cu x_m (vom vedea că m este dat de instrucțiunea 2 sau 5 din schema procedurală divide-et-impera), atunci:
 - a) dacă v este vârf extern etichetat (X_i, X_{i+1}) , atunci $a \in (x_i, x_{i+1})$ (variabila X_i este interpretată ca având valoarea x_i) și calculul se termină cu insucces;
 - b) dacă $a = x_m$, atunci calculul se termină cu succes;
 - c) dacă $a < x_m$, atunci rădăcina subarborelui stâng devine vârf curent;
 - d) dacă $a > x_m$, atunci rădăcina subarborelui drept devine vârf curent.

Exemplu. Arborele corespunzător căutării liniare pentru $n = 4$ este reprezentat în figura 5.2, iar cel corespunzător căutării binare pentru $n = 6$ este reprezentat în figura 5.3.

sfex

Vârfurile de pe frontieră unui arbore de decizie pentru căutare se mai numesc și *vârfuri pendante*. În continuare prezentăm câteva proprietăți ale arborilor de decizie pentru căutare.

Lema 5.1. Fie t arborele de decizie pentru căutare cu n vârfuri corespunzător căutării binare. Dacă $2^{h-1} \leq n < 2^h$, atunci înălțimea lui t este h .

Demonstrație. Procedăm prin inducție după n . Dacă $n = 1$, atunci afirmația din lemură este evident adevarată. Presupunem $n > 1$. Valoarea m corespunzătoare rădăcinii este $\lceil \frac{n}{2} \rceil$. Din definiția părții întregi superioare rezultă următoarele inegalități:

$$2^{h-2} \leq m < 2^{h-1} + 1. \quad (5.1)$$

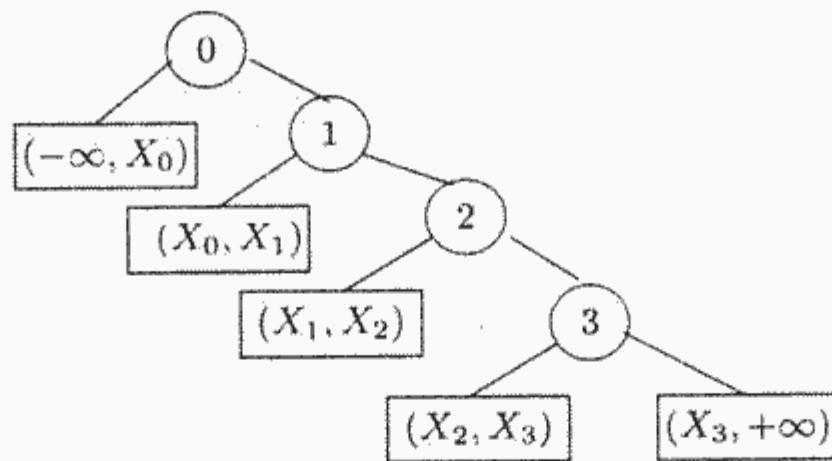


Figura 5.2: Arborele corespunzător căutării liniare

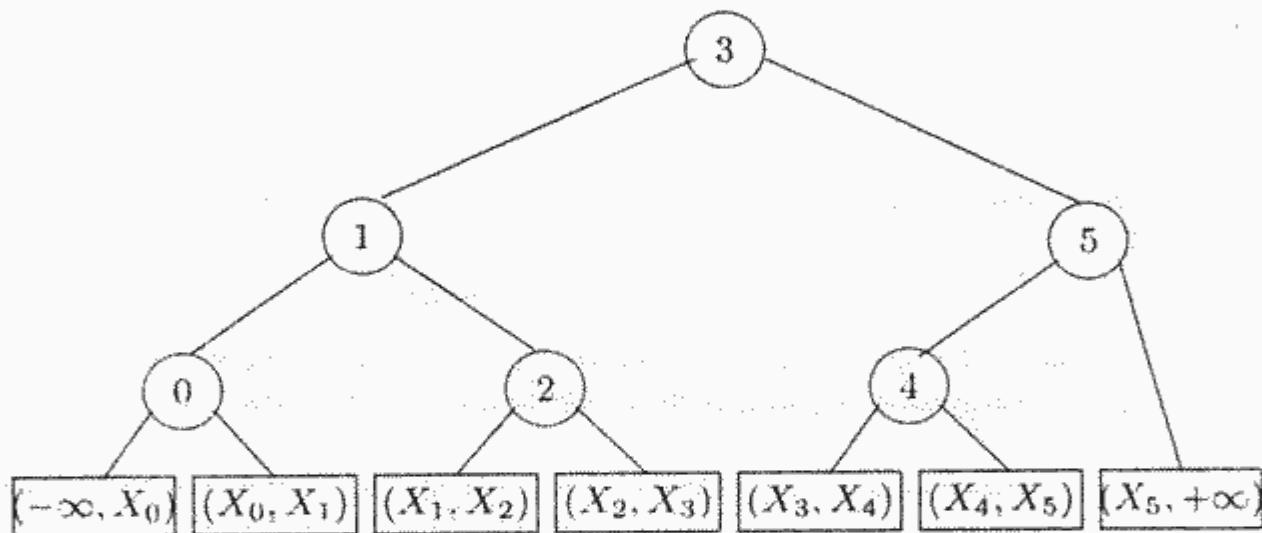


Figura 5.3: Arborele corespunzător căutării binare

Subarborii rădăcinii au m și respectiv $n - m - 1$ vârfuri cu proprietatea $m - 1 \leq n - m - 1 \leq m$. Dacă $m = n - m - 1$, atunci $m = \frac{n-1}{2}$. Deoarece $n \leq 2^h - 1$, rezultă că $m \leq 2^{h-1} - 1 < 2^{h-1}$. Dacă $m - 1 = n - m - 1$, atunci $m = \frac{n}{2} < 2^{h-1}$. Rezultă $m < 2^{h-1}$ în toate cazurile. Aplicând ipoteza inductivă, rezultă că subarborele cel mai înalt (cel cu m vârfuri și aflat la stânga rădăcinii) are înălțimea $h - 1$. Din definiția înălțimii arborelui binar, rezultă că înălțimea lui t este h . sfdem

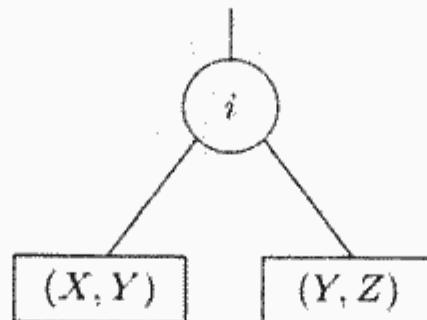
Corolar 5.1. *Timpul de execuție pentru cazul cel mai nefavorabil al căutării binare este $O(\log_2 n)$.*

Definiția 5.2. *Fie t un arbore de decizie pentru căutare. Lungimea internă a lui t , notată $\text{LungInt}(t)$, este suma lungimilor drumurilor de la rădăcină la vârfurile interne. Lungimea externă a lui t , notată $\text{LungExt}(t)$, este suma lungimilor drumurilor de la rădăcină la vârfurile de pe frontieră (pendante).*

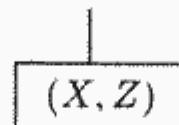
Lema 5.2. *Fie t un arbore de decizie pentru căutare cu n vârfuri interne. Atunci:*

$$\text{LungExt}(t) - \text{LungInt}(t) = 2n.$$

Demonstrație. Procedăm prin inducție după n . Pentru $n = 1$ avem $\text{LungInt}(t) = 0$ și $\text{LungExt}(t) = 2$. Presupunem $n > 1$. Fie i un vârf intern cu ambii fi pe frontieră. Înlocuim subarborele:



cu subarborele:



Noul arbore t' este un arbore de decizie cu $n - 1$ vârfuri interne și conform ipotezei inducitive avem

$$\text{LungExt}(t') - \text{LungInt}(t') = 2(n - 1).$$

Deoarece $\text{LungExt}(t) = \text{LungExt}(t') + k + 2$ și $\text{LungInt}(t) = \text{LungInt}(t') + k$, unde k este lungimea drumului de la rădăcină la i , rezultă:

$$\begin{aligned} \text{LungExt}(t) - \text{LungInt}(t) &= \text{LungExt}(t') + k + 2 - (\text{LungInt}(t') + k) \\ &= 2n - 2 + k + 2 - k \\ &= 2n. \end{aligned}$$

sfdem

Lema 5.3. Lungimea internă minimă a unui arbore de decizie cu n vârfuri interne este:

$$(n + 1)(h - 1) - 2^h + 2$$

unde $h = \lceil \log_2(n + 1) \rceil$.

Demonstrație. Considerăm formula:

$$x + x^2 + \cdots + x^k = \frac{x^{k+1} - x}{x - 1}.$$

Derivăm:

$$1 + 2x + \cdots + kx^{k-1} = \frac{kx^{k+1} - (k + 1)x^k + 1}{(x - 1)^2}.$$

Înmulțim cu x :

$$x + 2x^2 + \cdots + kx^k = x \cdot \frac{kx^{k+1} - (k + 1)x^k + 1}{(x - 1)^2}.$$

Luăm $k = h - 1$ și $x = 2$:

$$2 + 2 \cdot 2^2 + \cdots + (h-1) \cdot 2^{h-1} = 2^h(h-2) + 2.$$

Lungimea internă minimă a unui arbore de decizie este suma primilor n termeni din seria:

$$0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + \cdots$$

și corespunde arborelui binar complet. Dacă presupunem $n+1 = 2^h$, atunci această sumă este egală cu:

$$2 + 2 \cdot 2^2 + \cdots + (h-1)2^{h-1} = 2^h(h-2) + 2.$$

Pentru cazul general trebuie să scădem suma drumurilor care unesc rădăcina cu vârfurile de pe nivelul h care lipsesc:

$$2^h(h-2) + 2 - (2^h - 1 - n)(h-1) = (n+1)(h-1) - 2^h + 2.$$

sfdem

Corolar 5.2. Lungimea externă minimă a unui arbore de decizie este:

$$(n+1)(h+1) - 2^h.$$

sfdem

Demonstrație. Se aplică lemele precedente.

Teorema 5.1. Problema căutării are timpul de execuție în cazul cel mai nefavorabil $\Omega(\log n)$ în modelul arborilor de decizie pentru căutare.

Demonstrație. Arboarele binar cu lungimea internă minimă considerat în lema 5.3 are și înălțime minimă. În acest moment, concluzia teoremei rezultă din faptul că $2^{h-1} \leq n < 2^h$, unde n este numărul de noduri, iar h este înălțimea arborelui.

sfdem

Presupunem că orice x_i este căutat cu aceeași probabilitate $\frac{q}{n}$ și că a poate apartine oricărui interval (x_i, x_{i+1}) (presupunem $x_0 = -\infty, x_{n+1} = \infty$) cu aceeași probabilitate $\frac{1-q}{n+1}$. Cu aceste ipoteze, timpul mediu de execuție este dat de următoarea teoremă:

Teorema 5.2. Fie t un arbore de decizie pentru căutare. Timpul mediu de execuție a căutărilor este:

$$T_t^{med}(n) = \frac{2(n+q)}{n(n+1)} \cdot \text{LungExt}(t) + 1 - 4q.$$

Demonstrație. O căutare constă în parcurgerea unui drum de la rădăcină spre frontieră. Deoarece în fiecare nod se fac două comparații, iar la sfârșit încă o comparație, rezultă următoarea formulă pentru timpul mediu de execuție :

$$\begin{aligned} T_t^{med}(n) &= 1 + \frac{q}{n} \cdot 2 \cdot \text{LungInt}(t) + \frac{1-q}{n+1} \cdot 2 \cdot \text{LungExt}(t) \\ &= \dots \\ &= \frac{2(n+q)}{n(n+1)} \cdot \text{LungExt}(t) + 1 - 4q. \end{aligned}$$

sfdem

Teorema 5.3. *Căutarea binară este optimă din punctul de vedere al timpului mediu de execuție în modelul arborilor de decizie pentru căutare.*

Demonstrație. Se observă că arboarele de decizie asociat căutării binare are lungimea externă minimă.

sfdem

Corolar 5.3. *Timpul mediu de execuție pentru căutarea binară este $\Theta(\log_2 n)$.*

Corolar 5.4. *Problema căutării are timpul mediu de execuție $\Omega(\log n)$ în modelul arborilor de decizie pentru căutare.*

5.2.1 Căutare prin interpolare

Să presupunem că avem de căutat un cuvânt w într-un dicționar. În general, procedăm astfel: deschidem dicționarul la întâmplare. Dacă observăm că pagina la care am deschis conține cuvinte mai mici decât w (în ordinea lexicografică indusă de cea alfabetică), atunci vom sări un număr de pagini suficient de mare pentru a ne apropiia de cuvântul căutat w . În cazul în care pagina deschisă conține cuvinte mai mari se sare un număr de pagini înapoi. Procedeul continuă până când se găsește pagina care îl conține pe w . Numărul de pagini sărit la fiecare pas depinde de distanța la care se află w față de cuvintele de pe pagina curentă.

Metoda de căutare prin interpolare se bazează pe ideea de mai sus. Descrierea algoritmului este dată de schema de la metoda *divide-et-impera*, unde m se determină prin formula:

$$m = (p-1) + \left\lceil \frac{a - s[p-1]}{s[q+1] - s[p-1]} (q+1-p) \right\rceil.$$

Este necesar să considerăm două elemente artificiale $s[-1]$ și $s[n]$ cu $s[-1] < s[0]$ și $s[n-1] < s[n]$.

De remarcat că algoritmului de căutare prin interpolare nu-i putem atașa un arbore de decizie care să fie determinat numai de numărul n de elemente din S , deoarece acesta depinde și de modul de dispersie al elementelor din S . De aceea, instrumentele prin care se analizează această metodă diferă de cele de la *divide-et-impera*. Se poate alege S și a astfel încât căutarea prin interpolare să degenerizeze în căutare liniară. De aici rezultă că are un timp de execuție de $O(n)$ în cazul cel mai nefavorabil. În ipoteza că elementele x_0, \dots, x_{n-1} sunt repartizate uniform în intervalul $[x_{-1}, x_n]$ ($x_{-1} = s[-1]$, $x_n = s[n]$) rezultă un timp mediu de execuție $O(\log \log n)$ [Mel84a].

5.3 Arbore binari de căutare

Arborii $T(0, n-1)$ din definiția arborilor de decizie pentru căutare sunt transformați în structuri de date înălțătoare asemănătoare cu cele definite în secțiunea 5.2. Aceste structuri pot fi definite într-o manieră independentă:

Definiția 5.3. Un arbore binar de căutare este un arbore binar cu proprietățile:

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

În continuare descriem implementările celor trei operații peste această structură.

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

```
function poz(t, a)
    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
```

Funcția poz ia valoarea $NULL$, dacă $a \notin S$ și adresa nodului care conține pe a în caz contrar.

Operațiile de inserare și de ștergere trebuie să păstreze invariantă următoarea proprietate:

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

Pentru a realiza operația de inserare se caută intervalul căruia îi aparține x . Dacă în timpul procesului de căutare se găsește un nod $*p$ cu $p->elt = x$, atunci arboarele nu suferă nici o modificare (deoarece $x \in S$ implică $S \cup \{x\} = S$). Fie p adresa nodului de pe frontieră care definește 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 figura 5.4.

Algoritmul care realizează operația de inserare are următoarea descriere:

```
procedure insereaza(t, x)
    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

```

Functia nodNou() creeaza un nod al arborelui binar si intoarce adresa acestuia:

```

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

```

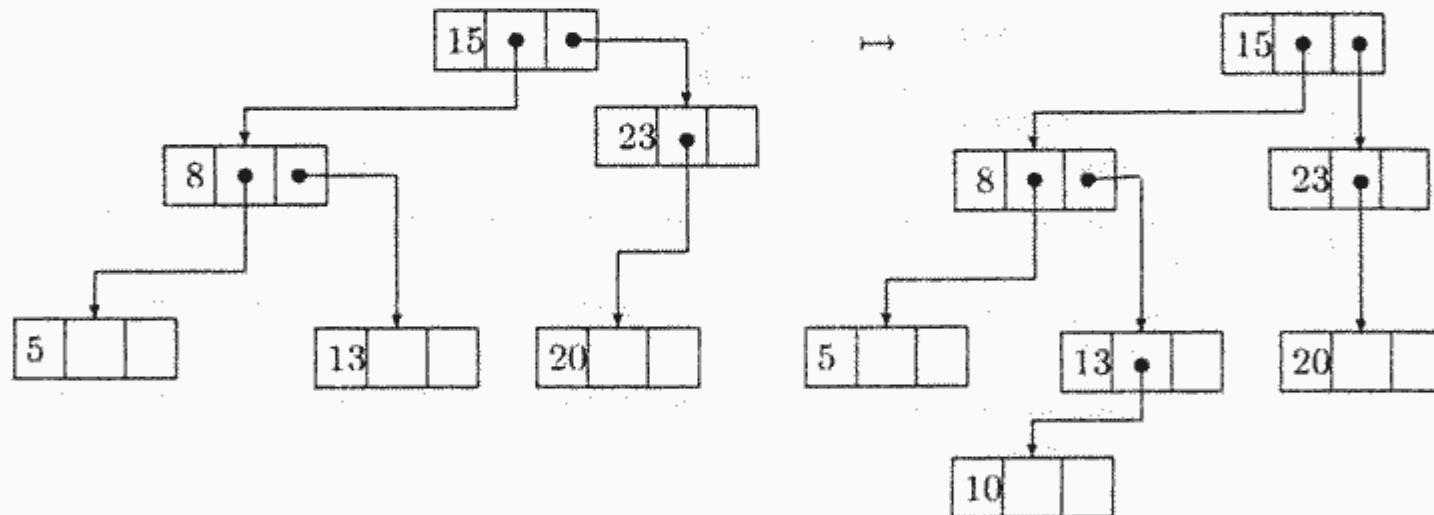


Figura 5.4: Inserare într-un arbore binar de căutare

Operația de ștergere se realizează într-un mod asemănător. Se caută x în arboarele care reprezintă mulțimea S . Dacă nu se găsește un nod $*p$ cu $p->elt = x$, atunci arboarele rămâne neschimbăt (deoarece $x \notin S$ implică $S \setminus \{x\} = S$). Fie p referința la nodul care conține pe x . Se disting următoarele trei cazuri:

1. $*p$ nu are fiu (figura 5.5.a). Se șterge nodul $*p$.
2. $*p$ are un singur fiu (figura 5.5.b). Părintele lui $*p$ este legat direct la fiul lui $*p$.
3. $*p$ are doi fiu (figura 5.5.c). Se determină cea mai mare valoare dintre cele mai mici decât x . Fie aceasta y . Ea se găsește memorată în ultimul nod $*q$ din lista inordine a subarborelui din stânga lui $*p$. Se transferă informația y în nodul $*p$ după care se elimină nodul $*q$ ca în primele două cazuri.

Acum algoritmul de căutare are următoarea descriere:

```

procedure elimina(t, x)
    if (t ≠ NULL)
        then p ← t
            while ((p ≠ NULL) and (x ≠ p->elt)) do
                parintep ← 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 elimCazisau2(parintep, p)
                        else q ← p->stg
                            parinteq ← p
                            while (q->drp ≠ NULL) do
                                parinteq ← q
                                q ← q->drp
                            p->elt ← q->elt
                            elimCazisau2(parinteq, q)
                end
            procedure elimCazisau2(parintep, p)
                if (p=t)
                    then if (t->stg ≠ NULL)
                        then t ← t->stg
                        else t ← t->drp
                    else if (p->stg ≠ NULL)
                        then if (parintep->stg = p)
                            then parintep->stg ← p->stg
                            else parintep->drp ← p->stg
                        else if (parintep->stg = p)
                            then parintep->stg ← p->drp
                            else parintep->drp ← p->drp
                delete(p)
            end

```

De remarcat că ambele operații, inserarea și eliminarea, necesită o etapă de căutare.

Observație. Structura de date utilizată aici se mai numește și *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 informație intervalele deschise corespunzătoare căutărilor fără succes, nu au mai fost incluse în definiția structurii de date din următoarele motive: simplitatea prezentării, economie de memorie și, atunci când interesează, intervalele pot fi determinate în timpul procesului de căutare. Sugerăm cititorului, ca exercițiu, să modifice algoritmul de căutare astfel încât, în cazul căutărilor fără succes, să ofere la ieșire intervalul deschis căruia îi aparține *a*.

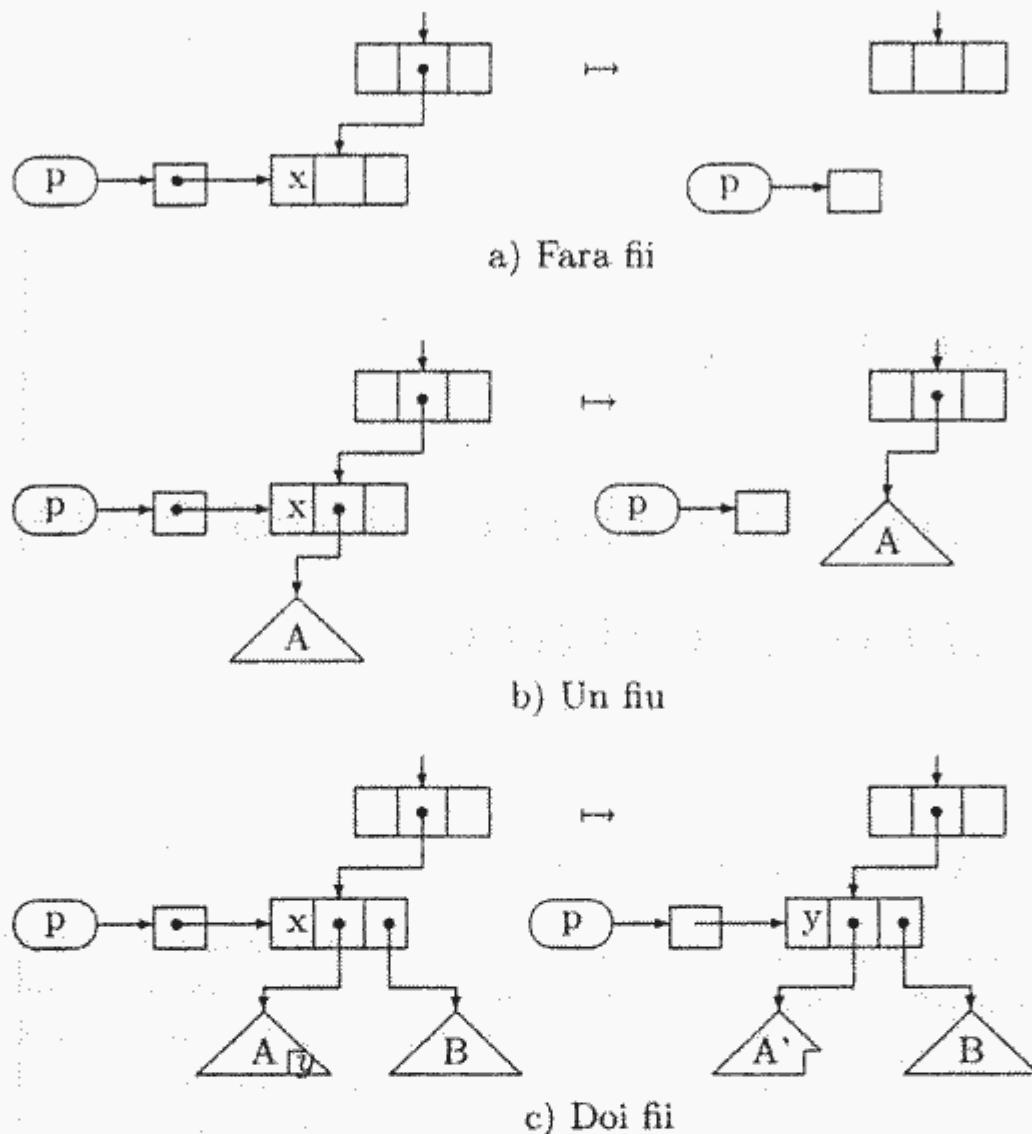


Figura 5.5: Ștergere dintr-un arbore binar de căutare

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 și în nodurile de pe frontieră. Informațiile din nodurile de pe frontieră sunt în ordine crescătoare de la stânga la dreapta și putem gândi că ele corespund intervalelor $(-\infty, x_0], (x_0, x_1], \dots, (x_{n-1}, +\infty)$. Algoritmul de căutare într-o astfel de structură va face testul $p->val = a$ numai dacă $*p$ este un nod pe frontieră. În cazul în care avem egalitate rezultă că a aparține mulțimii S ; altfel, a aparține intervalului deschis mărginit la dreapta de $p->elt$. Pentru $S = \{5, 8, 13, 15, 20, 23\}$, structura de date este reprezentată schematic în figura 5.6.

sfobs

Exercițiul 5.3.1. Să se descrie proceduri pentru operațiile de căutare, inserare și ștergere pentru arbori binari de căutare orientați pe frontieră. Operațiile vor păstra proprietatea ca fiecare nod să aibă exact doi fiți.

5.3.1 Exerciții

Exercițiul 5.3.2. Un *nod complet* este un nod care are doi fiți. Să se arate că într-un arbore binar numărul nodurilor complete este egal cu numărul nodurilor

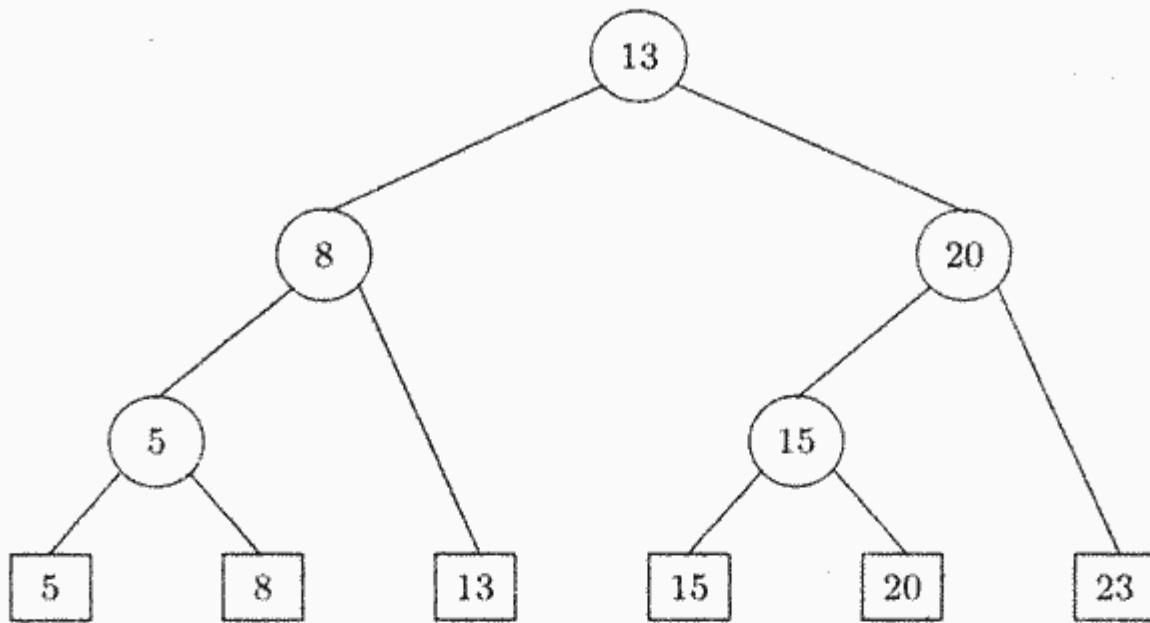


Figura 5.6: Arbore orientat pe frontieră

frunză minus 1.

Exercițiul 5.3.3. Să se scrie un subprogram care determină cel mai mare element dintr-un arbore binar de căutare.

Exercițiul 5.3.4. Să se scrie un subprogram care determină cel mai mic element dintr-un arbore binar de căutare.

Exercițiul 5.3.5. Presupunem că se dorește efectuarea unui experiment care să verifice problemele care apar la execuțiile aleatorii de perechi de operații inserare/ștergere. Următoarea strategie nu este perfect aleatoare dar e destul de aproape. Se construiește 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 ștergeri. 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 execuție al acestei operații?
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 operație? Ce se poate spune despre timpul de execuție al acestei operații?
3. Care este o bună alegere pentru α ? De ce?

Exercițiul 5.3.6. Să se scrie un subprogram care determină elementele k dintr-un arbore binar de căutare cu proprietatea $k_1 \leq k \leq k_2$ pentru k_1 și k_2 dați. Care este timpul de execuție al subprogramului?



Capitolul 6

Tipuri de date avansate pentru căutare

6.1 Arbori echilibrați

Considerăm arbori binari de căutare. Este posibil ca în urma operațiilor de inserare și de ștergere structura arborelui binar de căutare să se modifice foarte mult și operația de căutare să nu mai poată fi executată în timpul $O(\log_2 n)$. Un exemplu în care căutarea binară degenerăază în căutare liniară este ilustrat în figura 6.1. Suntem interesați să găsim algoritmi pentru inserție și ștergere care să mențină o structură „echilibrată” a arborilor, astfel încât operația de căutare să se execute în timpul $O(\log n)$ întotdeauna. Mai întâi, definim formal astfel de clase. Menționăm că în definițiile următoare nu este necesar ca arborii să fie binari; ei pot fi de diferite arități.

Reamintim că înălțimea unui arbore t , pe care o notăm cu $h(t)$, este lungimea drumului maxim de la rădăcină la un vîrf de pe frontieră.

Definiția 6.1. Fie \mathcal{C} o mulțime de arbori. \mathcal{C} se numește clasă de arbori echilibrați (balanșați), dacă pentru orice $t \in \mathcal{C}$, înălțimea lui t este mai mică decât sau egală cu $c \cdot \log n$, unde c este o constantă ce poate depinde de clasa \mathcal{C} (dar nu depinde de t) și n este numărul de vîrfuri din t .

Definiția 6.2. O clasă \mathcal{C} de arbori echilibrați se numește $O(\log n)$ -stabilă dacă există algoritmi pentru operațiile de căutare, inserare și de ștergere care necesită timpul $O(\log n)$ și arborii rezultați în urma execuției acestor operații fac parte din clasa \mathcal{C} .

În continuare prezentăm câteva clase $O(\log n)$ -stabile.

6.1.1 Arbori AVL

Această clasă a fost definită de G.M. Adelson-Velskii și E.M. Landis în 1962. În această subsecțiune ne referim la arbori binari de căutare orientați pe noduri interne.

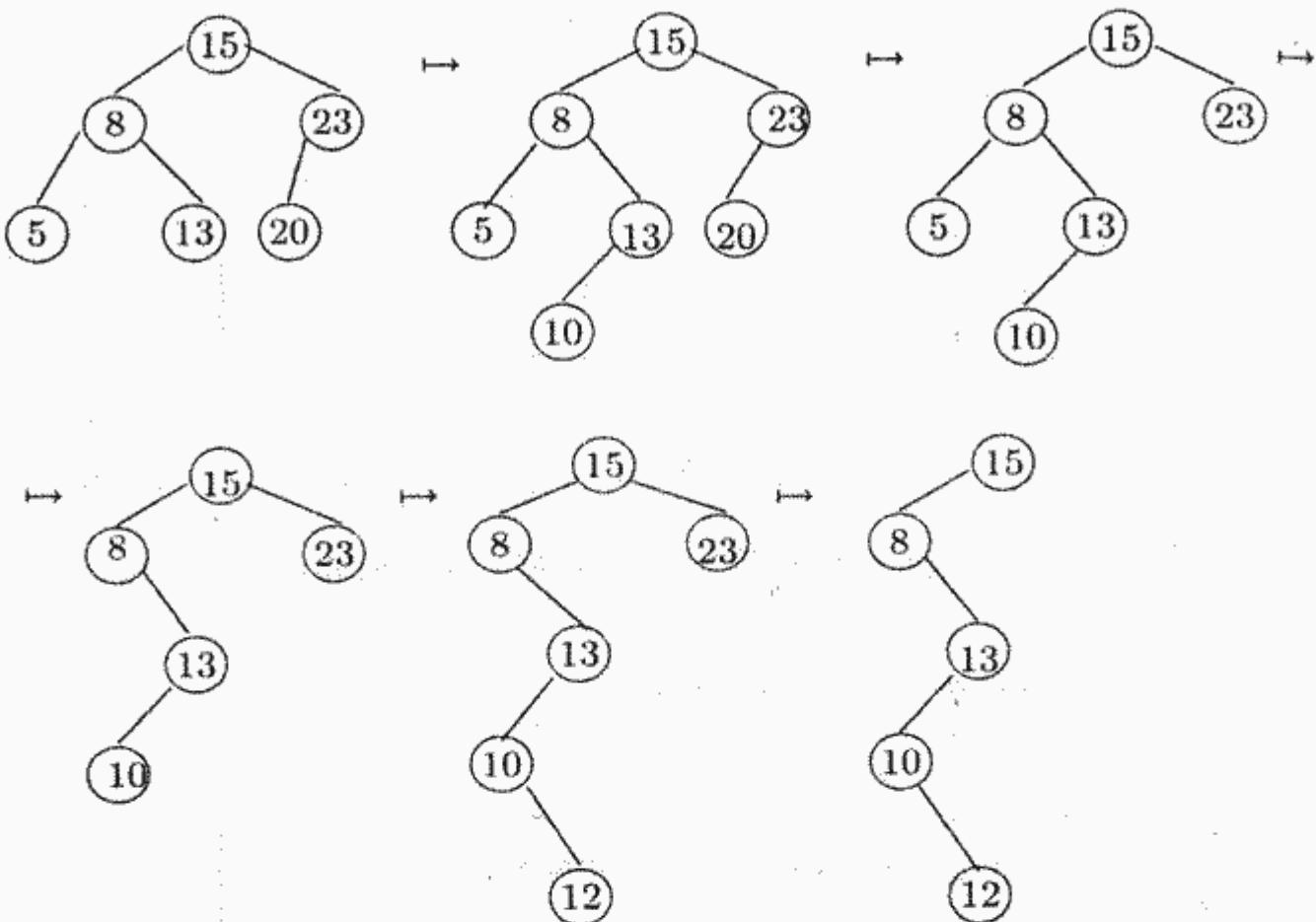


Figura 6.1: Degenerarea căutării binare în căutare liniară

Deși nodurile externe ale arborelui de decizie, corespunzătoare intervalelor deschise, nu sunt incluse în structura de date, le considerăm la determinarea înălțimii (figura 6.2).

Definiția 6.3. Un arbore binar este AVL-echilibrat (pe scurt arbore AVL), dacă pentru orice vârf, diferența dintre înălțimea subarborelui din stânga vârfului și cea a subarborelui din dreapta este -1 , 0 sau 1 . Numim această diferență factor de echilibrare.

Următorul rezultat arată că arborii AVL sunt echilibrați.

Teorema 6.1. Pentru orice arbore AVL t , cu n noduri interne, are loc $h(t) = \Theta(\log_2 n)$.

Demonstrație. Un arbore binar de înălțime h are cel mult 2^h vârfuri pe frontieră și cel mult $2^h - 1$ vârfuri interne. De aici rezultă:

$$h \geq \log_2(n + 1) = \Omega(\log_2 n) \quad (6.1)$$

Pentru a determina cea de-a doua margină, vom afla mai întâi structura unui arbore AVL de înălțime h , cu număr minim de noduri. Notăm acest arbore cu t_h^m . Procedăm prin inducție după h . Dacă $h = 1$, atunci t_1^m este format dintr-un singur nod intern corespunzător elementului x_1 și două pe frontieră corespunzătoare intervalelor $(-\infty, x_0)$ și $(x_0, +\infty)$. Dacă $h = 2$, atunci t_2^m are două noduri interne

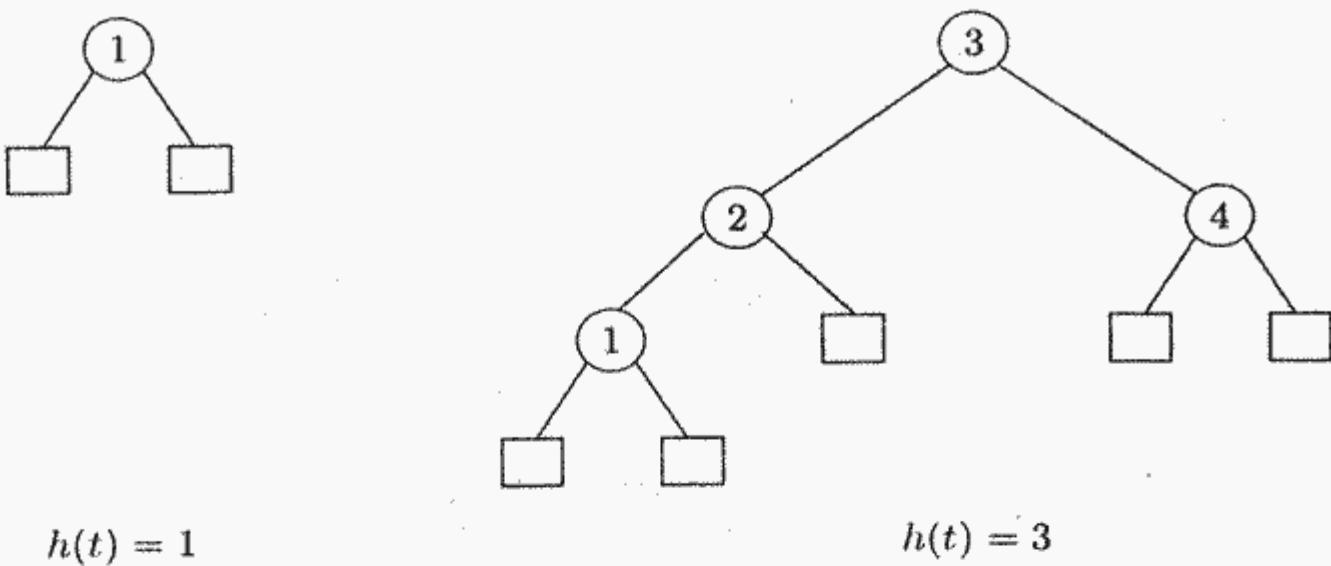


Figura 6.2: Calculul înălțimii arborilor AVL

corespunzătoare elementelor $x_0 < x_1$ și trei vârfuri pe frontieră corespunzătoare intervalelor $(-\infty, x_0)$, (x_0, x_1) , $(x_1, +\infty)$. Remarcăm că t_1^m are $Fib(3) - 1$ noduri interne iar t_2^m are $Fib(4) - 1$ noduri interne, unde $Fib(k)$ este al k -lea număr Fibonacci. Presupunem $h > 2$. Presupunem că subarborele din stânga rădăcinii lui t_h^m are înălțimea $h - 1$ și subarborele din dreapta rădăcinii are înălțimea $h - 2$ (deoarece t_h^m are număr minim de noduri, rezultă că nu pot avea ambii înălțimea $h - 1$). Deoarece t_h^m are număr minim de noduri, rezultă că subarborii rădăcinii au număr minim de noduri. Fără să restrângem generalitatea putem presupune că acești subarbore sunt t_{h-1}^m și t_{h-2}^m . Rezultă că numărul de vârfuri interne ale lui t_h^m este $Fib(h+1) - 1 + Fib(h) - 1 + 1 = Fib(h+2) - 1$. Deci t_h^m coincide cu al $h+2$ -lea arbore Fibonacci. Rezultă:

$$n \geq Fib(h+2) - 1 = \left\lceil \frac{\Phi^{h+2}}{\sqrt{5}} + \frac{1}{2} \right\rceil - 1 > \frac{\Phi^{h+2}}{\sqrt{5}} - \frac{3}{2} \quad (6.2)$$

unde $\Phi = \frac{1+\sqrt{5}}{2}$ („numărul de aur”). Din (6.2) obținem:

$$h < \frac{1}{\log_2 \Phi} \cdot \log_2 \left(\sqrt{5}(n + \frac{3}{2}) \right) - 2 = O(\log_2 n). \quad (6.3)$$

Relațiile (6.1) și (6.3) implică concluzia din teoremă.

sfdem

6.1.1.1 Conservarea proprietății de arbore binar AVL-echilibrat

Instrumente de reechilibrare – rotații. În perspectiva conservării proprietății de arbore binar de căutare AVL-echilibrat, operațiile **insereaza** și **elimina** trebuie modificate, astfel încât după o astfel de operație arborele să rămână AVL-echilibrat. Adăugarea la operațiile **insereaza** și **elimina** a acestor mecanisme de conservare a tipului AVL a arborelui asupra căruia operează nu schimbă complexitatea acestora.

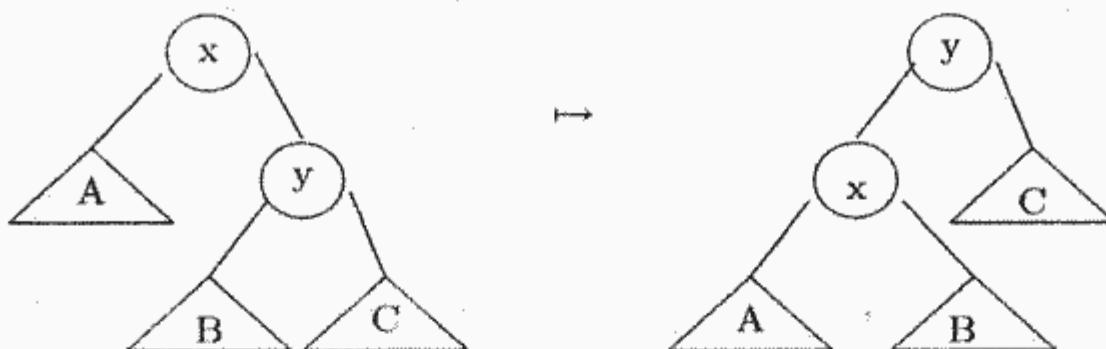


Figura 6.3: Rotație simplă la stânga

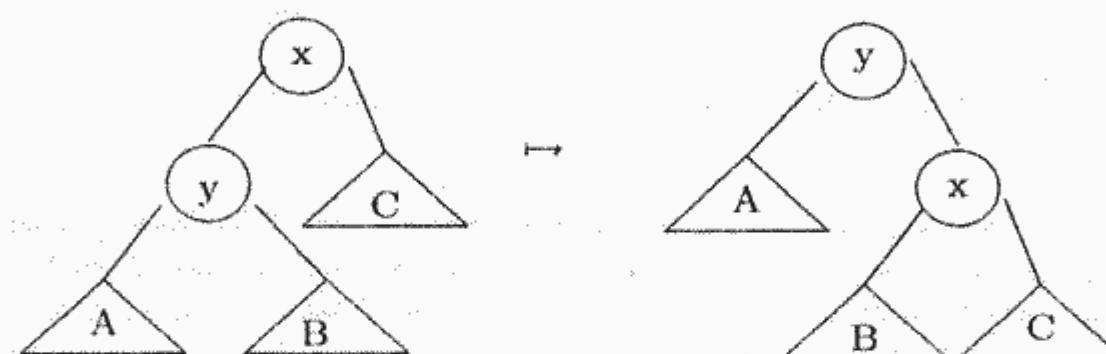


Figura 6.4: Rotație simplă la dreapta

Modificările necesare refacerii structurii de arbore binar echilibrat, după efectuarea unei operații de inserție sau ștergere, se numesc *rotații*. Rotațiile pot fi simple sau duble (figurile 6.3, 6.4, 6.5, 6.6).

Subprogramele care implementează operațiile implicate de rotații sunt următoarele:

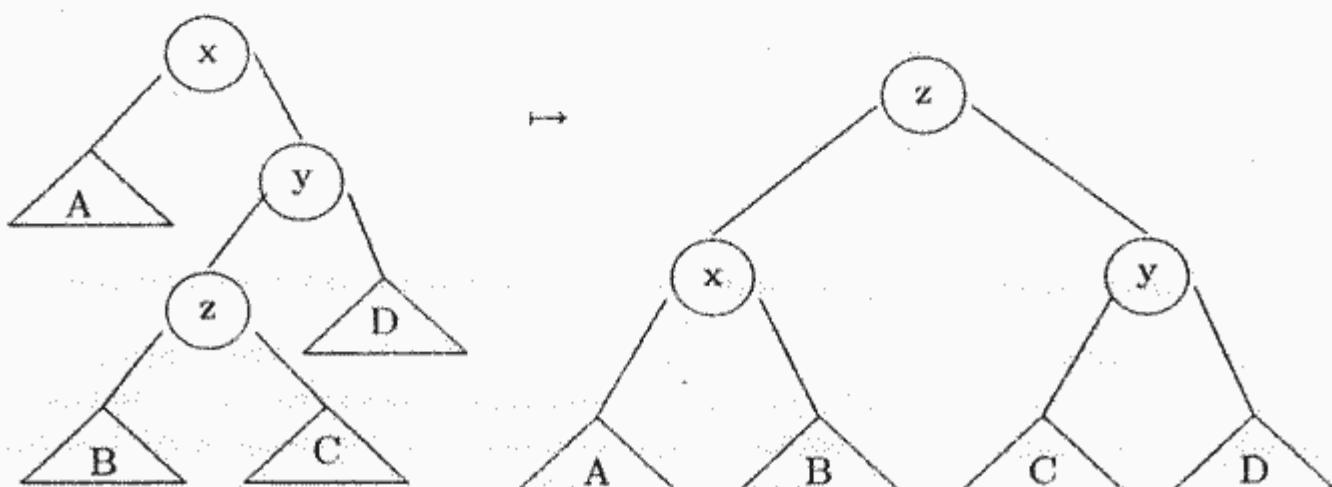


Figura 6.5: Rotație dublă la stânga

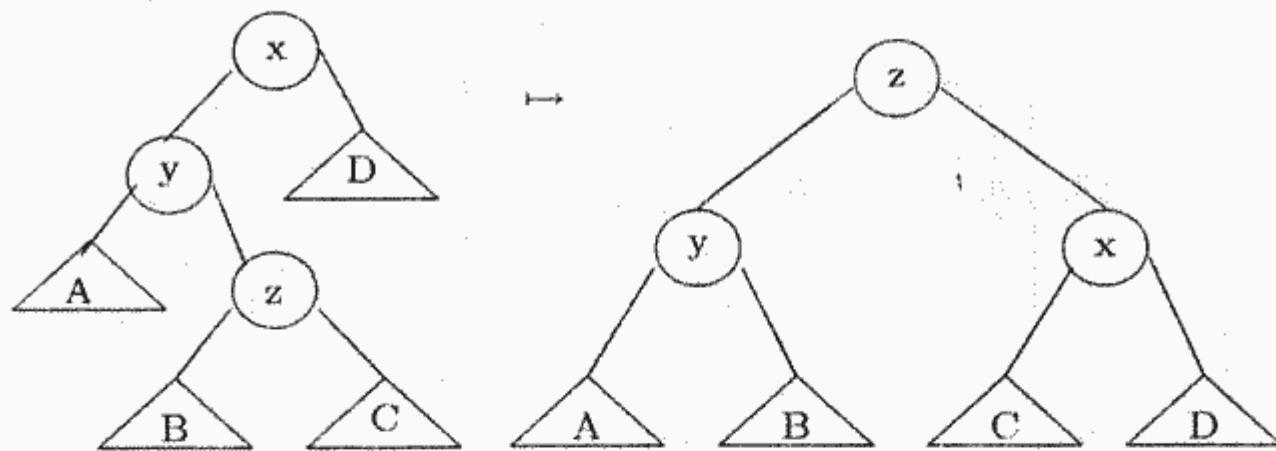


Figura 6.6: Rotație dublă a la dreapta

```

procedure rotatieSimplaStg(x)
    y ← x->drp
    x->drp ← y->stg
    y->stg ← x
    x ← y /* se schimba radacina */
end

procedure rotatieSimplaDrp(x)
    y ← x->stg
    x->stg ← y->drp
    y->drp ← x
    x ← y
end

procedure rotatieDublaStg(x)
    y ← x->drp
    z ← y->stg
    x->drp ← z->stg
    y->stg ← z->drp
    z->stg ← x
    z->drp ← y
    x ← z
end

procedure rotatieDublaDrp(x)
    y ← x->stg
    z ← y->drp
    x->stg ← z->drp
    y->drp ← z->stg
    z->drp ← x
    z->stg ← y
    x ← z
end

```

6.1.1.2 Tehnici de reechilibrare a arborilor AVL

Inserarea. După inserarea unui nod terminal într-un arbore AVL este necesară cel mult o rotație. Identificarea nodului asupra căruia trebuie aplicată rotația este problema-cheie.

Definiția 6.4. Se numește factor de echilibru al unui nod v și se notează cu $bf(v)$ diferența dintre înălțimea subarborelui stâng și înălțimea subarborelui drept: $bf(v) = h(v \rightarrow \text{stg}) - h(v \rightarrow \text{drp})$.

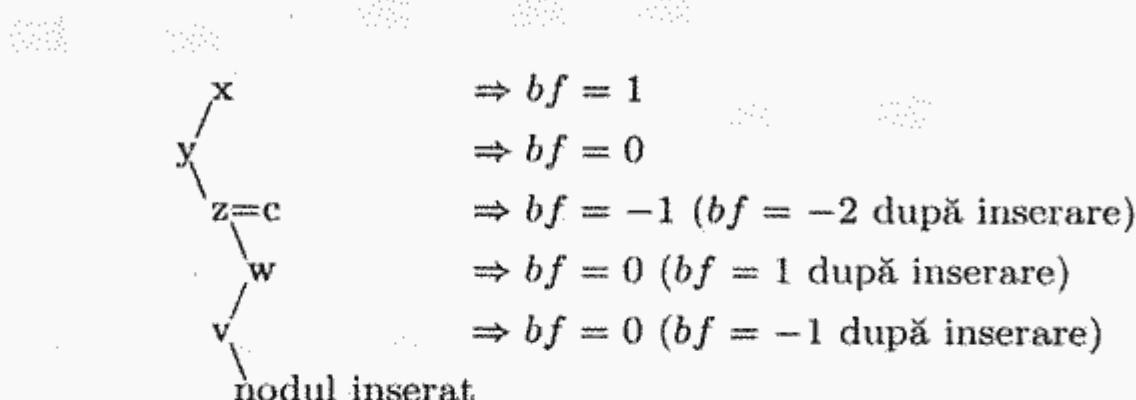


Figura 6.7: Determinarea nodului critic

Observație. După inserare, pentru nodul terminal v și nodul w s-au schimbat doar înălțimea și factorul de echilibru, aceștia rămânând totuși echilibrați. Nodul z devine însă dezechilibrat, deoarece $bf(z)$ devine -2 . sfobs

Definiția 6.5. Se numește nod critic primul nod c cu $bf(c) \neq 0$ întâlnit la o parcurgere a ramurii care leagă nodul inserat de rădăcină, parcursa efectuându-se de la nod spre rădăcină. $bf(c)$ este cel calculat înainte de inserare.

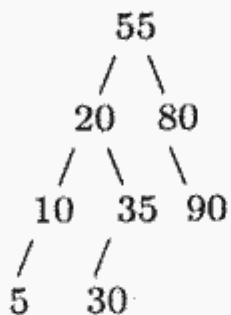
Observație. Toate nodurile din ramură care sunt pe nivele inferioare nodului critic vor avea după inserare $bf = 1$ sau $bf = -1$. sfobs

În cazul nodului critic c există două situații:

1. Nodul critic va fi perfect echilibrat ($bf(c) = 0$), dacă dezechilibrul creat de nodul inserat anulează dezechilibrul inițial al nodului. În acest caz, nu este nevoie de rotație (el completează un gol în arbore);
2. Factorul de echilibru devine $bf(c) = 2$ sau $bf(c) = -2$, atunci când nodul inserat mărește dezechilibrul arborelui (s-a inserat nodul în subarborele cel mai înalt). În acest caz, se aplică o rotație în urma căreia se schimbă structura subarborelui cu rădăcina c , astfel încât pentru noua rădăcină c' , factorul de echilibru $bf(c')$ devine 0. Adâncimea subarborelui devine egală cu adâncimea acestuia înainte de inserare. Astfel, nodurile care se află pe drumul de la c la rădăcina arborelui rămân cu factorul de echilibru identic cu cel dinainte de inserare.

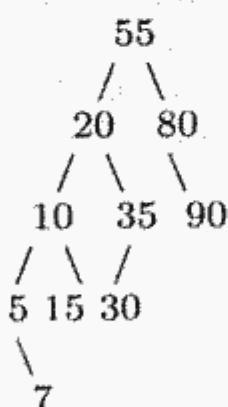
În concluzie, problema conservării proprietății de echilibru a unui arbore AVL în care s-a efectuat o inserare se rezolvă aplicând o rotație asupra nodului critic. Rotația se aplică *numai atunci* când inserarea dezechilibrează acest nod. Costul suplimentar care trebuie suportat rezultă din necesitatea ca în fiecare nod să se memoreze factorul de echilibru *bf*. Acești factori de echilibru trebuie actualizați la fiecare inserție și rotație.

Exemplu. Se dă arborele cu următoarea structură:

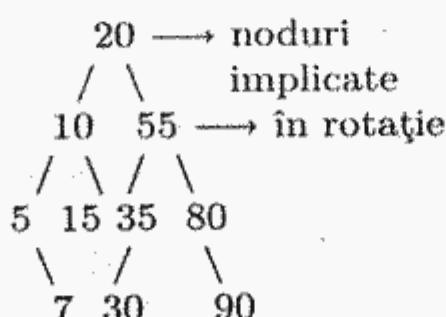


Să se insereze nodurile 15, apoi 7 și să se echilibreze arborele.

Inserăm prima valoare 15. Comparam mai întâi cu 55: e în stânga lui și ambele părți sunt echilibrate. Pentru a doua valoare de inserat, 7, nodul critic este 55.

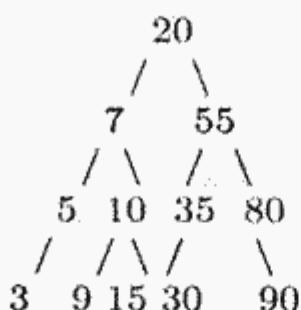


Este necesară aplicarea unei rotații asupra rădăcinii. Facem o identificare cu una din figurile de echilibrare prezentate anterior. Astfel 55→x; 20→y etc. Rezultă:



În urma unei rotații simple, factorii de echilibru implicați în rotație devin zero. Fie o a treia valoare de inserat, 3, apoi a patra, 9. Nodul critic pentru 3 este nodul de cheie 5. 3 echilibrează nodul critic, deci nu este nevoie de rotație. Nodul corespunzător lui 9 are nodul critic cel cu cheia 10 și dezechilibrează subarborele

nodului critic. După o rotație dublă la dreapta aplicată acestui subarbore, rezultă arborele:



sfex

Ștergerea. Operația de ștergere într-un arbore de căutare AVL echilibrat implică mai multe rotații. Invităm cititorul să analizeze cazul ștergerii unui nod dintr-un arbore AVL-echilibrat.

Teorema 6.2. Clasa arborilor AVL este $O(\log n)$ -stabilă.

Demonstrație. În urma efectuării unei operații de inserare sau de ștergere singurele noduri care-și modifică factorii de echilibrare se află pe drumul de la rădăcină la vârful inserat/șters. Se memorează acest drum într-o stivă în timpul etapei de căutare. După efectuarea operației, se parcurge acest drum în sens invers și se refac factorul de echilibru pentru nodurile dezechilibrate. Refacerea factorilor de echilibru se realizează cu ajutorul rotațiilor.

sfdem

6.1.2 Arbori bicolori

În această subsecțiune considerăm o clasă de arbori pentru căutare în care echilibrarea este menținută cu ajutorul unei colorări adecvate ale nodurilor. De fapt, sunt suficiente numai două culori pentru a putea defini o clasă $O(\log n)$ -stabilă. Deoarece aceste culori au fost inițial roșu și negru, arborii sunt cunoscuți sub numele de *red-black-trees*.

Definiția 6.6. Un arbore bicolor este un arbore binar de căutare care satisface următoarele proprietăți:

1. Nodurile sunt colorate. Un nod are sau culoarea roșie sau culoarea neagră.
2. Nodurile pendante (acestea sunt considerate ca făcând parte din structură) sunt colorate cu negru.
3. Dacă un nod este roșu, atunci fiile săi sunt colorați cu negru.
4. Pentru orice nod, drumurile de la acel nod la nodurile de pe frontieră au același număr de noduri colorate cu negru.

Un exemplu de arbore bicolor este reprezentat în figura 6.8. Nodurile reprezentate cu cercuri albe reprezintă nodurile colorate cu roșu. Dacă v este un nod într-un arbore bicolor t , atunci notăm cu $hn(v)$ numărul de noduri negre aflate pe un drum de la v la un nod de pe frontieră, excluzând v . Definiția lui hn nu depinde de alegerea drumului datorită condiției 4 din definiție.

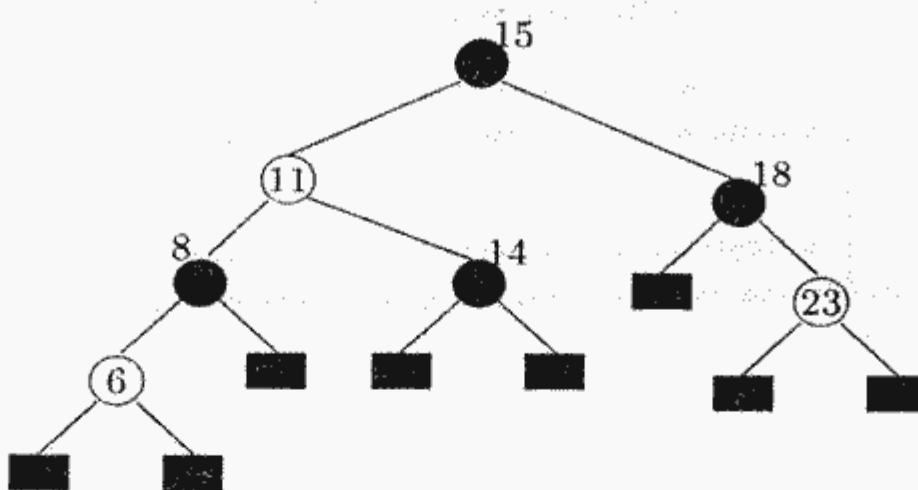


Figura 6.8: Arbore bicolor

Lema 6.1. Fie t un arbore bicolor. Orice subarbore al lui t are cel puțin $2^{hn(v)} - 1$ noduri interne, unde v este rădăcina subarborelui.

Demonstrație. Procedăm prin inducție după înălțimea h a subarborelui. Dacă $h = 0$, atunci v este singurul nod al subarborelui și este un nod pendant. Numărul de noduri interne este 0 și coincide cu $2^{hn(v)} - 1 = 2^0 - 1$. Presupunem $h > 0$. Distingem două situații.

1. v este roșu. Atunci fiile lui v sunt negre. Fie x și y fiile lui v . Avem $hn(v) = hn(x) + 1 = hn(y) + 1$. Din ipoteza inductivă, fiecare subarbore al lui v are cel puțin $2^{hn(v)} - 1$ noduri interne. Rezultă că t are cel puțin $2 \cdot (2^{hn(v)} - 1) + 1 = 2^{hn(v)} - 1$ noduri interne.
2. v este negru. Dacă x și y sunt ambii roșii atunci $hn(v) = hn(x) = hn(y)$. Din ipoteza inductivă, fiecare subarbore al lui v are cel puțin $2^{hn(v)} - 1$ noduri interne. Rezultă că t are cel puțin $2 \cdot (2^{hn(v)} - 1) + 1 > 2^{hn(v)} - 1$. Celelalte cazuri se tratează asemănător.

sfdem

Teorema 6.3. Un arbore bicolor cu n noduri interne are înălțimea cel mult $2 \log_2(n+1)$.

Demonstrație. Fie t un arbore bicolor cu rădăcina v și înălțimea h . Din lema 6.1 rezultă că t are cel puțin $2^{hn(v)} - 1$ noduri interne. Din condiția 3 din definiția arborilor bicolori, rezultă că pe orice drum de la v la un nod de pe frontieră cel puțin jumătate dintre noduri sunt negre. Avem $hn(v) \geq \frac{h}{2}$ care implică $n \geq 2^{hn(v)} - 1 \geq 2^{\frac{h}{2}} - 1$. De aici $h \leq 2 \log_2(n+1)$.

sfdem

Teorema 6.4. Clasa arborilor bicolori este $O(\log n)$ -stabilă.

Demonstrație. Din teorema 6.3 rezultă că operația de căutare, care se face exact ca la arborii de căutare obișnuiți, se realizează în timpul $O(\log n)$. Mai trebuie să arătăm că operațiile de inserare și de ștergere pot fi realizate în timpul $O(\log n)$ și în urma realizării acestor operații rezultă arbori bicolori.

sfdem

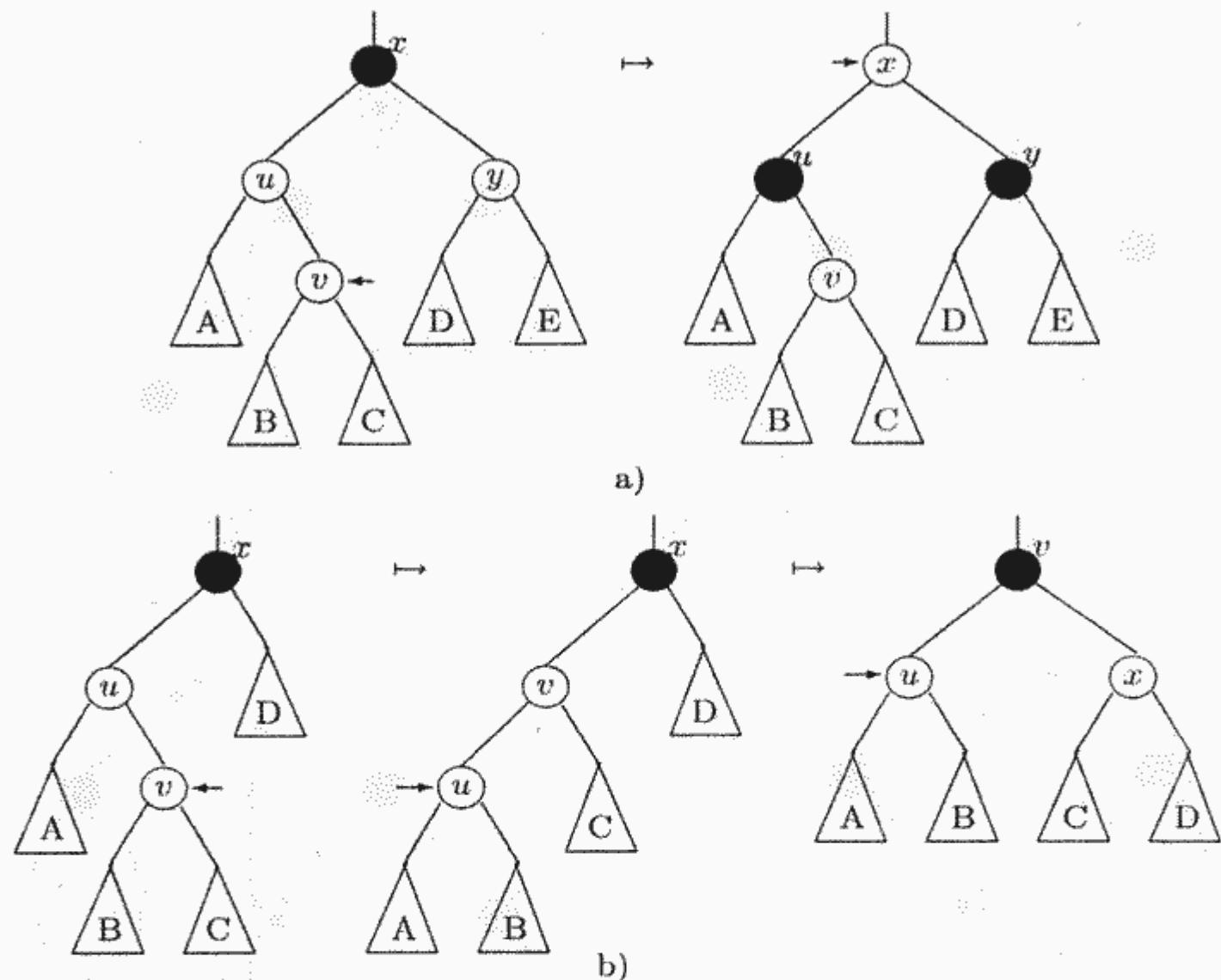


Figura 6.9: Inserarea într-un arbore bicolor – restabilirea proprietăților

Inserarea. Este realizată în două etape. În prima etapă se face inserarea noului nod exact ca în cazul arborilor de căutare obișnuiți. Nodul inserat va primi culoarea roșie. În urma inserării este posibil ca proprietatea 3 din definiție să nu mai fie respectată de noul arbore. În etapa a doua se restabilește proprietatea 3 cu ajutorul operațiilor din figura 6.9 sau a simetricelor lor. Aceste operații constau în recolorări de noduri sau combinații de rotații simple.

Subprogramul care realizează operația de inserare a unui nod într-un arbore bicolor este descrisă în continuare:

```

procedure insereazaBicolor(t, v)
    /* insereaza v în mod ușual (ca într-un arbore binar de căutare) */
    insereaza(t, v)
    /* restabilește proprietatea de arbore bicolor */
    v->culoare ← rosu
    u ← v->parinte
    x ← u->parinte

```

```

while (v ≠ t și u->culoare = rosu)
    if ( u = x->stg )
        then y ← x->drept
            if ( y->culoare = rosu )
                then /* schimba culorile */
                    u->culoare ← negru
                    y->culoare ← negru
                    x->culoare ← rosu
                    /* v urca */
                    v ← x
                else /* y este negru */
                    if ( v = u->drp)
                        then /* v urca si rotatie */
                            v ← v->parinte
                            rotatieSimplaStg(v)
                            v->culoare ← negru
                            x->culoare ← rosu
                            rotatieSimplaDrp(x)
                    else /* se repeta secventa de instructiuni de pe ramura then
                           cu stanga si dreapta interschimbate */
                        t->culoare ← negru
    end

```

Ștergere. Și această operație este realizată în două etape. Prima etapă seamănă cu cea de la arborii de căutare obișnuiți, dar acum trebuie ținut cont de faptul că nodurile pendante fac parte din structură. În urma ștergerii este posibil ca proprietatea 4 din definiție să nu mai fie respectată de noul arbore. În partea a doua se restabilește această proprietate cu ajutorul operațiilor din figura 6.10. Nodurile reprezentate cu cercuri duble pot fi roșii sau negre. Cazul a) este transformat într-unul din cazurile b), c), d) printr-o rotație. În cazul b) nodul pentru care nu are loc proprietatea 4 este deplasat spre rădăcină cu un nivel prin recolorarea unui nod. Cazul c) este transformat în d) printr-o interschimbare de culori și o rotație. Apariția cazului d) și rezolvarea lui conduce la restabilirea proprietății pentru întregul arbore.

Invităm cititorul să scrie subprograme pentru operația de ștergere.

6.1.3 2-3-arbore

2-3-arborii constituie o clasă de structuri de date arborescente $O(\log n)$ -stabilă ce generalizează arborii binari de căutare.

Definiția 6.7. Un 2-3-arbore este un arbore de căutare care satisface următoarele proprietăți:

1. Fiecare nod intern este de aritate 2 sau 3 (are 2 sau 3 fii). Un nod $*v$ de aritate 2 memorează o singură valoare $v->eltStg$, iar un nod de aritate 3 memorează două valori: $v->eltStg$ și $v->eltMij$ cu $v->eltStg < v->eltMij$.
2. Fiii unui nod $*v$ îi vom numi fiu stânga (notat $*(v->stg)$), mijlociu $*(v->mij)$ și dreapta $*(v->drp)$). Un nod de aritate 2 are fiu stânga și mijlociu.

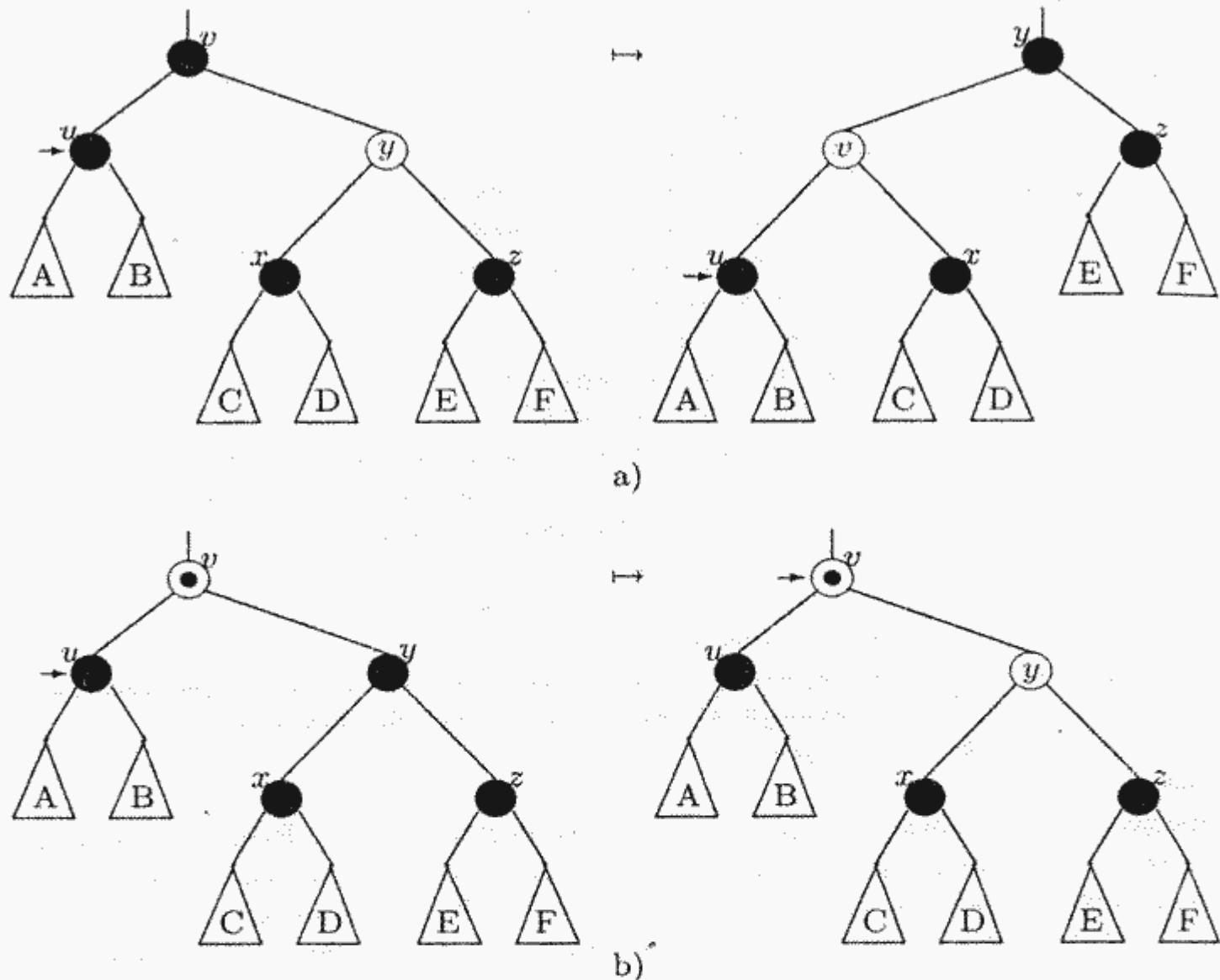


Figura 6.10: Stergerea într-un arbore bicolor – restabilirea proprietăților

3. Pentru orice nod intern $*v$ de aritate 2, valorile memorate în subarborele cu rădăcina în $*(v \rightarrow stg)$ sunt mai mici decât $v \rightarrow eltStg$ și valorile din subarborele cu rădăcina în $*(v \rightarrow mij)$ sunt mai mari decât $v \rightarrow eltStg$.
4. Pentru orice nod intern $*v$ de aritate 3 urez loc:
 - valorile din subarborele cu rădăcina $v \rightarrow stg$ sunt mai mici decât $v \rightarrow eltStg$;
 - valorile din subarborele cu rădăcina $v \rightarrow mij$ sunt mai mari decât $v \rightarrow eltStg$ și mai mici decât $v \rightarrow eltMij$;
 - valorile din subarborele cu rădăcina $*(v \rightarrow drp)$ sunt mai mari decât $v \rightarrow eltMij$.
5. Toate nodurile de pe frontieră se află pe același nivel.

Un exemplu de 2-3-arbore este reprezentat grafic în figura 6.11. Nodurile reprezentate grafic prin pătrate nu intră în definiția reprezentării, ele având rolul numai de a sugera mai bine structura de 2-3-arbore. Se poate stabili o corespondență între aceste noduri și elementele mulțimii reprezentate. Facem presupunerea ca în cazul nodurilor de aritate 2, câmpul $v \rightarrow eltMij$ să memoreze o valoare artificială $MaxVal$ mai mare decât orice element din mulțimea univers.

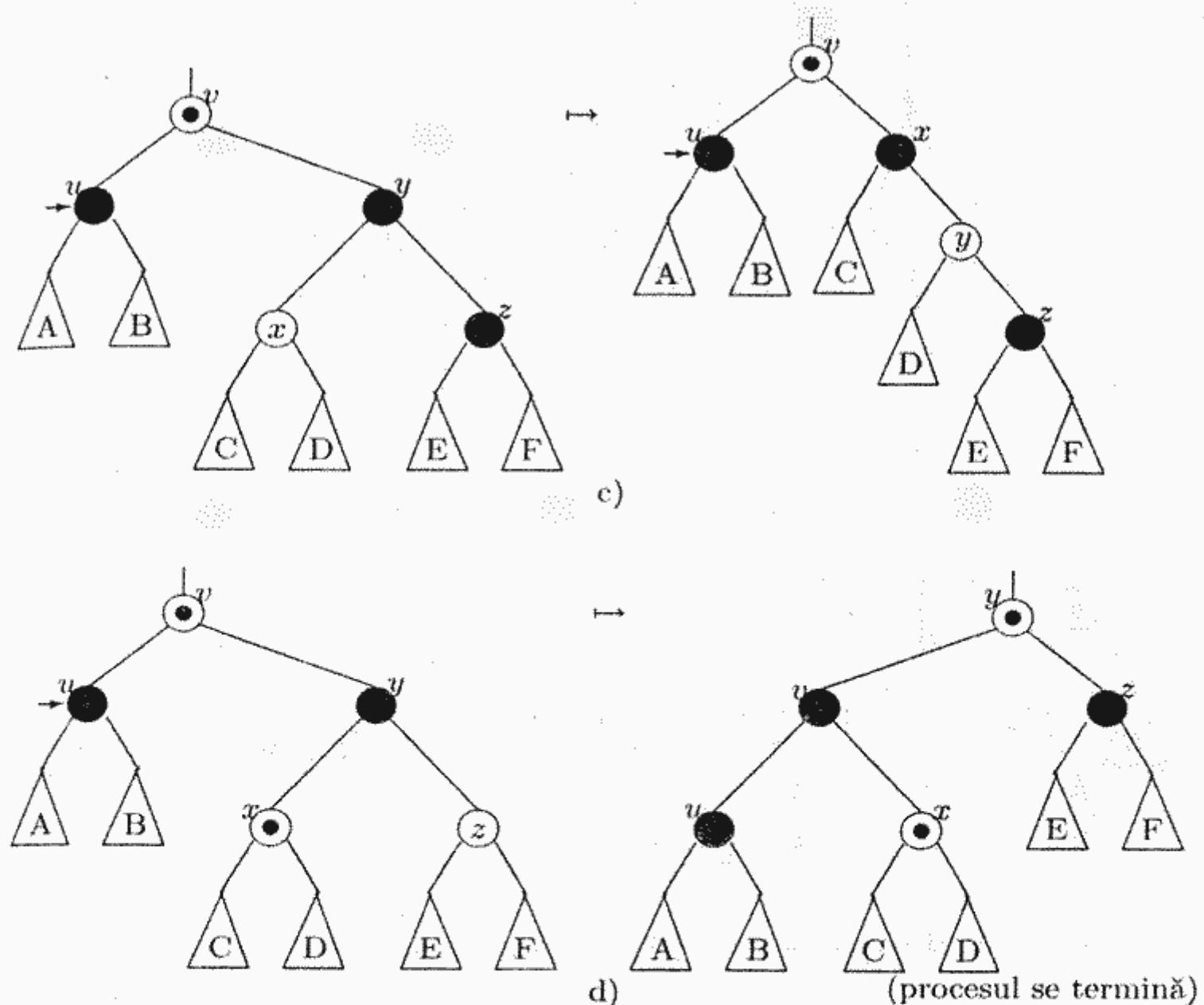


Figura 6.10: Ștergerea într-un arbore bicolor – restabilirea proprietăților (continuare)

Operația de căutare se realizează într-o manieră asemănătoare cu cea de la arborii binari de căutare. Procesul de căutare pentru a în arborele t pleacă din rădăcină și în fiecare nod vizitat $*v$ se compară a cu valorile memorate în $*v$:

- dacă $a = v->eltStg$ sau $a = v->eltMij$, atunci căutarea se termină cu succes;
- dacă subarborele în care se caută este vid, atunci căutarea se termină cu insucces;
- dacă $a < v->eltStg$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v->stg$;
- dacă $v->eltStg < a < v->eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v->mij$;
- dacă $a > v->eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v->drp$.

Algoritmul care realizează toate acestea este următorul:

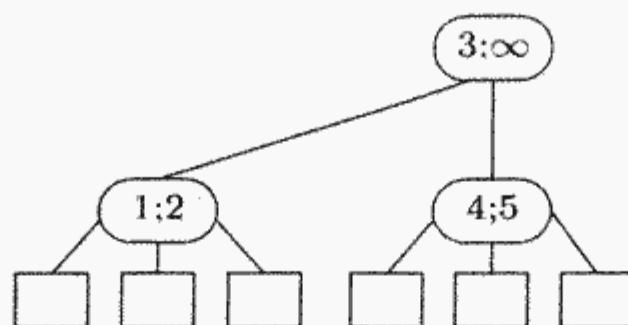


Figura 6.11: Exemplu de 2-3-arbore

```

function poz(t, a)
  p ← t
  while (p ≠ NULL)
    if ((p->eltStg = a) or (p->eltMij = a))
      then return p
    else if (a < p->eltStg)
      then p ← p->stg
    else if (a < p->eltMij)
      then p ← p->mij
    else p ← p->drp
  return p
end
  
```

Operația de inserare este ceva mai complicată și se realizează după următorul scenariu:

1. Se caută în arborele t elementul ce urmează a fi inserat x . Drumul de la rădăcină la frontieră parcurs în timpul căutării este memorat într-o stivă. Dacă x este memorat deja în t atunci operația de inserare se termină fără a modifica arborele.
 2. În continuare se parcurge înapoi drumul memorat în stivă și la fiecare pas se realizează o reuniune de doi arbori: unul care are rădăcina y de aritate 1 și subarborele lui t cu rădăcina în nodul $*v$ memorat în vârful stivei. Inițial avem $y = x$ și subarborele său vid. Întâlnim următoarele cazuri:
 - a) Nodul $*v$ este de aritate 2. Atunci nodurile y și $*v$ fuzionează formând un singur nod de aritate 3 (figura 6.12). În acest caz, operația de inserare se termină.
 - b) Nodul $*v$ este de aritate 3. Notăm cu A subarborele lui y , cu B, C, D subarboreii lui $*v$ și cu x_1, x_2 valorile memorate în $*v$. Pentru simplitatea prezentării notăm cei doi subarborei prin $[y](A)$ respectiv prin $[x_1; x_2](B, C, D)$. Nodul $*v$ se multiplică în două noduri de aritate 2, ale căror roluri în continuare diferă pentru fiecare din următoarele subcazuri:
 - $y < x_1$. În funcție de ce fel de fiu este $*v$, se formează următoarele perechi de subarborei (figura 6.13):
 - fiu stânga: $[x_1]([y](A, B))$ și $[x_2](C, D)$.
 - fiu mijlociu sau dreapta: $[y](A, B)$ și $[x_1]([x_2](C, D))$.
- În continuare se consideră $y = x_1$.

- $y > x_2$. Similar cazului precedent.
- $x_1 < y < x_2$. În funcție de ce fel de fiu este la rândul său tatăl lui $*v$, se formează următoarele perechi de subarbore (figura 6.14):
 - fiu stânga: $[y]([x_1](B, C))$ și $[x_2](A, D)$.
 - fiu mijlociu sau dreapta: $[x_1](B, C)$ și $[y]([x_2](A, D))$.

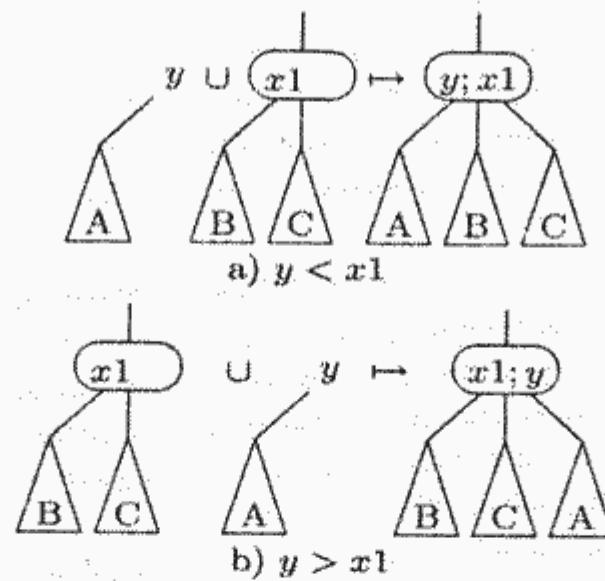
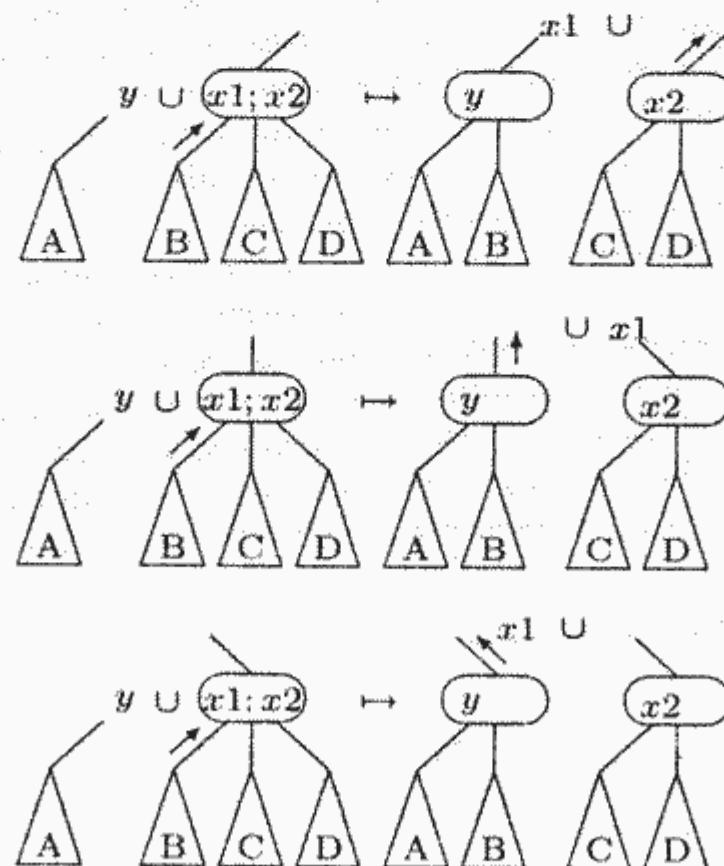
În continuare, y rămâne neschimbat.

Dacă $v = t$ ($*v$ este rădăcină), atunci se creează un nou nod care va avea ca fiu pe y și $*v$.

Stergerea se poate realiza prin următoarea strategie:

1. Se caută în arborele t elementul ce urmează a fi șters x . Drumul de la rădăcină la frontieră parcurs în timpul căutării este memorat într-o stivă. Dacă x nu este în t , atunci operația de stergere se termină fără a modifica arborele.
2. Fie $*v$ nodul unde este memorat x . Distingem situațiile:
 - a) $*v$ nod pe frontieră (figura 6.15).
 - i) $*v$ are aritate 3: dacă $x = v \rightarrow eltStg$, atunci se transferă valoarea din $v \rightarrow eltMij$ în $v \rightarrow eltStg$ și se pune $MaxVal$ în $v \rightarrow eltMij$, iar dacă $x = v \rightarrow eltMij$ atunci se memorează $MaxVal$ în $v \rightarrow eltMij$.
 - ii) $*v$ de aritate 2. Putem presupune că după stergere $*v$ devine de aritate 1 (are numai subarborele stânga). Acest tip de nod va fi „deplasat” pe drumul înapoi spre rădăcină până când este unit cu un frate de aritate 2 sau va avea tatăl de aritate 3 ori devine rădăcină. Deplasarea spre rădăcină a lui v se face aplicând operații de tipul celor din figura 6.17. Dacă tatăl lui $*v$ are aritatea 3 atunci se aplică operația din figura 6.18 și operația se termină. La fel, dacă la un moment dat $*v$ are un frate de aritate 3, atunci se aplică operația din figura 6.16 și operația se termină. Dacă nodul $*v$ de aritate 1 ajunge rădăcină atunci se șterge.
 - b) $*v$ este nod intern. Dacă $x = v \rightarrow eltStg$, atunci se înlocuiește x cu cea mai mare valoare din subarborele cu rădăcina în $v \rightarrow stg$ (sau cea mai mică valoare din subarborele cu rădăcina în $v \rightarrow mij$), iar dacă $x = v \rightarrow eltMij$, atunci se înlocuiește x cu cea mai mare valoare din subarborele cu rădăcina în $v \rightarrow mij$ (sau cea mai mică valoare din subarborele cu rădăcina în $v \rightarrow drp$). Valoarea care îl va înlocui pe x este memorată într-un nod de pe frontieră și, după înlocuire, va fi eliminată ca în primul caz.

Teorema 6.5 ([Cro92]) Clasa 2-3-arborilor este $O(\log n)$ stabilă.

Figura 6.12: Inserare într-un 2-3-arbore: cazul când v are aritatea 2Figura 6.13: Inserare într-un 2-3-arbore: v de aritate 3 și $y < v->eltStg$

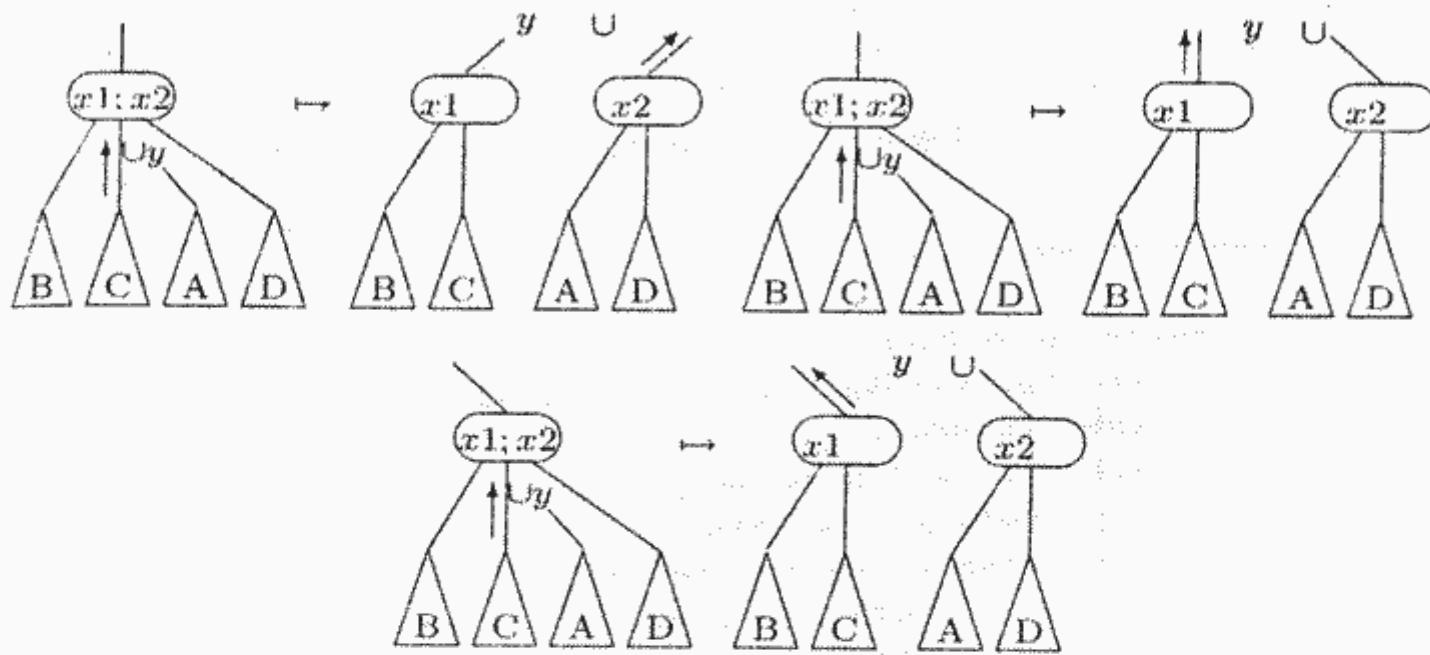


Figura 6.14: Inserare într-un 2-3-arbore: v de aritate 3 și $v->eltStg < y < v->eltMij$.

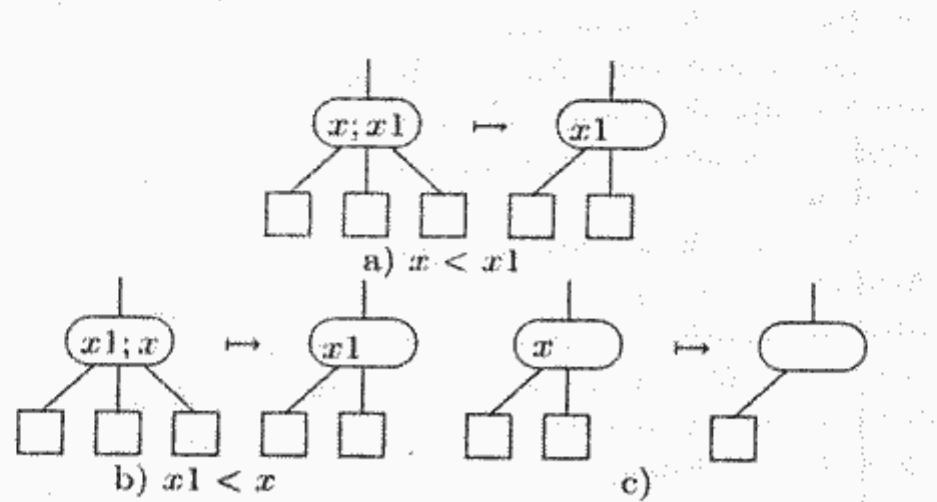
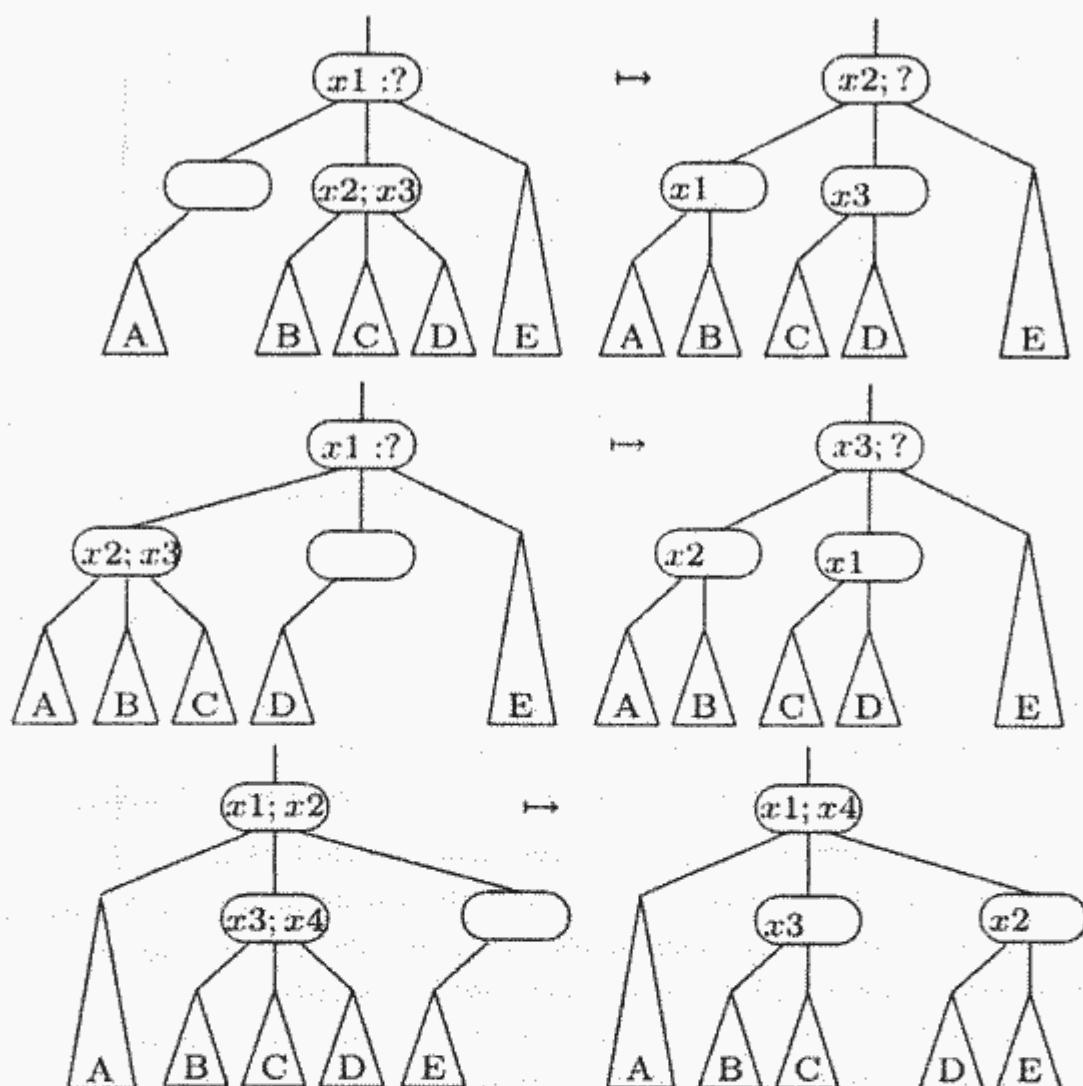
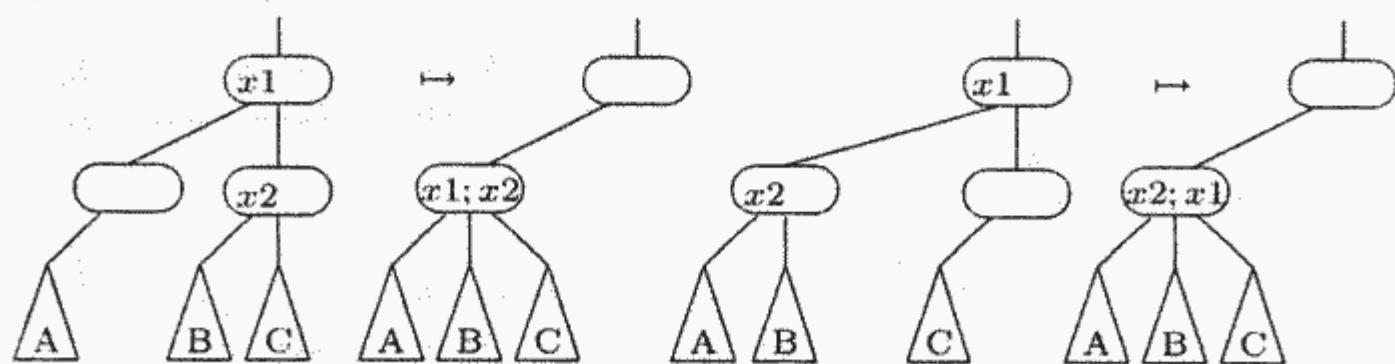


Figura 6.15: Stergere într-un 2-3-arbore: x este memorat într-un nod pe frontieră

Figura 6.16: Ștergere într-un 2-3-arbore: rotații când v are aritatea 3Figura 6.17: Ștergere într-un 2-3-arbore: combinarea nodurilor când v și tatăl său au aritatea 2

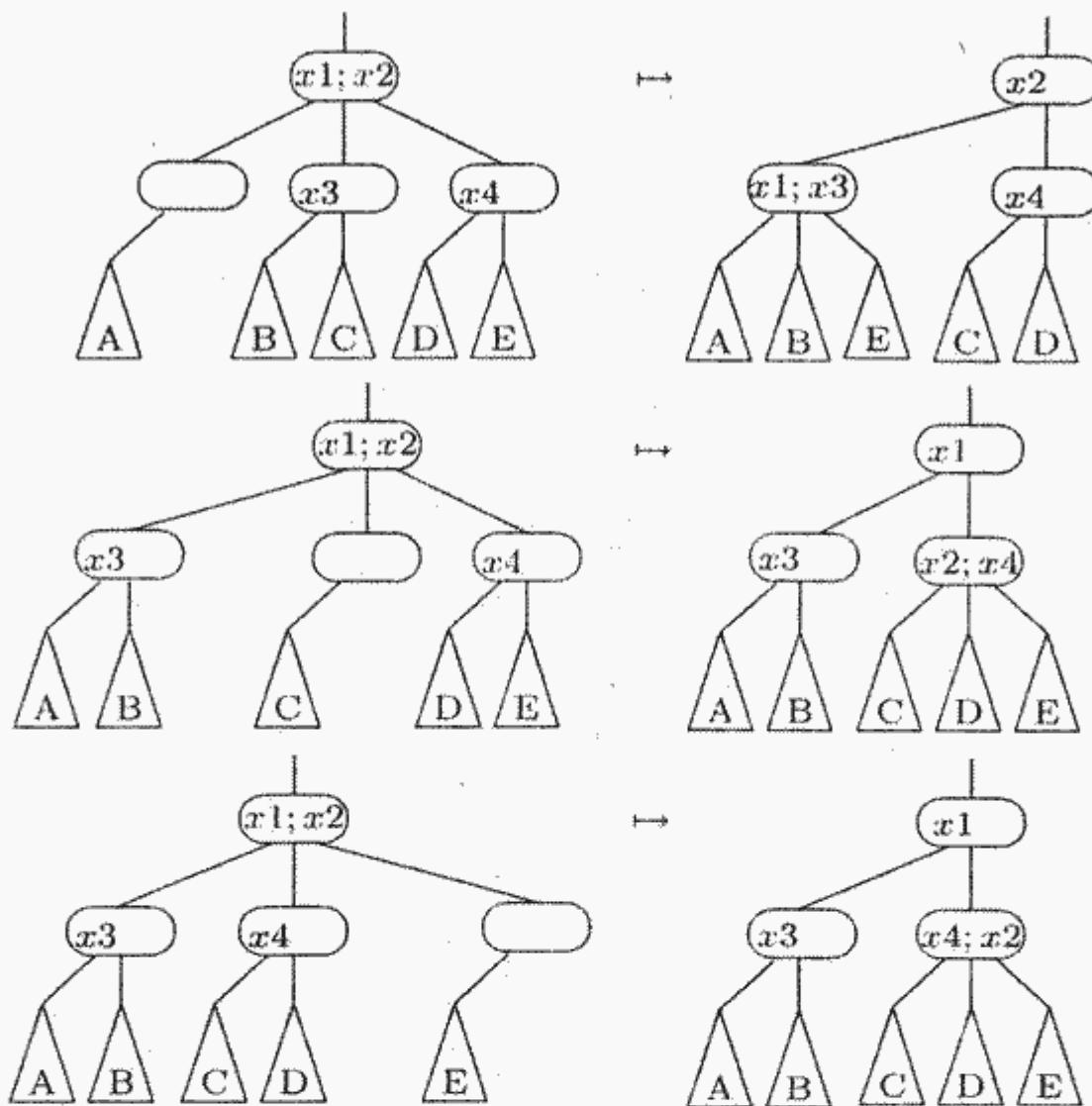


Figura 6.18: Ștergere într-un 2-3-arbore: combinarea nodurilor când v are aritatea 2, iar tatăl său aritatea 3

6.1.4 B-arbore

B-arborii sunt frecvent utilizati la indexarea unei colecții de date. Un index ordonat este un fișier secvențial. Pe măsură ce dimensiunea acestuia crește, cresc și dificultățile de administrare. Soluția este construirea unui index cu mai multe nivele, iar instrumentele sunt B-arborii. Algoritmii de căutare într-un index nestratificat nu pot depăși performanța $O(\log_2 n)$ intrări/ieșiri. Indexarea multistratificată (vezi figura 6.19) are ca rezultat algoritmi de căutare de ordinul $O(\log_d n)$ intrări/ieșiri, unde d este dimensiunea elementului indexului.

Definiția 6.8. Un B-arbore este un arbore cu rădăcină, în care:

1. fiecare nod intern are un număr variabil de chei și fi;
2. cheile dintr-un nod intern sunt memorate în ordine crescătoare;
3. fiecare cheie dintr-un nod intern are asociat un fiu care este rădăcina unui subarbore ce conține toate nodurile cu chei mai mici sau egale cu cheia respectivă dar mai mari decât cheia precedentă;
4. fiecare nod intern are un fiu extrem-dreapta, care este rădăcina unui subarbore ce conține toate nodurile cu chei mai mari decât oricare cheie din nod;

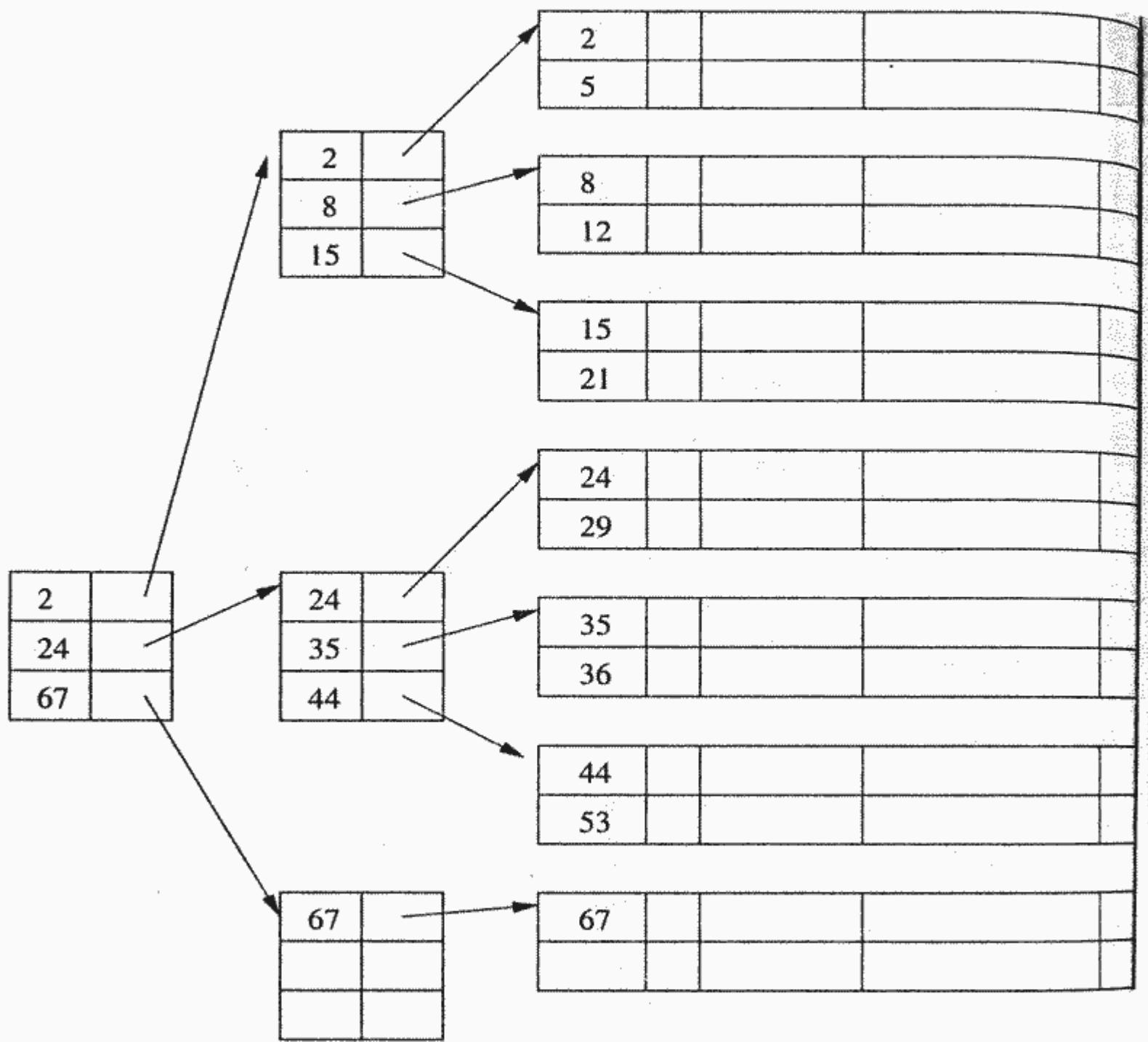


Figura 6.19: Index organizat pe 3 nivele

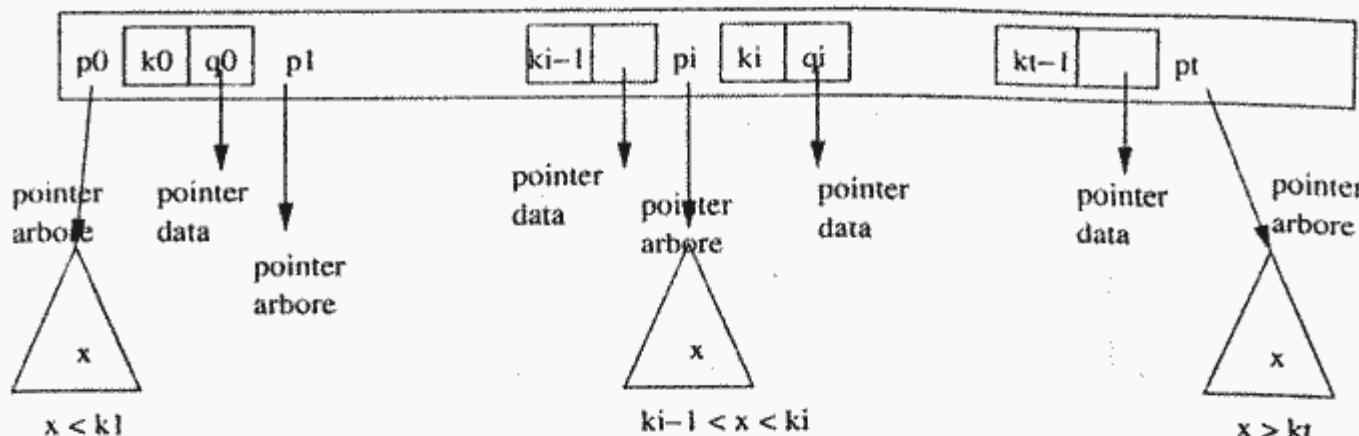


Figura 6.20: Structura unui nod al unui B-arbore

5. fiecare nod intern are cel puțin un număr de $f - 1$ chei (f fi), $f =$ factorul de minimizare;
6. doar rădăcina poate avea mai puțin de $f - 1$ chei (f fi);
7. fiecare nod intern are cel mult $2f - 1$ chei (f fi);
8. lungimea oricărui drum de la rădăcină la o frunză este aceeași.

Dacă fiecare nod necesită accesarea discului, atunci B-arborii vor necesita un număr minim de astfel de accesări. Factorul de minimizare va fi ales astfel încât dimensiunea unui nod să corespundă unui multiplu de blocuri ale dispozitivului de memorare. Această alegere optimizează accesarea discului.

Înălțimea h unui B-arbore cu n noduri și $f > 1$ satisfacă relația $h \leq \log_f \frac{n+1}{2}$.

Un nod v al unui B-arbore poate fi implementat cu o structură statică formată din câmpurile `nrChei`, `tipNod`, `cheie[nrChei]`, `data[nrChei]` și `fiu[nrChei+1]`. Câmpul `tipNod` conține o valoare $tipNod \in \{frunza, interior\}$, câmpul `nrChei` conține numărul t al cheilor din nodul v , tabloul `cheie[nrChei]` conține valorile cheilor memorate în nod $(k_0, k_1, \dots, k_{t-1})$, tabloul `data[nrChei]` conține pointerii la structurile care conțin datele asociate nodului v ($(q_0, q_1, \dots, q_{t-1})$), iar tabloul `fiu[nrChei]` conține pointerii la structurile care implementează fiii nodului v ((p_0, p_1, \dots, p_t)) (vezi figura 6.20).

Căutarea într-un B-arbore este asemănătoare cu căutarea într-un arbore binar. Deoarece timpul de căutare depinde de adâncimea arborelui, `căutareBarbore` are timpul de execuție $O(\log_f n)$ ([CLR93]).

```

cautareBarbore(v, k)
    i ← 0
    while (i < v->nrChei and k > v->cheie[i]) do i ← i+1
    if (i < v->nrChei and k = v->cheie[i]) then return (v, i)
    if (v->tipNod = frunza)
        then return NULL
        else citesteMemorieExterna(v->fiu[i])
            return cautareBarbore(v->fiu[i], k)
end

```

Operația `creeazaBarbore` creează un B-arbore vid cu rădăcina t fără fi. Timpul de execuție este $O(1)$.

```

creeazaBarbore(t)
    new t
    t->tipNod ← frunza
    t->nrChei ← 0
    scrieMemorieExterna(t)
end

```

Dacă un nod v este încărcat la maxim ($2f - 1$ chei), pentru a insera o cheie nouă este necesară spargerea acestuia. Prin spargere, cheia mediană a nodului v este mutată în părintele u al acestuia (v este al i -lea fiu). Este creat un nou nod w și toate cheile din v situate la dreapta cheii mediane sunt mutate în w . Cheile din v situate la stânga cheii mediane rămân în v . Nodul nou w devine fiu imediat la dreapta cheii mediane care a fost mutată în părintele u , iar v devine fiu imediat la stânga cheii mediane care a fost mutată în părintele u . Spargerea transformă nodul cu $2f - 1$ chei în două noduri cu $f - 1$ chei (o cheie este mutată în părinte). Timpul de execuție este $O(t)$ unde $t = \text{constant}$.

```

procedure spargeNod(u, i, v)
    new nod w
    w->tipNod ← v->tipNod
    w->nrChei ← f-1
    for j ← 0 to f-2 do w->cheie[j] ← v->cheie[j+f]
    if (v->tipNod = interior)
        then for j ← 0 to f-1 do w->fiu[j] ← v->fiu[j+f]
    v->nrChei ← f-1
    for j ← u->nrChei downto i+1 do u->fiu[j+1] ← u->fiu[j]
    u->fiu[i+1] ← w
    for j ← u->nrChei-1 downto i do u->cheie[j+1] ← u->cheie[j]
    u->cheie[i] ← v->cheie[f-1]
    u->nrChei ← u->nrChei + 1
    scrieMemorieExterna(u)
    scrieMemorieExterna(v)
    scrieMemorieExterna(w)
end

```

Dacă un nod conține mai puțin de $2f - 1$ chei, pentru a insera o cheie nouă în subarborele care are rădăcina nodul respectiv, se folosește subprogramul insereaza_inNodIncomplet(v, k) :

```

procedure insereaza_inNodIncomplet(v, k)
    i ← v->nrChei-1
    if (v->tipNod = frunza)
        then while (i >= 0 and k < v->cheie[i]) do
            v->cheie[i+1] ← v->cheie[i]
            i ← i-1
            v->cheie[i+1] ← k
            v->nrChei ← v->nrChei + 1
            scrieMemorieExterna(v)
    else while (i >= 0 and k < v->cheie[i]) do i ← i-1

```

```

        i ← i+1
        citesteMemorieExterna(v->fiu[i])
        if (v->fiu[i]->nrChei = 2f-1)
            then spargeNod(v, i, v->fiu[i])
                if (k > v->cheie[i]) then i ← i+1
                insereaza_inNodIncomplet(v->fiu[i], k)
    end

```

Pentru a efectua o inserție într-un B-arbore trebuie întâi să găsim nodul în care urmează să se face inserția. Pentru aceasta se aplică un algoritm similar cu `cautareBarbore`. Apoi cheia urmează să fie inserată. Dacă nodul determinat anterior conține mai puțin de $2f - 1$ chei, se face inserarea în nodul respectiv. Dacă acest nod conține $2f - 1$ chei, urmează spargerea acestuia. Procesul de spargere poate continua până în rădăcină. Pentru a evita două citiri de pe disc ale aceluiași nod, algoritmul sparge fiecare nod plin ($2f - 1$ chei) întâlnit la parcurgea *top-down* în procesul de căutare a nodului în care urmează să se face inserarea. Timpul de spargere a unui nod este $O(f)$. Rezultă pentru inserție complexitatea timp $O(f \log n)$.

```

procedure insereazaBarbore(t, k)
    v ← t
    if (v->nrChei = 2f-1)
        then new nod u
            t ← u;
            u->tipNod ← interior
            u->nrChei ← 0
            u->fiu[0] = v
            spargeNod(u, 0, v)
            insereaza_inNodIncomplet(u, k)
        else insereaza_inNodIncomplet(v, k)
    end

```

Operația de stergere a unei chei se desfășoară astfel:

1. dacă nodul-gazdă al cheii ce urmează să fie stearsă nu este frunză, atunci se efectuează o interschimbare între aceasta și succesorul ei în ordinea naturală (crescătoare) a cheilor. Deoarece cheia succesoare se află într-o frunză, operația se reduce la stergerea unei chei dintr-o frunză.
2. se sterge intrarea corespunzătoare cheii;
3. dacă nodul curent conține cel puțin $f - 1$ chei, operația de stergere se consideră terminată;
4. dacă nodul curent conține mai puțin decât $f - 1$ chei, se consideră frații vecini;
5. dacă unul din frații vecin are mai mult decât $f - 1$ chei, atunci se redistribuie una dintre intrările acestui frate în nodul-părinte și una din intrările din nodul părinte se redistribuie nodului curent (deficitar);
6. dacă ambii frații au exact $f - 1$ chei, atunci se unește nodul curent cu unul dintre frații vecini și cu o intrare din părinte;
7. dacă nodul-părinte devine deficitar (conține mai puțin decât $f - 1$ chei), acesta devine nod curent și se reiau pașii 5-6.

6.1.5 Exerciții

Exercițiu 6.1.1. Într-un arbore AVL vid se inserează în ordine cheile $1, 2, \dots, 2^{k-1}$. Să se arate că arborele rezultat este perfect echilibrat.

Exercițiu 6.1.2. Câți biți per nod sunt necesari pentru memorarea înălțimii nodului pentru un arbore AVL cu n noduri? Care este cel mai mic arbore AVL care depășește un contor de opt biți pentru memorarea înălțimii?

Exercițiu 6.1.3. Să se scrie un subprogram care generează un arbore AVL cu număr minim de noduri și de înălțime dată h . Care este timpul de execuție al programului?

Exercițiu 6.1.4. Să se scrie subprograme complete care să realizeze operațiile de inserare și de ștergere într-un 2-3 arbore.

Exercițiu 6.1.5. [HSAF93] Să se definească o alternativă orientată pe frontieră la definiția 2-3 arborilor. Fiecare nod de pe frontieră conține exact un element, iar semnificațiile valorilor din nodurile interne sunt: $v\uparrow.valstg$ este cea mai mare valoare memorată într-un nod de pe frontieră subarborelui din stânga și $v\uparrow.valdrp$ este cea mai mare valoare memorată într-un nod de pe frontieră subarborelui din mijloc. De asemenea, toate nodurile de pe frontieră se vor afla pe același nivel.

1. Să se definească o structură de date care să reprezinte un 2-3 arbore orientat pe frontieră.
2. Să se scrie subprograme care să realizeze operațiile de căutare, inserare și ștergere într-un 2-3 arbore reprezentat în acest fel.
3. Să se arate că noua definiție păstrează proprietatea de clasă $O(\log n)$ -stabilă.

Exercițiu 6.1.6. Să se scrie programe care să construiască reprezentarea orientată pe frontieră a unui 2-3 arbore pornind de la reprezentarea pe noduri interne și, reciproc, să construiască reprezentarea orientată pe noduri interne a unui 2-3 arbore pornind de la reprezentarea pe frontieră.

6.2 Dispersia (hashing)

Dispersia este o tehnică aplicată în implementarea tabelelor de simboluri. O *tabelă de simboluri* este un tip de date abstract în care obiectele sunt perechi (*nume, attribute*) și asupra cărora se pot executa următoarele operații: căutarea unui nume în tabelă, regăsirea atributelor unui nume, modificarea atributelor unui nume, inserarea unui nou nume și a atributelor sale și ștergerea unui nume și a atributelor sale. Exemple tipice pentru tabelele de simboluri sunt dicționarele de sinonime (tezaurele) (nume = cuvânt și attribute = sinonime) și tabela de simboluri a unui compilator (nume = identificator și attribute = (valoare inițială, lista liniilor în care apare etc.)).

Întrucât operația de căutare se realizează după nume, notăm cu U mulțimea tuturor numelor. Implementarea tabelelor de simboluri prin tehnica dispersiei presupune:

- un tablou ($T[i] \mid 0 \leq i \leq p - 1$), numit și *tabelă de dispersie (hash)*, pentru memorarea numelor și a referințelor la mulțimile de attribute corespunzătoare;

- o funcție $h : \mathbb{U} \rightarrow [0, p-1]$, numită și *funcție de dispersie (hash)*, care asociază unui nume o adresă în tabelă.

O submulțime $S \subseteq \mathbb{U}$ este reprezentată în modul următor: pentru fiecare $x \in S$ se determină $i = h(x)$ și numele x va fi memorat în componenta $T[i]$. Valoarea funcției de dispersie se mai numește și *index* sau *adresă în tabelă*. Dacă pentru două elemente diferite x și y avem $h(x) = h(y)$, atunci spunem că între cele două elemente există *coliziune*. Pentru ca operația de dispersie să fie eficientă, trebuie rezolvate corect următoarele două probleme: alegerea funcției de dispersie și rezolvarea coliziunilor.

6.2.1 Alegerea funcției de dispersie

Prezentăm sumar câteva tehnici elementare de alegere a funcției de dispersie.

Trunchierea. Se ignoră o parte din reprezentarea numelui. De exemplu, dacă numele sunt reprezentate prin secvențe de cifre și $p = 1000$, atunci $f(x)$ poate fi numărul format din ultimele trei cifre ale reprezentării: $f(62539194) = 194$.

„Folding”. Reprezentarea numelui este partionată în câteva părți și apoi aceste părți sunt combinate pentru a obține indexul. De exemplu, reprezentarea 62539194 este partionată în părțile 625, 391 și 94. Combinarea părților ar putea consta, de exemplu, în adunarea lor: $625 + 391 + 94 = 1110$. În continuare se poate aplica și trunchierea: $1110 \mapsto 110$. Deci $f(62539194) = 110$.

Aritmetică modulară. Se convertește reprezentarea numelui într-un număr și se ia ca rezultat restul împărțirii la p . Este de preferat ca p să fie prim, pentru a avea o repartizare cât mai uniformă a elementelor în tabelă.

Exemplu. Presupunem că un nume este reprezentat printr-un sir de caractere ASCII. Notăm cu $\text{int}(c)$ funcția care întoarce codul zecimal al caracterului c .

```
function hash(x)
    h ← 0
    for i ← 0 to lung(x)-1 do
        h ← h + int(x[i])
    return h%p
end
```

sfex

Multiplicare. Fie w cel mai mare număr ce poate fi memorat într-un cuvânt al calculatorului (de exemplu, $w = 2^{32}$, dacă cuvântul are 32 de biți). Funcția de dispersie prin multiplicare este dată de

$$h(x) = \left\lfloor p \cdot \left\{ \frac{x}{w} \right\} \right\rfloor$$

unde $\{y\} = y - \lfloor y \rfloor$ și A este o constantă convenabil aleasă. De obicei, se ia A prim cu w și p putere a lui 2.

6.2.2 Coliziunea

Există două tehnici de bază pentru rezolvarea coliziunii: înlățuirea și adresarea deschisă.

Dispersie cu înlățuire. Numele cu aceeași adresă sunt memorate într-o listă liniară simplu înlățuită (figura 6.21). Notăm cu s .nume câmpul care memorează numele unui simbol s și cu s .latr câmpul ce memorează lista (sau adresa listei) de atribută a lui s .

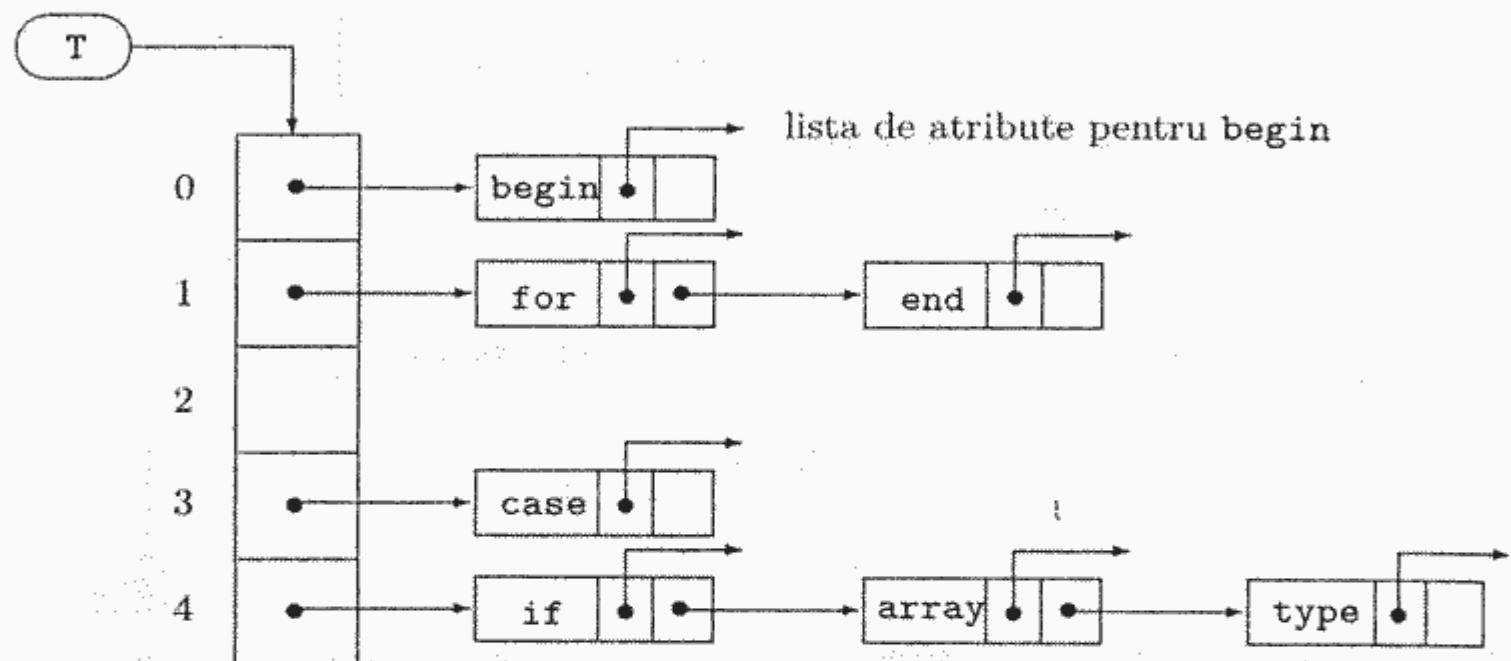


Figura 6.21: Dispersie cu înlățuire

Căutarea unui nume în tabelă se realizează în două etape: mai întâi se calculează adresa în tabelă corespunzătoare numelui și apoi se caută secvențial în lista simplu înlățuită.

```

function poz(x, T)
  i ← hash(x)
  p ← T[i]
  while (p ≠ NULL) do
    if (p->nume = x)
      then return p
    else p ← p->succ
  return NULL
end
  
```

Regăsirea atributelor unui nume presupune mai întâi căutarea numelui în tabelă:

```

function attribute(x, T)
  p ← poz(x, T)
  if (p ≠ NULL)
    
```

```

        then return p->latr
    else return NULL
end

```

Operația de adăugare a unui element x în tabelă presupune calculul lui $i = h(x)$ și inserarea acestuia în lista $T[i]$. Pentru ca operația de adăugare să se realizeze cu cât mai puține operații, inserarea se va face la începutul listei.

```

procedure insereaza(x, xatr, T)
    i ← hash(x)
    new(p)
    p->nume ← x
    p->latr ← xatr
    p->succ ← T[i]
    T[i] ← p
end

```

Operația de stergere a numelui x presupune căutarea pentru x în lista $T[h(x)]$ și eliminarea nodului corespunzător. Odată cu eliminarea numelui sunt eliminate și atributele acestuia.

```

procedure elimina(x, T)
    i ← hash(x)
    p ← T[i]
    predp ← NULL
    while ((p ≠ NULL) and (p->nume ≠ x)) do
        predp ← p
        p ← p->succ
    if (p ≠ NULL)
        then delete(p->latr) /* elimina toata lista */
        if (p = T[i])
            then T[i] ← p->succ
            else predp->succ ← p->succ
    delete(p)
end

```

Exercițiul 6.2.1. Să se scrie subprograme care să realizeze modificări ale atributelor corespunzătoare unui nume. Aceste modificări vor include adăugarea de noi atribut, stergerea de atribut sau înlocuirea de atribut.

Teorema 6.6. Presupunem că tabela T conține elementele mulțimii $S \subset U$. Operațiile de adăugare și de stergere au complexitatea timp pentru cazul cel mai nefavorabil egală cu $O(\#S)$.

Pentru a calcula complexitatea medie, presupunem că funcția de dispersie distribuie uniform elementele din U și că numele apar cu aceeași probabilitate ca argumente ale operațiilor de căutare.

Teorema 6.7. Numărul mediu de comparații pentru o căutare cu succes este aproximativ $1 + \frac{\beta}{2}$, unde $\beta = \frac{\#S}{p}$ este factorul de încărcare al tăbelei.

Demonstrație. *Metoda 1.* Știm că o căutare cu succes într-o listă liniară de lungime m necesită în medie $\frac{m+1}{2}$. Lungimea medie a unei liste este β , iar probabilitatea ca elementul căutat să aparțină unei liste este $\frac{1}{p}$ (se presupune o dispersie uniformă). Numărul mediu de comparații pentru o căutare cu succes este:

$$1 + \sum_{i=0}^{p-1} \frac{\beta + 1}{2} \cdot \frac{1}{p} = 1 + \frac{\beta + 1}{2}$$

Metoda 2. Presupunem pentru moment că inserările elementelor se fac la sfârșitul listelor. Căutarea pentru al i -lea element inserat necesită un număr de comparații egal cu 1 plus lungimea listei la sfârșitul căreia a fost adăugat. La momentul inserării elementului i , lungimea medie a listelor este $\frac{i-1}{p}$. Rezultă că numărul mediu de comparații pentru o căutare cu succes este:

$$\frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{p} \right) = 1 + \frac{\beta}{2} - \frac{1}{2p}$$

unde $n = \#S$.

sfdem

Exercițiul 6.2.2. Să se realizeze o implementare a dispersiei cu înlănțuire utilizând arbori binari de căutare în loc de liste liniare simplu înlănțuite.

Dispersie cu adresare deschisă. Pentru un nume $x \in U$ se definește o secvență $h(x, i)$, $i = 0, 1, 2, \dots$, de poziții în tabelă. Această secvență, numită și secvență de încercări, va fi cercetată ori de câte ori apare o coliziune. De exemplu, pentru operația de adăugare, se caută cel mai mic i pentru care locația de la adresa $h(x, i)$ este liberă. O metodă uzuală de definire a secvenței $h(x, i)$ constă dintr-o combinație liniară:

$$h(x, i) = (h_1(x) + c \cdot i) \bmod p$$

unde $h_1(x)$ este o funcție de dispersie, iar c o constantă. Se pot considera și forme pătratice:

$$h(x, i) = (h_1(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod p$$

O altă secvență de încercări des utilizată este următoarea:

$$h(x), h(x) - 1, \dots, 0, p - 1, p - 2, \dots, h(x) + 1$$

unde h este o funcție de dispersie.

Teorema 6.8. Pentru dispersia cu adresare deschisă, numărul mediu de încercări pentru o căutare fără succes este cel mult $\frac{1}{1-\beta}$ ($\beta < 1$).

Demonstrație. Vom presupune că $h(x, 0), h(x, 1), h(x, 2), \dots, h(x, p-1)$ realizează o permutare a mulțimii $\{0, 1, 2, \dots, p-1\}$. Aceasta asigură faptul că o inserare se realizează cu succes ori de câte ori există o poziție liberă în tabelă. Notăm cu X variabila aleatoare care întoarce numărul de încercări în poziții ocupate și cu p_i

probabilitatea $Pr(X = i)$. Evident, dacă $i > n$ ($n = \#S$), atunci $p_i = 0$. Numărul mediu de încercări M în poziții ocupate este:

$$\begin{aligned} 1 + \sum_{i=0}^n i \cdot p_i &= \\ 1 + \sum_{i=0}^{\infty} i \cdot p_i &= \\ 1 + \sum_{i=0}^{\infty} i \cdot Pr(X = i) &= \\ 1 + \sum_{i=0}^{\infty} i \cdot (Pr(X \geq i) - Pr(X \geq i+1)) &= \\ 1 + \sum_{i=1}^{\infty} Pr(X \geq i) &= \\ 1 + \sum_{i=1}^{\infty} q_i \end{aligned}$$

Avem $q_1 = \frac{n}{p}$, $q_2 = \frac{n}{p} \cdot \frac{n-1}{p-1} \leq \left(\frac{n}{p}\right)^2$, ... de unde rezultă

$$M = 1 + \sum_{i=1}^{\infty} \beta^i = \frac{1}{1-\beta}$$

sfdem

Corolar 6.1. [Knu76] Pentru dispersia cu adresare deschisă liniară, numărul mediu de încercări pentru o căutare cu succes este $\frac{1}{2} \left[1 + \frac{1}{1-\beta} \right]$.

6.2.3 Exerciții

Exercițiul 6.2.3. [HSAF93] Să se proiecteze o reprezentare a unei tabele de simboluri care să permită căutarea, inserarea și ștergerea unui nume întreg x în timpul $O(1)$. Se presupune $0 \leq x < m$ și că sunt disponibile $n+m$ registre de memorie, unde n este numărul de inserări.

Indicație. Se vor utiliza două tablouri $(a[i] \mid 0 \leq i < n)$ și $(b[j] \mid 0 \leq j < m)$ cu semnificațiile: dacă x este cel de-al i -lea nume inserat, atunci $a[i] = x$ și $b[x] = i$. Se va evita inițializarea tablourilor, deoarece aceasta necesită timpul $O(n+m)$.

Exercițiul 6.2.4. [HSAF93] Fie $S = \{x_0, \dots, x_{n-1}\}$ și $T = \{y_0, \dots, y_{r-1}\}$. Se presupune $0 \leq x_i < m$, $0 \leq i < n$, și $0 \leq y_j \leq m$, $0 \leq j < r$. Utilizând ideea de la exercițiul 6.2.3 să se scrie un algoritm care să determine dacă $S \subseteq T$. Complexitatea timp a algoritmului va fi $O(n+r)$. Deoarece $S = T$ dacă și numai dacă $S \subseteq T$ și $T \subseteq S$, rezultă că se poate testa dacă două mulțimi sunt echivalente în timp liniar. Care este complexitatea spațiu a algoritmului?

Exercițiul 6.2.5. [Knu76] Pentru scrierea unui compilator FORTRAN se utilizează o tabelă de simboluri pentru memorarea numelor de variabile care apar în programul FORTRAN ce se compilează. Aceste nume pot avea cel mult 8 caractere. S-a luat decizia ca dimensiunea tabelei să fie 128 și funcția de dispersie să fie $h(x) = \text{"caracterul cel mai semnificativ al numelui } x\text{"}$. Este o idee bună? Justificați răspunsul.

Exercițiul 6.2.6. [Knu76] Fie $h(x)$ o funcție de dispersie și $q(x)$ o funcție de argument x cu proprietatea că x poate fi determinat dacă se cunoște $h(x)$ și $q(x)$. De exemplu, în cazul împărțirii modulare avem $h(x) = x \bmod p$ și $q(x) = \lfloor \frac{x}{p} \rfloor$, iar în cazul dispersării prin multiplicare vom lua $h(x)$ rangurile semnificative ale lui $\{\frac{A \cdot x}{w}\}$ și $q(x)$ celealte ranguri (când p este putere a lui 2). Să se arate că dacă se utilizează înlățuirea, atunci este suficient să se memoreze în fiecare înregistrare $q(x)$ în loc de x .

Exercițiul 6.2.7. [CLR93] O tabelă de dispersie de dimensiune p este utilizată pentru a memora n elemente, $n \leq \frac{p}{2}$. Se presupune că se utilizează adresarea deschisă pentru rezolvarea coliziunilor.

1. Presupunând că dispersia este uniformă, arătați că, pentru $i = 0, 1, \dots, n-1$, probabilitatea ca la i -a inserare să necesite mai mult de k încercări este cel mult 2^{-k} .
2. Notăm cu X_i variabila aleatoare care întoarce numărul de încercări necesare la i -a inserare și prin X variabila aleatoare $\max_i X_i$. Să se arate că:
 - a) $Pr(X > \log n) \leq \frac{1}{n}$.
 - b) $M(X) = O(\log n)$.

Exercițiul 6.2.8. Un număr mare de ștergeri într-o tabelă de dispersie cu înlățuire conduce la o tabelă aproape vidă și deci la irosire a spațiului de memorie. O soluție este înjumătățirea tabelei. În mod similar, un număr mare de inserări conduce la dublarea dimensiunii tabelei. Presupunem că tabela este dublată când numărul elementelor memorate în tabelă este de două ori mai mare decât dimensiunea tabelei. Care este criteriul pentru înjumătățirea tabelei?

6.3 Arbori digitali (tries)

Până acum am studiat numai structuri de date pentru reprezentarea mulțimilor S , astfel încât pentru operațiile de căutare, inserare și stergere să existe algoritmi eficienți din punctul de vedere al complexității timp. În această subsecțiune vom studia o structură de date care, în anumite situații, reduce semnificativ spațiul de memorie necesar reprezentării mulțimii S . Considerăm exemplul când S este un dicționar. Se poate reduce spațiul necesar pentru memorarea dicționarului dacă pentru cuvintele cu rădăcina comună, aceasta este reprezentată o singură dată. Definiția arborilor digitali are ca punct de plecare această idee.

Un arbore digital este o structură de date care se bazează pe reprezentarea digitală (cu cifre) a elementelor din mulțimea univers. Denumirea de *trie* (în unele cărți *trie*) a fost dată de E. Fredkin (CACM, 3, 1960, pp. 490-500) și constituie o parte din expresia din limba engleză *information-retrieval*. Un arbore digital este de fapt un arbore cu rădăcină ordonat k -ar (fiecare vârf are cel mult k succesiuni), unde k este numărul de cifre (litere dintr-un alfabet) necesare pentru reprezentarea elementelor mulțimii univers. Se presupune că toate elementele sunt reprezentate prin secvențe de cifre (litere) de aceeași lungime m . Astfel, mulțimea univers conține m^k elemente.

Exemplu. Presupunem că elementele mulțimii univers sunt codificate prin secvențe de trei cifre din alfabetul $\{0, 1, 2\}$. Mulțimea de chei $S = \{121, 102, 211, 120, 210, 212\}$ este reprezentată prin arborele din figura 6.22.

sfex }

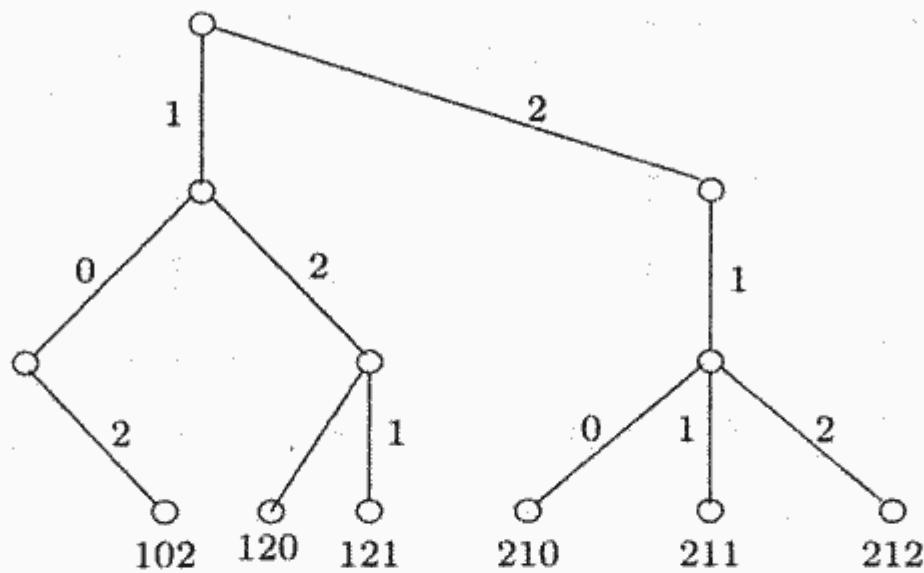


Figura 6.22: Arbore digital

Reprezentarea cuvintelor prin arbori digitali aduce o economie de memorie numai în cazul când există multe prefixe comune. În figura 6.23 este reprezentat un exemplu când spațiul ocupat de arboarele digital este mai mică decât cel ocupat de lista liniară a cuvintelor.

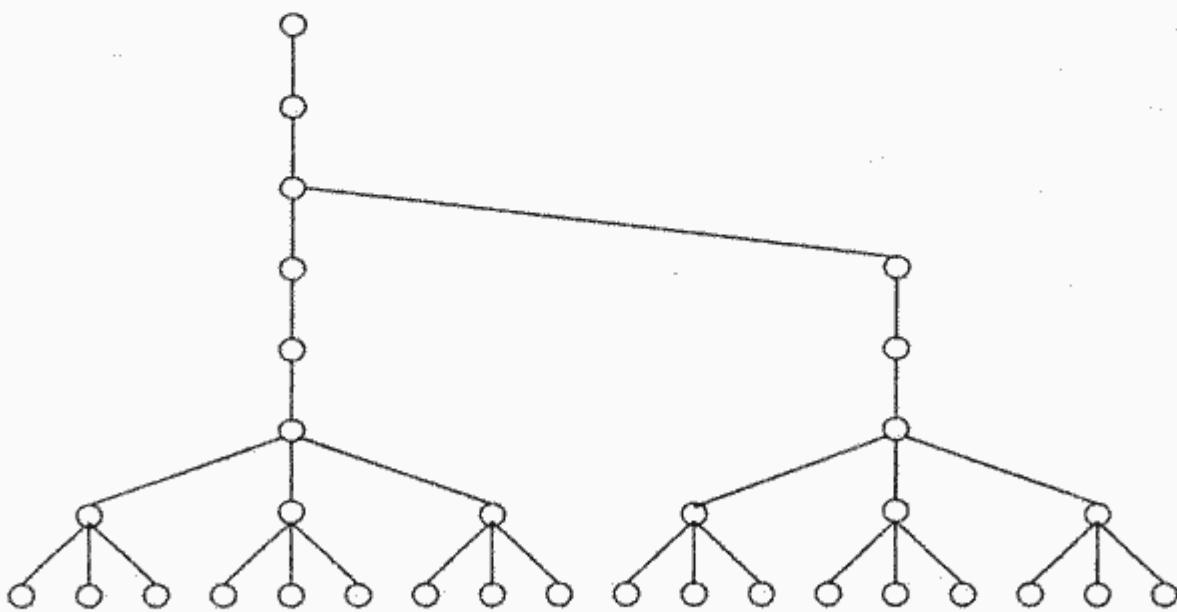


Figura 6.23: Cazul când există multe prefixe comune

Un arbore tri poate fi reprezentat în memoria calculatorului printr-o listă înlățuită în care fiecare nod v are k câmpuri de legătură. ($v.\text{succ}[j] \mid 0 \leq j < k$), ce memorează adresele fililor lui v . Pentru simplitatea prezentării, presupunem că alfabetul este $\{0, \dots, k-1\}$. Arboarele din figura 6.22 este reprezentat prin structura

din figura 6.24. Elementele din mulțimea S sunt gădite drept chei, iar nodurile de pe frontieră memorează informațiile asociate acestor chei.

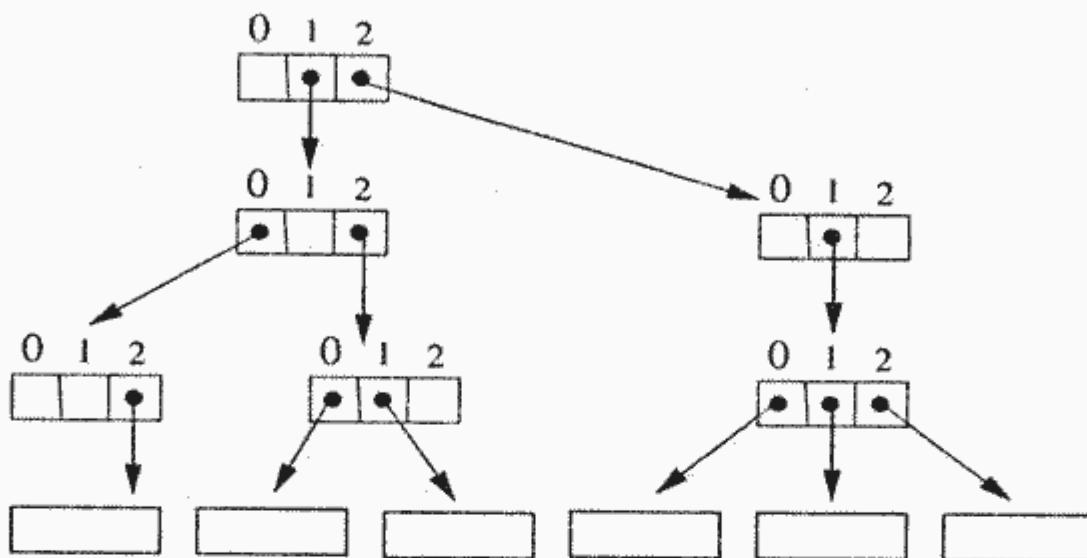


Figura 6.24: Structura de date pentru arborele din figura 6.22

Căutarea. Căutarea pentru un element a în structura t constă în încercarea de a parcurge drumul în arbore descris de secvență $(a[0], \dots, a[m - 1])$. Parcursarea completă a drumului înseamnă căutare cu succes ($a \in S$), iar parcursarea parțială are semnificația căutării fără succes:

```
function poz(a, m, t)
begin
    i ← 0
    p ← t
    while ((p ≠ NULL) and (i < m)) do
        p ← p->succ[a[i]]
        i ← i+1
    return p
end
```

Complexitatea timp pentru cazul cel mai nefavorabil este $O(m)$. De notat că aceasta nu depinde de numărul cuvintelor n . Pentru ca această căutare să fie mai eficientă decât căutarea binară trebuie ca $n > 2^m$.

Inserarea. Programul care realizează operația de inserare a unui cuvânt x la structura t simulează parcursarea drumului descris de secvență $(x[0], \dots, x[m - 1])$. Pentru acele componente $x[i]$ pentru care nu există noduri în t se va adăuga un nou nod ca succesor celui corespunzător lui $x[i - 1]$.

Eliminarea. Considerăm mai întâi un exemplu. Dacă din structura din figura 6.22 eliminăm elementul reprezentat de 102, atunci este necesară și eliminarea a două noduri (cele corespunzătoare componentelor 0 și 2), iar ștergerea elementului 210 presupune eliminarea unui singur nod (cel corespunzător lui 0). Deci un element

care urmează a fi eliminat este împărțit în două: un prefix care este comun și altor elemente care există în structură, și deci nu trebuie distrus, și un sufix care nu mai aparține niciunui element și deci poate fi șters. Astfel, strategia pe care o urmează programul descris aici este următoarea:

- se parcurge drumul descris de elementul x ce urmează a fi șters și se memorează acest drum într-o stivă;
- parcurge acest drum înapoi, utilizând stiva, și dacă pentru un nod de pe acest drum toți succesorii sunt egali cu *nil*, atunci se elimină acel nod.

Observație. Se poate elibera utilizarea stivei, dacă la parcurgerea „dus” se determină ultimul nod care are mai mult de un succesor și litera corespunzătoare acestui nod. Rezultă că ștergerea se poate realiza cu $O(1)$ spațiu suplimentar. sfobs

Atât inserarea, cât și eliminarea se realizează în timpul cel mai nefavorabil $O(n)$.

6.3.1 Cazul cheilor cu lungimi diferite

Există multe situații practice când cheile nu aceeași lungime. De exemplu, pentru un dicționar de sinonime, cheile sunt cuvinte și, evident, acestea au lungimi diferite. În plus, este foarte posibil ca un cuvânt să fie prefix al altui cuvânt. În acest caz se pune problema evidențierii nodurilor care corespund cheilor. Soluția cea mai la indemână este de a adăuga un nou câmp la structura unui nod. Noul câmp va avea următoarea semnificație: dacă nodul corespunde unei chei, atunci câmpul va referi informația asociată cheii (de exemplu, lista sinonimelor); altfel, câmpul va avea valoarea NULL. Un fragment de astfel de structură este reprezentat în figura 6.25. Algoritmii de căutare, inserare și eliminare se obțin din cei descriși în cazul cheilor de lungimi egale prin adăugarea operațiilor de testare și de actualizare a câmpului nou adăugat.

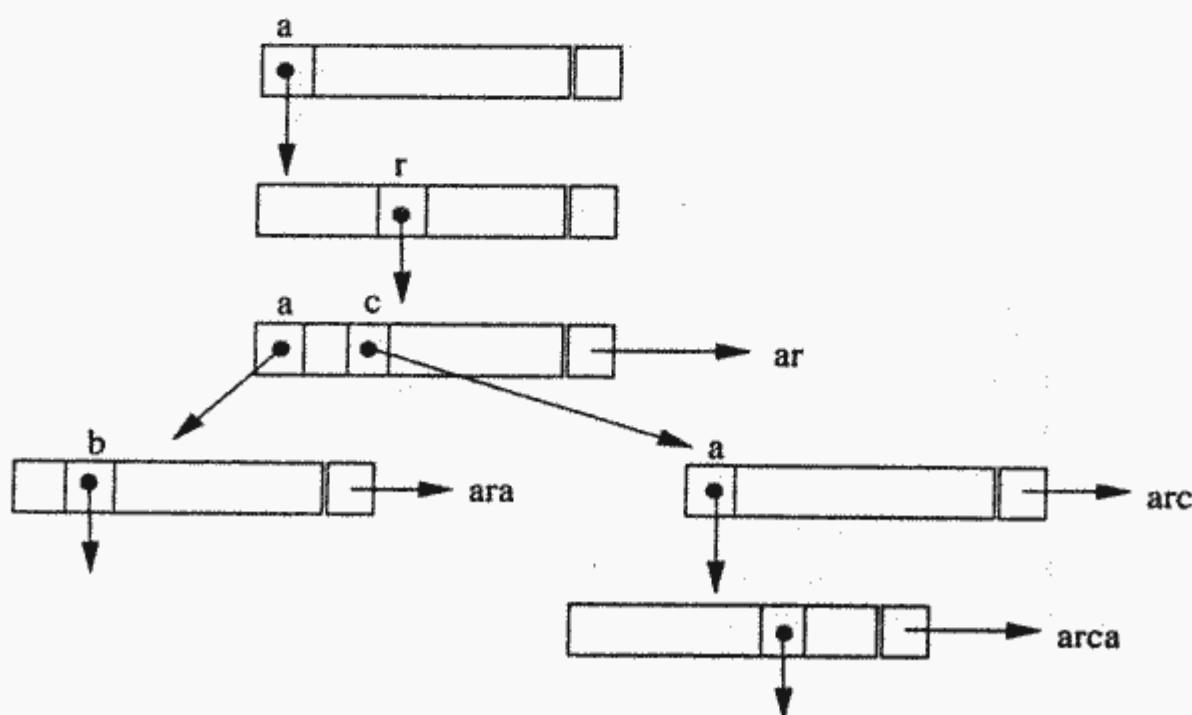


Figura 6.25: Arbore digital memorând chei cu lungimi diferite

6.3.2 Compactarea lanțurilor

O structură rară este o structură care memorează puține chei. Într-o structură rară apar multe lanțuri formate din noduri intermediare cu un singur succesor. Un exemplu de structură rară este reprezentat în figura 6.26. Atât spațiul de memorie cât și timpul de realizare a operațiilor pot fi îmbunătățite prin eliminarea acestor lanțuri. Astfel, în structură vor fi păstrate numai nodurile intermediare care au cel puțin doi succesi. Aceste noduri sunt utilizate pentru a distinge între cheile cu un prefix comun, descris de drumul de la rădăcină la nod în arborele inițial, și care diferă prin litera de la poziția corespunzătoare nodului. Deoarece această poziție nu mai este egală cu lungimea drumului de la rădăcină la nod, trebuie ca ea să fie memorată în nod. În plus, pentru că procesul de căutare a unei chei nu mai implică o parcurgere completă a cheii, există posibilitatea ca același nod de pe frontieră să fie atins cu chei diferite. Aceasta este eliminată prin memorarea cheii în nodurile de pe frontieră; atingerea nodului de pe frontieră este însoțită de testarea dacă cheia memorată coincide cu cea căutată. Așadar, prețul plătit este un câmp nou în structura nodului, câmp care memorează poziția din cheie testată pentru a decide succesorul pe care se continuă procesul de căutare, și memorarea cheilor în nodurile de pe frontieră. Rezultatul compactării structurii din 6.26 este reprezentat în figura 6.27.

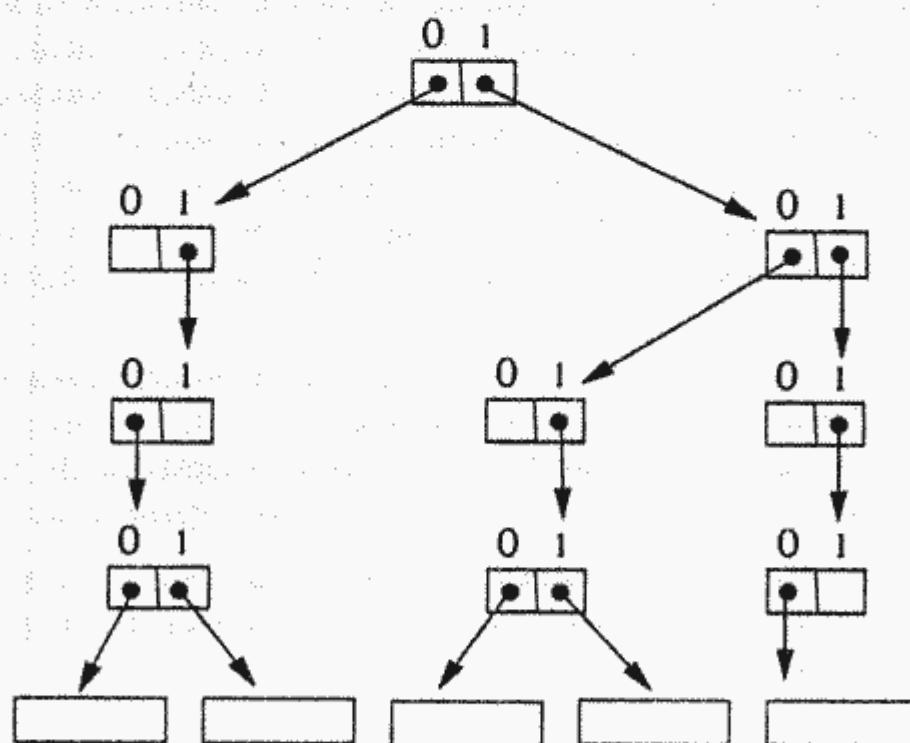


Figura 6.26: Structură rară

Căutarea. Operația de căutare este similară cazului necompactat, cu deosebirea că în fiecare nod este testată poziția din memorată de acel nod iar atingerea unui nod de pe frontieră este însoțită de testarea cheii. De exemplu, căutarea cheii $a = 1010$ în structura din figura 6.27 se realizează astfel: Se pleacă din rădăcină. Poziția memorată în rădăcină este 0. Deoarece $a[0] = 1$, următorul nod interogat este fiul

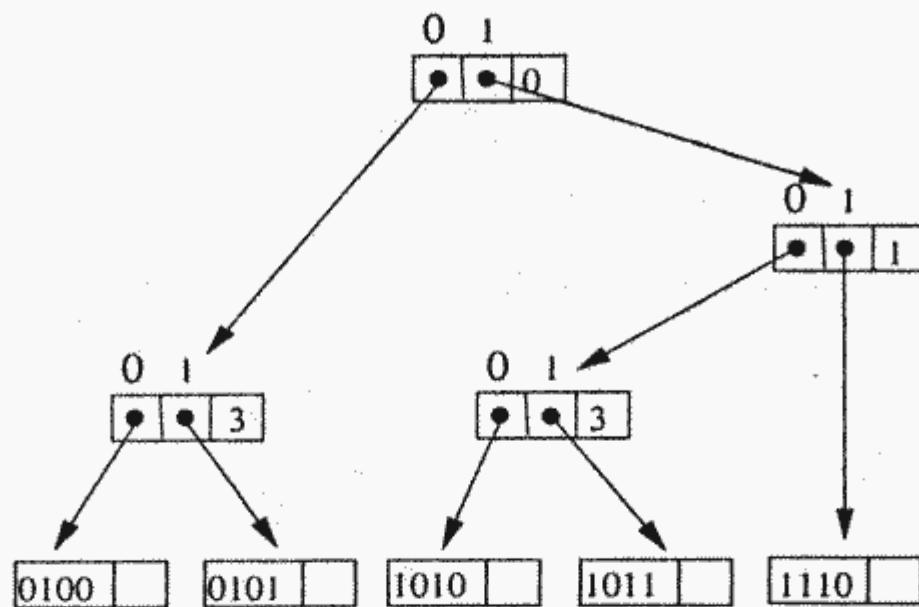


Figura 6.27: Structura din 6.26 compactată

drept. Poziția memorată de acesta este 1 și pentru că $a[1] = 0$, următorul nod este fiul stâng. Poziția memorată aici este 3 și pentru că $a[3] = 0$, următorul nod este fiul stâng. Acest nod este pe frontieră și testăm dacă a coincide cu cheia memorată în nod. Deoarece avem egalitate, căutarea se termină cu succes. De notat că același nod de pe frontieră poate fi atins și cu cheia 1000.

Inserarea. Presupunem că dorim inserarea cheii $a = 0111$ în structura din figura 6.27. În primul pas se realizează o operație de căutare a acestei chei. Căutarea se termină în nodul ce memorează cheia 0101. Cele două chei diferă pe poziția 2. Trebuie să inserăm un nou nod care să distingă după această poziție. Deoarece $0 < 2 < 3$, acest nod va fi fiu-stânga pentru rădăcină și nod-tată pentru nodul care distinge după poziția 3. Acesta din urmă va fi fiu-stânga pentru nodul nou inserat. Mai adăugăm un nou nod care memorează noua cheie a și care va fi fiu-dreapta pentru nodul ce face distincție după poziția 2. Rezultatul este reprezentat în figura 6.28.

Eliminarea. Se realizează într-o manieră asemănătoare cazului necompactat. De exemplu, eliminarea cheii 1011 din structura din figura 6.27 se realizează astfel. Se caută nodul care memorează această cheie. În procesul de căutare se memorează drumul de la rădăcină la acest nod într-o stivă. Stiva ne ajută la parcurgerea drumului înapoi. Nodul care memorează cheia este eliminat și se merge în nodul tată. Acest nod este eliminat și se merge în nodul-tată, deoarece a rămas cu un singur succesor, nodul 1010. Cum acest are doi succesi, operația de eliminare se termină.

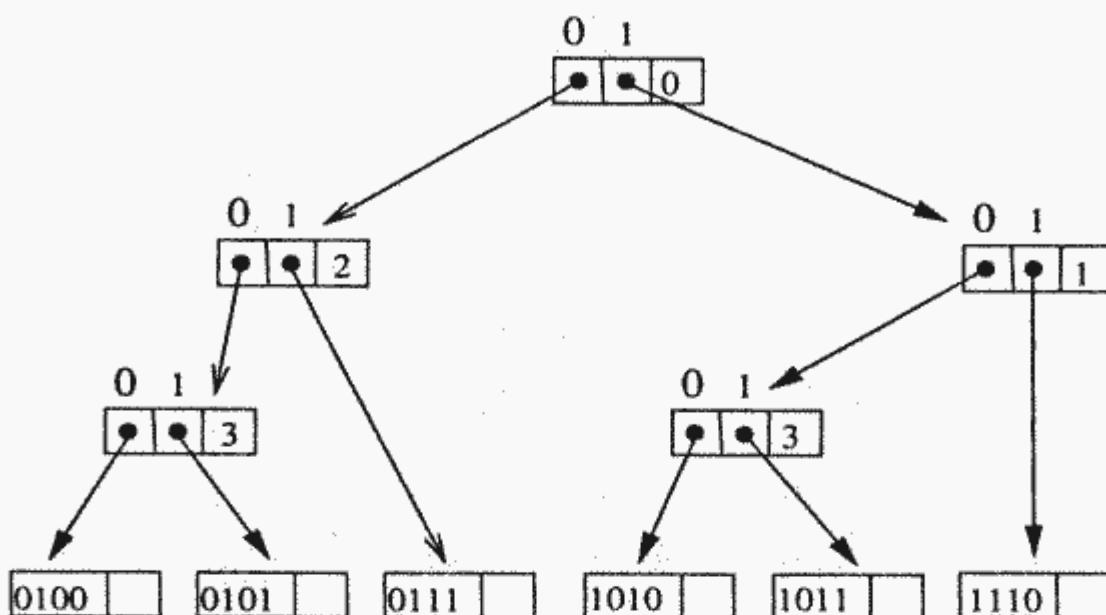


Figura 6.28: Adăugarea cheii 0111

6.3.3 Structuri Patricia

Un *arbore digital binar* este un arbore digital peste alfabetul $\{0, 1\}$. Exemplele din subsecțiunea 6.3.2 sunt arbori digitali binari. Într-un arbore digital binar, fiecare nod intern are exact doi fiți. Acești arbori au următoarea proprietate: numărul nodurilor de pe frontieră este cu 1 mai mare decât numărul nodurilor interne. Dacă mai adăugăm un nod intern, atunci putem să memorăm cheile în nodurile interne, reducând astfel numărul nodurilor. Nodul pe care îl adăugă îl punem ca rădăcină. Astfel rădăcina va avea totdeauna un singur fiu, anume pe cel stâng (stabilit convențional). Poziția corespunzătoare rădăcinii este -1 . Memorarea cheilor în nodurile interne se face astfel încât poziția nodului care va memora cheia este mai mică decât sau egală cu poziția nodului tată în arborele inițial. După eliminarea nodurilor de pe frontieră, legăturile se refac pe principiul „legătura urinează cheia”. Noua structură obținută se numește *Patricia* (*Practical Algorithm to Retrieve Information Coded in Alphanumeric*). O structură Patricia corespunzătoare arborelui 6.27 este reprezentată în figura 6.30. Unul dintre mărele avantaje ale acestei structuri este acela că avem un singur tip de nod. În continuare, sunt explicate operațiile pe acest exemplu. Această structură va fi luată ca exemplu pentru explicarea operațiilor.

Căutarea. Presupunem că se caută cheia $x = 1010$. Din rădăcina a se coboară direct în fiul stâng b . Deoarece poziția din b este 0 și $x[0] = 1$, se coboară în fiul drept f . Cum $x[1]$ este 0, următorul nod este e . Poziția din e este 3 și pentru că $x[3] = 0$, ne deplasăm din nou în b . Pentru că poziția din b este 0 care este mică decât ultima poziție testată 3, procesul de parcurgere a structurii se termină. Se compară x cu cheia memorată în b . Deoarece avem egalitate, căutarea se termină cu succes.

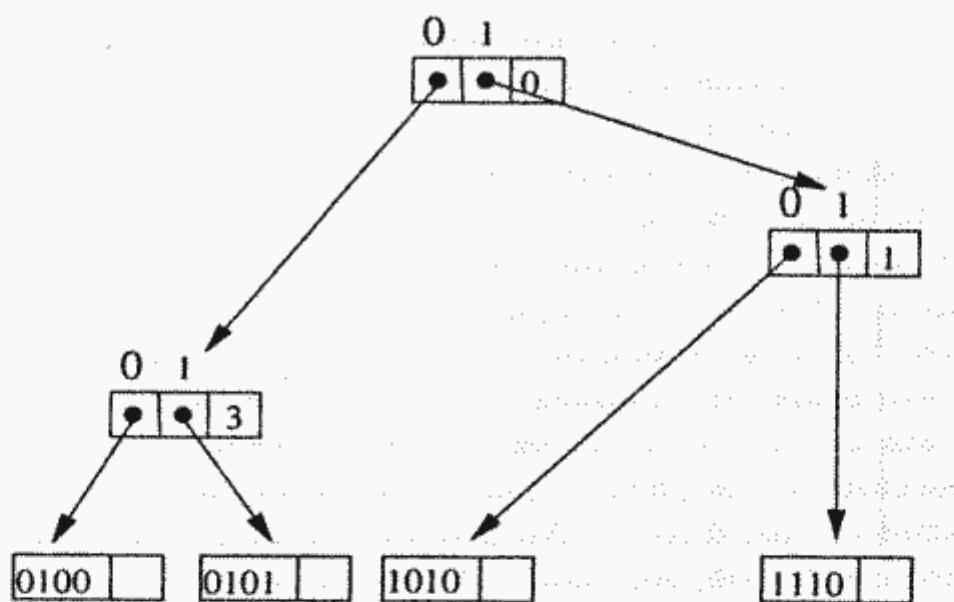


Figura 6.29: Eliminarea cheii 1011

Inserarea. Inserarea cheii $x = 1100$ presupune următorii pași. Ca de obicei, mai întâi se caută cheia în structură. Operația de căutare se termină în nodul f . Prima poziție pe care cheia din f și x diferă este $j = 2$. Se repetă căutarea pentru cheia scurtă formată cu primele $j - 1 = 1$ caractere din x . Această căutare se termină în b și ultima mutare făcută pe fiul drept. Vom insera un nou nod ca fiu drept al lui b . Noul nod va memora cheia x , va face distincție pentru poziția $j = 2$. Deoarece $x[j] = 0$, legătura pe fiul stâng va fi la el însuși. Legătura pe fiul drept va fi la b , nodul pe care s-a oprit căutarea. Rezultatul este reprezentat în figura 6.31.

Eliminarea. Eliminarea cheii 0111 presupune mai întâi căutarea sa. Aceasta se termină în nodul c . Deoarece ultima mutare a fost tot din nodul c pe fiul drept, rezultă că acest nod poate fi eliminat foarte simplu. Legătura de la nodul tată b este dusă direct la fiul stâng al lui c , d . Rezultatul este reprezentat în figura 6.32.

Teorema 6.9. Presupunem că, pornind de la structura vidă, se creează o structură Patricia prin n inserări de chei generate aleator. Atunci operația de căutare necesită $O(\log n)$ comparații în medie.

6.3.4 Exerciții

Exercițiul 6.3.1. (Extinderea tipului de dată.) Fie t un arbore digital. Notăm cu $M(t)$ mulțimea reprezentată de acest arbore. Să se scrie următoarele subprograme:

- 1) **reuniune**(t_1, t_2) care realizează operația $M(t) = M(t_1) \cup M(t_2)$.
- 2) **intersectie**(t_1, t_2) care realizează operația $M(t) = M(t_1) \cap M(t_2)$.
- 3) **minus**(t_1, t_2) care realizează operația $M(T) = M(t_1) \setminus M(t_2)$.

Exercițiul 6.3.2. Să se scrie o procedură **triInLista**(t, L) care având la intrare un arbore digital t construiește lista liniară L ce conține toate elementele din mulțimea reprezentată de t , în ordine lexicografică.

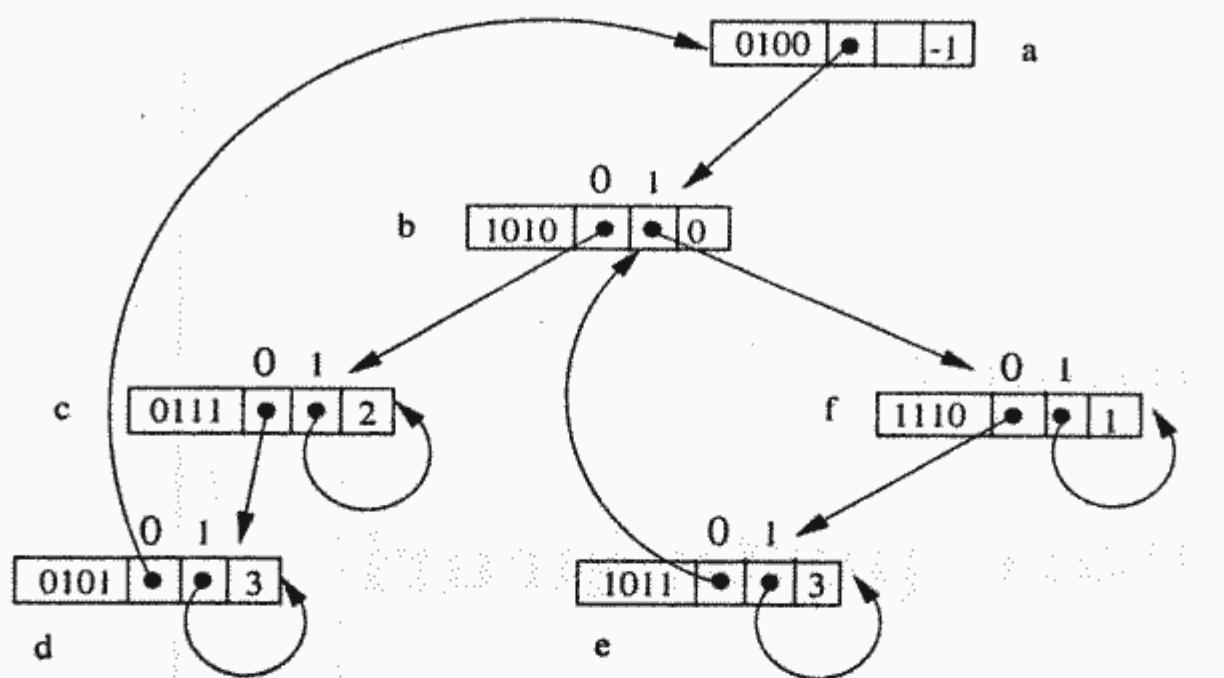


Figura 6.30: Structură Patricia

Exercițiul 6.3.3. Să se descrie în limbaj algoritmic operațiile pentru arborii digitali cu chei de lungimi diferite.

Exercițiul 6.3.4. Să se descrie în limbaj algoritmic operațiile pentru arborii digitali compactați.

Exercițiul 6.3.5. Explicați modul în care se elimină cheia 1010 din structura Patricia 6.30.

Exercițiul 6.3.6. Să se descrie în limbaj algoritmic operațiile pentru structurile Patricia.

6.4 Referințe bibliografice

Deși scrisă în 1973, [Knu76] rămâne o carte de referință în domeniu. Multe aspecte ale căutării studiate în acest capitol sunt inspirate de aici. Completări la structurile de date studiate aici, precum și alte structuri, mai pot fi găsite în [Mel84a, Sed88, HSAF93, Hei96, Wei92]. Pentru aspectele teoretice privind structurile de căutare recomandăm [Knu76, Mel84a, Mel79].

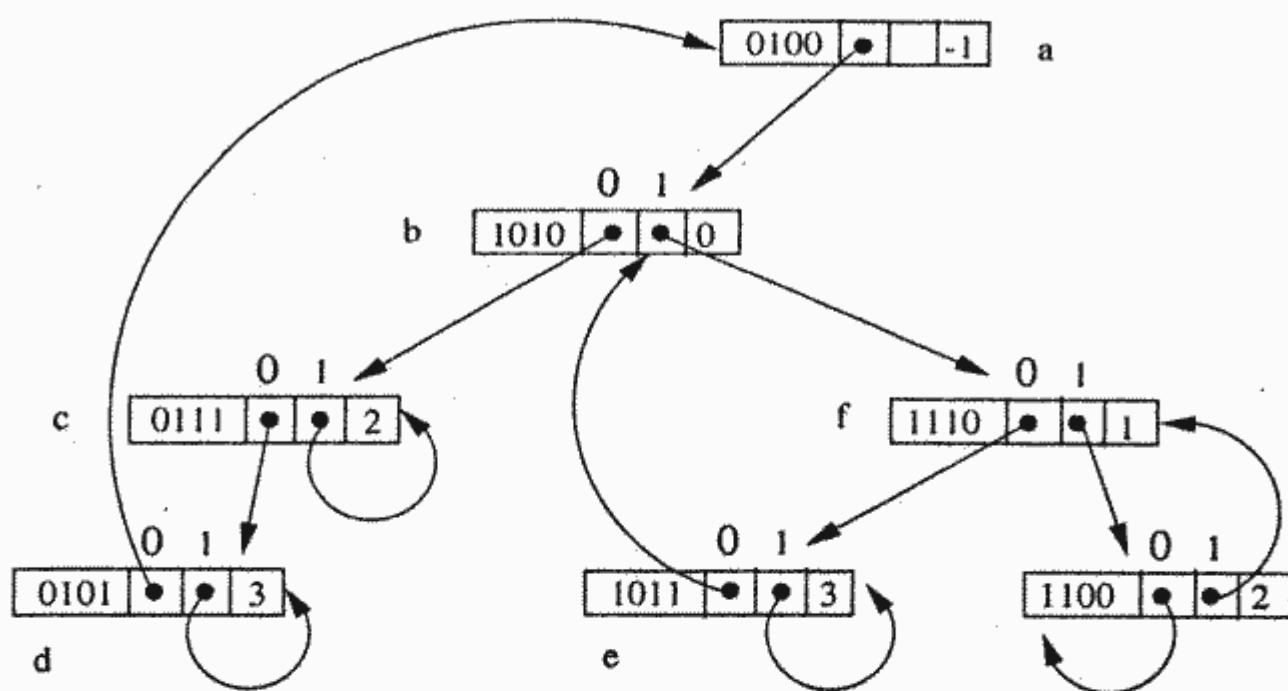


Figura 6.31: Structură Patricia: inserarea cheii 1100

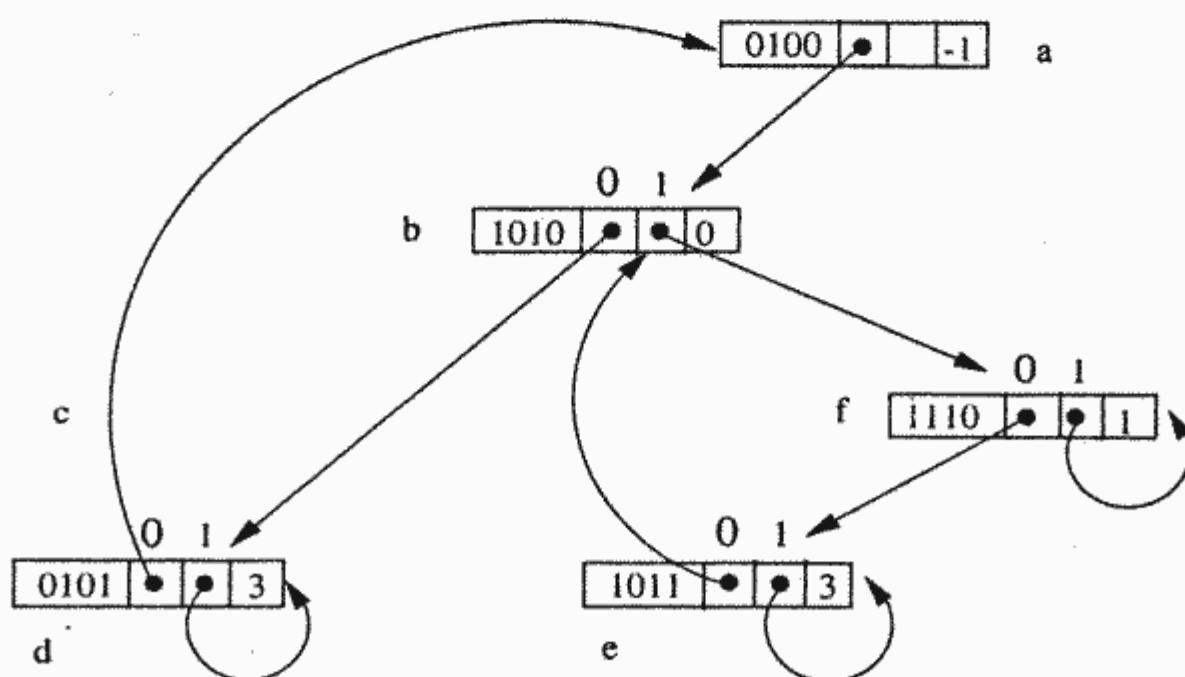


Figura 6.32: Structură Patricia: eliminarea cheii 0111



Capitolul 7

Căutare peste şiruri

Considerăm şirurile ca tipuri de date abstracte în care:

- obiectele sunt secvențe $x_0 \dots x_{n-1}$ cu componentele x_i aparținând unui tip abstract ale cărui elemente le numim *caractere*;
- setul de operații include inserarea și ștergerea de caractere, concatenarea a două şiruri și regăsirea unui şir ca subşir al altuia.

Sistemele de procesare a textelor utilizează acest tip pentru editarea textelor. Un text poate fi privit ca un şir de litere, cifre, și caractere speciale. Tipul **String** și fișierele text constituie reprezentări ale tipului abstract şir. Un caz particular îl constituie şirurile binare, construite peste alfabetul $\{0, 1\}$. Acestea sunt utilizate la memorarea imaginilor grafice, de exemplu. În acest capitol prezentăm algoritmi pentru o singură operație peste şiruri și anume regăsirea unui şir ca subşir al altuia, cunoscută și sub numele de *căutare peste şiruri*.

Fie tipul abstract **Character** în care obiectele sunt caracterele. Fără să restrângem generalitatea, presupunem că şirurile sunt reprezentate prin tablouri unidimensionale de caractere. Problema căutării peste şiruri poate fi formulată astfel: fiind date două şiruri $s = s_0 \dots s_{n-1}$ (subiectul sau textul) și $p = p_0 \dots p_{m-1}$ (patternul), să se proiecteze un algoritm eficient care să decidă dacă p este subşir al lui s . În cazul când patternul apare de mai multe ori în text, interesează poziția unei apariții (de regulă, prima). De exemplu, în textul $s = aabaccabaac$ patternul $p = bbaabbcc$ nu apare niciodată, iar patternul aba apare de două ori.

Algoritmii care rezolvă problema de căutării peste şiruri au o istorie interesantă [Sed88]. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste şiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată. Algoritmul dat de Knuth, Morris și Pratt a fost publicat abia în 1976. Între timp, R.S. Boyer și J.S. Moore (și independent W. Gosper) au descoperit un algoritm care este mult mai rapid în

multe situații. În 1980, R.M. Karp și M.O. Rabin au proiectat un algoritm cu o descriere foarte simplă și care poate fi extins la texte și patternuri bidimensionale, deci foarte util la procesarea imaginilor grafice.

7.1 Căutarea naivă

Un algoritm simplu dar, după cum vom vedea, ineficient este următorul: pentru fiecare poziție posibilă în text se testează dacă patternul se potrivește peste subtextul care începe la acea poziție.

```
function pmNaiv(s, n, p, m)
    i ← 0
    while (i < n-m) do
        i ← i+1
        j ← 0
        while (s[i+j] ≠ p[j]) do
            if (j = m-1)
                then return i /* a gasit prima aparitie a lui p*/
            else j ← j + 1
        return -1 /* p nu apare in s */
    end
```

Procedura PMNaiv va întoarce valoarea 0, dacă patternul nu este subșir al subiectului (textului), sau poziția i de la care începe prima apariție a patternului în subiect (text). Complexitatea timp a algoritmului $O(n \cdot m)$.

7.2 Algoritmul Knuth-Morris-Pratt

Ideea algoritmului este următoarea: Atunci când s-a întâlnit o nepotrivire, i.e., caracterul curent p_j din pattern este diferit de caracterul curent s_i din text, caracterele $s_{i-1}, \dots, s_{i-j+1}$ din text sunt cunoscute deoarece ele există în pattern. Poziția de start pentru următoarea comparație dintre pattern și text poate fi determinată explotând această informație. Ca urmare, întoarcerile în text sunt eliminate. Astfel că algoritmul va consta din două etape:

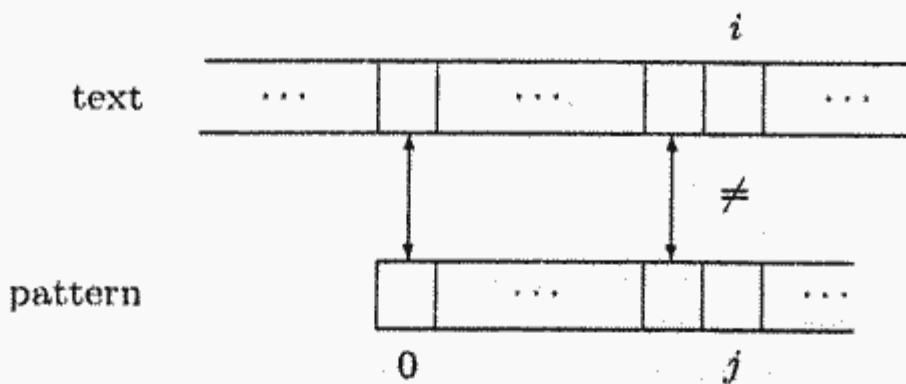
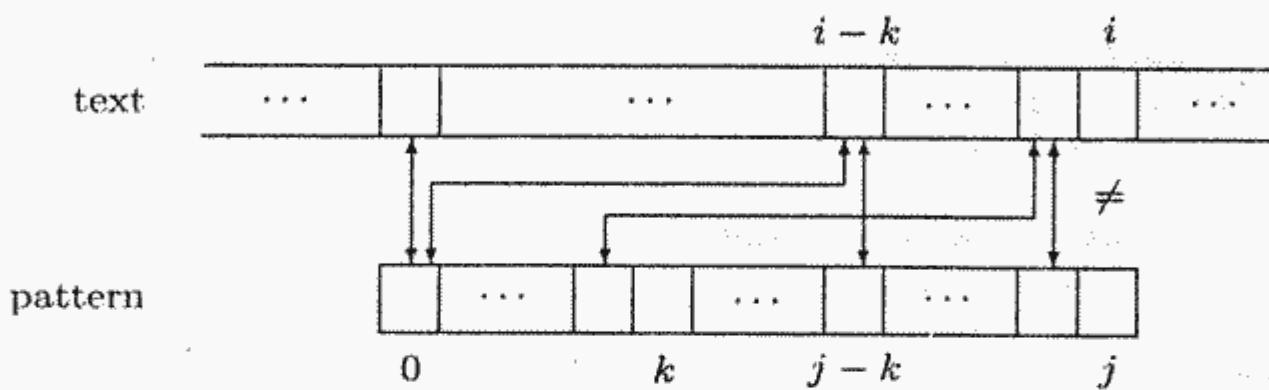
1. Se proceseaza mai întâi patternul în $O(m)$ și se creează o structură de date.
2. Folosind structura de date creată la prima etapă, se procesează textul fără a mai executa întoarceri.

Există două moduri de a preciza o apariție a unui pattern în text:

- prin poziția de start: spunem că p se potrivește în s la poziția de start i ;
- prin poziția de sfârșit: spunem că p se potrivește în s la poziția de sfârșit i .

O nepotrivire o reprezentăm grafic ca în figura 7.1. Avem $s_{i-j} = p_0, \dots, s_{i-1} = p_{j-1}, s_i \neq p_j$. Vom utiliza această informație pentru a determina cel mai lung prefix propriu al lui p care se potrivește în s la poziția de sfârșit i . Notăm acest prefix cu $F[i, j]$. Determinarea funcției $F[i, j]$ se face într-o manieră recursivă. Pentru moment presupunem $p = p_0 \cdots p_j$.

1. $F[i, 0]$ este sirul vid (de lungime zero).

Figura 7.1: Reprezentarea grafică a nepotrivirii $s_i \neq p_j$ Figura 7.2: $s_i \neq p_k \Rightarrow F[i, j] = F[i, k]$

2. Presupunem $j > 0$. Se consideră cel mai mare prefix q al lui p care se potrivește în s la poziția de sfârșit $i - 1$. Presupunem $q = p_0 \cdots p_{k-1}$ (figura 7.2) și notăm $p' = qp_k$. Dacă $p_k = s_i$ atunci $F[i, j] = p'$. Altfel, p' se află în aceeași situație ca p și deci avem $F[i, j] = F[i, k]$.

Presupunem că pentru patternul p s-a determinat $F[i, j]$ pentru orice prefix $p_0 \cdots p_j$ al lui p . În momentul apariției unei nepotriviri $p_j \neq s_i$, $F[i, j] = p_0 \cdots p_k$ este cel mai mare prefix al lui $p_0 \cdots p_j$ care se potrivește în s la poziția de sfârșit i și deci se poate continua cu comparația dintre s_{i+1} și p_{k+1} .

Convenim să notăm $F[i, j] = k$ în loc de $F[i, j] = p_0 \cdots p_k$ (orice prefix este identificat în mod unic de poziția sa de sfârșit în pattern).

Cum poate fi calculată eficient funcția $F[i, j]$? Notăm cu $f[j]$ valoarea k determinată în pasul inductiv din definiția lui $F[i, j]$. Cunoscând f , $F[i, j]$ poate fi calculată foarte simplu:

```

k ← f[j]
while (k ≠ 0) and (p[k] ≠ s[i]) do
    k ← f[k]
F[i, j] ← k
  
```

Funcția f poate fi calculată explorând numai patternul: dacă $f[j] = k$, atunci $p_0 \cdots p_{k-1}$ este de fapt cel mai lung prefix al lui p care se potrivește în p la poziția de sfârșit $j - 1$. Ca și F , funcția f poate fi definită recursiv:

- Dacă $j = 0$, atunci $f[0] = -1$ (putem presupune că $p_0 \dots p_{k-1}$ desemnează sirul vid);
- Presupunem $j > 0$. Fie $k = f[j-1]$ (care este definită). Rezultă că $p_0 \dots p_{k-1}$ este cel mai lung prefix al lui $p_0 \dots p_j$ care se potrivește în p la poziția de sfârșit $j-2$. Dacă $p_k = p_{j-1}$, atunci $f[j] = k+1$. Altfel, se caută cel mai lung prefix al lui $p_0 \dots p_k$ care se potrivește în p la poziția de sfârșit $j-2$. Adică se face $k = f[k]$ și se repetă raționamentul de mai sus.

Datorită rolului pe care-l joacă în timpul căutării, funcția f se numește *funcție eșec (failure function)*. Subprogramul Pascal care calculează f are o structură foarte simplă:

```
procedure Determina_f (p, m, f)
  f[0] ← -1
  for j ← 1 to m-1 do
    k ← f[j-1]
    while ((k ≠ -1) and (p[j-1] ≠ p[k])) do
      k ← f[k]
    f[j] ← k+1
  end
```

Notăm cu KMP (de la inițialele celor trei autori) subprogramul care realizează efectiv operația de pattern-matching:

```
function KMP(s, n, p, m, f)
  i ← 0
  j ← 0
  while (i < n) do
    while (j ≠ -1) and (p[j] ≠ s[i]) do
      j ← f[j]
    if (j = m-1)
      then return i-m+1 /* gasit p in s */
    else i ← i+1
      j ← j+1
  return -1 /* p nu apare in s */
end
```

Funcția failure pentru patternul *abaabaaabc* este reprezentată în figura 7.3. Privind mai atent această funcție se observă că algoritmul poate fi îmbunătățit. De exemplu, dacă apare o nepotrivire între un caracter s_i din text și al cincilea caracter din pattern, atunci algoritmul continuă cu comparația dintre s_i și al doilea caracter din pattern. Dar noi știm că între acestea există nepotrivire pentru că al doilea și al cincilea caracter din pattern sunt identice.

Teorema 7.1. *Algoritmul KMP execută totdeauna cel mult $n + m$ comparații de caractere.*

Deși algoritmul KMP execută mai puține comparații decât cel naiv, în practică KMP nu se dovedește a fi mai rapid în mod semnificativ pentru că sunt foarte rare situațiile când patternul are o structură repetitivă.

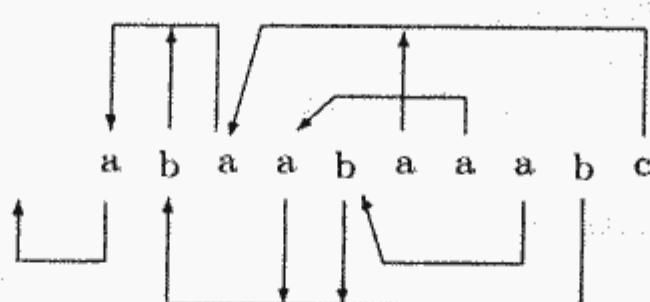


Figura 7.3: Funcția eșec pentru algoritmul KMP

7.3 Algoritmul Boyer-Moore

V I S U L U N E I N O P T I D E I A R N Ă
 I A R
 I A R
 I A R
 I A R
 I A R
 I A R
 I A R

Figura 7.4: Algoritmul Boyer-Moore

Caracterele patternului sunt explorate de la dreapta la stânga. Un exemplu ce arată modul de lucru al algoritmului este reprezentat în figura 7.4. Prima dată se compară R (ultimul caracter din pattern) cu S (al treilea caracter din text). Deoarece S nu apare în pattern, se deplasează patternul cu trei poziții (lungimea sa) la dreapta. Apoi se compară R cu caracterul spațiu. Nici caracterul spațiu nu apare în pattern aşa că acesta se deplasează din nou la dreapta cu trei poziții. Procesul continuă până când R este comparat cu I (al 21-lea caracter din text). Deoarece I apare în pattern pe prima poziție se deplasează patternul la dreapta cu două poziții. Apoi se compară R cu R, deci există potrivire. Se continuă comparația cu penultimul caracter din pattern (de fapt, al doilea) și precedentul din text (al 22-lea). Se obține din nou potrivire și se compară următoarele două caractere de la stânga (primul din pattern și al 21-lea din text). Deoarece există potrivire și patternul a fost parcurs complet, rezultă că s-a determinat prima apariție a patternului în text.

Ca și în cazul algoritmului KMP, există o etapă de preprocesare a patternului. Această preprocesare construiește o funcție care, pentru fiecare caracter din alfabet, precizează câte poziții este deplasat patternul la dreapta atunci când apare o nepotrivire. Notăm cu `detSalt(p, salt)` procedura care calculează funcția `salt`. Pentru patternul IAR avem $salt['R'] = 0$, $salt['A'] = 1$ și $salt['I'] = 2$. În ipoteza căutării repetitive pentru același pattern, este preferabil ca funcția `salt` să fie calculată o singură dată și apoi transmisă ca parametru algoritmului de căutare.

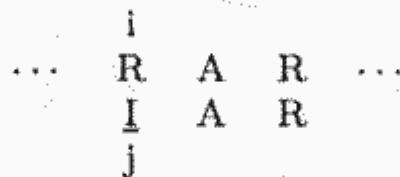
Descrierea algoritmului este foarte simplă:

```

function BM(s, n, p, m, salt)
    i ← m-1
    j ← m-1
    repeat
        if (s[i] = p[j])
            then i ← i-1
            j ← j-1
        else if (m-j > salt[s[i]])
            then i ← i+m-j
        else i ← i+salt[s[i]]
        j ← m-1
    until ((j<0) or (i>n-1))
    if (j < 0)
        then return i+1
    else return -1
end.

```

În cazul unei nepotriviri de forma



valoarea lui i este incrementată cu $3 - j = 3 > 0 = \text{salt}[R]$.

Teorema 7.2 ([Sed88]) *Algoritmul BM execută totdeauna cel mult $m+n$ comparații de caractere și aproximativ $\frac{n}{m}$ salturi când alfabetul nu este mic și patternul nu este prea lung.*

Observație. Dacă alfabetul are numai două caractere (cazul şirurilor binare), atunci performanțele algoritmului BM nu sunt cu mult mai bune decât cele ale căutării naive. În acest caz se recomandă împărțirea şirurilor în grupe cu un număr fixat de biți. Fiecare grupă reprezintă un caracter. Dacă dimensiunea unei grupe este k , atunci vor exista 2^k caractere, i.e. dintr-un alfabet mic obținem unul cu multe caractere. Totuși, k va trebui ales suficient de mic pentru ca dimensiunea tabelei de salturi să nu fie prea mare.

sfobs

7.4 Algoritmul Rabin-Karp

Acest algoritm utilizează tehnica tabelelor de dispersie (hash). Un simbol este o secvență de m caractere. Să ne imaginăm că toate simbolurile posibile sunt memorate într-o tabelă de dispersie foarte mare, astfel încât nu există coliziune. A testa dacă patternul p coincide cu un subşir de lungime m din text este echivalent cu a testa dacă funcția de dispersie h dă aceeași valoare pentru ambele simboluri. Există avantajul că pentru pattern funcția de dispersie este calculată o singură dată. Pentru ca algoritmul să fie eficient, funcția de dispersie trebuie definită în aşa fel

încât, atunci când se face o deplasare la dreapta în text, calculul valorii funcției pentru următorul simbol să fie cât mai simplu. Timpul necesar efectuării acestui calcul trebuie să fie cu mult mai mic decât cel necesar comparării a două siruri de lungime m .

Un mod de a defini funcția de dispersie este următorul: Se consideră fiecare sir de m caractere ca fiind reprezentarea unui număr întreg în baza d , unde d este numărul maxim de caractere. Numărul zecimal corespunzător sirului $s[i..i+m-1]$ este:

$$x = s[i]d^{m-1} + s[i+1]d^{m-2} + \dots + s[i+m-1]$$

Funcția de dispersie h va fi definită prin $h(s[i..i+m-1]) = x \bmod q$, unde q este un număr prim foarte mare. O deplasare la dreapta în text corespunde înlocuirii lui x cu:

$$(x - s[i]d^{m-1})d + s[i+m]$$

În formulele de mai sus s-a presupus caracterele $0, \dots, d-1$. Pentru a interpreta caracterele textului drept cifre, considerăm o funcție *index* care asociază fiecărui caracter numărul său de ordine în alfabet (similar funcției *Ord* din Pascal). Toate acestea conduc la următorul algoritm foarte simplu:

```

function RK(s, n, p, m)
    dlam1 ← 1 /* calculeaza d la puterea m-1 */
    for i ← 1 to m-1 do
        dlam1 ← (d*dlam1) mod q
    hp ← 0 /* h(p) */
    for i ← 0 to m-1 do
        hp ← (hp*d+index(p[i])) mod q
    hs ← 0 /* h(s[1..m] */
    for i ← 0 to m-1 do
        hs ← (hs*d+index(s[i])) mod q
    i ← 0
    while ((hp ≠ hs) and (i <= n-m)) do
        if (hp = hs)
            then return i
        hs ← (hs+d*q-index(s[i])*dlam1) mod q
        hs ← (hs*d+index(s[i+m])) mod q
        i ← i+1
    return -1
end

```

Adunarea cu $d * q$ la recalcularul lui hs se face pentru a fi siguri că se obține un număr pozitiv. Alegerea $d = 32$ s-a făcut pentru ca operația de înmulțire să se efectueze foarte simplu (intregii din Longint sunt reprezentați pe 32 de biți). Merită notat faptul că din cauza eventualelor coliziuni, testul final mai trebuie dublat de verificarea $p = s[i..i+m-1]$.

Deși teoretic, în cazul cel mai nefavorabil, algoritmul RK are o complexitate timp proporțională cu $n \cdot m$, în practică s-a dovedit a executa aproximativ $n + m$ pași.

7.5 Expresii regulate

În această secțiune considerăm cazul când patternul constituie doar o specificație a ceea ce se cauță în sensul că el desemnează o mulțime de şiruri pentru care se cauță. Numim o astfel de specificație *pattern generalizat*. Un exemplu de pattern generalizat îl constituie utilizarea caracterului *wildcard* (a se vedea și exercițiul 7.2). Un *wildcard* este un caracter special, să zicem „?”, care se potrivește peste orice alt caracter. De exemplu, patternul generalizat „ab?a” specifică de fapt mulțimea {abaa, abba, abca, abda, ...}. Orice subşir al textului care aparține acestei mulțimi se va potrivi peste acest pattern. Un alt mod de a specifica patternuri generalizate îl constituie expresiile regulate.

Definiția 7.1. *Mulțimea expresiilor regulate peste alfabetul A este definită recursiv astfel:*

- orice caracter din A este o expresie regulată;
- dacă e_1, e_2 sunt expresii regulate, atunci (e_1e_2) , $(e_1 + e_2)$ și e_1^* sunt expresii regulate.

Mulțimea de şiruri (limbajul) $L(e)$ definit de o expresie regulată e este definit recursiv astfel:

- dacă e este un caracter, atunci $L(e) = \{e\}$;
- dacă $e = (e_1e_2)$, atunci $L(e) = L(e_1)L(e_2) = \{w_1w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- dacă $e = (e_1 + e_2)$, atunci $L(e) = L(e_1) \cup L(e_2)$;
- dacă $e = e_1^*$, atunci $L(e) = \cup_k L(e_1^k)$, unde $L(e_1^0) = \{\varepsilon\}$, $L(e_1^{k+1}) = L(e_1^k)L(e_1)$ (ε este şirul vid (de lungime zero)),

Exemplu. Fie alfabetul $A = \{a, b, c\}$. Avem $L(a(b + a)c) = \{abc, aac\}$ și $L((ab)^*) = \{\varepsilon, ab, abab, ababab, \dots\} = \{(ab)^k \mid k \geq 0\}$.

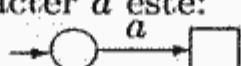
sfex

Deoarece $L(e_1(e_2e_3)) = L((e_1e_2)e_3)$, vom scrie $e_1e_2e_3$ pentru a desemna una din cele două expresii. Vom aplica astfel de convenții de omisire a parantezelor ori de câte ori este posibil.

În continuare ne vom ocupa de modul în care expresiile regulate pot fi utilizate în procesul de căutare într-un text. Mai întâi plecăm de la observația că patternul și funcția eșec de la algoritm KMP pot fi gândite ca reprezentând un automat. În figura 7.5 este reprezentat automatul corespunzător patternului și funcției eșec din figura 7.3. Acest automat este reprezentat ca un digraf în care vîrfurile reprezintă stările automatului iar arcele relația de tranziție. Tranzițiile descriu comportarea automatului. De exemplu, dacă automatul din figura 7.5 se află în starea 3 și caracterul curent din text este 'b', atunci automatul va trece în starea 4. Altfel va trece în starea 1. Dacă automatul ajunge în starea finală (desenată prin patrat), atunci s-a determinat o apariție a patternului în text.

Noțiunea de automat o extindem la expresii regulate după cum urmează:

1. Automatul definit de un caracter a este:



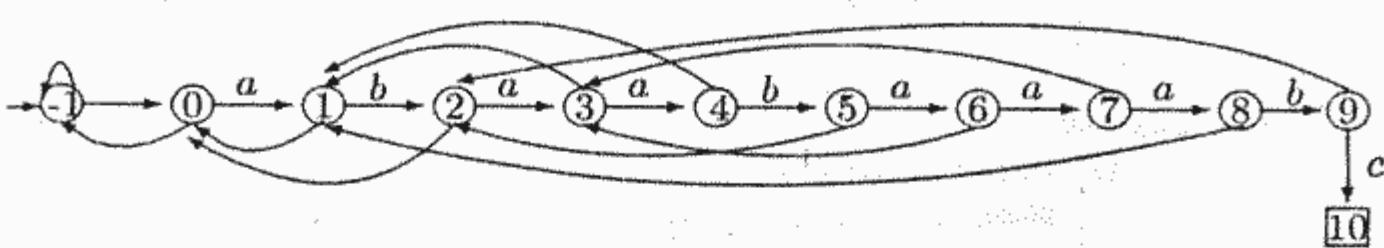


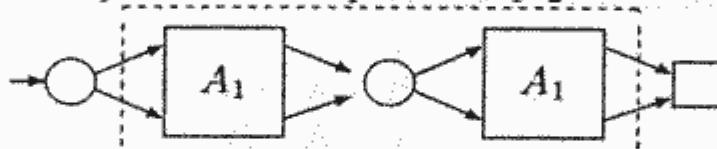
Figura 7.5: pattern și funcția eșec ca automat

2. Fie e_1 și e_2 două expresii regulate ale căror automate A_1 și A_2 sunt reprezentate ca mai jos:

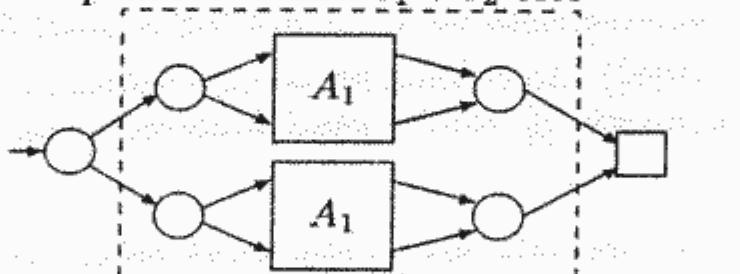


Atunci:

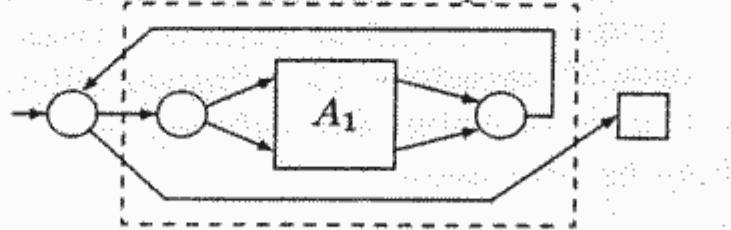
- a) automatul corespunzător compunerii e_1e_2 este



- b) automatul corespunzător sumei $e_1 + e_2$ este



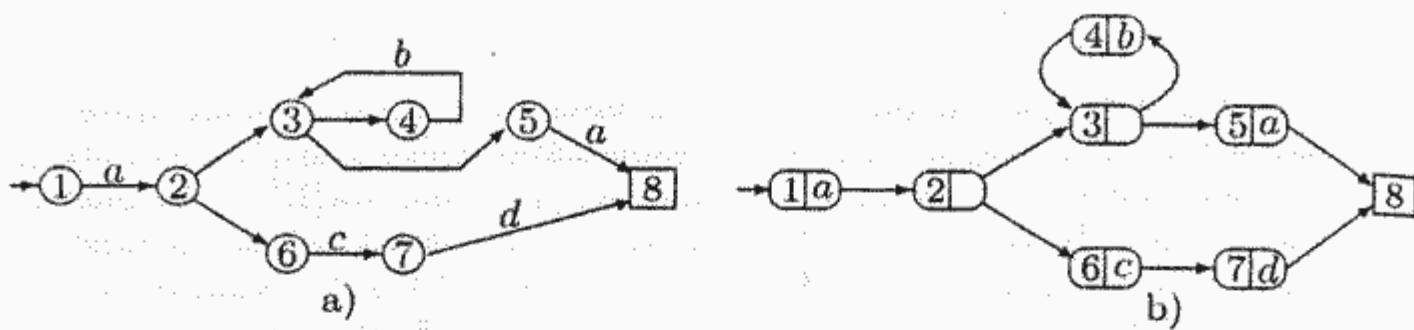
- c) automatul corespunzător închiderii e_1^* este



Exemplu. Automatul corespunzător expresiei $e = a(b^*a + cd)$ este reprezentat în figura 7.6.a (câteva arce și stări redondante au fost eliminate).

sfex

Întâlnim două tipuri de stări (vârfuri în digraf) în definiția unui automat asociat unei expresii: stări din care ies un singur arc etichetat cu un caracter și stări din care ies două arce neetichetate. Această proprietate este evidențiată mai bine prin reprezentarea automatelor ca în figura 7.6.b. Notăm totuși faptul că această proprietate rezultă din modul particular în care am construit automatul. În general, automatul asociat unei expresii nu este unic determinat, dar toate automatele corespunzătoare aceleiași expresii sunt izomorfe (din punctul de vedere al

Figura 7.6: Automatul corespunzător expresiei $a(b^*a + cd)$

comportării). Reprezentarea particulară de aici permite reprezentarea automatelor printr-un tablou unidimensional de structuri, unde o componentă a tabloului are trei câmpuri: unul pentru memorarea caracterului, iar celelalte două pentru memorarea stărilor următoare. Facem convenția ca -1 să reprezinte starea invalidă și 0 starea inițială. De exemplu, $A.stare[s].st_urm2 = -1$ poate semnifica faptul că din starea s pleacă un singur arc.

Exercițiul 7.5.1. Să se scrie un program care construiește automatul asociat unei expresii regulate.

Algoritmul de căutare a unui sir desemnat de o expresie regulată are următoarea descriere sumară. Utilizăm o coadă cu restricții la ieșire D (a se vedea și exercițiul 2.5.2), unde inserările se pot face și la început și la sfârșit, iar ștegerile/citirile numai la început. Fie q starea curentă a automatului, j poziția curentă în textul s , i poziția în textul s de început a patternului curent. Simbolul $\#$ va juca rolul de delimitator (el poate fi înlocuit cu starea invalidă -1). Inițial avem $D = (\#)$, $q = 1$ (prima stare după starea inițială 0), $i = j = 1$. La pasul curent se execută următoarele operații:

1. Dacă:

- din q pleacă două arce neetichetate, atunci inserează la început în D destinațiile celor două arce;
- din q pleacă un singur arc etichetat cu $s[j]$, atunci inserează la sfârșitul lui D destinația arcului;
- q este delimitatorul $\#$, atunci:
 - dacă $D = \emptyset$, atunci incrementează i , j devine noul i , inserează $\#$ în D și atribuie 1 lui q (aceasta corespunde situației când au fost epuizate toate posibilitățile de a găsi în text o apariție a unui sir specificat de pattern care începe la poziția i);
 - dacă $D \neq \emptyset$, atunci incrementează j și inserează $\#$ la sfârșitul lui D ;
- dacă q este starea finală, atunci s-a găsit o apariție a unui sir specificat de pattern care începe la poziția i .

2. Extrage starea de la începutul lui D și o memorează în D , după care reia pasul curent.

Exercițiul 7.5.2. Să se exemplifice execuția algoritmului de mai sus pentru automatul din figura 7.6 și textul $abcacde$.

Exercițiul 7.5.3. Să se scrie un program care implementează algoritmul de mai sus.

Exercițiul 7.5.4. Fie e o expresie regulată, A automatul corespunzător lui e și s un subiect.

1. Să se scrie un subprogram recursiv $\text{PozSf}(q, i)$: Integer care, pentru o stare q în A și o poziție i în subiect date, întoarce -1 , dacă nu există drum de la q la starea finală descris de un subșir al lui s ce începe la poziția i , și poziția de sfârșit al subșirului descris de un astfel de drum, atunci când există.
2. Să se scrie un program care utilizează PozSf pentru a determina o apariție în s a unui sir desemnat de e .
3. Să se modifice $\text{PozSf}(q, i)$, astfel încât să determine toate drumurile de la q la starea finală descrise de subșiruri ale lui s care încep la poziția i .

7.6 Mai multe patternuri și înlocuire

Considerăm două modificări ale problemei studiate în secțiunile anterioare: în loc de un singur pattern presupunem că se caută simultan pentru mai multe patternuri, iar apariția fiecărui pattern este înlocuită cu un alt sir. Această nouă versiune este tratată în [AS84]. Notăm cu $K = \{(x_0, y_0), (x_1, y_1), \dots, (x_{k-1}, y_{k-1})\}$ mulțimea perechilor (pattern, sir înlocuitor). Covenim că sirurile x_i și y_i să le numim cuvinte-cheie. Presupunem $(x_i \neq \epsilon)$, pentru orice i cu $0 \leq i < k$. Căutarea simultană pentru mai multe chei și înlocuirea acestora presupune rezolvarea problemei suprapunerii cheilor: când o cheie se suprapune peste altă cheie. În acest caz se va înlocui direct cheia mai lungă (evident, cele două chei au aceeași poziție de start).

Algoritmul prezentat este o generalizare a algoritmului KMP și presupune parcurserea a trei etape. În prima etapă se memorează cheile într-o structură arborescentă asemănătoare arborilor digitali. O astfel de structură este reprezentată în figura 7.7 pentru

$$K = \{(ABCDE, \alpha), (CDE, \beta), (BC, \gamma)\}.$$

Vârfurile arborelui le numim stări (de la terminologia automatelor) și le identificăm prin numere întregi nenegative. Mai considerăm în plus starea -1 care joacă rolul stării nedefinite. Rădăcina este starea 0 totdeauna. Arborele îl reprezentăm printr-o funcție $g[s, c]$ cu semnificația $g[s, c] = s'$ dacă și numai dacă există un arc de la starea s la starea s' etichetat cu caracterul c (în terminologia automatelor, acțiunea notată de c transformă starea s în starea s'). Altfel $g[s, c] = \text{Nedefinit}$. Această reprezentare este foarte simplă, dar are dezavantajul că multe intrări într-un tabel au valoarea Nedefinit . Structura arborescentă o completăm cu o funcție parțial definită $\text{out}[s]$ care, pentru o stare corespunzătoare terminării unei chei, înregistrează sirul înlocuitor. Pentru exemplul din figura 7.7 avem $\text{out}[5] = \alpha$, $\text{out}[8] = \beta$, $\text{out}[10] = \gamma$. Definiția funcției out urmează a fi completată în etapa a doua. Subprogramul care construiește arborele utilizează o procedură de inserare a unei perechi de chei, similară celei de la arbori digitali:

```

procedure adaugaCheie(x, y)
    m ← lung(x)
    stare ← 0
    j ← 1
    while (g[stare, x[j]] ≠ nedefinit) do
        stare ← g[stare, x[j]]
        j ← j+1
    for i ← j to m do
        starenoua ← starenoua+1
        for c ← primulCaracter to ultimulCaracter do
            g[starenoua, c] ← nedefinit
            out[starenoua] ← NULL
            g[stare, x[i]] ← starenoua
            stare ← starenoua
        out[stare] ← y
    end
procedure constrGraf(chei, k, g, out)
begin
    starenoua ← 0
    for c ← primulCaracter to ultimulCaracter do
        g[0, c] ← nedefinit
    for i ← 1 to k do
        adaugaCheie(chei[i, 1], chei[i, 2])
    for c ← primulCaracter to ultimulCaracter do
        if (g[0, c] = nedefinit) then g[0, c] ← 0
    end
    S-a presupus chei[i, 1] =  $x_i$  și Chei[i, 2] =  $y_i$ .

```

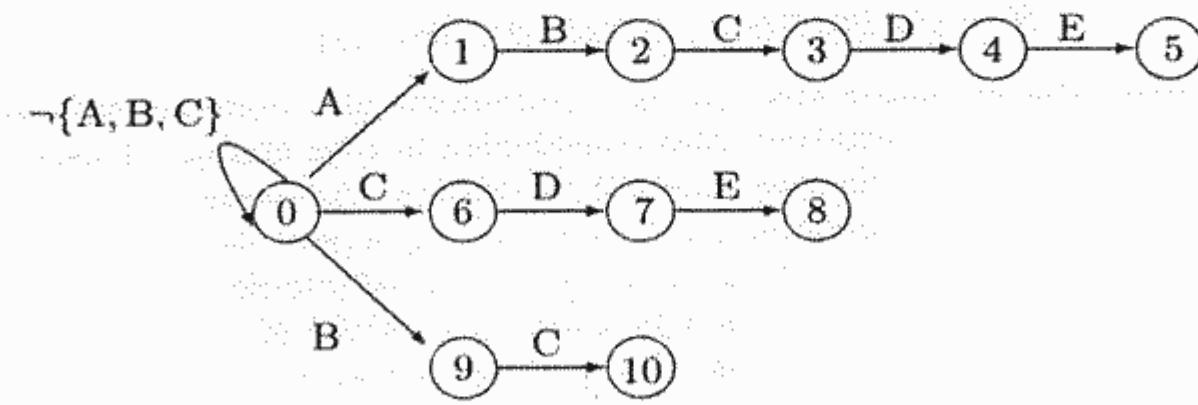


Figura 7.7: Arborele cheilor.

În etapa a doua se construiește funcția eșec $f[s]$ cu semnificația că dacă apare nepotrivire în starea s , atunci următoarea stare va fi $f[s]$. Algoritmul care construiește funcția eșec este similar celui de la KMP, numai că acum ordinea de determinare a valorilor funcției este cea dată de explorarea BFS a arborelui. Funcția eșec pentru exemplul de mai sus este reprezentată în figura 7.8. Tot în această etapă este

completată și definiția funcției *out* care va înregistra, pentru fiecare stare, sirul ce urmează a fi scris în textul rezultat, corespunzător acelei stări. Pentru stările $s = g[0, c]$ punem $out[s] = 'c'$. Pentru exemplul nostru avem $out[1] = 'A'$, $out[6] = 'C'$ și $out[9] = 'B'$. În continuare, funcția *out* se calculează odată cu funcția *esec*. Avem $out[2] = out[3] = 'A'$, $out[7] = 'CD'$, $out[4] = 'A'\gamma'D'$. Valorile $out[5] = \alpha$, $out[8] = \beta$ și $out[10] = \gamma$ au fost calculate la prima etapă. Programul care realizează etapa a doua este următorul:

```

procedure constrFctEsec(g, out, f)
    q ← coadaVida()
    for c ← primulCaracter to ultimulCaracter do
        stare1 ← g[0, c]
        if ((stare1 ≠ 0) and (stare1 ≠ nedefinit))
            then insereaza(q, stare1)
            f[stare1] ← 0
            if (out[stare1] = NULL)
                then new(out[stare1])
                    out[stare1] ← c
    while (not esteCoadaVida(q)) do
        stare3 ← citeste(q)
        elimina(q)
        for c ← primulCaracter to ultimulCaracter do
            stare1 ← g[stare3, c]
            if ((stare1 ≠ Nedefinit) and (stare1 ≠ 0))
                then insereaza(q, stare1)
                    if out[stare1] ≠ NULL
                        then f[stare1] ← 0
                        else stare2 ← f[stare3]
                        out[stare1] ← out[stare3]
                        while (g[stare2, c] = nedefinit) do
                            stare2 ← f[stare2]
                            concatenate(out[stare1], out[stare2])
                        f[stare1] ← g[stare2, c]
                        if (f[stare1] = 0)
                            then concatenate(out[stare1], c)
    end

```

A treia etapă constă în procesarea textului, găsirea și înlocuirea cheilor.

```

procedure MPMR(g, out, f, s, n)
    stare ← 0
    for i ← 1 to n do
        while (g[stare, s[i]] = nedefinit) do
            scrie(out[stare])
            stare ← f[stare]
        stare ← g[stare, s[i]]
        if (stare = 0) then scrie(s[i])
        while (stare ≠ 0) do

```

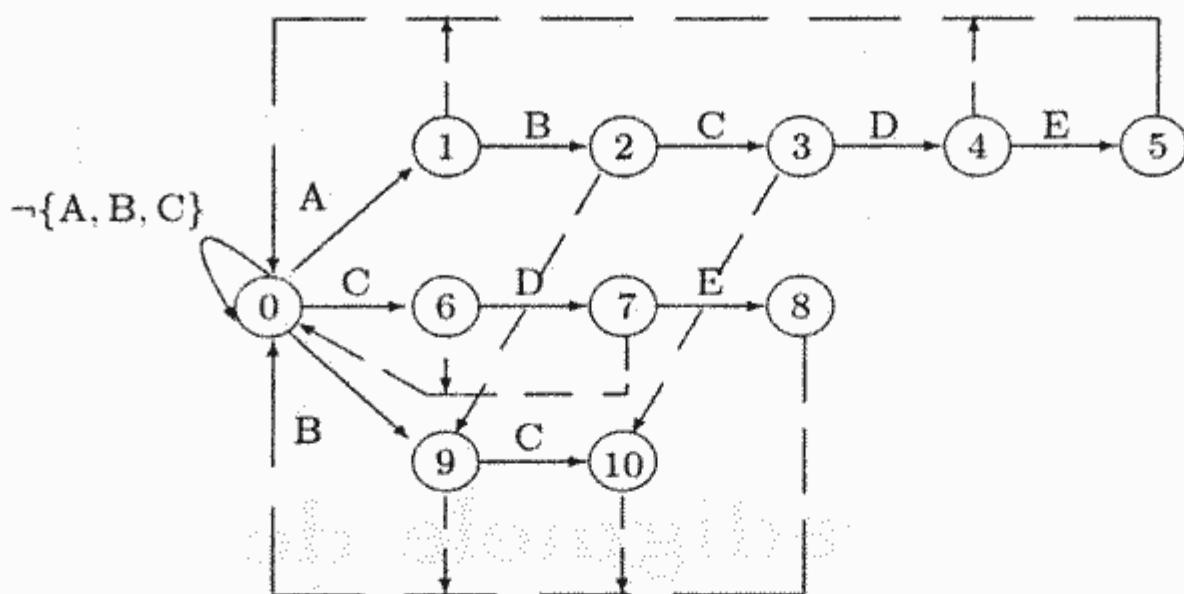


Figura 7.8: Funcția eșec

```

    scrie(out[stare])
    stare ← f[stare]
end

```

Exercițiul 7.6.1. Să se aplique algoritmul MPMR pentru textul 'BABCDEXABCAB' și mulțimea de chei din exemplu.

Algoritmii de preprocessare necesită un timp proporțional cu suma lungimilor cheilor, iar algoritmul de căutare și înlocuire realizează cel mult $2n$ tranziții de stări [AS84].

7.7 Exerciții

Exercițiul 7.1. Să se implementeze un algoritm de căutare naivă care să cerceteze patternul de la dreapta la stânga.

Exercițiul 7.2. Să se modifice algoritmii de pattern matching pentru cazul în care patternul conține un caracter *wildcard* (orice caracter din text se potrivește peste el).

Exercițiul 7.3. Să se implementeze o versiune a algoritmului Rabin-Karp pentru patternuri în texte bidimensionale. Se presupune că atât patternul, cât și textul sunt dreptunghiuri de caractere.

Exercițiul 7.4. Să se proiecteze un algoritm liniar care determină dacă un text T este o rotație ciclică a altui text T' . De exemplu, tara și rata sunt rotații ciclice unul altuia.

Exercițiul 7.5. ([CLR93]) Cu w^i notăm concatenarea şirului w cu el însuși de i ori. De exemplu, $(abc)^4 = abcabcabcabc$. Numim *factor de repetiție* al şirului w un număr r cu $w = u^r$, unde u este un şir și $r > 0$. Notăm cu $\rho(w)$ cel mai mare factor de repetiție al şirului w . Să se proiecteze un algoritm eficient care, pentru un şir $w[1..n]$ dat, determină $\rho(w[1..i])$ pentru $i = 1, 2, \dots, n$.

Exercițiul 7.6. Să se modifice algoritmii de pattern-matching-simplu, astfel încât să fie determinate toate aparițiile patternului în text și să fie înlocuite cu un alt sir dat.

7.8 Referințe bibliografice

Acest capitol, exceptând secțiunea dedicată patternurilor multiple, urmează linia din [Sed88]. Patternurile multiple și înlocuirea sunt tratate în [AS84]. Mai pot fi consultate [Baa78, CLR93, AHU74].



Capitolul 8

Despre paradigmile de proiectare

8.1 Aspecte generale

Descrierea unui algoritm care rezolvă o problemă presupune ca etapă intermediară construcția modelului matematic corespunzător problemei (a se vedea figura 8.1). Necesitatea construcției modelului matematic este impusă de următoarele motive:

- *eliminarea ambiguităților și inconsistențelor.* De multe ori, problema este descrisă informal (verbal). De aici, anumite aspecte ale problemei pot fi omise sau formulate ambiguu. Construcția modelului matematic evidențiază toate aceste lipsuri și, în acest fel, conduce la eliminarea lor;
- *utilizarea instrumentelor matematice de investigare.* Parcurgerea drumului de la formularea problemei la găsirea soluției nu este întotdeauna simplă sau predictabilă. Instrumentele matematice de investigare oferă un cadru sistematic și sigur pentru determinarea structurii analitice a soluției și apoi a modului de obținere a acesteia;
- *diminuarea efortului la scrierea programului.* Descrierea soluției în termenii modelului matematic ușurează foarte mult muncă de definire și apoi de descriere a algoritmului (programul).

O paradigmă de proiectare a algoritmilor se bazează pe un anumit tip de model matematic și pune la dispoziție procedee prin care se poate construi și implementa (descrie ca program) un model particular corespunzător unei probleme. Descrierea unui model matematic indexmodel matematic cuprinde următoarele trei aspecte [BD62]:

1. *conceptual:* presupune identificarea și definirea conceptelor care descriu componente din domeniul problemei;
2. *analitic:* presupune găsirea tuturor relațiilor între concepte care conduc la găsirea și descrierea soluției;
3. *computațional:* presupune evaluarea calitativă a algoritmului ce construiește soluția.

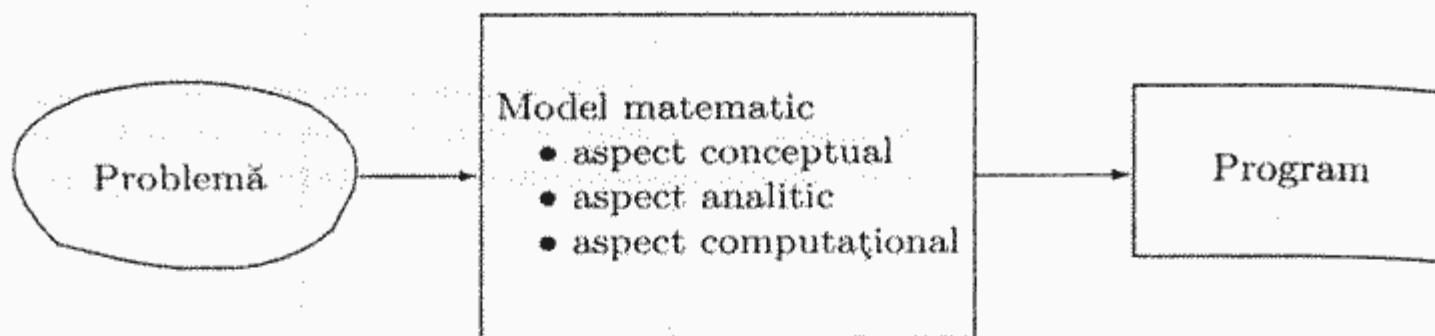


Figura 8.1: Drumul de la problemă la program

Cele trei aspecte se reflectă în etapele ce trebuie parcuse în rezolvarea unei probleme și pe care le-am discutat în capitolul de introducere.

Următoarele capitole prezintă cele mai cunoscute paradigmă de proiectare: *greedy*, *divide-et-impera*, programare dinamică, *backtracking* și *branch-and-bound*. Pentru fiecare paradigmă sunt prezentate modelul matematic și un set de studii de caz. În acest capitol prezentăm, ca exemplu, paradigma eliminării pentru a evidenția aspectele enumerate mai sus.

8.2 Un exemplu simplu de paradigmă: eliminarea

8.2.1 Primul studiu de caz: problema celebrității

8.2.1.1 Descrierea problemei

Se consideră o colectivitate formată din n persoane. O celebritate este o persoană care este cunoscută de oricare altă persoană și nu cunoaște pe nimeni. Problema constă în a scrie un program care să determine dacă există o celebritate în cadrul colectivității prin întrebări de forma: „Persoana i cunoaște persoana j ?”. Programul va pune un număr cât mai mic de întrebări.

8.2.1.2 Modelul matematic

Aspectul conceptual. Fără să restrângem generalitatea, presupunem că persoanele colectivității sunt identificate prin numerele $0, 1, 2, \dots, n-1$. O întrebare de forma „Persoana i cunoaște persoana j ?“ va fi reprezentată de următoarea funcție:

$$\text{cunoaste} : \{0, 1, 2, \dots, n-1\} \times \{0, 1, 2, \dots, n-1\} \rightarrow \{\text{true}, \text{false}\}.$$

Aspectul analitic. Notăm cu CAND mulțimea persoanelor care pot candida la calitatea de celebritate. Inițial, orice persoană poate candida la această calitate. După primirea răspunsului la o întrebare există următoarele două posibilități:

- Răspunsul este adevărat, i.e., persoana i cunoaște pe j . Atunci i nu poate fi celebritate și poate fi eliminat din CAND.
- Răspunsul este fals. Atunci j nu poate fi celebritate și deci poate fi eliminat din CAND.

În ambele cazuri se observă că este eliminat un posibil candidat. Procesul de eliminare poate fi repetat până când rămâne un singur candidat.

Aspectul computațional. Obținerea unui singur candidat se poate realiza prin $n - 1$ întrebări. Notăm că nu putem trage concluzia încă dacă singurul candidat rămas este celebritate sau nu. Mai trebuie puse întrebări suplimentare pentru a afla acest lucru. Această verificare se poate face prin cel mult $2 \cdot (n - 1)$ întrebări. Rezultă că determinarea celebrității se poate face punând cel mult $3 \cdot (n - 1)$ întrebări.

8.2.1.3 Implementare

Deoarece la sfârșit CAND va avea un singur element, ea poate fi gestionată de o variabilă de tip întreg cand. Se pleacă de la ipoteza că persoana candidat este 0 și apoi se pun întrebări între posibilul candidat și persoana următoare pentru a afla noul candidat. Pentru ultimul candidat se verifică dacă este celebritate prin întrebări duble între el și fiecare dintre celelalte $n - 1$ persoane.

Funcția **Celebritate** întoarce valoarea -1 , dacă nu există celebritate; altfel întoarce numărul corespunzător celebrității.

```
function celebritate(n)
    cand ← 0
    for j ← 1 to n-1 do
        if (cunoaste(i, j))
            then cand ← j
    i ← 0
    while (cand >= 0) and (i < n) do
        if (not cunoaste(i, cand) or cunoaste(cand, i))
            then cand ← -1
        i ← i+1
    return cand
end
```

8.2.1.4 Exerciții

Exercițiul 8.2.1. În timpul execuției subprogramului **Celebritate**, în partea care verifică dacă unicul candidat este celebritate este posibil ca anumite întrebări să se repete, ele mai fiind puse la determinarea candidatului. Este posibilă atât eliminarea acestei redundanțe cât și păstrarea timpului de execuție și a spațiului de memorie utilizat?

Exercițiul 8.2.2. Să se găsească similarități între problema celebrității și cea din exercițiul 2.9.10.

8.2.2 Al doilea studiu de caz: permutarea maximală

8.2.2.1 Prezentarea problemei

Se consideră o colectivitate formată din n persoane. Fiecare persoană din colectivitate are un singur preferat în colectivitate. Este posibil ca o persoană să se poate

prefera pe sine însăși. Un grup de persoane din colectivitate se numește *omogen*, dacă fiecare membru al grupului este preferat exact al unui singur membru din acel grup. Se pune problema alegerii unui grup omogen cu număr maxim de persoane.

8.2.2.2 Modelul matematic

Aspectul conceptual. Notăm cu A mulțimea finită reprezentând colectivitatea de persoane și cu $f : A \rightarrow A$ funcția care reprezintă relația de preferință: $f(i) = j$ înseamnă că j are ca preferat pe i (sau, echivalent, i este preferat de j). Problema constă în a determina o submulțime X a lui A cu proprietățile:

- i) $f(X) \subseteq X$ (cei care preferă persoane din X se află de asemenea în X);
- ii) $f|_X$ este injectivă (orice membru al grupului are un singur preferat);
- iii) X are număr maxim de elemente peste clasa submulțimilor lui A care satisfac i) și ii).

Prin $f|_X$ am notat restricția funcției f la submulțimea X . Remarcăm faptul că primele două condiții, i) și ii), sunt echivalente cu faptul că $f|_X : X \rightarrow X$ este bijecție (permutare).

Aspectul analitic. Vom aplica o strategie asemănătoare cu cea din problema celebrității: se vor elimina pe rând din A elementele nedorite până când se va obține X . Rămâne de determinat criteriul de eliminare. Pornim cu analiza problemei inițiale. Distingem următoarele două cazuri:

- $f(A) = A$. Atunci $X = A$.
- există $j \in A$ cu proprietatea că pentru orice $i \in A$, $f(i) \neq j$. Problema P' cu instanță $A' = A \setminus \{j\}$, $f|_{A'}$ și problema inițială au aceeași soluție.

Așadar, putem porni cu $CAND = A$ și la fiecare pas să eliminăm un j care nu aparține imaginei funcției $f|_{CAND}$. Procesul de eliminare continuă atât cât este posibil și la sfârșit vom avea $X = CAND$.

Aspectul computațional. Vom arăta mai întâi că testul de eliminare pentru un j fixat poate fi efectuat în timpul $O(1)$. Presupunem, fără să restrângem generalitatea, că $A = \{0, 1, \dots, n - 1\}$. Mulțimea $CAND$ poate fi reprezentată prin vectorul caracteristic:

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

Cu o mică modificare, vectorul x poate fi utilizat și pentru reprezentarea funcției $f|_{CAND}$:

$$x[i] = \begin{cases} f(i), & \text{dacă } i \in CAND \\ -1, & \text{altfel} \end{cases}$$

Cu această reprezentare, verificarea condiției $(\exists i \in CAND) f(i) = j$, pentru un j fixat, necesită în cazul cel mai nefavorabil n comparații. Acest număr poate fi redus la 1 dacă vom considera un tablou suplimentar b cu următoarea semnificație:

$$b[j] = \#\{i \in CAND \mid f(i) = j\}$$

Acum, verificarea condiției $f(\text{CAND}) = \text{CAND}$ se poate realiza printr-o singură parcurgere a tabloului b : dacă există j cu $x[j] \geq 0$ și $b[j] = 0$, atunci j va fi eliminat din CAND. Rezultă că problema determinării permutării maximale poate fi rezolvată în timpul $O(n^2)$.

8.2.2.3 Implementare

Programul permMax are următoarea descriere:

```

procedure permMax(f, n, x)
    for j ← 0 to n-1 do
        b[j] ← 0
    for i ← 0 to n-1 do
        b[f[i]] ← b[f[i]] + 1
    for i ← 0 to n-1 do
        x[i] ← f[i]
    existaj ← true
    while (existaj) do
        existaj ← false
        j ← 0
        while ((j < n) and not existaj) do
            if ((x[j] ≥ 0) and (b[j] = 0))
                then existaj ← true
                x[j] ← -1
                b[f[j]] ← b[f[j]] - 1
            j ← j+1
    end

```

8.2.2.4 Exerciții

Exercițiul 8.2.3. Funcția f poate fi reprezentată printr-un digraf: există arc de la i la j dacă și numai dacă $f(i) = j$. Să se arate că determinarea permutării maximale este echivalentă cu determinarea tuturor circuitelor în digraful asociat.

Exercițiul 8.2.4. Să se proiecteze un algoritm care să determine toate circuitele în digraful asociat funcției f . Să se precizeze complexitățile timp și spațiu pentru algoritmul proiectat.

Exercițiul 8.2.5. Poate fi utilizat algoritmul de la exercițiul 8.2.4 la determinarea tuturor circuitelor într-un digraf oarecare?

8.2.3 Prezentarea generală a paradigmăi

Strategiile aplicate în studiile de caz precedente sunt foarte asemănătoare. Aici vom arăta cum aceste strategii pot fi generalizate la nivel de paradigmă.

8.2.3.1 Modelul matematic

Aspectul conceptual. Problemele pentru care poate fi aplicată strategia eliminării sunt modelate după următoarea schemă:

Se consideră o mulțime S cu n elemente. Se pune problema determinării existenței unei submulțimi maximale $A \subseteq S$ care satisface o proprietate $P(A)$ (când există).

Aspectul analitic. Se caută o condiție C cu proprietatea $C(x) \Rightarrow x \notin A$. Fie CAND submulțimea obținută din S prin eliminarea elementelor x ce nu satisfac $C(x)$. Apoi se verifică dacă submulțimea CAND satisface proprietatea P ; dacă da, atunci $A = \text{CAND}$.

Aspectul computațional. Presupunem că verificarea condiției $C(x)$ se face în timpul $O(n^p)$. Determinarea submulțimii de candidați CAND se face în timpul $O(n^{p+1})$. Dacă verificarea proprietății P se face în timpul $O(n^q)$, atunci determinarea lui A necesită timpul $O(n^{p+1} + n^q)$.

8.2.3.2 Implementare

Programul care descrie soluția dată de strategia eliminării este foarte simplu și are următoarea descriere schematică:

```
function sol(S)
    CAND ← S
    for each x din CAND do
        if C(x) // x satisfac criteriul de eliminare
            then CAND ← CAND \ {x}
    if P(CAND) // CAND satisfac P
        then return CAND
        else return ∅
end
```

8.3 Alte considerații privind paradigmele de proiectare

Așa cum s-a observat deja din studiul de caz precedent, construcția modelului matematic nu se poate face totdeauna cu o delimitare netă între cele trei aspecte: conceptual, analitic și computațional. De fapt, în [BD62] nici nu se recomandă o astfel de delimitare. Este mult mai natural și mai eficient ca toate cele trei aspecte să fie dezvoltate simultan. Este posibil ca dezvoltarea relațiilor analitice între concepte să evidențieze anumite aspecte ale problemei care nu au fost complet sau corespunzător conceptualizate. În acest caz o revizuire a conceptelor este necesară. De asemenea, anumite performanțe computaționale pot fi îmbunătățite prin găsirea unor reprezentări echivalente conceptual, dar mai performante.

De multe ori nici cele două faze ale rezolvării problemei – modelul matematic și implementarea – nu pot fi considerate separat. O implementare defectuoasă poate reduce performanțele soluției descrise de model sau, dimpotrivă, o implementare bună poate îmbunătăți performanțele soluției analitice. În alte cazuri, aspectul computațional poate fi realizat numai după descrierea implementării.

8.4 Referințe bibliografice

Considerațiile despre necesitatea construirii modelului matematic, ca pas intermediar pe drumul problemă – program, se bazează pe cele din [BD62].

Formalizarea tehnicii de eliminare ca metodă apare pentru prima dată aici. Cele două probleme apar ca exemple independente în [Man89]. Problema celebrității, formalizată în limbajul teoriei grafurilor, apare ca exercițiu în [CLR93, CLR00, MS91, Baa78, BG00]. Testarea primalității unui număr întreg utilizând testul Fermat [Luc93] constituie un alt exemplu de algoritm bazat pe tehnica eliminării.



Capitolul 9

Algoritmi greedy

9.1 Memorarea eficientă a programelor

9.1.1 Descrierea problemei

N programe (fișiere) urmează a fi memorate pe o bandă magnetică. Citirea unui program presupune citirea tuturor programelor aflate înaintea sa și deci timpul de regăsire este suma timpilor de citire ai tuturor acestor programe (inclusiv cel căutat). *Timpul mediu* de regăsire este media aritmetică a celor n timpi de regăsire. Problema constă în găsirea unei aranjări a celor n programe astfel încât timpul mediu de regăsire să fie minim.

9.1.2 Modelul matematic

Problema poate fi descrisă în următoarea manieră mai abstractă. Considerăm n obiecte notate cu $0, 1, \dots, n - 1$ de dimensiuni L_0, \dots, L_{n-1} , respectiv, care urmează a fi aranjate într-o listă liniară. Dacă obiectele sunt aranjate în ordinea $(\pi(0), \dots, \pi(n - 1))$, atunci timpul t_k de regăsire al obiectului $\pi(k)$ (care se află pe poziția k) este $\sum_{j=0}^k L_{\pi(j)}$ și timpul mediu de regăsire al tuturor obiectelor este $TM = \frac{1}{n} \sum_{k=0}^{n-1} t_k$. Problema constă în determinarea unei permutări $\pi = (\pi(0), \dots, \pi(n - 1))$ astfel încât, aranjând obiectele în ordinea dată de π , timpul mediu de regăsire să fie minim.

Scriem $SUMA(\pi) = \sum_{k=0}^{n-1} t_k$ detaliat:

$$\begin{aligned} SUMA(\pi) &= L_{\pi(0)} + && (k = 0) \\ &+ L_{\pi(1)} + && (k = 1) \\ &+ L_{\pi(0)} + L_{\pi(1)} + L_{\pi(2)} + && (k = 2) \end{aligned}$$

$$\cdots + L_{\pi(0)} + L_{\pi(1)} + \cdots + L_{\pi(n-1)} \quad (k = n - 1)$$

Timpul mediu depinde direct de această sumă. Este ușor de văzut că obiectele de la începutul listei contribuie de mai multe ori la sumă. Intuiția ne spune că suma este cu atât mai mică cu cât elementele de la începutul listei sunt mai mici. De aici, rezultă următoarea strategie de aranjare a obiectelor în lista liniară:

- dacă până la momentul $i - 1$ s-a construit deja permutarea parțială $(\pi(0), \dots, \pi(i - 1))$, atunci la momentul i se va alege obiectul $\pi(i) = k$ cu dimensiunea L_k minimă peste mulțimea obiectelor nearanjate în listă.

Corectitudinea strategiei este dată de următoarea teoremă:

Teorema 9.1. *Dacă $L_{\pi(0)} \leq L_{\pi(1)} \leq \cdots \leq L_{\pi(n-1)}$, atunci*

$$SUMA(\pi) = \min \{ SUMA(\pi') \mid \pi' \text{ o permutare a mulțimii } \{0, 1, \dots, n - 1\} \}$$

Demonstrație. Aplicăm metoda reducerii la absurd. Fie π o permutare optimă pentru care există $i < j$ cu $L_{\pi(i)} > L_{\pi(j)}$. Notăm cu π' permutarea π înmulțită cu transpoziția (i, j) . Făcând calculele, se obține $SUMA(\pi) > (SUMA(\pi'))$. Contradicție.

sfdem

9.1.3 Implementare

Algoritmul corespunzător strategiei de mai sus este descris de schema procedurală `memprog`:

```

procedure memprog(L, n, π)
begin
    S ← {0, ..., n - 1}
    while (S ≠ ∅) do
        alege k ∈ S astfel încât L[k] = min{L[j] | j ∈ S}
        S ← S \ {k}
        π[i] ← k
    end
end

```

O procesare care ordonează a obiectele în ordinea crescătoare a dimensiunilor conduce la alegerea obiectului k în timpul $O(1)$. Rezultă că problema poate fi rezolvată în timpul cel mai nefavorabil $O(n \log n)$.

9.2 Prezentare intuitivă a paradigmiei

Principalele ingrediente ale paradigmiei algoritmilor greedy sunt următoarele:

1. S – o mulțime de intrări;
2. C – un tip de date cu proprietățile:
 - a) obiectele din C reprezintă submulțimi ale lui S ;
 - b) operațiile includ inserarea ($X \cup \{x\}$) și eliminarea ($X \setminus \{x\}$).

3. Clasa de probleme la care se aplică include probleme de optim. Convenim să luăm ca exemplu următoarea descriere schematică:

Intrare: S ;

Ieșire: O submulțime maximală B din C care optimizează o funcție f cu valori reale.

În subsecțiunea 9.10 vom vedea că trebuie adăugate anumite condiții suplimentare. Pentru moment, această descriere sumară este suficientă pentru scopul nostru.

Pentru problema memorării programelor, considerăm $S = \{(i, j) \mid 0 \leq i, j < n\}$, $X \in C$ dacă și numai dacă $(\forall(i, j), (i', j') \in X) i = i' \iff j = j'$. O pereche (i, j) are semnificația „pe poziția i se află obiectul j ”. Cu alte cuvinte, presupunem că la intrare avem toate posibilitățile în care putem aranja obiectele. O submulțime $X \in C$ corespunde unei aranjări parțiale: pe o poziție i se află un singur obiect (relația X este funcțională) și un obiect j se află pe o singură poziție (relația X este injectivă). O submulțime maximală B desemnează o permutare π . Funcția f peste submulțimi maximale este *SUMA*. Ea poate fi extinsă la mulțimi arbitrară X prin $f(X) = \sum(t_k(X) \mid (\exists \ell) (k, \ell) \in X)$, unde $t_k(X) = \sum(L_j \mid (\exists i) (i, j) \in X \wedge j \leq k)$.

4. Paradigma se bazează pe următoarele două proprietăți:

a) *Proprietatea de alegere locală.* Soluția problemei se obține făcând alegeri optime locale (de aici și denumirea de *greedy*=„lacom”). O alegere optimă locală poate depinde de alegerile de până atunci, dar nu și de cele viitoare. Alegerile optime locale nu asigură automat că soluția finală realizează optimul global, adică constituie o soluție a problemei. Trebuie demonstrat acest fapt. De regulă, aceste demonstrații nu sunt foarte simple. Aceasta este un inconvenient major al metodei *greedy*. Algoritmii sunt relativ simpli, dar demonstrarea faptului că aceștia rezolvă întradevăr problema de optim asociată este deseori dificilă. În subsecțiunea 9.10 vom prezenta un cadru formal pentru aceste demonstrații.

O alegere optimă locală (alegere *greedy*) pentru problema memorării programelor constă în selectarea la pasul i a unei perechi $i \mapsto j$ cu proprietate că L_j este minim peste $\{L_{j'} \mid j' \text{ neales}\}$. Observăm că această alegere depinde numai de alegerile făcute până atunci. În final a trebuit să demonstrăm că aceste alegeri produc o permutare optimă.

b) *Proprietatea de substructură optimă.* Soluția optimă a problemei conține soluțiile optime ale subproblemelor.

Pentru problema memorării programelor, dacă B este o permutare optimă, atunci orice $B' \subseteq B$ este o permutare optimă pentru submulțimea de obiecte i cu $i \mapsto j \in B'$.

9.3 Arbore ponderați pe frontieră optimi

Studiul de caz prezentat în această secțiune are un grad de generalitate destul de mare astfel că poate fi considerat mai degrabă ca fiind o subparadigmă. Vom vedea că multe dintre problemele prezentate în secțiunile următoare sunt de fapt cazuri

particulare de arbori ponderați pe frontieră optimi. Din acest motiv, prezentarea este făcută direct la nivelul abstract.

9.3.1 Modelul matematic

Considerăm arbori binari cu proprietatea că orice vârf are 0 sau 2 succesi și vârfurile de pe frontieră au ca informații (etichete, ponderi) numere, notate cu $\inf(v)$. Convenim să numim acești arbori ca fiind *ponderați pe frontieră*. Pentru un vârf v din arborele t notăm cu d_v lungimea drumului de la rădăcina lui t la vârful v . *Lungimea externă ponderată* a arborelui t este:

$$\text{LEP}(t) = \sum_{v \text{ pe frontieră lui } t} d_v \cdot \inf(v).$$

Modificăm acești arbori etichetând vârfurile interne cu numere ce reprezintă suma etichetelor din cele două vârfuri fii. Adică, pentru orice vârf intern v avem $\inf(v) = \inf(v_1) + \inf(v_2)$, unde v_1, v_2 sunt fiii lui v (a se vedea și figura 9.1).

Lema 9.1. *Fie t un arbore binar ponderat pe frontieră. Atunci*

$$\text{LEP}(t) = \sum_{v \text{ intern în } t} \inf(v)$$

Demonstrație. Se procedează prin inducție după n , numărul de vârfuri de pe frontieră lui t .

Baza inducției. Presupunem $n = 2$. Relația este evidentă.

Pasul inductiv. Presupunem că t are $n+1$ vârfuri pe frontieră. Fie v_1 și v_2 două vârfuri de pe frontieră cu același predecesor imediat (tată) v_3 . Avem $d_{v_1} = d_{v_2} = d$ și $\inf(v_3) = \inf(v_1) + \inf(v_2)$. Considerăm arborele t' obținut din t prin eliminarea vârfurilor v_1 și v_2 . Acum vârful v_3 se află pe frontieră lui t' . Conform ipotezei inductive avem:

$$\text{LEP}(t') = \sum_{v \text{ intern în } t'} \inf(v) \quad (9.1)$$

Utilizăm (9.1) pentru a calcula lungimea externă ponderată a lui t :

$$\begin{aligned} \text{LEP}(t) &= \sum_{v \text{ pe frontieră lui } t} d_v \cdot \inf(v) \\ &= \sum_{v \text{ pe frontieră lui } t, v \neq v_1, v_2} d_v \cdot \inf(v) + d \cdot \inf(v_1) + d \cdot \inf(v_2) \\ &= \sum_{v \text{ pe frontieră lui } t, v \neq v_1, v_2} d_v \cdot \inf(v) + (d-1)(\inf(v_1) + \inf(v_2)) + \inf(v_3) \\ &= \sum_{v \text{ pe frontieră lui } t, v \neq v_1, v_2} d_v \cdot \inf(v) + (d-1)\inf(v_3) + \inf(v_3) \\ &= \sum_{v \text{ pe frontieră lui } t'} d_v \cdot \inf(v) \end{aligned}$$

$$\begin{aligned}
 &= \text{LEP}(t') + \inf(v_3) \\
 &= \sum_{v \text{ intern în } t'} \inf(v) + \inf(v_3) \\
 &= \sum_{v \text{ intern în } t} \inf(v)
 \end{aligned}$$

S-a ținut cont de faptul că interiorul lui t este format din interiorul lui t' la care se adaugă v_3 . sfdein

Exemplu. Fie arborele din figura 9.1. Lungimea externă ponderată a arborelui t

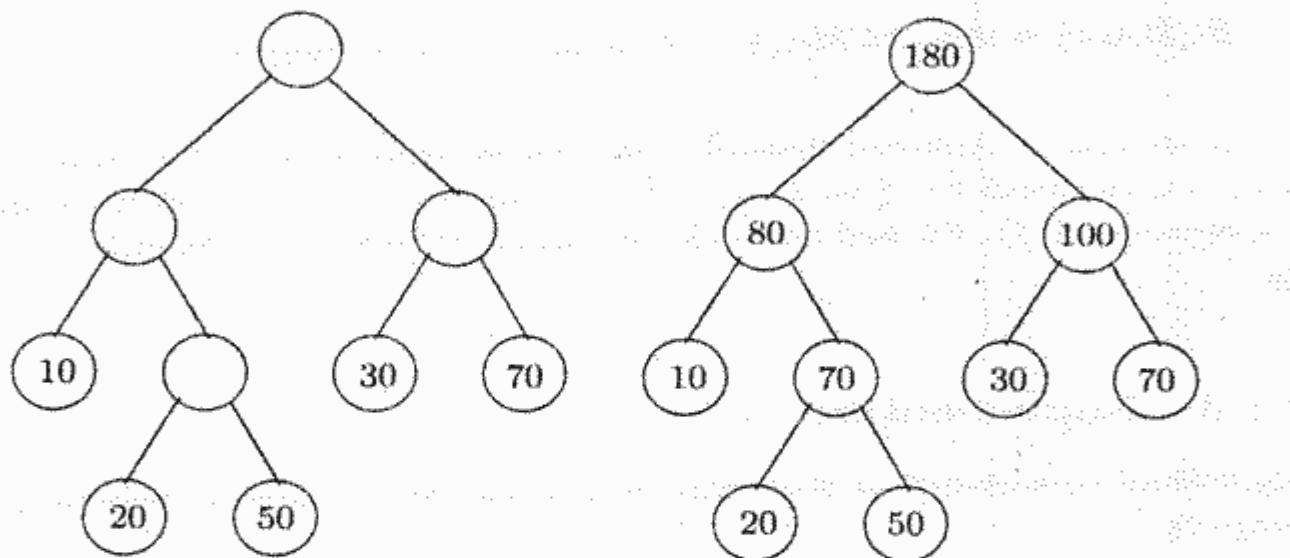


Figura 9.1: Arbore ponderat pe frontieră, înainte și după modificare

este egală cu $80 + 70 + 180 + 100 = 10 \cdot 2 + 20 \cdot 3 + 50 \cdot 3 + 30 \cdot 2 + 70 \cdot 2 = 430$. sfex

În continuare ne ocupăm de următoarea problemă:

Fie dată $x = (x_0, \dots, x_{n-1})$ o secvență (listă liniară) de numere. Problema constă în determinarea unui arbore binar ponderat pe frontieră care are ca informații în cele n vârfuri de pe frontieră numerele x_0, \dots, x_{n-1} și cu lungimea externă ponderată minimă.

O metodă total ineficientă ar putea fi generarea tuturor arborilor binari cu n vârfuri pe frontieră etichetate cu elementele secvenței x și alegerea unuia cu lungimea externă ponderată minimă.

Exercițiul 9.3.1. Să se arate că mulțimea arborilor binari ponderați pe frontieră cu elementele secvenței x este finită.

Pentru a obține un algoritm eficient este necesar să studiem mai mult proprietățile arborilor cu LEP minimă. Prin $\mathcal{T}(x)$ notăm mulțimea arborilor binari care au ca informații în vârfurile de pe frontieră numerele din secvența x .

Lema 9.2. Fie t un arbore din $\mathcal{T}(x)$ cu LEP minimă și v_1, v_2 două vârfuri pe frontieră lui t . Dacă $\inf(v_1) < \inf(v_2)$ atunci $d_{v_1} \geq d_{v_2}$.

Demonstrație. Presupunem $d_{v_1} < d_{v_2}$. Notăm $d_1 = d_{v_1}$ și $d_2 = d_{v_2}$. Fie t' arborele obținut din t prin interschimbarea vârfurilor v_1 și v_2 . Avem:

$$\begin{aligned} \text{LEP}(t') &= \sum_{v \text{ pe frontiera lui } t'} d_v \cdot \text{inf}(v) \\ &= \sum_{v \text{ pe frontiera lui } t', v \neq v_1, v_2} d_v \cdot \text{inf}(v) + d_1 \cdot \text{inf}(v_2) + d_2 \cdot \text{inf}(v_1) \\ &= \sum_{v \text{ pe frontiera lui } t} d_v \cdot \text{inf}(v) - d_1 \cdot \text{inf}(v_1) - d_2 \cdot \text{inf}(v_2) + d_1 \cdot \text{inf}(v_2) + \\ &\quad d_2 \cdot \text{inf}(v_1) \\ &= \text{LEP}(t) - (d_1 - d_2) \cdot (\text{inf}(v_1) - \text{inf}(v_2)) \\ &< \text{LEP}(t) \end{aligned}$$

Contradicție: s-a obținut un arbore cu lungime externă ponderată mai mică. Rezultă $d_1 \geq d_2$. sfdem

Lema 9.3. Presupunem $x_0 \leq x_1 \leq \dots \leq x_{n-1}$. Există un arbore în $T(x)$ cu LEP minimă și în care vârfurile x_0 și x_1 sunt vârfuri frate.

Demonstrație. Fie t un arbore cu LEP minimă. Fie v_i vârful etichetat cu x_i ($\text{inf}(v_i) = x_i$) și d_i distanța de la rădăcină la vârful v_i , $i = 0, \dots, n-1$. Deoarece $x_i \leq x_{i+1}$ rezultă, conform lemei 9.2, $d_i \geq d_{i+1}$ (în caz de egalitate $x_i = x_{i+1}$ considerăm pe locul i vârful mai depărtat de rădăcină). Fie v_i vârful frate al vârfului v_0 . Avem $d_1 \geq d_i$ (deoarece $x_1 \leq x_i$) și $d_1 \leq d_0 = d_i$ (deoarece $x_1 \geq x_0$ și v_0 și v_i sunt vârfuri frate) care implică $d_1 = d_i$. În arborele t interschimbăm vârfurile v_1 și v_i și obținem un arbore t' care satisfac concluzia lemei. sfdem

Lema 9.3 ne conduce direct la ideea algoritmului. Presupunem $x_0 \leq x_1 \leq \dots \leq x_{n-1}$. Știm că există un arbore optim t în care x_0 și x_1 sunt memorate în vârfuri frate. Tatăl celor două vârfuri va memora $x_0 + x_1$. Stergând cele două vârfuri ce memorează x_0 și x_1 se obține un arbore t' . Fie $t1'$ un arbore optim pentru secvența $y = (x_0 + x_1, x_2, \dots, x_{n-1})$ și $t1$ arborele obținut din $t1'$ prin „agățarea” a două vârfuri cu informațiile x_0 și x_1 de vârful ce memorează $x_0 + x_1$. Avem $\text{LEP}(t1') \leq \text{LEP}(t')$ care implică $\text{LEP}(t1) = \text{LEP}(t1') + x_0 + x_1 \leq \text{LEP}(t') + x_0 + x_1 = \text{LEP}(t)$. Dar cum t este optim, rezultă $\text{LEP}(t1) = \text{LEP}(t)$ și de aici t' este optim pentru secvența y . Considerăm în loc de secvențe de numere secvențe de arbori și obținem algoritmul:

```

procedure lep(x, n)
  1: B ← {t(x_0), ..., t(x_{n-1})}
  2: while #B > 1 do
  3:   alege t1, t2 din B cu inf(rad(t1)), inf(rad(t2)) minime
  4:   construiește arborele t în care subarborii rădăcinii
  5:   sunt t1, t2 și inf(rad(t)) = inf(rad(t1)) + inf(rad(t2))
  6:   B ← (B \ {t1, t2}) ∪ {t}
end

```

În procedura 1ep, $t(x_i)$ desemnează arborele format dintr-un singur vârf etichetat cu x_i iar $rad(t)$ rădăcina arborelui t . Inițial se consideră n arbori cu un singur vârf, care memorează numerele x_i , $i = 0, \dots, n - 1$. Pasul de alegere greedy constă în selectarea a doi arbori cu etichetele din rădăcină minime și construirea unui nou arbore ce va avea rădăcina etichetată cu suma etichetelor din rădăcinile celor doi arbori și pe cei doi arbori ca subarbori ai rădăcinii (figura 9.2).

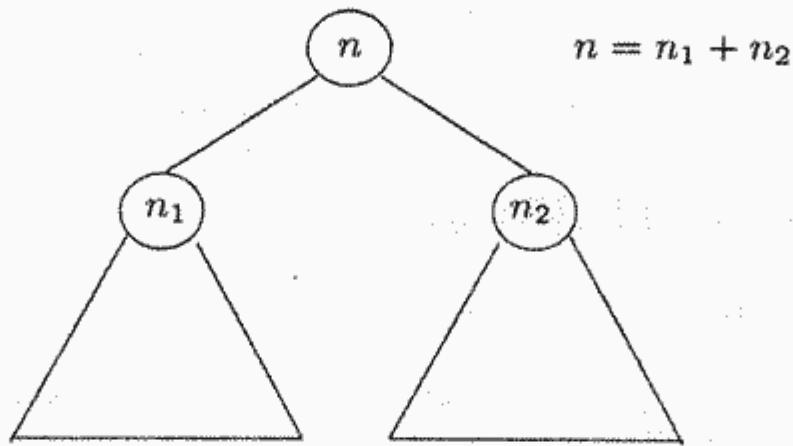


Figura 9.2: Construcția arborelui din pasul de alegere greedy

Discuția de mai sus conduce la următorul rezultat:

Teorema 9.2. *Fie $t^m(x)$ unicul element din mulțimea calculată de schema procedurală 1ep. Arboarele $t^m(x)$ are proprietatea:*

$$\text{LEP}(t^m(x)) = \min\{\text{LEP}(t) | t \in T(x)\} \quad (9.2)$$

9.3.2 Implementare

Descrierea ca program a algoritmului de mai sus nu are prea mare importanță practică. De fapt, aşa cum vom vedea și în secțiunile următoare, soluția de mai sus constituie mai mult un submodel ce va fi utilizat la rezolvarea unor probleme practice.

Totuși câteva comentarii sunt necesare. Astfel:

- dacă mulțimea B este implementată printr-o listă liniară, atunci în cazul cel mai nefavorabil operația 3 este are timpul de execuție $O(n)$, iar operația 6 are timpul de execuție $O(1)$;
- dacă mulțimea B este implementată printr-o listă liniară ordonată, atunci în cazul cel mai nefavorabil operația 3 are timpul de execuție $O(1)$, iar operația 6 are timpul de execuție $O(n)$;
- dacă mulțimea B este implementată printr-un heap, atunci în cazul cel mai nefavorabil operația 3 are timpul de execuție $O(\log n)$, iar operația 6 are timpul de execuție $O(\log n)$.

Rezultă că pentru implementarea mulțimii B , heapul este alegerea cea mai bună.

Literă	Frecvență	Literă	Cod
H	1	H	010
A	4	A	1
R	2	R	000
B	3	B	001
U	1	U	011
a)		b)	

Figura 9.3: Codificarea caracterelor din textul HARABABURA

9.4 Arbori Huffman

9.4.1 Descrierea problemei

N mesaje M_0, \dots, M_{n-1} sunt recepționate cu frecvențele f_0, \dots, f_{n-1} . Mesajele sunt codificate cu șiruri (cuvinte) construite peste alfabetul $\{0, 1\}$ cu proprietatea că pentru orice $i \neq j$, codul mesajului M_i nu este un prefix al codului lui M_j . O astfel de codificare se numește *independentă de prefix* („prefix-free”). Notăm cu d_i lungimea codului mesajului M_i . Lungimea medie a codului este $\sum_{i=0}^{n-1} f_i \cdot d_i$. Problema constă în determinarea unei codificări cu lungimea medie minimă.

9.4.2 Modelul matematic

Unei codificări îi putem asocia un arbore binar cu proprietatea că mesajele corespund nodurilor de pe frontieră, iar un cod descrie drumul de la rădăcină la mesajul corespunzător: 0 înseamnă deplasarea la fiul stâng, iar 1 deplasarea la fiul drept. Nodurile de pe frontieră arborelui sunt etichetate cu frecvențele mesajelor corespunzătoare: drumul de la rădăcină la un nod de pe frontieră descrie exact codul mesajului asociat acestui nod. Acum este ușor de văzut că determinarea unui cod optim coincide cu determinarea unui arbore ponderat pe frontieră optim.

Exemplu. Codurile Huffman pot fi utilizate la scrierea comprimată a textelor. Considerăm textul HARABABURA. Mesajele sunt literele din text, iar frecvențele sunt date de numărul de apariții ale fiecărei litere în text (a se vedea figura 9.3a).

Construcția arborelui Huffman este reprezentată în figura 9.4. Codurile obținute sunt reprezentate în figura 9.3b.

sfex

9.4.3 Implementare

Presupunem că intrarea este memorată într-un tabel T de structuri cu două câmpuri: $T[i].mes$ reprezentând mesajul i , iar $T[i].f$ frecvența mesajului i . Pentru implementare recomandăm reprezentarea arborilor prin tablouri. Notăm cu H tabloul reprezentând arboarele Huffman. Semnificația câmpului $H[i].elt$ este următoarea: dacă i este nod intern, atunci $H[i].elt$ reprezintă informația calculată din nod, iar

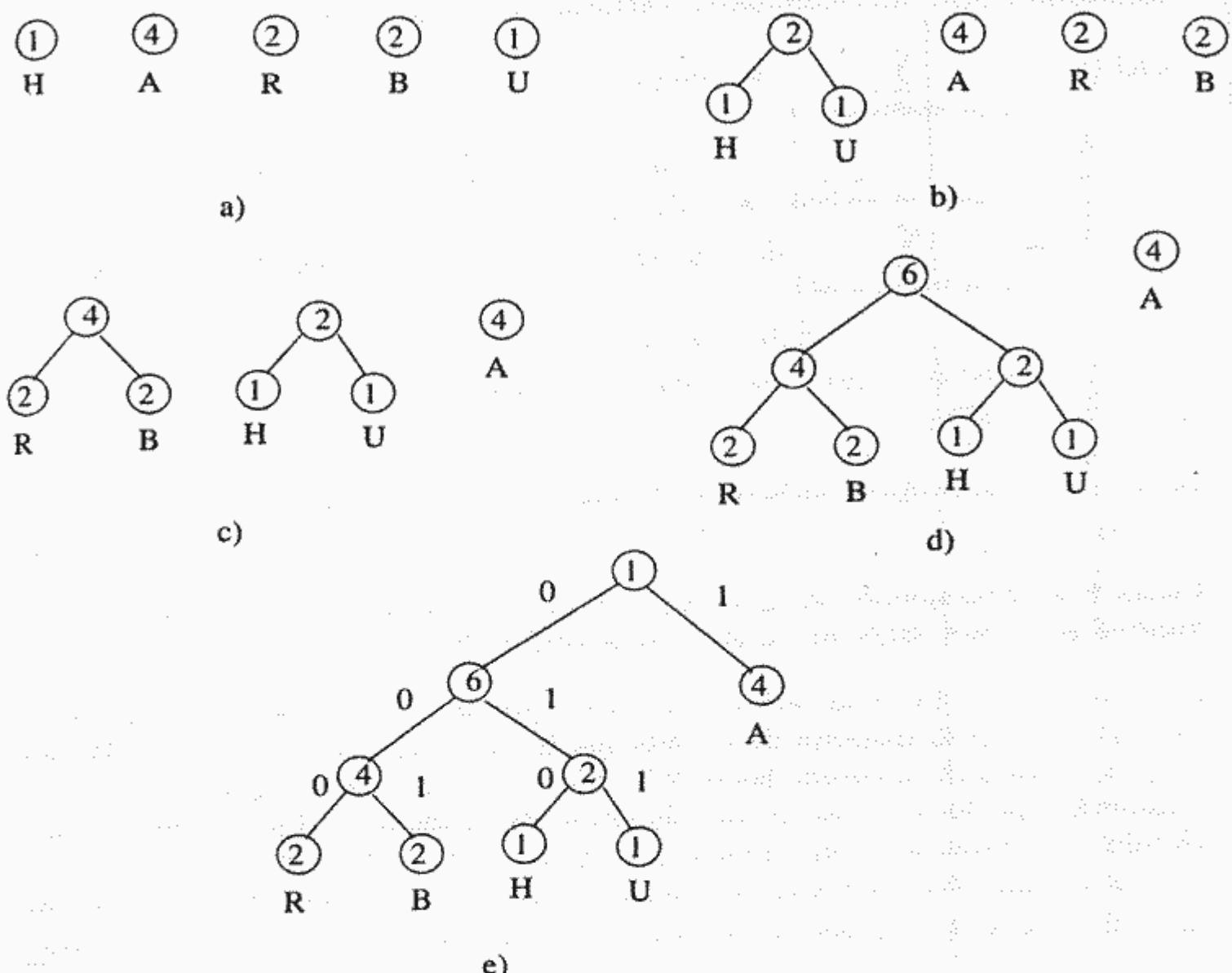


Figura 9.4: Construcția arborelui Huffman pentru HARABABURA

dacă i este pe frontieră (coresponde unui mesaj), atunci $H[i].elt$ este adresa din T a mesajului corespunzător. În ultimul caz, informația din nod este $T[H[i].elt].f$. Notăm cu $val(i)$ funcția care întoarce informația din nodul i , calculată ca mai sus. Tabloul H , care în final va memora arborele Huffman corespunzător codurilor optime, va memora pe parcursul construcției acestuia colecțiile intermedie de arbori. Astfel că, în timpul execuției algoritmului de construcție a arborelui, H are trei părți (figura 9.5): prima parte a tabloului va fi un min-heap care va conține rădăcinile arborilor din colecție. Partea de mijloc va conține nodurile care nu sunt rădăcini. Cea de-a treia parte este vidă și constituie zona în care partea din mijloc se poate extinde. Un pas al algoritmului de construcție ce realizează selecția greedy presupune parcurgerea următoarelor etape:

1. Mută rădăcina cu informația cea mai mică pe prima poziție liberă din zona a treia, să zicem k . Aceasta este realizată de următoarele operații:

a) copie rădăcina de pe prima poziție din heap pe poziția k :

$$H[k] \leftarrow H[1]$$

$$k \leftarrow k + 1$$

b) aduce ultimul element din heap pe prima poziție:

$$\begin{aligned} H[1] &\leftarrow H[m] \\ m &\leftarrow m - 1 \end{aligned}$$

c) reface min-heapul apelând subprogramul de introducere în min-heap.

2. Mută din nou rădăcina cu informația cea mai mică pe prima poziție liberă din zona a treia, dar fără a o elimina din min-heap:

$$\begin{aligned} H[k] &\leftarrow H[1] \\ k &\leftarrow k + 1 \end{aligned}$$

3. Construiește noua rădăcină și o memorează pe prima poziție în min-heap (în locul celei mutate mai sus).

4. Reface min-heapul apelând subprogramul de introducere în min-heap.

<i>heapul rădăcinilor</i>	<i>noduri care nu sunt rădăcini</i>	<i>zonă vidă</i>
---------------------------	-------------------------------------	------------------

Figura 9.5: Organizarea tabloului H

Algoritmul rezultat are timpul de execuție $O(n \log n)$.

9.5 Interclasare optimă pe două căi

9.5.1 Descrierea problemei

Se consideră m liste liniare sortate a_0, \dots, a_{m-1} care conțin n_0, \dots, n_{m-1} , respectiv, elemente dintr-o mulțime total ordonată. Interclasarea celor m secvențe constă în execuția repetată a următorului proces: se extrag din mulțime două liste și se pune în locul lor lista obținută prin interclasarea acestora. Procesul se continuă până când se obține o singură listă sortată cu cele $n_0 + \dots + n_{m-1}$ elemente. Problema constă în determinarea unei alegeri pentru care numărul total de transferuri de elemente să fie minim.

Un exemplu este dat de *sortarea externă*. Presupunem că avem de sortat un volum mare de date ce nu poate fi încărcat în memoria internă. Soluția este următoarea: se partionează colecția de date în mai multe liste ce pot fi ordonate cu unul dintre algoritmii de sortare internă din capitolul 4. Listele sortate sunt memorate în fișiere pe suport extern. Sortarea întregii colecții se face prin interclasarea fișierelor ce memorează listele sortate.

9.5.2 Modelul matematic

Mai întâi considerăm problema interclasării a două liste sortate:

Fie date două liste sortate $x = (x_0, \dots, x_{m-1})$ și $y = (y_0, \dots, y_{n-1})$ ce conțin elemente dintr-o mulțime total ordonată. Să se construiască o listă sortată $z = (z_0, \dots, z_{n+m-1})$ care să conțină cele $m + n$ elemente ce apar în x și y .

Utilizăm notația $z = \text{merge}(x, y)$ pentru a nota faptul că z este rezultatul interclasării listelor x și y . O prezentare a algoritmului de interclasare a două liste ordonate se găsește în 10.2. Numărul de comparații executate de algoritm este cel mult $m + n - 1$, iar numărul de elemente transferate este $m + n$.

Revenim la problema interclasării a m liste. Mai întâi considerăm un exemplu. Fie $m = 5, n_0 = 20, n_1 = 60, n_2 = 70, n_3 = 40, n_4 = 30$. Un mod de alegere a listelor pentru interclasare este următorul:

$$\begin{aligned} b_0 &= \text{merge}(a_0, a_1) \\ b_1 &= \text{merge}(b_0, a_2) \\ b_2 &= \text{merge}(a_3, a_4) \\ b &= \text{merge}(b_1, b_2) \end{aligned}$$

Numărul de transferuri al acestei soluții este $(20 + 60) + (80 + 70) + (40 + 30) + (150 + 70) = 80 + 150 + 70 + 220 = 520$. Există alegeri mai bune? Răspunsul este afirmativ și invităm cititorul să găsească o astfel de alegere.

Unei alegeri i se poate ataşa un arbore binar în modul următor:

- informațiile din vârfuri sunt lungimi de secvențe,
- vârfurile de pe frontieră corespund secvențelor inițiale a_0, \dots, a_{m-1} ,
- vârfurile interne corespund secvențelor intermediare.

Pentru soluția de mai sus avem arboarele din figura 9.6a. Se observă ușor că aceștia sunt arbori ponderați pe frontieră și numărul de transferuri de elemente corespunzător unei alegeri este egală cu LEP a arborelui asociat. Așadar, alegerea optimă corespunde arborelui cu LEP minimă. Algoritmul greedy care interclasează optim cele m liste este următorul:

```
procedure intercl0pt(x, n)
begin
    B ← {a0, ..., an-1}
    while (#B > 1) do
        alege x1, x2 din B cu lungimi minime
        intercl2(x1, x2, y)
        B ← (B \ {x1, x2}) ∪ {y}
    end
```

Pentru exemplul de mai sus, soluția optimă dată de algoritmul greedy este:

$$\begin{aligned} b_0 &= \text{merge}(a_0, a_4) \\ b_1 &= \text{merge}(a_3, b_0) \\ b_2 &= \text{merge}(a_1, a_2) \\ b &= \text{merge}(b_1, b_2) \end{aligned}$$

și are asociat arboarele cu lungime externă ponderată minimă din figura 9.6b. Numărul de comparații este $50 + 90 + 130 + 220 = 490$.

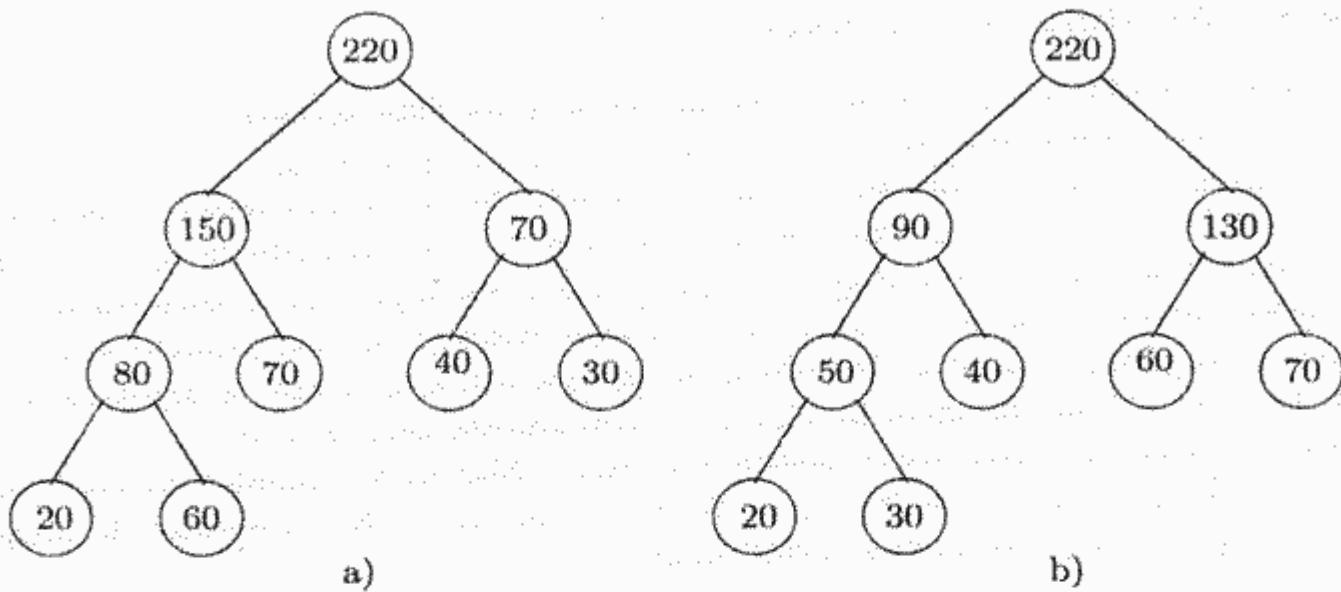


Figura 9.6: Arbori asociati algoritmilor de interclasare

9.6 Problema rucsacului I (varianta continuă)

9.6.1 Descrierea problemei

Se consideră un rucsac de capacitate M și n obiecte notate cu $0, 1, \dots, n - 1$ de dimensiuni (greutăți) w_0, w_1, \dots, w_{n-1} . Dacă în rucsac se pune o parte fracționară x_i din obiectul i , $0 \leq x_i \leq 1$, atunci se obține un profit $p_i \cdot x_i$ ($p_i > 0$). Umplerea rucsacului cu fracțiunile (cantitățile) x_0, \dots, x_{n-1} aduce profitul total $\sum_{i=0}^{n-1} p_i x_i$. Problema constă în a determina părțile fracționare x_0, \dots, x_{n-1} care aduc un profit total maxim.

9.6.2 Modelul matematic

Problema poate fi formulată ca o problemă de optim, în modul următor:

- funcția obiectiv:

$$\max \sum_{i=0}^{n-1} p_i x_i$$

- restricții:

$$\begin{aligned} \sum_{i=0}^{n-1} w_i x_i &\leq M \\ 0 \leq x_i &\leq 1 \text{ pentru } i = 0, \dots, n - 1 \end{aligned}$$

Dacă $\sum_{i=0}^{n-1} w_i \leq M$, atunci profitul maxim se obține când $x_i = 1, 0 \leq i \leq n - 1$. De aceea vom presupune că $\sum_{i=0}^{n-1} w_i > M$. În acest caz, nu toate fracțiunile x_i pot fi egale cu 1. În plus, rucsacul poate fi umplut exact, i.e., putem alege x_i astfel încât $\sum_{i=0}^{n-1} w_i x_i = M$.

Vom prezenta două strategii greedy: una care nu determină întotdeauna soluția optimă – pentru a evidenția că nu întotdeauna alegerea locală cea mai lacomă conduce la soluția optimă – și una care dă întotdeauna soluția optimă.

9.6.2.1 Soluția 1

La fiecare pas se va introduce în rucsac obiectul care aduce profit maxim. La ultimul pas, dacă obiectul nu încape în totalitate, se va introduce numai acea parte fracționară a sa, care umple exact rucsacul. Descrierea algoritmului este dată de următorul algoritm:

```

procedure rucsac_I1(w, p, x, n)
    S ← {0, ..., n - 1}
    for i ← 0 to n-1 do
        x[i] ← 0
        C ← 0
    while ((C < M) and (S ≠ ∅)) do
        * : alege  $i \in S$  care maximizează profitul peste  $S$ 
        S ← S \ {i}
        if  $(C + w[i] \leq M)$ 
            then  $C \leftarrow C + w[i]$ 
                  x[i] ← 1
            else  $C \leftarrow M$ 
                  x[i] ←  $\frac{M - C}{w[i]}$ 
    end

```

Procedura **rucsac_I1** are dezavantajul că nu determină optimul întodeauna.

Exemplu. Presupunem $n = 3, M = 10$, iar dimensiunile și profiturile obiectelor date de următorul tabel:

	0	1	2
w_i	6	4	8
p_i	3	4	6

Algoritmul **rucsac_I1** va determina soluția $x = (0, \frac{1}{2}, 1)$ care produce profitul $\sum p_i x_i = \frac{1}{2} \cdot 4 + 1 \cdot 6 = 8$. Se observă că vectorul $x' = (0, 1, \frac{3}{4})$ produce un profit mai bun: $\sum p_i x'_i = 1 \cdot 4 + \frac{3}{4} \cdot 6 = \frac{17}{2} > 8$. sfex

9.6.2.2 Soluția 2

La fiecare pas va fi introdus în rucsac obiectul care aduce profit maxim pe unitatea de capacitate (greutate) utilizată, adică obiectul care maximizează fracția $\frac{p_i}{w_i}$ peste mulțimea obiectelor neintroduse încă. Algoritmul corespunzător acestei strategii se obține din **rucsac_I1** prin înlocuirea liniei * : cu

alege $i \in S$ care maximizează profitul pe unitatea de greutate peste S .

Corectitudinea strategiei este dată de următorul rezultat.

Teorema 9.3. *Procedura **rucsac_I2** determină soluția optimă (cu profit maxim).*

Demonstrație. Presupunem $\frac{p_0}{w_0} \geq \dots \geq \frac{p_{n-1}}{w_{n-1}}$. Fie $x = (x_0, \dots, x_{n-1})$ soluția generată de procedura **Rucsac_I2**. Dacă $x_i = 1, 0 \leq i < n$, atunci evident că această

soluție este optimă. Altfel, fie j primul indice pentru care $x_j \neq 1$. Din algoritm, se observă că $x_i = 1$ pentru orice $0 \leq i < j$ și $x_i = 0$ pentru $i > j$. Fie $y = (y_0 \dots y_{n-1})$ o soluție optimă (care maximizează profitul). Avem $\sum_{i=0}^{n-1} y_i w_i = M$. Fie k cel mai mic indice pentru care $x_k \neq y_k$. Există următoarele posibilități:

- $k < j$. Rezultă $x_k = 1$ și de aici $y_k \neq x_k$ implică $y_k < x_k$.
- $k = j$. Deoarece $\sum x_i \cdot w_i = M$ și $x_i = y_i$, $1 \leq i < j$, rezultă că $y_k < x_k$ (altfel $\sum y_i \cdot w_i > M$ care constituie o contradicție).
- $k > j$. Rezultă $\sum_{i=0}^{n-1} y_i \cdot w_i > \sum_{i=0}^j x_i \cdot w_i = M$. Contradicție.

Deci, toate situațiile conduc la concluzia $y_k < x_k$ și $k \leq j$. Mărim y_k cu diferența până la x_k și scoatem această diferență din secvența $(y_{k+1}, \dots, y_{n-1})$, astfel încât capacitatea utilizată să rămînă tot M . Rezultă o nouă soluție $z = (z_0, \dots, z_{n-1})$ care satisfacă:

$$z_i = x_i, \quad 0 \leq i \leq k$$

$$\sum_{k < i \leq n-1} (y_i - z_i) \cdot w_i = (x_k - y_k) \cdot w_k$$

Aveam:

$$\begin{aligned} \sum_{i=0}^{n-1} z_i \cdot p_i &= \sum_{i=0}^{n-1} y_i \cdot p_i + \sum_{0 \leq i < k} z_i \cdot p_i + z_k \cdot p_k + \sum_{k < i < n} z_i \cdot p_i - \sum_{0 \leq i < k} y_i \cdot p_i - y_k \cdot p_k - \\ &\quad - \sum_{k < i < n} y_i \cdot p_i \\ &= \sum_{i=0}^n y_i \cdot p_i + (z_k - y_k) \cdot p_k \cdot \frac{w_k}{w_k} - \sum_{k < i < n} (y_i - z_i) \cdot p_i \cdot \frac{w_i}{w_i} \\ &\geq \sum_{i=0}^{n-1} y_i \cdot p_i + (z_k - y_k) \cdot w_k \frac{p_k}{w_k} - \sum_{k < i < n} (y_i - z_i) \cdot w_i \cdot \frac{p_k}{w_k} \\ &= \sum_{i=0}^{n-1} y_i \cdot p_i \end{aligned}$$

Deoarece y este soluție optimă, rezultă $\sum_{i=0}^{n-1} z_i p_i = \sum_{i=0}^{n-1} y_i p_i$. Soluția z are următoarele două proprietăți:

- este optimă, și
- coincide cu x pe primele k poziții (y coincidea cu x numai pe primele $k-1$ poziții).

Procedeul de mai sus este repetat (considerând z în loc de y) până când se obține o soluție optimă care coincide cu x .

sfdem

Implementare. Timpul de execuție al algoritmului **rucsac_I2** este $O(n^2)$. Dar dacă intrările satisfac $\frac{p_0}{w_0} \geq \dots \geq \frac{p_{n-1}}{w_{n-1}}$, atunci algoritmul **rucsac_I2** necesită timpul $O(n)$. La acesta trebuie adăugat timpul de preprocesare (ordonare) care este $O(n \log n)$.

9.7 Secvențializarea optimă a activităților cu termen de realizare și profit

9.7.1 Descrierea problemei

Considerăm n activități numerotate cu $0, 1, \dots, n - 1$. Presupunem că realizarea fiecărei activități durează o unitate de timp și că prima activitate începe la momentul zero. Realizarea activității i aduce un profit $p(i) > 0$, $i = 0, \dots, n - 1$. Există constrângere ca fiecare activitate i să fie terminată la termenul-limită $d(i)$ ($d(i)$ reprezintă durata maximă care este la dispoziție pentru a executa i). Problema constă în determinarea unei liste liniare $(s(0), \dots, s(k - 1))$, care constituie o secvențializare a realizării activităților, astfel încât orice activitate este realizată în termen, i.e., $d(s(i)) \geq i + 1$, și profitul dat de aceasta este maxim.

Exemplu. Presupunem $n = 4$. Termenele de execuție și profiturile sunt date în următorul tabel:

i	0	1	2	3
$d(i)$	2	3	1	2
$p(i)$	20	35	35	25

O soluție este $(0, 3, 1)$ cu profitul $20 + 25 + 35 = 80$, iar o altă soluție este $(2, 3, 1)$ cu profitul $35 + 25 + 35 = 95$. Există secvențializări ce aduc un profit mai mare?

sfex

9.7.2 Modelul matematic

Numim *secvență (soluție) acceptabilă* o listă $(s(0), \dots, s(k - 1))$ cu proprietatea $d(s(i)) \geq i + 1$, $i = 0, \dots, k - 1$, i.e., orice activitate i este realizată în termen. Suntem interesați în a găsi o listă liniară s^* cu proprietatea:

$$\sum_i p(s^*(i)) = \max \left\{ \sum_i p(s(i)) \mid s \text{ soluție acceptabilă} \right\}$$

Numim aceste secvențe *soluții optime*. Următorul rezultat arată că, pentru o soluție acceptabilă, dacă ordonăm activitățile după termenele de predare, atunci se obține tot o soluție acceptabilă.

Teorema 9.4. Fie $s = (s(0), \dots, s(k - 1))$ o secvențializare a k lucrări. Dacă s este acceptabilă și $\pi = (\pi(0), \dots, \pi(k - 1))$ este o permutare a mulțimii $\{0, \dots, k - 1\}$ cu proprietatea $d(s(\pi(0))) \leq \dots \leq d(s(\pi(k - 1)))$, atunci $s(\pi) = (s(\pi(0)), \dots, s(\pi(k - 1)))$ este o secvență acceptabilă.

Demonstrație. Presupunem că există $i < j$ cu $d(s(i)) > d(s(j))$. Considerăm $s' = (\dots, s'(i) = s(j), \dots, s'(j) = s(i), \dots)$ (în rest coincide cu s). Din $d(s(j)) > j > i$ rezultă $d(s'(i)) > i + 1$ și din $d(s(i)) > d(s(j)) > j$ rezultă $d(s'(j)) > j + 1$.

Repetând raționamentul de mai sus, la un moment dat vom obține secvența care satisface concluzia teoremei.

sfdem

Observație. Teorema de mai sus este adevărată și în cazul când timpii de execuție ai activităților pot fi diferiți.

sfobs

Teorema 9.4 ne permite să facem următoarea modificare asupra definițiilor pentru soluție acceptabilă și soluție optimă. Fie $B \subseteq \{0, \dots, n-1\}$. Spunem că B este o *soluție acceptabilă* dacă există o secvență acceptabilă $(s(0), \dots, s(k-1))$ cu proprietatea că $B = \{s(0), \dots, s(k-1)\}$. B este *soluție optimă* dacă există o secvență optimă $s^* = (s^*(0), \dots, s^*(k-1))$ cu proprietatea că $B = \{s^*(0), \dots, s^*(k-1)\}$. Cu aceste definiții putem descrie algoritmul greedy care rezolvă secvențializarea optimă a activităților. Pasul de alegere greedy va selecta de fiecare dată o activitate care aduce un profit maxim și formează o soluție acceptabilă cu celelalte activități alese până în acel moment:

```

procedure secvact(p, d, n, B)
    X ← {0, ..., n - 1}
    B ← ∅
    while (X ≠ ∅) do
        alege i din X cu profit maxim
        X ← X \ {i}
        if B ∪ {i} este soluție acceptabilă
            then B ← B ∪ {i}
    end

```

Corectitudinea algoritmului **secvact** este dată de următoarea teoremă.

Teorema 9.5. Fie B mulțimea determinată de algoritmul **secvact**. Orice secvențializare acceptabilă a lui B este optimă.

Demonstrație. Presupunem $B = \{i_0, \dots, i_{k-1}\}$ și fie $B' = \{j_0, \dots, j_{k'-1}\}$ o soluție optimă. Ideea demonstrației este următoarea: dacă $B' \neq B$ și cele două mulțimi au r elemente comune, atunci se construiește o soluție optimă B'' care are $r+1$ elemente comune cu B . Procedeul se repetă până când se obține o soluție optimă care coincide cu B . Putem presupune, fără să restrângem generalitatea, că $p(i_0) \geq \dots \geq p(i_{k-1})$ și că $p(j_0) \geq \dots \geq p(j_{k'-1})$. În plus, dacă $p(j_a) = p(j_{a+1})$, atunci $j_a < j_{a+1}$. De asemenea, presupunem că algoritmul determină activitățile tot în această ordine (dintre toate activitățile cu același profit o va alege pe cea cu numărul de ordine – indicele – mai mic). Există următoarele posibilități:

1. $B \subset B'$. Din modul cum lucrează algoritmul **secvact**, rezultă că B' nu este acceptabilă (la B nu se mai poate adăuga nimic astfel încât să se poată obține o secvențializare acceptabilă).
2. $B' \subset B$. Rezultă că B' nu este soluție optimă.
3. B și B' incomparabile. Vom arăta că putem alege $i_a \in B \setminus B'$ și $j_b \in B' \setminus B$ cu $d(j_b) \leq d(i_a)$. Alegem i_a cu $a < \min\{k, k'\}$, $i_a \neq j_a$ și a este cel mai mic cu această proprietate. Fie $T = \{\ell_0, \dots, \ell_{t-1}\} = \{\ell \neq i_a \mid d(\ell) \leq d(i_a)\}$. Presupunem $d(\ell_0) \leq \dots \leq d(\ell_{t-1})$. Distingem următoarele două subcazuri:

- a) $T \subseteq B$. Rezultă că pentru orice $j_b \in B' \setminus B$ avem $d(j_b) > d(i_a) > t$. Fie s o secvențializare acceptabilă a lui B' (ordonată crescător după termenele de execuție). Urmează că secvența $(s(0) = \ell_0, \dots, s(t-1) = \ell_{t-1}, i_a, s(t), \dots)$ este acceptabilă și de aici rezultă $B' \cup \{i_a\}$ acceptabilă (în orice secvență acceptabilă a lui B' poate fi inserat și i_a). Dar aceasta contrazice faptul că B' este optimă.
- b) $\exists j_b \in T$ cu $j_b \notin B$. Din definiția mulțimii T rezultă $d(i_a) \geq d(j_b)$. Deci $B'' = B' \setminus \{j_b\} \cup \{i_a\}$ este acceptabilă. Din modul de alegere a lui i_a rezultă $p(i_a) \geq p(j_b)$ care implică B'' optimă.

Deci în singurul caz posibil am obținut o soluție optimă care are a elemente comune cu B (B' avea numai $a - 1$ elemente comune cu B). Repetând procedeul, vom obține la un moment dat o soluție optimă $B^* = B$. Acum teorema este demonstrată complet. sfdem

9.7.3 Implementare

Datorită teoremei 9.4 putem determina direct secvența optimă s^* cu $d(s^*(0)) \leq \dots \leq d(s^*(k-1))$. Vom utiliza un tablou s și o variabilă k cu proprietatea că $S = \{s[0], \dots, s[k-1]\}$ și $d(s[0]) \leq \dots \leq d(s[k-1])$. Testarea proprietății dacă $S \cup \{i\}$ este admisibilă se face prin inserarea lui i în tabloul s astfel încât ordonarea nedescrescătoare a termenilor de execuție să fie posibilă. Pentru ca algoritmul de inserare să poată fi descris uniform, este util de considerat o activitate -1 cu $d(-1) = p(-1) = -1$. Strategia de căutare este următoarea:

- se caută secvențial indicele i_0 cu:

$$d(s[i_0]) \leq d(i) \leq d(s[i_0 + 1]);$$

- se verifică proprietatea:

$$(\forall j) i_0 + 1 \leq j < k \Rightarrow d(s[j]) \geq j + 1;$$

- dacă rezultatul evaluării este *true*, atunci inserează i la poziția $i_0 + 1$.

Condițiile de mai sus pot fi combinate într-o singură:

- se determină secvențial de la dreapta la stânga primul i_0 cu proprietatea:

$$d[s[i_0]] \leq d[i] \vee d[s[i_0]] = i_0$$

- dacă procesul de căutare se termină cu $d[s[i_0]] \leq d[i]$, atunci din $d[s[j]] \geq j \wedge d[s[j]] \neq j$ rezultă $d[s[j]] \geq j + 1$, pentru orice $j > i_0$. În acest caz se inserează i pe poziția $i_0 + 1$.

Alegerea activităților de profit maxim este mai ușoară dacă acestea se consideră sortate după profit: $p[0] \geq \dots \geq p[n-1]$. Descrierea strategiei de mai sus este următoarea:

```

procedure secvact(p, d, n, s, k)
    p[-1] ← 0
    d[-1] ← 0
    s[-1] ← 0
    for i ← 1 to n-1 do
        j ← k
        while ((d[s[j]] > d[i]) and (d[s[j]] > j)) do
            j ← j-1
        if (d[s[j]] ≤ d[i])
            then i0 ← j
            for j ← k downto i0+1 do
                s[j+1] ← s[j]
            s[i0+1] ← i
            k ← k+1
    end

```

O buclă `for` execută $O(k)$ operații și deci timpul de execuție al algoritmului este $O(1) + \dots + O(n) = O(n^2)$. Timpul de execuție poate fi redus aproape de $O(n)$, reprezentând multimile prin arbori și utilizând algoritmii *union* și *find* pentru operațiile asupra multimilor (secțiunea 2.10).

9.8 Problema instructorului de schi

9.8.1 Descrierea problemei

Un instructor de schi are la dispoziție n perechi de schiuri pe care trebuie să le distribue la n elevi începători. Problema instructorului constă în faptul că distribuirea trebuie făcută în aşa fel încât suma diferențelor absolute dintre înălțimea elevului și lungimea schiurilor atribuite să fie minimă.

9.8.2 Modelul matematic

Notăm cu x_0, \dots, x_{n-1} înălțimile elevilor și cu y_0, \dots, y_{n-1} lungimile schiurilor. Problema revine la rezolvarea următoarei probleme de optim:

$$\min_{\pi \in S_n} \sum_{i=0}^{n-1} |x_i - y_{\pi(i)}|,$$

unde prin S_n am notat mulțimea permutărilor elementelor $\{0, \dots, n-1\}$. Strategia greedy propusă este următoarea: la cel mai mic elev fără schiuri se atrbuie cea mai scurtă pereche de schiuri disponibilă. Descrierea acestei strategii este:

```

procedure ski(x, y, n, B)
    X ← {x_0, ..., x_{n-1}}
    Y ← {y_0, ..., y_{n-1}}
    B ← ∅
    while ((X ≠ ∅) and (Y ≠ ∅)) do

```

```

x ← min X
X ← X \ {x}
y ← min Y
Y ← Y \ {y}
B ← B ∪ {(x, y)}
end

```

Fără să restrângem generalitatea, presupunem $x_0 \leq x_1 \leq \dots \leq x_{n-1}$. Cu această presupunere, criteriul de alegere locală se reduce la a considera elevii în ordine crescătoare. Corectitudinea strategiei de mai sus este dată de următoarea teoremă:

Teorema 9.6. Notăm cu $S(\pi)$ suma $\sum_{i=0}^{n-1} |x_i - y_{\pi(i)}|$. Dacă $y_{\pi(0)} \leq \dots \leq y_{\pi(n-1)}$, atunci:

$$S(\pi) = \min\{S(\pi') \mid \pi' \text{ permutare a mulțimii } \{0, 1, \dots, n-1\}\}.$$

Demonstrație. Fie π o permutare oarecare cu $i < j, y_{\pi(i)} > y_{\pi(j)}$. Vom arăta că $S(\pi') \leq S(\pi)$, unde π' este produsul lui π cu transpoziția (i, j) . Distingem următoarele cazuri:

1. $x_i \leq x_j \leq y_{\pi(i)}$.

a) $y_{\pi(j)} < x_i \leq x_j \leq y_{\pi(i)}$. Avem:

$$\begin{aligned} |y_{\pi(i)} - x_i| + |x_j - y_{\pi(j)}| &= |y_{\pi(i)} - x_j| + |x_i - y_{\pi(j)}| + 2(x_j - x_i) \\ &\geq |y_{\pi(i)} - x_j| + |x_i - y_{\pi(j)}| \end{aligned}$$

b) $x_i \leq y_{\pi(j)} \leq x_j \leq y_{\pi(i)}$. Avem:

$$\begin{aligned} |y_{\pi(i)} - x_i| + |x_j - y_{\pi(j)}| &= |y_{\pi(i)} - x_j| + |y_{\pi(j)} - x_i| + 2(x_j - y_{\pi(j)}) \\ &\geq |y_{\pi(i)} - x_j| + |y_{\pi(j)} - x_i| \end{aligned}$$

c) $x_i \leq x_j \leq y_{\pi(j)} \leq y_{\pi(i)}$. Avem:

$$|y_{\pi(i)} - x_i| + |y_{\pi(j)} - x_j| = |y_{\pi(j)} - x_i| + |y_{\pi(i)} - x_j|$$

2. $y_{\pi(j)} < y_{\pi(i)} < x_j$. Se analizează la fel ca mai sus.

În toate cazurile rezultă:

$$|x_i - y_{\pi(i)}| + |x_j - y_{\pi(j)}| \geq |x_i - y_{\pi(j)}| + |x_j - y_{\pi(i)}|$$

Acum are loc:

$$\begin{aligned} S(\pi') &= \sum_{k \neq i, j} |x_k - y_{\pi(k)}| + |x_i - y_{\pi(j)}| + |x_j - y_{\pi(i)}| \\ &\leq \sum_{k \neq i, j} |x_k - y_{\pi(k)}| + |x_i - y_{\pi(i)}| + |x_j - y_{\pi(j)}| \\ &= S(\pi) \end{aligned}$$

Am obținut următoarea proprietate:

$$(\forall i, j) i < j, y_{\pi(i)} > y_{\pi(j)}, \pi' = \pi \cdot (i, j) \Rightarrow S(\pi') \leq S(\pi)$$

Rezultă că permutarea π cu proprietatea $y_{\pi(0)} \leq \dots \leq y_{\pi(n-1)}$ realizează minimul funcției S . sfdem

Corolar 9.1. *Dacă*

$$(\forall i, j) x_i \leq y_j \text{ sau } (\forall i, j) y_j \leq x_i,$$

atunci $S(\pi) = \text{const.}$

Exercițiul 9.8.1. [MS91] Să se arate că algoritmul **ski** minimizează de asemenea și funcția $S(\pi) = \max\{|x_i - y_{\pi(i)}| \mid 0 \leq i \leq n - 1\}$ (diferența pentru cazul cel mai nefavorabil).

9.9 Arborele parțial de cost minim

Următoarea problemă este cunoscută ca *arborele parțial de cost minim*: dat un graf ponderat $G = (V, E)$ cu funcția de cost $c : E \rightarrow \mathcal{R}$, să se determine un arbore parțial de cost minim.

9.9.1 Modelul matematic

Reamintim că un arbore parțial al lui G este un subgraf conex fără cicluri ce include toate vârfurile lui G . Fie A o mulțime de muchii. O muchie $\{i, j\}$ este *sigură* pentru A dacă $A \cup \{\{i, j\}\}$ este o submulțime a unui arbore parțial de cost minim. Următorul algoritm generic, bazat pe strategia greedy, determină arborele parțial descris ca o mulțime de arce:

```

procedure APCM(V, E, c)
    A ← ∅
    E₁ ← E
    while (V, A) nu formează arbore parțial do
        determină o muchie {i, j} ∈ E₁ sigură A
        E₁ ← E₁ \ {{i, j}}
        A ← A ∪ {{i, j}}
    end

```

Teorema 9.7. (V, A) determinat de algoritmul APCM este arbore parțial de cost minim pentru $G = (V, E)$.

9.9.2 Implementare

Se cunosc doi algoritmi eficienți bazați pe schema de mai sus:

- *Algoritmul lui Kruskal*: mulțimea A este o colecție de arbori (pădure). Pasul de alegere locală selectează muchia de cost minim care nu formează circuite cu muchiile alese până în acel moment. O implementare eficientă a algoritmului lui Kruskal se obține reprezentând A printr-o structură *union-find*:

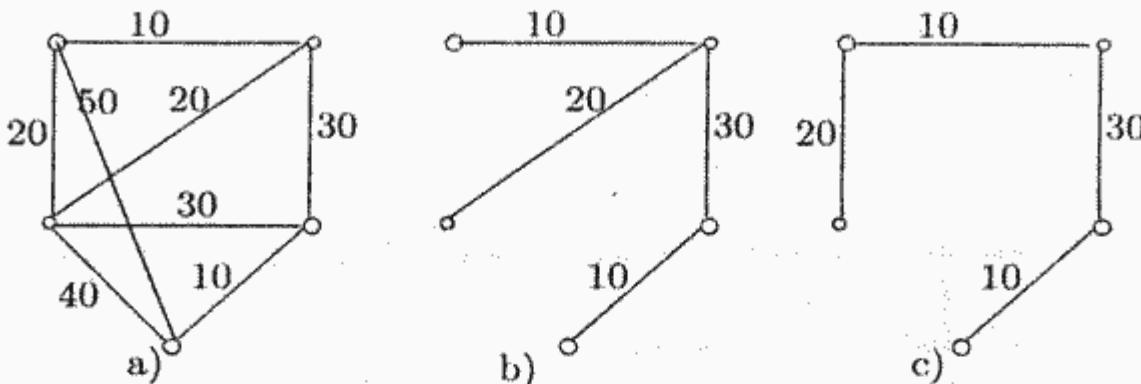


Figura 9.7: Arbore parțial de cost minim

```

procedure Kruskal(V,E,c)
    A ← ∅
    for each i ∈ V do single(i)
    sortează E crescător după c
    for each {i,j} ∈ E în ordine crescătoare do
        if (find(A, i) ≠ find(A, j))
            then union(A,i,j)
    end

```

- *Algoritmul lui Prim:* mulțimea A este arbore. Pasul de alegere locală selectează muchia de cost minim care, împreună cu celelalte alese până în acel moment, păstrează proprietatea de arbore. O implementare eficientă a algoritmului lui Prim se obține prin întreținerea unei structuri min-heap, pe care o notăm cu Q . Fiecare vârf i i se asociază o valoare $cheie(i)$ ce reprezintă minimul dintre costurile muchiilor care unesc acel vârf cu un vârf din arborele construit până în acel moment. Pe timpul execuției algoritmului, Q va memora, pe baza valorilor $cheie(i)$ definite mai sus, vâfurile care nu sunt în arbore. Mulțimea A va fi $A = \{\{j, parinte(j)\} \mid j \in V \setminus \{r\} \setminus Q\}$, unde r este rădăcina arboreului (aleasă arbitrar) și $parinte(j)$ este adresa vârfului care realizează valoarea $cheie(j)$:

```

procedure Prim(V,E,c)
    Q ← V
    for each i ∈ Q do cheie[i] ← ∞
    cheie[r] ← 0
    parinte[r] ← -1
    while Q ≠ ∅ do
        if (j ∈ Q and c(i,j) < cheie[j])
            then parinte[j] ← i
            cheie[j] ← c({i,j})
    end

```

În figura 9.7a este reprezentat un graf ponderat G , în fig. 9.7b arborele parțial minim determinat cu algoritmul lui Kruskal și în fig. 9.7c arborele parțial minim determinat cu algoritmul lui Prim (rădăcina este vârful din stânga-sus).

9.10 Prezentare formală a paradigmelor

9.10.1 Modelul matematic

Fie S o mulțime finită de intrări și \mathcal{C} o colecție de submulțimi ale lui S . Spunem că \mathcal{C} este *accesibilă* dacă satisfacă *axioma de accesibilitate*:

$$(\forall X \in \mathcal{C}) X \neq \emptyset \Rightarrow (\exists x \in X) X \setminus \{x\} \in \mathcal{C} \quad (9.3)$$

Dacă \mathcal{C} este accesibilă, atunci perechea (S, \mathcal{C}) se numește *sistem accesibil*. De exemplu, perechea (S, \mathcal{C}) definită în secțiunea 9.2 pentru problema memorării programelor este un sistem accesibil.

Exercițiul 9.10.1. Fie $G = (V, E)$ un graf. Definim $S(G) = E$ și $\mathcal{C}(G) = \{X \subseteq E \mid (V, X)$ este arbore}. Să se arate că $(S(G), \mathcal{C}(G))$ este sistem accesibil.

Exercițiul 9.10.2. Să se definească un sistem accesibil pentru arborii ponderați pe frontieră.

O submulțime $X \in \mathcal{C}$ se numește *bază* dacă este maximală, i.e., nu există $x \in S \setminus X$ cu $X \cup \{x\} \in \mathcal{C}$. O submulțime $X \in \mathcal{C}$ care nu este bază se numește *extensibilă*. Cu alte cuvinte, dacă X este extensibilă, atunci există $y \in S \setminus X$ astfel încât $X \cup \{y\} \in \mathcal{C}$. Clasa de probleme pentru care se pot defini algoritmi greedy este definită de următoarea schemă:

Se consideră date un sistem accesibil (S, \mathcal{C}) și o *funcție obiectiv* $f : \mathcal{C} \rightarrow \mathcal{R}$. Problema constă în determinarea unei baze $B \in \mathcal{C}$ care satisfacă:

$$f(B) = \text{optim}\{f(X) \mid X \text{ bază în } \mathcal{C}\}$$

În general, prin optim vom înțelege minim sau maxim. Strategia *greedy* constă în găsirea unui criteriu de selecție a elementelor din S care candidează la formarea bazei optime (care dă optimul pentru funcția obiectiv). Acest criteriu este numit *alegere greedy* sau *alegere a optimului local*. Formal, optimul local are o următoarea definiție [MS91]:

$$f(X \cup \{x\}) = \text{optim}\{f(X \cup \{y\}) \mid y \in S \setminus X, X \cup \{x\} \in \mathcal{C}\} \quad (9.4)$$

Această strategie este descrisă schematic de următorul algoritm:

```

procedure greedy(S, B)
    S1 ← S
    B ← ∅
    while (B este extensibilă) do
        alege un optim local x din S1 conform cu 9.4
        S1 ← S1 \ {x}
        B ← B ∪ {x}
    end

```

Din păcate, numai condiția de accesibilitate nu asigură existența întotdeauna a unui criteriu de alegere locală care să conducă la determinarea unei baze optime. Aceasta înseamnă că, pentru anumite probleme, putem proiecta algoritmi *greedy* care nu furnizează soluția optimă, ci o bază pentru care funcția obiectiv poate avea valori apropiate de cea optimă. Acesta este cazul, de exemplu, pentru anumite probleme NP-complete (a se vedea secțiunea 13.3).

9.10.1.1 Matroizi

În această subsecțiune vom prezenta o clasă particulară de probleme pentru care algoritmii *greedy* conduc la determinarea unei baze optime.

Perechea $M = (S, \mathcal{C})$ se numește *matroid*, dacă satisfac următoarele condiții:

- proprietatea de ereditate:

$$X \in \mathcal{C} \wedge X \neq \emptyset \Rightarrow (\forall x \in X) X \setminus \{x\} \in \mathcal{C},$$

- proprietatea de interschimbare:

$$X, Y \in \mathcal{C} \wedge \#X < \#Y \Rightarrow (\exists y \in Y \setminus X) X \cup \{y\} \in \mathcal{C}.$$

Exercițiul 9.10.3. Să se studieze dacă (S, \mathcal{C}) definit în secțiunea 9.2 pentru memorarea programelor este un matroid.

Exercițiul 9.10.4. Fie $G = (V, E)$ un graf. Definim $S(G) = E$ și $\mathcal{C}(G) = \{X \subseteq E \mid (V, X) \text{ este aciclic}\}$. Să se arate că $(S(G), \mathcal{C}(G))$ este un matroid. Rămâne afirmația adevărată dacă în loc de „aciclic” considerăm arbore?

Un *matroid ponderat* este un matroid $M = (S, \mathcal{C})$ în care fiecare element $x \in S$ are asociată o pondere $w(x)$. Problemele pe care le considerăm aici sunt definite după următorul şablon:

Se consideră un matroid ponderat $M = (S, \mathcal{C}, w)$. Dacă $X \subseteq S$, atunci definim $w(X) = \sum_{x \in X} w(x)$. Problema constă în determinarea unei baze $B \in \mathcal{C}$ care satisfacă:

$$w(B) = \max\{w(X) \mid X \text{ bază în } \mathcal{C}\}$$

Determinarea optimului local constă în selectarea unui $x \in S \setminus X$ cu proprietatea:

$$w(x) = \max\{w(y) \mid y \in S \setminus X, X \cup \{y\} \in \mathcal{C}\}.$$

Deoarece optimul local este determinat numai pe baza ponderilor elementelor rămasă, algoritmul *greedy* constă în testarea tuturor elementelor din S în ordinea descrescătoare a ponderilor $w(x)$:

```

procedure greedyMatr(S, w, B)
    sortează S descrescător după w(x)
    B ← ∅
    for each x ∈ S considerat în ordine descrescătoare după w(x) do
        if (B ∪ {x} ∈ C) then B ← B ∪ {x}
    end
  
```

Teorema 9.8. Dacă $M = (S, \mathcal{C}, w)$ este un matroid ponderat, atunci algoritmul `greedyMatr` determină o bază optimă.

Demonstrație. Fie x primul element din S ales de algoritmul `greedyMatr`. Mai întâi arătăm următoarea propoziție: există o bază optimă A care conține x . Fie A' o bază optimă. Dacă $x \in A'$, atunci luăm $A = A'$. Presupunem că $x \notin A'$. Luăm pentru început $A = \{x\}$. Din definiția algoritmului `greedyMatr`, $A \in \mathcal{C}$. Utilizând proprietatea de interschimbare în mod repetat, adăugăm elemente din A' la până când $\#A = \#A'$. Există $y \in A'$ cu proprietatea $A = A' \setminus \{x\} \cup \{y\}$. Din modul de alegere a lui x , avem $w(x) \geq w(y)$ care implică:

$$w(A) = w(A') - w(x) + w(y) \geq w(A').$$

Deoarece A' este optimă, rezultă $w(A) = w(A')$ și de aici avem că A este optimă. Acum demonstrația propoziției este terminată.

Propoziția de mai sus ne spune că algoritmul `greedy` procedează corect când îl alege pe x la primul pas pentru că acesta aparține unei baze optime. Fie acum $M' = (S', w', \mathcal{C}')$, unde $S' = \{y \in S \mid \{x, y\} \in \mathcal{C}\}$, $w' = w|_{S'}$ (restricția lui w la S') și $\mathcal{C}' = \{X \in \mathcal{C} \mid X \cup \{x\} \in \mathcal{C}\}$. Aplicând proprietatea de ereditate rezultă că M' este matroid și, dacă A este o bază optimă a lui M , atunci $A \setminus \{x\}$ este o bază optimă pentru M' (matroizii au proprietatea de substructură optimă). Fie B baza calculată de algoritmul `greedyMatr`. Aplicând un raționament inductiv, obținem că $B \setminus \{x\}$ este bază optimă pentru M' . De aici rezultă că B este bază optimă pentru M .

sfdem

9.10.2 Evaluarea timpului de execuție

Presupunem că pasul de alegere `greedy` selectează elemente x în timpul $O(k^p)$ unde $k = \#S_1$ și că testarea condiției $B \cup \{x\} \in \mathcal{C}$ se face în timpul $O(\ell^q)$, unde $\ell = \#B$. Avem $k + \ell \leq n$. Multimile S și B pot fi reprezentate astfel încât costul operațiilor $B \cup \{x\}$ și $S_1 - \{x\}$ să fie egal cu $O(1)$. Deoarece pasul de alegere este executat de n ori rezultă că strategia are timpul de execuție

$$\begin{aligned} T(n) &= O(n^p + 1^q) + \cdots + O(1^p + n^q) \\ &= O(1^p + \cdots + n^p + 1^q + \cdots + n^q) \\ &= O(n^{p+1} + n^{q+1}) = O(n^{\max(p+1, q+1)}) \end{aligned}$$

Operațiile peste multimile S_1 și B vor trebui descrise în funcție de reprezentările concrete ale acestora.

În cazul matroizilor ponderați, timpul de execuție pentru cazul cel mai nefavorabil poate fi redusă la $O(n \log n)$, dacă testarea condiției $B \cup \{x\} \in \mathcal{C}$ se face în timpul $O(1)$.

9.11 Exerciții

Exercițiul 9.1. [MS91] (Alocarea optimă a fișierelor pentru rețelele de calculatoare)
Se consideră o rețea cu n noduri și o mulțime de fișiere la care au acces toate

nodurile. Se presupune că se cunoaște în devans o secvență de cereri de regăsire/modificare a informației din fișiere. Pentru fiecare cerere se cunoaște nodul care a inițiat cererea, fișierul invocat și numărul de biți ce urmează a fi transferați. O *schemă de alocare* constă într-o atribuire a fiecărui fișier la unul sau mai multe noduri. Având mai multe copii ale aceluiași fișier avem un avantaj în regăsirea informației: costul unei regăsiri este zero, dacă fișierul este local, și este egal cu numărul de biți transferați, dacă fișierul este accesat de la distanță. Dar multiplicarea fișierelor crește costul operației de modificare, întrucât fiecare copie trebuie modificată și astfel numărul de biți accesati pentru modificare este înmulțit cu numărul de copii aflate la distanță. De asemenea, existența mai multor copii duce la creșterea siguranței în exploatare; dar aceasta se întâmplă până la un punct deoarece este o probabilitate foarte mică să cadă toate nodurile stației. Funcția care dă câștigul în siguranță depinde de numărul de copii și se supune următorului principiu: fiecare copie nou adăugată aduce un câștig mai mic decât copia anterioară. Costul unei scheme de alocare se obține prin însumarea costurilor cererilor de regăsire/modificare din secvența dată din care se scade câștigul în siguranță.

Să se proiecteze un algoritm greedy care să determine o schemă de alocare optimă și să se demonstreze corectitudinea sa.

Exercițiul 9.2. [HS84] (*Schimbarea banilor*) Fie $A_n = \{a_0, \dots, a_{n-1}\}$ o mulțime finită de tipuri de monezi. De exemplu: $a_0 = 100$ lei, $a_1 = 50$ lei etc. Presupunem că a_i este întreg pentru orice i , $0 \leq i < n$. Pentru fiecare tip există o infinitate de monezi. Se pune problema ca pentru un număr întreg C dat, să se determine numărul minim de monezi a căror sumă este exact C (C poate fi schimbată numai cu monezi de tipuri din A_n).

1. Să se arate că dacă $a_0 > \dots > a_{n-1}$ și $a_{n-1} \neq 1$, atunci există $C > 0$ pentru care problema nu are soluție.
2. Să se arate că dacă $a_{n-1} = 1$, atunci problema are întotdeauna soluție.
3. Să se proiecteze un algoritm greedy pentru cazul particular $A_n = \{k^{n-1}, \dots, k^0\}$, $k > 1$.
4. Să se arate că algoritmul găsit la 3 nu determină întotdeauna soluția optimă pentru cazul general.

Exercițiul 9.3. [HS84] (*Memorarea programelor II*) Se consideră n programe de lungimi $\ell_0, \dots, \ell_{n-1}$ ce urmează a fi memorate pe o bandă. Un program i este regăsit cu frecvența f_i . Dacă programele sunt memorate în ordinea $\pi(0), \pi(1), \dots, \pi(n-1)$, atunci timpul mediu de regăsire este:

$$T = \frac{\sum_{j=0}^{n-1} f_{\pi(j)} \cdot \sum_{k=0}^j \ell_{\pi(k)}}{\sum_{j=0}^{n-1} f_j}$$

Să se scrie un algoritm greedy care determină ordinea de memorare ce minimizează timpul mediu de regăsire. Să se demonstreze corectitudinea algoritmului.

Exercițiul 9.4. [CLR93] Se consideră o mulțime $S = \{0, 1, 2, \dots, n-1\}$ de activități care utilizează aceeași resursă. Fiecare activitate i are un timp de start s_i și un timp de terminare t_i . Dacă o activitate i este selectată, atunci ea se va desfășura în perioada de timp dată de intervalul semideschis $[s_i, t_i)$. Două activități i și j

sunt *compatibile* dacă intervalele $[s_i, t_i)$ și $[s_j, t_j)$ nu se acoperă, i.e., sau $s_i \geq t_j$ sau $s_j \geq t_i$. Să se proiecteze un algoritm greedy care să determine o mulțime cu număr maxim de activități compatibile. Să se dovedească corectitudinea algoritmului.

Exercițiul 9.5. Următorul algoritm, cunoscut sub numele de algoritm Dijkstra, determină drumurile minime într-un digraf ponderat $D = (\langle V, A \rangle, \ell)$ care pleacă dintr-un vârf i_0 dat, în cazul când ponderile $\ell[i, j]$ sunt ≥ 0 . Pentru fiecare vârf i , $d[i]$ va fi lungimea drumului minim de la i_0 la i și $p[i]$ va fi predecesorul lui i pe drumul minim de la i_0 la i . Q este un min-heap cu cheile date de valorile $d[i]$.

```

procedure Dijkstra(D, i0, d, p)
begin
    for i ← 1 to n do
        p[i] ← 0
        d[i] ← ∞
    d[i0] ← 0
    Q ← V
    while Q ≠ ∅ do
        i ← citeste(Q)
        elimina(Q)
        for fiecare j ∈ listaDeAdiacenta(i) do
            if (d[j] > d[i] + ℓ[i, j])
            then d[j] ← d[i] + ℓ[i, j]
                p[j] ← i
    end

```

Se cere:

- Să se exemplifice execuția algoritmului Dijkstra pentru digraful din figura 9.8.
- Notăm cu $\delta(i_0, i)$ lungimea drumului minim de la i_0 la i (când există) și cu $S(t)$ mulțimea vârfurilor i eliminate din Q până la momentul t . Să se arate că dacă $i \in S(t)$, atunci $d[i] = \delta(i_0, i)$.
- Să se arate că algoritmul Dijkstra determină corect drumurile minime care pleacă din i_0 (adică $d[i] = \delta(i_0, i)$ pentru orice i , după terminare).
- Să se determine complexitatea algoritmului Dijkstra.
- Să se arate că algoritmul Dijkstra este un algoritm greedy.

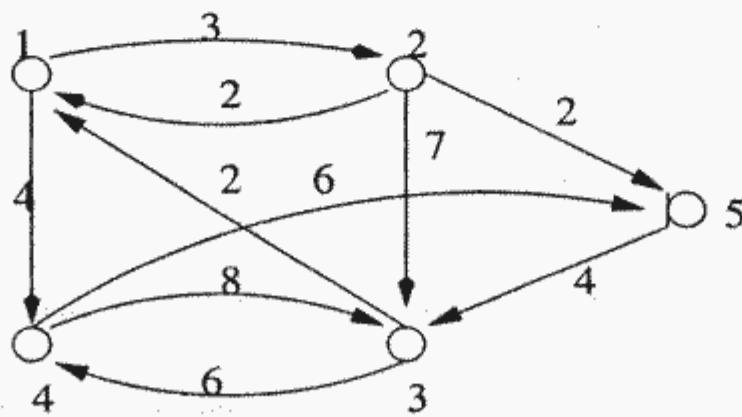


Figura 9.8: Exercițiul 9.5

9.12 Referințe bibliografice

Prezentarea din acest capitol se bazează pe [HS84, MS91, CLR93, CLR00, Sah98, Hei96, Sed88, LG86]. Pentru aspectele teoretice privind metoda greedy recomandăm [MS91, Cro92, CLR93].



Capitolul 10

Divide-et-impera

10.1 Prezentare generală

10.1.1 Modelul matematic

Paradigma *divide-et-impera* (în engleză *divide-and-conquer*) constă în divizarea problemei inițiale în două sau mai multe subprobleme de dimensiuni mai mici, apoi rezolvarea în aceeași manieră (recursivă) a subproblemelor și combinarea soluțiilor acestora pentru a obține soluția problemei inițiale. Divizarea unei probleme se face până când se obțin subprobleme de dimensiuni mici ce pot fi rezolvate prin tehnici elementare. Paradigma poate fi descrisă schematic astfel:

```
procedure divideEtImpera(P, n, S)
    if (n ≤ n0)
        then rezolvă subproblema P prin tehnici elementare
    else împarte P în P1, ..., Pa de dimensiuni n1, ..., na
        divideEtImpera(P1, n1, S1)
        ...
        divideEtImpera(Pa, na, Sa)
        combină S1, ..., Sa pentru a obține S
    end
```

Exemple tipice de aplicare a paradigmii *divide-et-impera* sunt algoritmii de parcursere a arborilor binari și algoritmul de căutare binară în mulțimi total ordonate. Deoarece descrierea strategiei are un caracter recursiv, aplicarea ei trebuie precedată de o generalizare de tipul problemă \mapsto subproblemă prin care dimensiunea problemei devine o variabilă liberă.

Vom presupune că dimensiunea n_i a subproblemei P_i satisfacă $n_i \leq \frac{n}{b}$, unde $b > 1$. În acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce asigură proprietatea de terminare a subprogramului recursiv. De asemenea, descrierea recursivă ne va permite utilizarea inducției pentru demonstrarea corectitudinii.

10.1.2 Evaluarea timpului de execuție

Presupunem că divizarea problemei în subprobleme și asamblarea soluțiilor necesită timpul $O(n^k)$. Timpul de execuție $T(n)$ al algoritmului **divideEtImpera** este dat de următoarea relație de recurență:

$$T(n) = \begin{cases} O(1) & , \text{dacă } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + O(n^k) & , \text{dacă } n > n_0 \end{cases} \quad (10.1)$$

Teorema 10.1. Dacă $n > n_0$ atunci:

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \text{dacă } a > b^k \\ O(n^k \log_b n) & , \text{dacă } a = b^k \\ O(n^k) & , \text{dacă } a < b^k \end{cases} \quad (10.2)$$

Demonstrație. Fără să restrângem generalitatea presupunem $n = b^m \cdot n_0$. De asemenea mai presupunem că $T(n) = cn_0^k$ dacă $n \leq n_0$ și $T(n) = aT\left(\frac{n}{b}\right) + cn^k$ dacă $n > n_0$. Pentru $n > n_0$ avem:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + cn^k \\ &= aT(b^{m-1}n_0) + cn^k \\ &= a(aT(b^{m-2}n_0) + c\left(\frac{n}{b}\right)^k) + cn^k \\ &= a^2T(b^{m-2}n_0) + c\left[a\left(\frac{n}{b}\right)^k + n^k\right] \\ &= \dots \\ &= a^mT(n_0) + c\left[a^{m-1}\left(\frac{n}{b^{m-1}}\right)^k + \dots + a\left(\frac{n}{b}\right)^k + n^k\right] \\ &= a^m cn_0^k + c\left[a^{m-1}b^k n_0^k + \dots + a(b^{m-1})^k n_0^k + (b^m)^k n_0^k\right] \\ &= cn_0^k a^m \left[1 + \frac{b^k}{a} + \dots + \left(\frac{b^k}{a}\right)^m\right] \\ &= ca^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i \end{aligned}$$

unde am renoscut cn_0^k prin c . Distingem cazurile:

1. $a > b^k$. Seria $\sum_{i=0}^m \left(\frac{b^k}{a}\right)^i$ este convergentă și deci sirul sumelor parțiale este convergent. De aici rezultă $T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$.
2. $a = b^k$. Rezultă $a^m = b^{km} = cn^k$ și de aici $T(n) = O(n^k m) = O(n^k \log_b n)$.
3. $a < b^k$. Avem $T(n) = O(a^m (\frac{b^k}{a})^m) = O(b^{km}) = O(n^k)$.

Acum teorema este demonstrată complet.

sfdem

10.2 Sortare prin interclasare (Merge Sort)

10.2.1 Modelul matematic

Considerăm cazul când lista liniară ce urmează să fie sortată este reprezentată ca tablou. Știm că prin interclasarea a două liste sortate obținem o listă sortată ce conține toate elementele listelor de intrare. Ideea este de a utiliza interclasarea în etapa de asamblare a soluțiilor: în urma rezolvării recursive a subproblemelor rezultă liste ordonate și prin interclasarea lor obținem lista inițială sortată. Primul pas constă în generalizarea problemei. Presupunem că se cere sortarea unui tablou $a[p..q]$ cu p și q variabile libere, în loc de sortarea tabloului $a[0..n-1]$ cu n variabilă legată (deoarece este dată de intrare). Divizarea problemei constă în împărțirea listei de sortat în două subliste $a[p..m]$ și $a[m+1..q]$, de preferat de lungimi aproximativ egale. Noi vom lua $m = \left\lfloor \frac{p+q}{2} \right\rfloor$. Așa cum am spus, faza de combinare a soluțiilor constă în interclasarea celor două subliste, după ce ele au fost sortate recursiv prin același procedeu. Pasul de bază este dat de faptul că sublistele de lungime 1 sunt ordonate prin definiție:

```

procedure mergeSort(a, p, q)
    if (p < q)
        then m ←  $\left\lfloor \frac{p+q}{2} \right\rfloor$ 
            mergeSort(a, p, m)
            mergeSort(a, m+1, q)
            interclasează subtablourile  $(a[p], \dots, a[m]), (a[m+1], \dots, a[q])$ 
            utilizând un tablou temporar
    end

```

Pasul de divizare a problemei în subprobleme se face în timpul $O(1)$. Pentru faza de asamblare a soluțiilor, reamintim că interclasarea a două secvențe ordonate crescător se face în timpul $O(m_1 + m_2)$, unde m_1 și m_2 sunt lungimile celor două secvențe. Aplicând teorema 10.1 pentru $a = 2, b = 2, k = 1$ rezultă că algoritmul `mergeSort` va efectua sortarea unui tablou de lungime n în timpul $O(n \log_2 n)$.

10.2.2 Implementare

Listele parțiale obținute în timpul interclasării vor fi memorate temporar într-o variabilă tablou auxiliară. După terminarea procesului de interclasare, lista rezultat va fi copiată în locul subsecvențelor de intrare. Astfel programul va utiliza $O(n + \log_2 n)$ memorie suplimentară (tabloul auxiliar și stiva cu apelurile recursive).

```

procedure intercl2(a, p, q, m, temp)
    i ← p
    j ← m+1
    k ← -1
    while ((i ≤ m) and (j ≤ q)) do
        k ← k+1
        if (a[i] ≤ a[j])
            then temp[k] ← a[i]

```

```

        i ← i+1
    else temp[k] ← a[j]
        j ← j+1
    while (i ≤ m) do
        k ← k+1
        temp[k] ← a[i]
        i ← i+1
    while (j ≤ n) do
        k ← k+1
        temp[k] ← a[j]
        j ← j+1
end
procedure mergeSort(a, p, q)
    if (p < q)
        then m ←  $\left\lfloor \frac{p+q}{2} \right\rfloor$ 
            mergeSort(a, p, m)
            mergeSort(a, m+1, q)
            intercl2(a, p, q, m, temp)
            for i ← p to q do
                a[i] ← temp[i-p]
end

```

Exercițiul 10.2.1. Să se rescrie `mergeSort` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlăncuită. Să se compare eficiența nouului algoritm cu cel corespunzător reprezentării cu tablouri.

10.2.2.1 Variantă nerecursivă

În continuare vom căuta să găsim o variantă nerecursivă pentru `mergeSort`. Pentru aceasta să observăm că arborele subproblemelor generat de metodă (echivalent, arborele apelurilor recursive) este un arbore binar în care vârfurile de pe frontieră corespund subsecvențelor de lungime 1. Varianta nerecursivă va proceda la parcurgerea acestui arbore în maniera “bottom-up”, i.e., parcurge nivel cu nivel de la frontieră spre rădăcină: întâi sunt vizitate toate vârfurile de pe nivelul cel mai mare (cel imediat superior frontierei), apoi după vizitarea vârfurilor de pe nivelul h se trece la vizitarea vârfurilor de pe nivelul $h - 1$. Vizitarea unui vârf al arborelui constă de fapt în interclasarea a două subtablouri. Deoarece adresa unui vârf poate fi calculată printr-o relație simplă, nu este nevoie de gestionarea unei cozi care să memoreze adresele vârfurilor ce urmează a fi vizitate. De remarcat că nivelul vizitat curent în arbore dă și lungimea subsecvențelor sortate deja. Frontieră corespunde subsecvențelor de lungime 1, nivelul imediat superior subsecvențelor de lungime 2, următorul subsecvențelor de lungime 4 și aşa mai departe (figura 10.1).

Pentru a lucra corect, utilizăm un tablou temporar `temp` cu următorul scop: dacă se interclasează subsecvențe din secvența `a`, atunci rezultatul interclasării va fi memorat în `temp`, iar dacă se interclasează subsecvențe din `temp`, atunci rezultatul

va fi memorat în **a**. În acest fel, listele intermediare (sortate parțial) vor fi memorate alternativ în cele două tablouri **a** și **temp**.

Faza de interclasare a subsecvențelor de pe un nivel este realizată în modul următor: se interclasează primele două subsecvențe, apoi se interclasează a treia subsecvență cu a patra și a.m.d. S-a preferat o descriere mai condensată a metodei de interclasare.

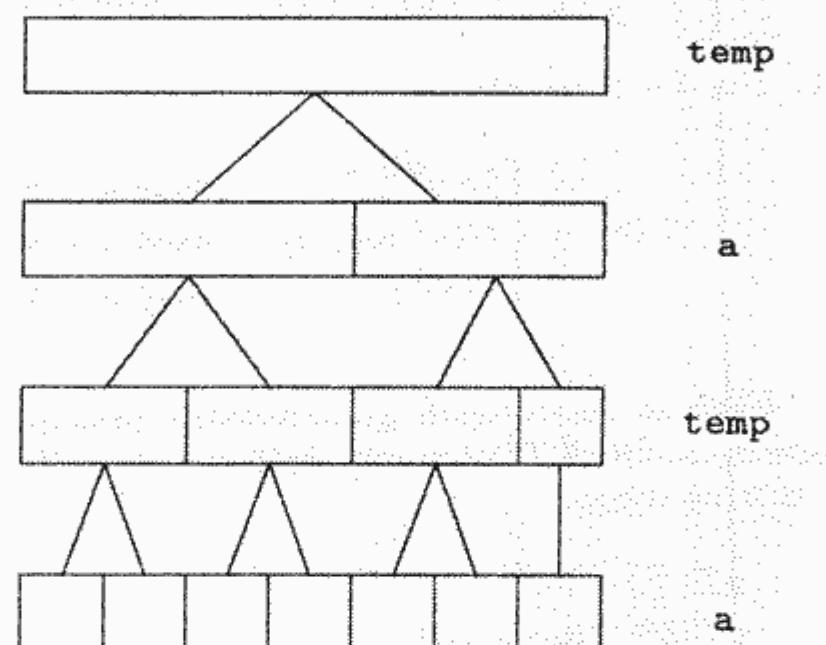


Figura 10.1: Merge Sort nerecursiv

```

procedure intercl(secvin, secvout, n, ls)
    i ← 0
    j ← ls
    k ← 0
    repeat
        t ← j
        u ← j+ls
        if (u > n) then u ← n
        while ((i < t) and (j < u)) do
            if (secvin[i] ≤ secvin[j])
                then secvout[k] ← secvin[i]
                    i ← i+1
                    k ← k+1
            else secvout[k] ← secvin[j]
                    j ← j+1
                    k ← k+1
        while (i < t) do
            secvout[k] ← secvin[i]
            i ← i+1
            k ← k+1
        while (j < u) do
    
```

```

secvout[k] ← secvin[j]
j ← j+1
k ← k+1
i ← u
j ← i+ls
until (j > n-1)
for j ← i to n-1 do
    secvout[j] ← secvin[j]
end

procedure mergeSortNerec(a, p, q)
    ina ← true
    ls ← 1
    while (ls < n) do
        if (ina)
            then intercl(a,temp,n,ls)
            else intercl(temp,a,n,ls)
        ina ← not ina
        ls ← ls*2
        if (not ina)
            then for i ← 0 to n-1 do
                a[i] ← temp[i]
    end

```

Exercițiul 10.2.2. Memorarea alternativă a listelor intermediare în cele două tablouri a și b reduce considerabil numărul transferurilor. Se poate aplica aceeași tehnică și în cazul subprogramului recursiv `mergeSort?`? Dacă răspunsul este da, să se arate cum, iar dacă este nu, să se dea o justificare a acestuia.

Exercițiul 10.2.3. Să se rescrie `mergeSortNerec` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înălțuită. Să se evidențieze avantaje/dezavantaje ale utilizării acestei reprezentări.

10.3 Sortarea rapidă (Quick Sort)

10.3.1 Modelul matematic

Ca și în cazul algoritmului *Merge Sort*, vom presupune că trebuie sortat tabloul $a[p..q]$. Divizarea problemei constă în alegerea unei valori x din $a[p..q]$ și determinarea prin interschimbări a unui indice k cu proprietățile:

- $p \leq k \leq q$ și $a[k] = x$;
- $\forall i : p \leq i \leq k \Rightarrow a[i] \leq a[k]$;
- $\forall j : k < j \leq q \Rightarrow a[k] \leq a[j]$;

Elementul x este numit *pivot*. În general, se alege pivotul $x = a[p]$, dar nu este obligatoriu. Partiționarea tabloului se face prin interschimbări care mențin invariante proprietăți asemănătoare cu cele de mai sus. Se consideră două variabile index: i cu care se parcurge tabloul de la stânga la dreapta și j cu care se parcurge

tabloul de la dreapta la stânga. Inițial se ia $i = p + 1$ și $j = q$. Proprietățile menținute invariante în timpul procesului de partitioare sunt:

$$\forall i' : p \leq i' < i \Rightarrow a[i'] \leq x \quad (10.3)$$

și

$$\forall j' : j < j' \leq q \Rightarrow a[j'] \geq x \quad (10.4)$$

Presupunem că la momentul curent sunt interogate elementele $a[i]$ și $a[j]$ cu $i < j$.

Distingem următoarele cazuri:

1. $a[i] \leq x$. Transformarea $i \leftarrow i + 1$ păstrează 10.3.
2. $a[j] \geq x$. Transformarea $j \leftarrow j - 1$ păstrează 10.4.
3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea $a[i] \leftrightarrow a[j]$ și se face $i \leftarrow i + 1$ și $j \leftarrow j - 1$, atunci ambele predicate 10.3 și 10.4 sunt păstrate.

Operațiile de mai sus sunt repetate până când i devine mai mare decât j . Considerând $k = i - 1$ și interschimbând $a[p]$ cu $a[k]$ obținem partitioarea dorită a tabloului.

După sortarea recursivă a subtablourilor $a[p..k - 1]$ și $a[k + 1..q]$ se observă că tabloul este sortat deja. Astfel partea de asamblare a soluțiilor este vidă.

Algoritmul rezultat este:

```

procedure quickSort(a, p, q)
  if (p < q)
    then determină prin interschimbări indicele k cu:
      p ≤ k ≤ q
      (∀)i : p ≤ i ≤ k ⇒ a[i] ≤ a[k]
      (∀)j : k < j ≤ q ⇒ a[k] ≥ a[j]
      quickSort(a, p, m)
      quickSort(a, m+1, q)
  end

```

Cazul cel mai nefavorabil se obține atunci când la fiecare partitioare se obține una din subprobleme cu dimensiunea 1. Deoarece operația de partitioare necesită $O(q - p)$ comparații, rezultă că pentru acest caz numărul de comparații este $O(n^2)$. Acest rezultat este oarecum surprinzător, având în vedere că numele metodei este „sortare rapidă”. Așa cum vom vedea, într-o distribuție normală, cazurile pentru care QuickSort execută n^2 comparații sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului. În continuare determinăm numărul mediu de comparații. Presupunem că $q + 1 - p = n$ (lungimea secvenței) și că probabilitatea ca pivotul x să fie al k -lea element este $\frac{1}{n}$ (fiecare element al tabloului poate fi pivot cu aceeași probabilitate $\frac{1}{n}$). Rezultă că probabilitatea obținerii subproblemelor de dimensiuni $k - p = i - 1$ și $q - k = n - i$ este $\frac{1}{n}$. În procesul de partitioare, un element al tabloului (pivotul) este comparat cu toate celelalte, astfel că sunt necesare $n - 1$ comparații. Acum numărul mediu de comparații se calculează prin formula:

$$T^{med}(n) = \begin{cases} (n - 1) + \frac{1}{n} \sum_{i=1}^n (T^{med}(i - 1) + T^{med}(n - i)) & , \text{dacă } n \geq 1 \\ 1 & , \text{dacă } n = 0 \end{cases}$$

Rezolvăm această recurență. Avem:

$$\begin{aligned} T^{med}(n) &= (n-1) + \frac{2}{n}(T^{med}(0) + \cdots + T^{med}(n-1)) \\ nT^{med}(n) &= n(n-1) + 2(T^{med}(0) + \cdots + T^{med}(n-1)) \end{aligned}$$

Trecem pe n în $n-1$:

$$(n-1)T^{med}(n-1) = (n-1)(n-2) + 2(T^{med}(0) + \cdots + T^{med}(n-2))$$

Scădem:

$$nT^{med}(n) = 2(n-1) + (n+1)T^{med}(n-1)$$

Împărțim prin $n(n+1)$ și rezolvăm recurența obținută:

$$\begin{aligned} \frac{T^{med}(n)}{n+1} &= \frac{T^{med}(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\ &= \frac{T^{med}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)} \right) \\ &= \cdots \\ &= \frac{T^{med}(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1 \cdot 2} + \cdots + \frac{2}{n(n+1)} \right) \\ &= 1 + 2 \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1} \right) - 2 \left(\frac{1}{1 \cdot 2} + \cdots + \frac{1}{n(n+1)} \right) \end{aligned}$$

Deoarece $1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log_2 n)$ și seria $\sum \frac{1}{k(k+1)}$ este convergentă (și deci sirul sumelor parțiale este mărginit), rezultă că $T(n) = O(n \log_2 n)$.

Am demonstrat următorul rezultat:

Teorema 10.2. Complexitatea medie a algoritmului QuickSort este $O(n \log_2 n)$.

10.3.2 Implementare

Subprogramul **partitioneaza** descrie algoritmul de divizare de mai sus:

```

procedure partitioneaza(a, p, q, k)
    x ← a[p]
    i ← p + 1
    j ← q
    while (i ≤ j) do
        if (a[i] ≤ x) then i ← i + 1
        if (a[j] ≥ x) then j ← j - 1
        if (i < j)
            then if ((a[i] > x) and (x > a[j]))
                then swap(a[i], a[j])
            i ← i + 1
            j ← j - 1
    k ← i-1
    a[p] ← a[k]
    a[k] ← x
end

```

Dacă se presupune că are loc $a[p - 1] \leq a[i] \leq a[q + 1]$, pentru orice i cu $p \leq i \leq q$ (aceasta implică existența inițială a două elemente artificiale $a[-1]$ și $a[n]$ cu $a[-1] \leq a[i] \leq a[n]$, $0 \leq i \leq n - 1$), atunci procedura de partiționare poate fi descrisă mai simplu astfel:

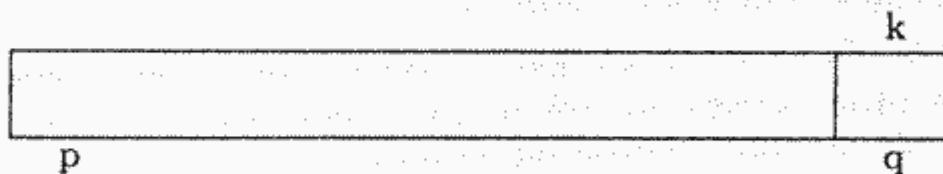
```

procedure partioneaza(a, p, q, k)
    x ← a[p]
    i ← p - 1
    j ← q + 1
    while (i < j) do
        repeat i ← i + 1 until (a[i] ≥ x)
        repeat j ← j - 1 until (a[j] ≤ x)
        if (i < j)
            then swap(a[i], a[j])
    k ← j
    a[p] ← a[k]
    a[k] ← x
end

```

10.3.3 Spațiul ocupat de algoritmul QuickSort

La execuția algoritmilor recursivi, importantă este și spațiul de memorie ocupat de stivă (a se vedea secțiunea 3). Considerăm spațiul de memorie ocupat de stivă în cazul cel mai defavorabil, $k = q$:



În acest caz, spațiul de memorie ocupat de stivă este $M(n) = c + M(n - 1)$, care implică $M(n) = O(n)$.

În general, pivotul împarte secvența de sortat în două subsecvențe. Dacă subsecvența mică este rezolvată recursiv, iar subsecvența mare este rezolvată iterativ, atunci consumul de memorie se reduce. Noul algoritm are următoarea descriere:

```

procedure QuickSort(a, p, q)
    while (p < q) do
        partioneaza(a, p, q, k)
        if (k-p > q-k)
            then QuickSort(a, k+1, q)
                q ← k-1
        else QuickSort(a, p, k-1)
                p ← k+1
    end

```

Spațiul de memorie ocupat de stivă pentru noul algoritm satisfacă relația $M(n) \leq c + M(n/2)$, de unde rezultă $M(n) = O(\log n)$.

Exercițiul 10.3.1. Dacă tabloul a conține multe elemente egale, atunci algoritmul **partitioneaza** realizează multe interschimbări inutile (de elemente egale). Să se modifice algoritmul astfel încât noua versiune să eliminate acest inconvenient.

Exercițiul 10.3.2. Să se modifice subprogramul de partitiorare de mai sus prin înlocuirea instrucțiunii **while** cu **repeat** și a celor două instrucțiuni **repeat** cu **while**. Atenție: partea de inițializare și partea de interschimbare vor fi modificate corespunzător pentru ca noul subprogram să rezolve corect problema partitiorii.

Exercițiul 10.3.3. Următorul program poate fi de asemenea utilizat cu succes pentru partitiorarea tabloului $a[p..q]$:

```
procedure partitioneaza(a, p, q, k)
    i ← p
    j ← q
    iinc ← 0
    jinc ← -1
    while (i < j) do
        if (a[i] > a[j])
            then swap(a[i], a[j])
            incaux ← iinc
            iinc ← -jinc
            jinc ← -incaux
        i ← i+iinc
        j ← j+jinc
    k ← i
end
```

Să se demonstreze că programul face o partitiorare corectă. Ce proprietăți invariante păstrează instrucțiunea **while**?

Acum, programul care descrie algoritmul QuickSort este următorul:

```
procedure quickSort(a, p, q)
    if (p < q)
        then partitioneaza(a, p, q)
            quickSort(a, p, k)
            quickSort(a, k+1, q)
    end
```

10.4 Selecționare

Următoarea problemă este o variantă a celei din exercițiul 4.1.8:

Să se scrie un algoritm care având la intrare un tablou ($a[i] \mid 0 \leq i < n$) și un număr $k \in \{0, 1, \dots, n-1\}$, schimbă ordinea elementelor din a astfel încât la ieșire pe poziția k se află cel de-al $k+1$ -lea număr cel mai mic (echivalent, cel de-al $n-1-k$ -lea cel mai mare), elementele aflate la stânga lui k sunt mai mici decât $a[k]$ și elementele aflate la dreapta lui k sunt mai mari decât $a[k]$.

Evident, orice algoritm de sortare rezolvă problema de mai sus. Deoarece cerințele pentru selecție sunt mai slabe decât cele de la ordonare, se pune firesc întrebarea dacă există algoritmi mai performanți decât cei utilizați la sortare. Condiția pe care trebuie să o satisfacă la ieșire tabloul a este formulată de:

$$(\forall i)(i < k \Rightarrow a[i] \leq a[k]) \wedge (i > k \Rightarrow a[i] \geq a[k])$$

Fie j o poziție cu proprietatea:

$$(\forall i)(i < j \Rightarrow a[i] \leq a[j]) \wedge (i > j \Rightarrow a[i] \geq a[j])$$

Un asemenea j poate fi obținut cu algoritmul de partitioare de la quickSort. Dacă $j = k$, atunci problema este rezolvată. Dacă $j < k$, atunci cel de-al k -lea cel mai mic element trebuie căutat în subtabloul $a[j+1..n]$, iar dacă $j > k$ atunci cel de-al k -lea cel mai mic element trebuie căutat în subtabloul $a[1..j-1]$. Aceasta conduce la următoarea formulare recursivă a algoritmului de selectare:

```
function selecteaza(a, p, q, k)
    partitioneaza(a, p, q, k1)
    if (k1 = k)
        then return a[k]
    else if (k1 < k)
        then selecteaza(a, k1+1, q, k)
    else selecteaza(a, p, k1-1, k)
end
```

Descrierea recursivă nu este avantajoasă, deoarece produce un consum de memorie suplimentară (stiva apelurilor recursive) ce poate fi eliminat prin derecursivare:

```
function selecteaza(a, n, k)
    p ← 0
    q ← n-1
    repeat
        partitioneaza(a, p, q, k1)
        if (k1 ≠ k)
            then if (k1 < k)
                then p ← k1+1
            else q ← k1-1
    until (k1 = k)
    return a[k]
end
```

Analiza algoritmului **selecteaza** este asemănătoare cu cea a algoritmului **quickSort**. În cazul cel mai nefavorabil necesită $O(n^2)$ timp, iar pentru complexitatea medie se cunoaște următorul rezultat:

Teorema 10.3 ([Sed88]) *Complexitatea timp medie a algoritmului **selecteaza** este $O(n)$. Mai precis, numărul de comparații este aproximativ:*

$$n + k \log\left(\frac{n}{k}\right) + (n - k) \log\left(\frac{n}{n - k}\right).$$

10.5 Linia orizontului

10.5.1 Descrierea problemei

Pe o suprafață orizontală se consideră n paralelipipede (se poate presupune că sunt clădiri), așezate astfel încât toate paralelipipedele au o suprafață laterală paralelă cu o aceeași dreaptă. În acest fel putem considera că toate paralelipipedele au față spre est. O sursă de lumină, aflată la o distanță destul de mare astfel încât razele de lumină sunt paralele și cad perpendicular pe fețele paralelipipedelor (soarele puțin timp după ce a răsărit), proiectează paralelipipedele pe un ecran (orizontul), presupus a fi suficient de mare. Se cere să se determine conturul superior al umbrei (linia orizontului).

10.5.2 Modelul matematic

În primul rând să observăm că problema poate fi redusă de la o problemă în spațiul tridimensional la o problemă în plan. Proiecția unui paralelipiped pe ecran este un dreptunghi cu baza pe axa Ox . Un astfel de dreptunghi poate fi specificat prin trei elemente: cele două proiecții $x_{stg} < x_{drp}$ ale vârfurilor pe axa Ox și înălțimea y . Deoarece un dreptunghi se poate afla parțial sau total în spatele altui dreptunghi, rezultă că proiecțiile se pot intersecta. În acest fel, linia orizontului va fi înfășurătoarea superioară a unei colecții de dreptunghiuri ce au bazele pe axa Ox . Pentru ca domeniul problemei să fie complet specificat, mai trebuie să vedem ce înseamnă linia orizontului. Pentru aceasta, vom răspunde mai întâi la întrebarea: din ce se compune linia orizontului? Considerăm colecția de dreptunghiuri din figura 10.2a. Linia orizontului corespunzătoare acestei colecții este reprezentată în figura 10.2b. Observăm că ea se compune din:

- două semidrepte incluse în axa Ox , care sunt extremitățile;
- segmente orizontale și verticale care sunt laturi, porțiuni de laturi sau porțiuni din axa Ox ;
- în plus linia orizontului are proprietatea că orice dreptunghi se află între Ox și linia orizontului.

Proiecțiile capetelor segmentelor orizontale pe axa Ox determină o secvență de puncte, pe care le notăm de la stânga la dreapta prin $x_1, x_2, \dots, x_k, x_{k+1}$ (s-a presupus că linia este formată de k segmente orizontale). Fiecare segment orizontal i are o înălțime y_i , $1 \leq i \leq k$. Dacă vom considera în plus $x_0 = -\infty$, $x_{k+2} = +\infty$, $y_0 = y_{k+1} = 0$, rezultă că secvența $x_0, y_0, x_1, y_1, \dots, x_k, y_k, x_{k+1}, y_{k+1}, x_{k+2}$ reprezintă în mod unic linia orizontului. Pentru exemplul nostru, linia este reprezentată de secvența $(-\infty, x_1, y_1, x_2, 0, x_3, y_3, \dots, x_8, 0, +\infty)$ (figura 10.3).

În general, linia orizontului poate fi reprezentată printr-o listă liniară de $2k + 1$ numere reale $(x_1, y_1, \dots, x_k, y_k, x_{k+1})$, cu $k \leq n$. Un dreptunghi poate fi considerat un caz particular de linie a orizontului ($k = 1$) și deci poate fi reprezentat de lista liniară (x_{stg}, y, x_{drp}) .

O colecție de dreptunghiuri va fi reprezentată de o variabilă tablou d , unde $d[i]$ este o structură cu câmpurile $d[i].x_{stg}$, $d[i].x_{drp}$, $d[i].y$, precum și de o variabilă de tip întreg n .

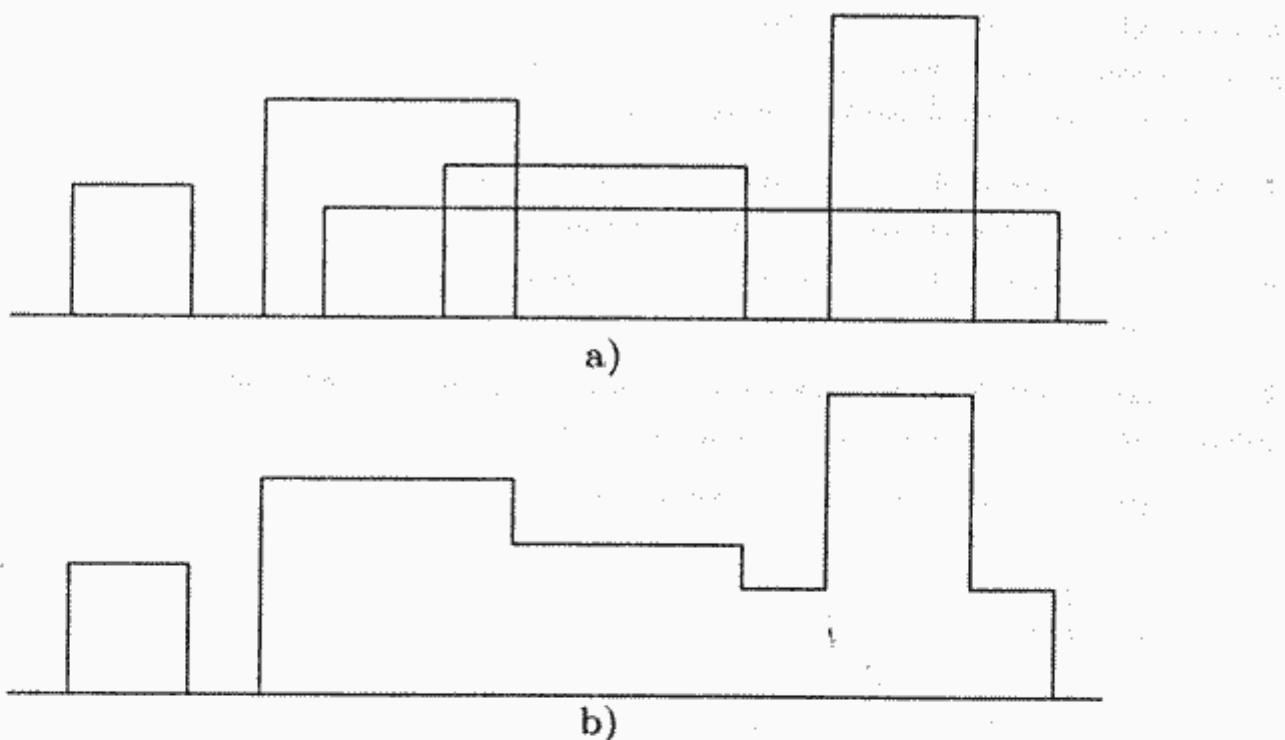


Figura 10.2: Linia orizontului

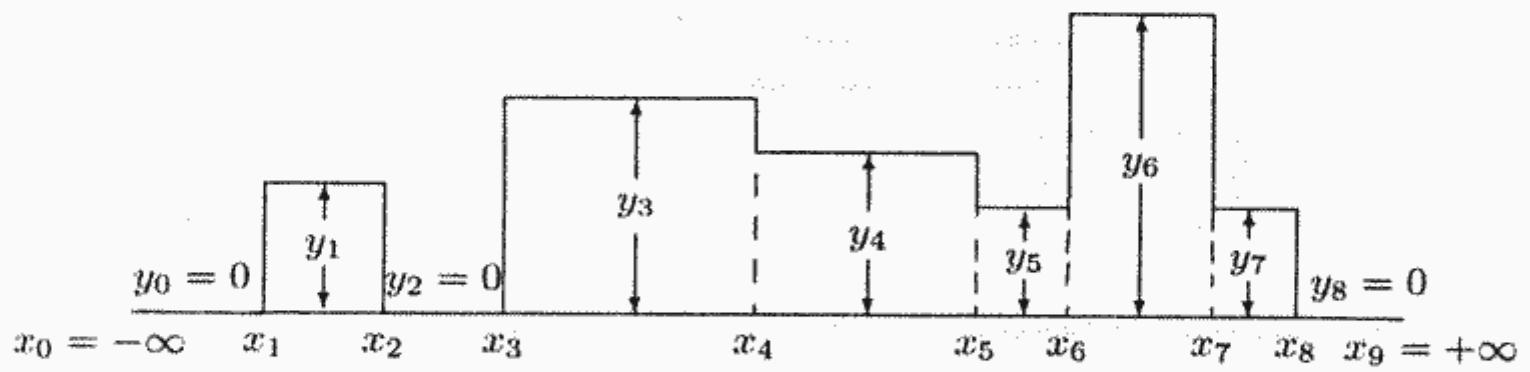


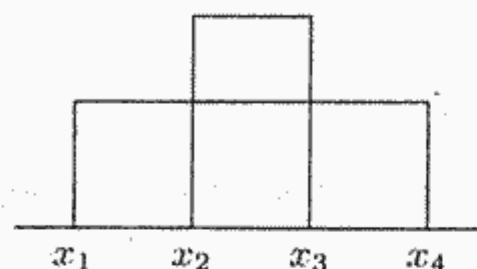
Figura 10.3: Reprezentarea liniei orizontului

Vom rezolva problema determinării liniei orizontului prin strategia *divide-et-impera*. Generalizăm presupunând date dreptunghiurile $(d[p], \dots, d[q])$ în loc de $(d[1], \dots, d[n])$. Există două moduri de a diviza problema:

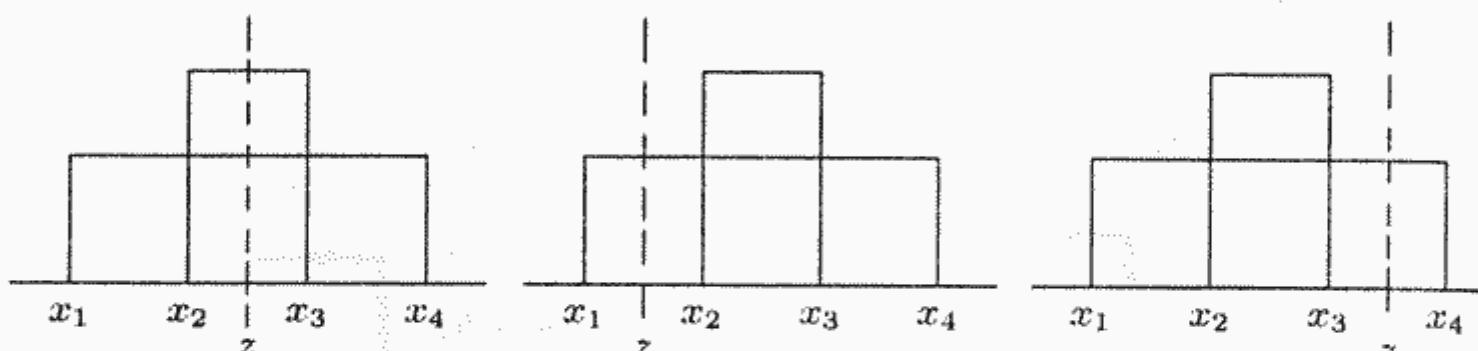
1. *Divizarea verticală*. Se consideră un z cu $d[p].xstg \leq z \leq d[p].xdrp$. O subproblemă va fi formată din colecția de dreptunghiuri aflate la stânga lui z și cealaltă subproblemă din colecția de dreptunghiuri aflate la dreapta lui z . Dreptunghiurile $d[i]$ cu $d[i].stg < z < d[i].drp$ (o latură verticală la stânga lui z și cealaltă la dreapta lui z) vor fi divizate în două dreptunghiuri prin considerarea unei laturi verticale duse prin z . Această strategie are avantajul că partea de asamblare este simplă (constă din concatenarea celor două linii ale orizontului obținute prin rezolvarea subproblemelor), dar are și mărele dezavantaj ca sunt cazuri când pasul de divizare nu conduce la micșorarea numărului de dreptunghiuri pentru subprobleme. Ca exemplu, considerăm următoarea colecția de două dreptunghiuri din figura 10.4a. Indiferent cum ar fi ales z între x_1 și x_4 (figura 10.4b), cel puțin una din subprobleme are

două sau mai multe dreptunghiuri (atenție, procesul de divizare poate conduce chiar la mărirea numărului de dreptunghiuri într-o subproblemă).

De remarcat totuși că o ordonare lexicografică conduce la o simplificare a pasului de divizare.



a)



b)

Figura 10.4: Un exemplu când divizarea verticală nu reduce numărul de dreptunghiuri

2. *Divizarea orizontală*. Pasul de divizare este foarte simplu: prima subproblemă este formată din primele $\left[\frac{n}{2}\right]$ dreptunghiuri, iar a două subproblemă din celelalte $n - \left[\frac{n}{2}\right]$ dreptunghiuri. Nu mai este necesară ordonarea dreptunghiurilor.

Acum este suficient să se determine $m = \left[\frac{p+q}{2}\right]$ și prima subproblemă va conține colecția (sublista) $d[p], \dots, d[m]$, iar a doua subproblemă va conține colecția (sublista) $d[m+1], \dots, d[q]$. Partea de asamblare este un pic mai dificilă pentru că necesită interclasarea celor două linii obținute prin rezolvarea subproblemelor.

Divizarea orizontală pare mai potrivită, motiv pentru care o alegem pentru determinarea liniei orizontului. Mai întâi considerăm un exemplu. Fie colecția din figura 10.5. Prin divizare se obțin subproblemele reprezentate în figura 10.6. Rezolvarea subproblemelor conduce la soluțiile din figura 10.7 care interclasate dau linia reprezentată în figura 10.8.

În continuare vom descrie modul în care se realizează interclasarea. Presupunem că prin rezolvarea subproblemelor s-au obținut liniile L1 : TLinieOrizont cu valoarea

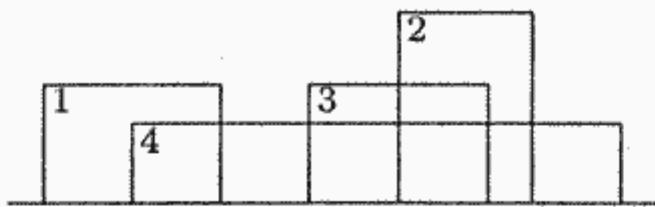


Figura 10.5: Problema inițială



Figura 10.6: Divizarea în subprobleme

$(x_1, y_1, \dots, x_{m1}, y_{m1}, x_{m1+1})$, respectiv $L2: TLinieOrizont$ cu valoarea $(x'_1, y'_1, \dots, x'_{m2}, y'_{m2}, x'_{m2+1})$. Procesul de interclasare are loc comparând două câte două componente orizontale ale celor două linii și alegându-o de fiecare dată pe cea cu înălțimea mai mare. Notăm că termenul „interclasare” este utilizat corect: cele două linii $L1$ și $L2$ sunt liste ordonate de segmente orizontale, iar rezultatul este tot o listă ordonată de segmente orizontale. Lungimea curentă a liniei rezultat este memorată de variabila m . Presupunem că la momentul curent se compară $L1[i] = x_{[\frac{i}{2}]+1}$ și $L2[j] = x'_{[\frac{j}{2}]+1}$. Pentru a simplifica scrierea, notăm $\left[\frac{i}{2}\right] + 1$ cu i_1 și $\left[\frac{j}{2}\right] + 1$ cu j_1 . De asemenea, notăm cu h_1 înălțimea curentă din $L1$, cu h_2 înălțimea curentă din $L2$, cu h înălțimea curentă din linia rezultat L și cu $h1, h2$, respectiv h variabilele care memorează aceste înălțimi. O variabilă k va indica acea linie care contribuie în pasul curent la linia rezultat: $k = 1$ indică faptul că linia $L1$ contribuie la formarea liniei L și $k = 2$ indică faptul că linia $L2$ contribuie la formarea liniei L . Distingem următoarele situații:

1. $x_{i_1} < x'_{j_1}$. În continuare, h_1 va deveni egală cu $L1[i+1]$. Dacă $i \geq 2 * m1 + 1$, atunci h_1 va deveni zero. Valoarea variabilei j rămâne neschimbată. Se disting subcazurile:

- a) $h_1 < h_2$. Dacă $k = 1$ (figura 10.9a), atunci $h > h_2$ și în continuare vom avea:

$k \leftarrow 2$

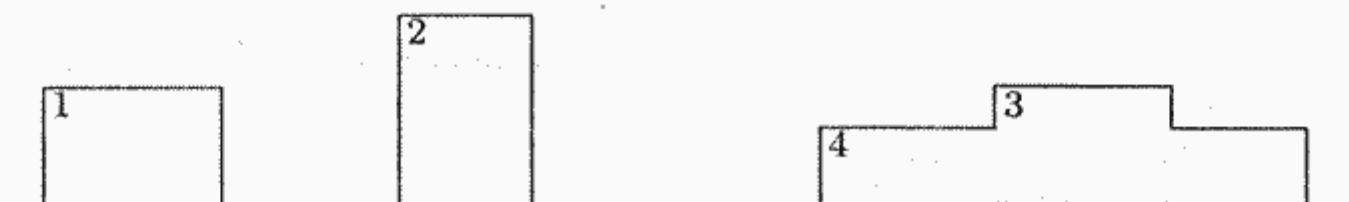


Figura 10.7: Rezolvarea subproblemelor

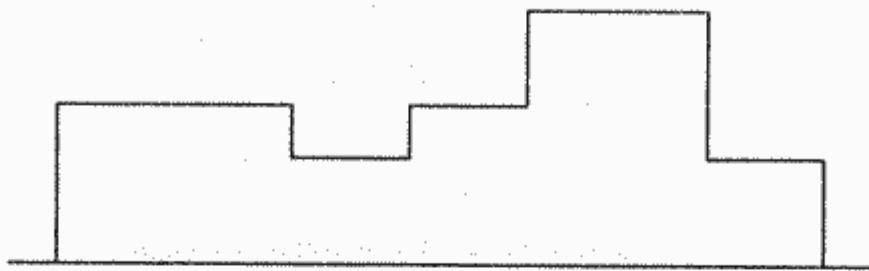


Figura 10.8: Asamblarea soluțiilor

```

L[m+1] ← L1[i]
L[m+2] ← h2
h ← h2
m ← m+2
i ← i+2
    
```

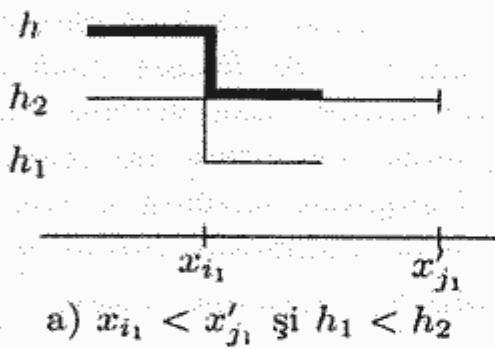
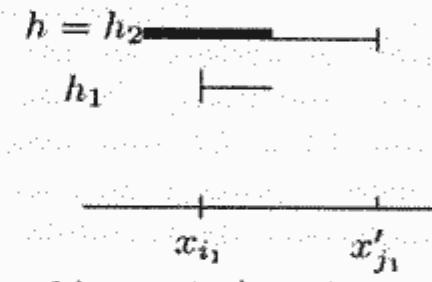
a) $x_{i_1} < x'_{j_1}$ și $h_1 < h_2$ b) $x_{i_1}' < x'_{j_1}$ și $h = h_2$

Figura 10.9: Cazul 1

Dacă $k = 2$ (figura 10.9b), atunci $h = h_2$ și în continuare k, m și h rămân neschimbate. Singura modificare care este $i \leftarrow i + 2$.

- b) $h_1 = h_2$. Dacă $k = 1$ (figura 10.10a), atunci $h > h_2$ și în continuare vom avea:

```

L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2
    
```

iar k rămâne neschimbată. Dacă $k = 2$ (figura 10.10b), atunci $h = h_2$ și în continuare k, m, h rămân neschimbate. Singura modificare este $i \leftarrow i + 2$.

- c) $h_1 > h_2$. Dacă $k = 1$ (figura 10.11a), atunci $h > h_2$ și în continuare vom avea:

```

L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2
    
```

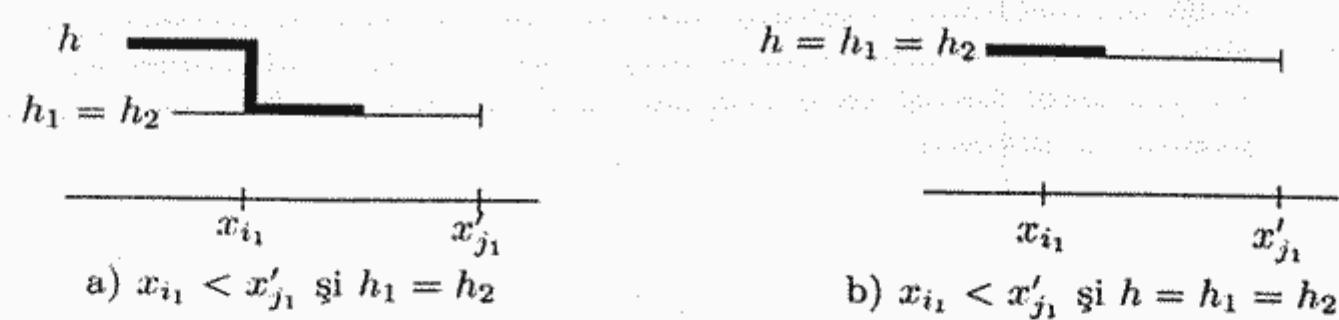


Figura 10.10: Cazul 1

iar k rămâne neschimbată. Dacă $k = 2$, atunci $h = h_2$ (figura 10.11b) și vom avea:

```

k ← 1
L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2

```

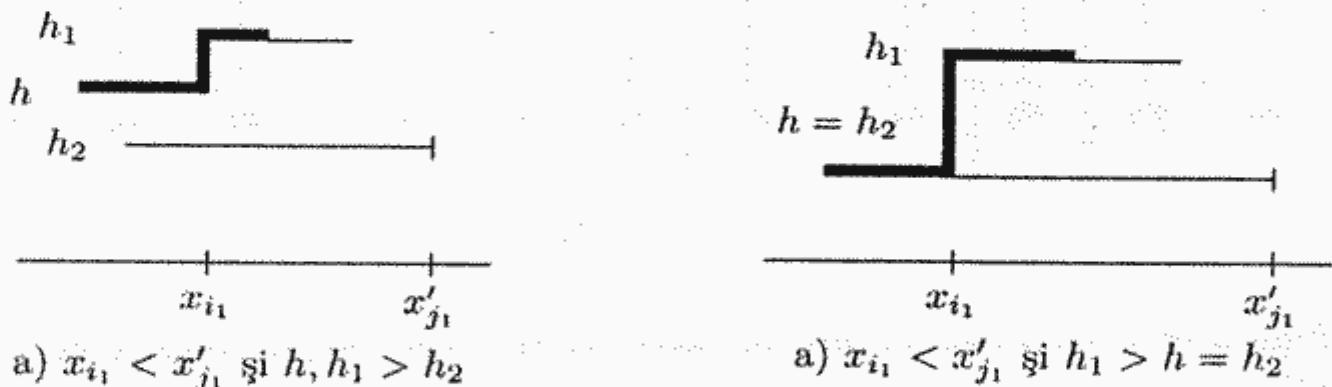


Figura 10.11: Cazul 1

Cazurile $x_{i_1} = x'_{j_1}$ și $x_{i_1} > x'_{j_1}$ se analizează la fel. Considerăm drept cazuri de bază următoarele două:

- Colecția nu are dreptunghiuri ($n = 0$). În acest caz, linia orizontală coincide cu axa Ox .
- Colecția are un singur dreptunghi ($n = 1$). Linia orizontului coincide cu reprezentarea dreptunghiului.

10.5.3 Implementare

Programul complet care determină linia orizontului se obține prin asamblarea secvențelor de mai sus. De notat că anumite variabile pot fi eliminate. De exemplu, se poate ține cont de faptul că $h1 = L1[i - 1]$, $h2 = L2[j - 1]$, $h = L[m]$. Considerând elementele suplimentare $L1[0] = L2[0] = L[0] = 0$, se poate renunța la utilizarea variabilelor $h1$, $h2$ și h .

APLICÂND TEOREMA 10.1 pentru cazul $a = b = 2$ și $k = 1$ obținem că algoritmul de determinare a liniei orizontului are complexitatea timp $O(n \log n)$.

Exercițiul 10.5.1. Descrierea algoritmului cu ajutorul unui subprogram recursiv are dezavantajul că utilizează două tablouri locale pentru memorarea liniilor rezultate din rezolvarea subproblemelor. Salvarea acestor tablouri în stiva sistemului reduce semnificativ performanțele programului. Să se eliminate acest inconvenient printr-un procedeu asemănător cu cel de la sortare prin interclasare.

10.6 Transformata Fourier discretă

10.6.1 Modelul matematic

Un semnal poate fi văzut din două puncte de vedere diferite:

1. domeniul timp;
2. domeniul frecvență.

Notăm cu $f(t)$ funcția care descrie semnalul în domeniul timp și cu $F(\nu)$ funcția care descrie semnalul în domeniul frecvenței. Se poate trece dintr-o descriere în alta prin *transformata Fourier*. Trecerea din domeniul timp în domeniul frecvență se face prin *transformata Fourier directă*:

$$F(\nu) = \mathcal{F}[f(t)] = \int_{-\infty}^{+\infty} f(t) e^{-2\pi i \nu t} dt, \quad (10.5)$$

iar trecerea de la domeniul frecvență la domeniul timp se face prin *transformata Fourier inversă*:

$$f(t) = \mathcal{F}^{-1}[F(\nu)] = \int_{-\infty}^{+\infty} F(\nu) e^{2\pi i \nu t} d\nu. \quad (10.6)$$

Transformata Fourier discretă se obține când se consideră $f(t)$ măsurată într-un număr finit de puncte: t_0, \dots, t_{n-1} cu $t_j = jT$, unde T este perioada de timp la care se fac măsurările. Notând $x_k = f(t_k)$, pentru $k = 0, \dots, n-1$, *transformata Fourier discretă directă* este dată de relația:

$$\mathcal{F}[f(t)] = \sum_{k=0}^{n-1} x_k e^{-2\pi i \frac{j}{n} k T} T = T \sum_{k=0}^{n-1} x_k e^{-2\pi i j k}. \quad (10.7)$$

Notăm:

$$W_j = \sum_{k=0}^{n-1} x_k e^{-2\pi i j k}. \quad (10.8)$$

Transformata Fourier discretă inversă este dată de relația:

$$x_k = \sum_{j=0}^{n-1} W_j T e^{\frac{2\pi i j k}{n}} \frac{1}{n T} = \frac{1}{n} \sum_{j=0}^{n-1} W_j e^{\frac{2\pi i j k}{n}} \quad (10.9)$$

Dacă se consideră polinomul de variabilă Y :

$$f(Y) = x_0 + x_1 Y + \cdots + x_{n-1} Y^{n-1} \quad (10.10)$$

și rădăcinile de ordinul n ale unității:

$$w_j = e^{\frac{2\pi i j}{n}}, j = 0, 1, \dots, n-1,$$

obținem

$$W_{n-j} = f(w_j), j = 0, 1, \dots, n-1, \quad (10.11)$$

unde am considerat $W_n = W_0$.

Rădăcinile de ordinul n ale unității satisfac următoarele proprietăți:

Lema 10.1. Dacă n este par, $n = 2m$, și $j < m$, atunci $-w_j = w_{j+m}$.

Demonstrație. Avem $w_{j+m}^2 = (e^{\frac{2\pi i(j+m)}{2m}})^2 = (e^{\frac{2\pi i j}{2m}})^2 (e^{\frac{2\pi i m}{2m}})^2 = (e^{\frac{2\pi i j}{2m}})^2 e^{\frac{2\pi i 2m}{2m}} = w_j^2$. Deoarece w_j și w_{j+m} sunt distințe, rezultă $w_j = -w_{j+m}$. sfdem

Lema 10.2. Dacă n este par, $n = 2m$, și $1 \leq j < m$, atunci w_j^2 este de asemenea o rădăcină $\neq 1$.

Demonstrație. Deoarece $w_j^n = 1$ rezultă $(w_j^2)^n = 1$. Pe de altă parte, $w_j^2 = (e^{\frac{2\pi i j}{n}})^2 = e^{\frac{2\pi i(j+j)}{n}} = w_{j+j} \neq 1$ pentru că $2 \leq j + j \leq n - 2$. sfdem

În continuare ne ocupăm de un mijloc de calcul eficient al valorilor $f(w_j)$. Reamintim că determinarea valorii unui polinom într-un punct se poate realiza prin n înmulțiri și n adunări, utilizând schema lui Horner (a se vedea exercițiul 10.2), și aceste valori sunt optime [HS84, Kro79]. De aici, calculul direct al celor n componente ale transformatei Fourier necesită timpul $O(n^2)$. Utilizând strategia *divide et impera*, vom proiecta un algoritm care necesită $O(n \log n)$ timp. Numim calculul dat de acest algoritm *transformata Fourier rapidă* și-l notăm FFT (Fast Fourier Transform). Considerăm mai întâi cazul $n = 2^r$. Utilizăm proprietățile de mai sus ale rădăcinilor unității pentru a despărții suma din definiția lui $f(w_j)$ în două subsume: suma coeficienților pari și suma coeficienților impari. Avem:

$$\begin{aligned} f(w_j) &= \sum_{m=0}^{\frac{n}{2}-1} x_{2m} w_j^{2m} + \sum_{m=0}^{\frac{n}{2}-1} x_{2m+1} w_j^{2m+1} \\ &= \sum_{m=0}^{\frac{n}{2}-1} x_{2m} w_j^{2m} + w_j \sum_{m=0}^{\frac{n}{2}-1} x_{2m+1} w_j^{2m} \end{aligned}$$

De fapt, expresia de mai sus corespunde scrierii polinomului $f(t)$ sub forma:

$$f(Y) = (x_0 + x_2 Y^2 + \dots + x_{n-2} Y^{n-2}) + Y(x_1 + x_3 Y^2 + \dots + x_{n-1} Y^{n-2}) = g(Y) + Yh(Y)$$

unde $g(Y)$ și $h(Y)$ sunt polinoame de grad $n-2$, i.e., de grad mai mic cu 1 decât cel al lui $f(Y)$. Evaluarea acestor polinoame se va face numai în punctele w_m , $m = 0, 1, \dots, \frac{n}{2} - 1 = 2^{r-1} - 1$. Considerăm liste de numere $x_0, x_2, x_4, \dots, x_{2^{r-2}}$, respectiv $x_1, x_3, x_5, \dots, x_{2^{r-1}}$. Seriile Fourier ale acestor liste constau din valorile polinoamelor:

$$g'(Y) = x_0 + x_2 Y + x_4 Y^2 + \dots + x_{n-2} Y^{\frac{n}{2}-1}$$

și

$$h'(Y) = x_1 + x_3 Y + x_5 Y^2 + \cdots + x_{n-1} Y^{\frac{n}{2}-1}$$

pentru rădăcinile de ordinul $\frac{n}{2}$ ale unității. Notăm cu w'_j , $j = 0, 1, \dots, \frac{n}{2} - 1$ aceste rădăcini. Se pune problema dacă putem calcula $f(w_j)$, utilizând $g'(w'_j)$ și $h'(w'_j)$. Pentru $j < \frac{n}{2}$ avem:

$$w_j^{2m} = e^{\frac{2\pi ij2m}{2^r}} = e^{\frac{2\pi ijm}{2^{r-1}}} = w'_j m$$

iar pentru $j \geq \frac{n}{2}$ are loc:

$$w_j^{2m} = -w_{j-\frac{n}{2}}^{2m} = -e^{\frac{2\pi i(j-\frac{n}{2})2m}{2^r}} = e^{\frac{2\pi i(j-\frac{n}{2})m}{2^{r-1}}} = w'_{j-\frac{n}{2}}$$

Acstea relații arată că valorile $f(w_j)$ pot fi calculate dacă se cunosc $g'(w'_j)$ și $h'(w'_j)$: $f(w_j) = g'(w'_j) + w_j h'(w'_j)$, dacă $j < \frac{n}{2}$ și $f(w_j) = g'(-w'_j) + w_j h'(-w'_{j-n})$ dacă $j \geq \frac{n}{2}$. Deci strategia divide-et-impera poate fi aplicată cu succes. Algoritmul rezultat este:

```

procedure FFT(f, n, w)
  if (n = 1)
    then f(w_0) ← x_0
  else n ← [n/2]
    g' ← x_0 + x_2 Y + ⋯ + x_{2n-2} Y^{n-1}
    h' ← x_1 + x_3 Y + ⋯ + x_{2n-1} Y^{n-1}
    FFT(g', n, w)
    FFT(h', n, w)
    for j ← 0 to n-1 do
      f(w_j) ← g'(w'_j) + w_j * h'(w'_j)
    for j ← n to 2*n-1 do
      f(w_j) ← g'(-w'_{j-n}) + w_j * h'(-w'_{j-n})
  end

```

În algoritmul de mai sus există dezavantajul că trebuie calculate atât rădăcinile unității de ordinul n , cât și cele de ordinul $\frac{n}{2}$. Acest dezavantaj poate fi eliminat dacă se schimbă instanța problemei. Plecând de la observația că dacă se cunoaște o rădăcină primitivă de ordinul n a lui 1, să zicem $w \neq 1$, atunci celelalte rădăcini sunt puteri ale lui w . Deci putem lua $w_j = w^j$. Instanța problemei se modifică în felul următor: în loc să considerăm dat numai polinomul f , presupunem date polinomul f de grad $n - 1$ și o rădăcină primitivă de ordinul n a unității. Ieșirea constă în determinarea valorilor $f(w^0), f(w^1), f(w^2), \dots, f(w^{n-1})$. De asemenea, se ține cont de faptul că dacă w este o rădăcină primitivă de ordinul n a unității, atunci w^2 este o rădăcină primitivă de ordinul $\frac{n}{2}$ a unității. Noul algoritm este:

```

procedure FFT(f, n, w)
  if (n = 1)
    then f(w_0) ← x_0
  else n ← [n/2]
    ⋯

```

```

 $g' \leftarrow x_0 + x_2Y + \cdots + x_{2n-2}Y^{n-1}$ 
 $h' \leftarrow x_1 + x_3Y + \cdots + x_{2n-1}Y^{n-1}$ 
FFT( $g'$ , n, w)
FFT( $h'$ , n, w2)
for  $j \leftarrow 0$  to  $n-1$  do
     $f(w^j) \leftarrow g'(w^{2j}) + w * h'(w^{2j})$ 
for  $j \leftarrow n$  to  $2n-1$  do
     $f(w^j) \leftarrow g'(w^{2j}) + w * h'(w^{2j})$ 
end

```

Teorema 10.4. Dacă n este o putere a lui 2, atunci transformata Fourier discretă pentru n măsurători poate fi calculată în timpul $O(n \log n)$.

Demonstrație. Se aplică teorema 10.1 pentru $a = 2 = b$ și $k = 1$.

sfdem

10.6.2 Implementare

Următoarele funcții descriu operații peste numere complexe:

- adunarea:

```

function ad(x,y)
    z.re ← x.re + y.re
    z.im ← x.im + y.im
    return z
end

```

- scăderea:

```

function sc(x,y)
    z.re ← x.re - y.re
    z.im ← x.im - y.im
    return z
end

```

- înmulțirea:

```

function inm(x,y)
    z.re ← x.re*y.re - x.im*y.im
    z.im ← x.re*y.im + x.im*y.re
    return z
end

```

Procedura recursivă care implementează schema FFT este următoarea:

```

procedure FFT(f, n, w, fw)
    if (n = 1)
        then fw[0] ← f[0]
    else n ← n / 2
        for i ← 0 to n-1 do
            g[i] ← f[2*i]
            h[i] ← f[2*i+1]
        inm(w,w,w1)
    end
end

```

```

FFT(g,n,w1,gw)
FFT(h,n,w1,hw)
w1 ← complex(1)
for j ← 0 to n-1 do
    w2 ← inm(w1,hw[j])
    fw[j] ← ad(gw[j],w2)
    fw[j+n] ← sc(gw[j],w2)
    w1 ← inm(w1,w)
end

```

10.7 Exerciții

Exercițiul 10.1. Se știe că maximul (minimul) dintr-o listă cu n elemente se poate determina făcând $n - 1$ comparații. Astfel, se poate scrie un program care calculează simultan și maximul și minimul făcând $2(n - 1)$ comparații. Să se proiecteze un algoritm *divide-et-impera* care determină simultan minimul și maximul. Algoritmul va trebui să execute $\frac{3n}{2} - 2$ comparații, dacă n este o putere a lui 2.

Exercițiul 10.2. (*Evaluarea polinoamelor*) Să se scrie un algoritm *divide-et-impera* care să calculeze valoarea unui polinom într-un punct. Care este eficiența algoritmului? Să se compare aceasta cu cea a algoritmului bazat pe schema lui Horner:

$$f(x) = a_0 + (a_1 + \cdots + (a_{n-1}x + a_{n-2})x + \cdots)x$$

Exercițiul 10.3. [MS91] (*Numere Fibonacci*) Să se scrie un program care calculează al n -lea număr Fibonacci $F(n)$ executând $\Theta(\log n)$ operații aritmetice.

Exercițiul 10.4. [MS91] (*Înmulțire rapidă*)

1. Arătați că două polinoame de gradul 1 pot fi înmulțite executând exact trei înmulțiri.
2. Utilizând 1, să se scrie un algoritm *divide-et-impera* care să înmulțească două polinoame de grad n în timpul $\Theta(n^{\log_2 3})$.
3. Utilizând 2, să se arate că doi întregi reprezentați binar pe n biți pot fi înmulțiti în timpul $\Theta(n^{\log_2 3})$, unde orice operație binară invocă un număr constant de biți.

Exercițiul 10.5. *Diametrul* unui arbore (graf conex fără cicluri) este lungimea celui mai lung drum între două vârfuri. Să se găsească un algoritm *divide-et-impera* pentru determinarea diametrului unui arbore.

Exercițiul 10.6. (*Algoritmul lui Strassen de înmulțire a matricilor*) Înmulțirea a două matrici A și B se poate realiza cu numai șapte înmulțiri:

- se calculează expresiile:

$$\begin{aligned} q_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ q_2 &= (a_{21} + a_{22})b_{11} \\ q_3 &= a_{11}(b_{12} - b_{22}) \\ q_4 &= a_{22}(-b_{11} + b_{21}) \\ q_5 &= (a_{11} + a_{12})b_{22} \\ q_6 &= (-a_{11} + a_{21})(b_{11} + b_{12}) \\ q_7 &= (a_{12} + a_{22})(b_{21} + b_{22}) \end{aligned}$$

- apoi se calculează elementele matricei $C = AB$:

$$\begin{aligned}c_{11} &= q_1 + q_4 - q_5 + q_7 \\c_{12} &= q_3 + q_5 \\c_{21} &= q_2 + q_4 \\c_{22} &= q_1 + q_3 - q_2 + q_6\end{aligned}$$

Să se arate că înmulțirea a două matrici $n \times n$ se poate face cu $O(n^{\log_2 7})$ înmulțiri.

Exercițiul 10.7. Se consideră un ecran alb/negru cu rezoluția 1024×1024 . Notiunea de fereastră este definită recursiv astfel:

- ecranul este o fereastră;
- un pixel este o fereastră;
- o fereastră de dimensiune $n \times n$ definește 4 ferestre (stânga-sus, dreapta-sus, stânga-jos și dreapta-jos) de dimensiuni $\frac{n}{2} \times \frac{n}{2}$.

Colorarea ferestrelor se poate realiza cu următorul set de instrucțiuni:

- © - șterge ecranul,
- a - subfereastra stânga-sus devine fereastră curentă,
- b - subfereastra dreapta-sus devine fereastră curentă,
- c - subfereastra dreapta-jos devine fereastră curentă,
- d - subfereastra stânga-jos devine fereastră curentă,
- ↑ - ultima fereastră vizitată înaintea celei curente devine fereastră curentă.

Se cere:

1. Să se arate că pentru orice dreptunghi de pe ecran, cu laturile paralele cu marginile ecranului, există o secvență de instrucțiuni care realizează colorarea dreptunghiului.
2. Să se proiecteze un algoritm care, pentru un astfel de dreptunghi dat, determină o secvență de instrucțiuni de lungime minimă care realizează colorarea dreptunghiului.

10.8 Referințe bibliografice

Prezentarea de aici se bazează pe [HS84, MS91, LG86]. În [Sed88], metoda este prezentată ca o aplicație a recursiei.



008

www.thruwayenviantart.com

Capitolul 11

Programare dinamică

11.1 Exemplu: drum optim într-o rețea piramidală de numere

Considerăm următoarea problemă foarte simplă. Fie $a = (a_0, \dots, a_{n-1})$ o secvență de numere întregi, astfel încât $n + 1 = \frac{k(k+1)}{2}$, și $R(a)$ rețeaua piramidală ce are nodurile etichetate cu numerele a_i așa cum este sugerat în figura 11.1. Se pune problema determinării unui drum din vârful piramidei la baza piramidei pentru care suma numerelor aflate pe drum este maximă. Dacă notăm cu \mathcal{D} mulțimea drumurilor de la vârf la bază și cu $\ell(d)$ suma numerelor aflate pe drumul d atunci funcția obiectiv este:

$$\max_{d \in \mathcal{D}} \ell(d)$$

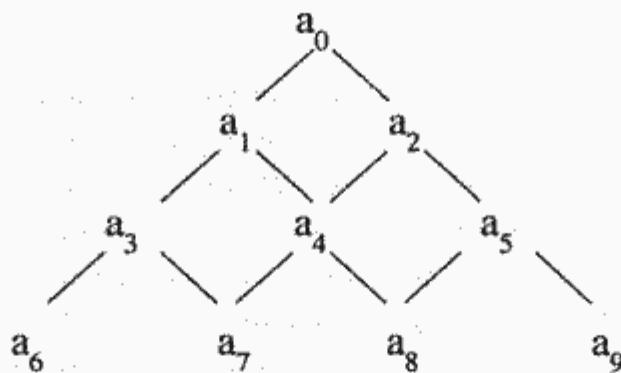
Prin stare a problemei vom înțelege subproblema $DMR(i)$ corespunzătoare subrețelei cu vârful în a_i . Funcția obiectiv a subproblemei $DMR(i)$ este:

$$\max_{d \in \mathcal{D}(i)} \ell(d)$$

unde $\mathcal{D}(i)$ este mulțimea drumurilor care pleacă din vârful a_i spre baza piramidei. Notăm cu $f(i)$ valoarea funcției obiectiv pentru $DMR(i)$. Fie $st(i)$ indicele fiului stâng al lui a_i și $dr(i)$ indicele fiului dreapta al lui a_i . Există relația evidentă $dr(i) = st(i) + 1$. Drumul optim ce pleacă din a_i va trece prin unul din cei doi fi ai lui a_i . Presupunem că trece prin $a_{st(i)}$. Subdrumul ce pleacă din $st(i)$ este la rândul său optim. Se raționează asemănător în cazul când drumul optim trece prin $a_{dr(i)}$ și rezultă următoarea relație de recurență:

$$f(i) = \max(f(st(i)), f(dr(i))) + a_i \quad (11.1)$$

Relația 11.1 poate fi dovedită utilizând inducția și reducerea la absurd. Dacă se rezolvă recursiv această relație, atunci $f(4)$ va fi calculată de două ori, $f(7)$ și $f(8)$ de trei ori și.a.m.d. De aceea valorile corespunzătoare drumurilor optime

Figura 11.1: $R(a)$ pentru $n = 10$.

vor fi memorate într-un tablou unidimensional și ordinea de calcul va fi de la baza piramidei spre vârf, adică în ordinea descrescătoare a indicilor.

Drumul optim b_0, b_1, \dots pentru o rețea de numere este determinat, pe baza tabloului cu valori optime, prin următorul algoritm:

```

b[0] ← a[0];
for j ← 1 to k do
  if (f[i]=f[st(i)]+b[j-1])
  then b[j] ← st(i)
  else b[j] ← dr(i)
  
```

11.2 Prezentarea intuitivă a paradigmei

Principalele ingrediente ale paradigmăi programare dinamică sunt următoarele:

1. *Clasa de probleme* la care se aplică include probleme de optim.
2. Definirea noțiunii de *stare*, care este de fapt o subproblemă, și asocierea funcții obiectiv pentru stare (subproblemă).
3. Definirea unei relații de tranziție între stări. O relație $s \rightarrow s'$, unde s și s' sunt stări, va fi numită *decizie*. O *politică* este o secvență de decizii consecutive, adică o secvență de formă $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$.
4. Aplicarea *Principiului de optim* pentru obține o relație de recurență. Principiul de optim (PO) afirmează că o subpolitica a unei politici optimale este la rândul ei optimală. Deoarece este posibil ca PO să nu aibă loc, rezultă că trebuie verificată validitatea relației de recurență.
5. *Calculul recurenței* rezolvând subproblemele de la mic la mare și memorând valorile obținute într-un tablou. Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime. Dacă în procesul de descompunere problemă \mapsto subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de algoritmul recursiv de câte ori apare.
6. Extragerea soluției optime din tablou utilizând proprietatea de *substructură optimă a soluției*, care afirmează că *soluția optimă a problemei include soluțiile optime ale subproblemelor sale*. De remarcat că proprietatea de substructură optimă este echivalentă cu principiul de optim.

Ca și în cazul celorlalte paradigmă, aplicarea programării dinamice se poate face în două moduri:

- direct: se definește noțiunea de *stare* (de cele mai multe ori ca fiind o subproblemă), se aplică principiul de optim pentru a deduce relațiile de recurență de tip 11.8 și apoi se stabilesc strategiile de calcul al valorilor și soluțiilor optime.
- prin comparare: se observă că problema este asemănătoare cu una dintre problemele cunoscute și se încearcă aplicarea strategiei în aceeași manieră ca în cazul problemei corespunzătoare.

11.3 Alocarea resurselor

11.3.1 Descrierea problemei

Pentru realizarea a p proiecte sunt disponibile r resurse. Alocarea a j resurse la proiectul i produce un profit $c[i, j]$. Problema constă în alocarea celor r resurse, astfel încât profitul total să fie maxim.

Problema poate fi reformulată ca o problemă de optim peste digrafuri. Considerăm mai întâi un exemplu.

Exemplu. Presupunem $p = 3$ și $r = 2$. Digaful asociat problemei este reprezentat în figura 11.2. Există un vîrf inițial s , care corespunde numărului inițial de resurse, și un vîrf final t , care corespunde numărului de resurse rămase nealocate (care este zero). Un vîrf intermediar corespunde numărului de resurse ce pot fi alocate în continuare. Arcele sunt etichetate cu profiturile aduse de resursele ce pot fi alocate unui anumit proiect. Nu pot fi alocate mai multe resurse decât sunt disponibile. Soluția optimă descrie un drum de la s la t .

sfex

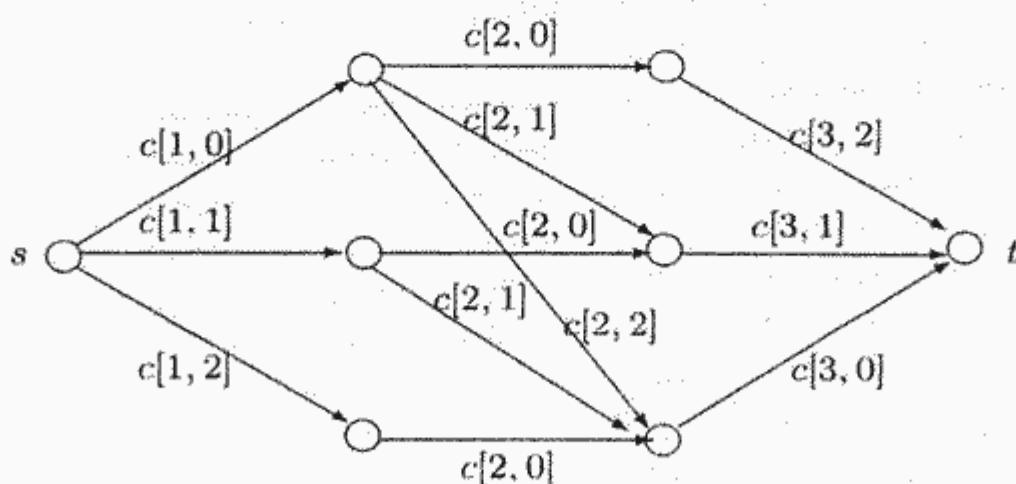


Figura 11.2: Alocarea resurselor – reprezentarea cu digrafuri

11.3.2 Modelul matematic

Construcția digrafului asociat problemei pentru cazul general poate fi dată în modul următor: mulțimea de vîrfuri ale digrafului este partajată în $p + 1$ submulțimi disjuncte: V_1, \dots, V_{p+1} . V_1 conține un singur vîrf s , V_{p+1} conține tot un singur

vârf t și V_i conține $r + 1$ vârfuri, $(i, j), 0 \leq j \leq r, 1 < i \leq p$. Pentru uniformitate, vom presupune $s = (1, r), t = (p + 1, 0)$. Un arc unește un vârf $(i, j) \in V_i$ cu un vârf $(i + 1, k) \in V_{i+1}$ numai dacă $j \geq k$ și în acest caz este etichetat cu $c[i, j - k]$. Semnificația unui arc $((i, j), (i + 1, k))$ cu eticheta $c[i, j - k]$ este următoarea: dacă la pasul i sunt disponibile j resurse și la proiectul i se alocă $j - k$ resurse, atunci se obține un profit $c[i, j - k]$ și rămân k resurse disponibile. Un astfel de graf este un caz particular de digraf etajat (*multi-stage graph*) definit după cum urmează:

Definiția 11.1. *Un digraf etajat este un digraf $G = (V, E)$, în care $V = V_1 \cup \dots \cup V_{p+1}$, cu proprietățile:*

- $V_i \cap V_j = \emptyset$ dacă $i \neq j$.
- $\#V_1 = \#V_{p+1} = 1$. Presupunem $V_1 = \{s\}, V_{p+1} = \{t\}$.
- dacă un arc are sursa în V_i , atunci are destinația în V_{i+1} .

Problema alocării resurselor poate fi generalizată acum la o problemă de optim peste digrafuri etajate.

Fie date un digraf etajat $G = (V, A)$ și o funcție de profit (cost) $c : A \rightarrow \mathcal{R}$. Profitul (costul) unui drum este suma profiturilor (costurilor) arcelor care descriu drumul. Problema constă în determinarea unui drum de la s la t cu profit maxim (cost minim).

Presupunem $V = \{0, 1, \dots, n - 1\}$, $s = 0$, $t = n - 1$ și că vâfurile de pe același etaj sunt numere consecutive.

Vom încerca să explicăm parcurgerea fiecărei etape a programării dinamice. Definim drept stare problema determinării drumurilor optime de la orice vârf $j \in X \subseteq V$ la t . Notăm această problemă prin $DODE(X)$ (Drumurile Optime în Digrafuri Etajate). Pentru i ($1 \leq i \leq p + 1$) și $j \in V_i$ notăm cu $D[i, j]$ drumul optim de la vârful j la t și cu $ValOpt[i, j]$ profitul (costul) acestui drum. $ValOpt[i, j]$ reprezintă valoarea asociată stării $DODE(\{j\})$. Valoarea asociată stării $DODE(X)$ va fi mulțimea $\{ValOpt[i, j] \mid j \in X\}$.

Ne punem problema găsirii relației analitice pentru valoarea optimă $ValOpt[i, j]$. Dacă $i = p + 1$, atunci singura valoare posibilă pentru j este n și avem $ValOpt[p + 1, n] = 0$. Considerăm $i < p$. Orice drum de la j la n începe cu un arc cu destinația în V_{i+1} . Notăm cu k destinația primului arc de pe drumul optim $D[i, j]$. Selectarea acestui arc poate fi gândită ca fiind decizia prin care starea $DODE(\{j\})$ este transformată în starea $DODE(\{k\})$. Aplicând principiul de optim, politica optimă aplicată stării $DODE(\{k\})$ este o subpolitică a politicii optime corespunzătoare stării $DODE(\{j\})$. Rezultă $ValOpt[i, j] = c[j, k] + ValOpt[i + 1, k]$. De aici, rezultă următoarea ecuație pentru $ValOpt$:

$$\begin{aligned} ValOpt[p + 1, n] &= 0 \\ ValOpt[i, j] &= \text{optim}\{c[j, k] + ValOpt[i + 1, k] \mid k \in V_{i+1}, (j, k) \in A\} \end{aligned} \quad (11.2)$$

unde $\text{optim} = \max$, dacă $c[j, k]$ reprezintă profitul, și $\text{optim} = \min$, dacă $c[j, k]$ reprezintă costul arcului (j, k) .

Pe baza relațiilor 11.2, calculul valorilor $ValOpt[i, j]$ rezultă din rezolvarea următoarelor subprobleme: $DODE(V_{p+1}), DODE(V_p \cup V_{p+1}), \dots, DODE(V_1 \cup \dots \cup V_{p+1}) = DODE(V)$. Deoarece vâfurile aflate pe un același etaj sunt numere consecutive, iar

etajele sunt considerate de la stânga la dreapta (de la s spre t), rezultă că drumurile optime se vor determina în ordinea descrescătoare a vâfurilor. Așa cum vom vedea la implementare, valorile $\text{ValOpt}[i, j]$ pot fi memorate într-un tablou unidimensional prin renunțarea la înregistrarea etajului la care aparține un vârf.

Utilizând 11.2, proprietatea de substructură optimă se caracterizează astfel: un drum optim care pleacă din vârful j conține drumurile optime care pleacă din orice vârf intermediar al său. Astfel, dacă se cunosc drumurile optime din $\text{DODE}(X)$ atunci se pot determina drumurile optime din $\text{DODE}(X \cup \{j\})$. De aici, determinarea unui drum optim de la s la t , după ce se cunosc valorile optime $\text{ValOpt}[i, j]$, se realizează prin determinarea tuturor drumurilor optime $D[i, j]$:

$$\begin{aligned} D[p+1, t] &= () \quad (\text{drumul vid}) \\ D[i, j] &= (\langle j, k \rangle, D[i+1, k]) \quad \text{dacă } \text{ValOpt}[i, j] = c[j, k] + \text{ValOpt}[i+1, k] \end{aligned}$$

unde un drum este reprezentat printr-o listă de arce. Notăm că drumurile pot fi determinate simultan cu valorile optime.

Aplicarea programării dinamice pentru determinarea drumurilor optime într-un digraf etajat este prezentată schematic de următorul algoritm:

```

procedure alocRes(V, E, c, ValOpt, D)
    ValOpt[p+1, n] ← 0
    D[p+1, n] ← listaVida()
    for i ← p downto 1 do
        for orice  $j \in V_i$  do
            determină  $k_0$  a.i.  $c[j, k_0] + \text{ValOpt}[i+1, k_0] =$ 
                optim{ $c[j, k] + \text{ValOpt}[i+1, k] \mid \langle j, k \rangle \in A\}$ 
            ValOpt[i, j] ←  $c[j, k_0] + \text{ValOpt}[i+1, k_0]$ 
            insereaza(D[i, j], 1, (j, k))
    end

```

Exercițiul 11.3.1. Să se descrie un algoritm pentru determinarea drumului optim într-un digraf etajat, bazat pe paradigma programării dinamice, considerând ca stare $\text{DODE}'(X) =$ problema determinării drumurilor optime de la s la orice $j \in X$.

11.3.3 Implementare

Pentru implementare, considerăm digraful G reprezentat prin listele de adiacență spre interior. Pentru reprezentarea funcției ValOpt este suficient să utilizăm un tablou unidimensional prin renunțarea la înregistrarea etajului la care aparține vârful j , i.e., $\text{ValOpt}[j] = \text{ValOpt}[i, j]$. Datorită proprietății de substructură optimă, nu este eficient să înregistram tot drumul optim care pleacă din vârful j . Dacă știm primul arc $\langle j, k \rangle$ de pe drumul optim, atunci parcurgem acest arc și apoi continuăm (recursiv) pe drumul optim care pleacă din k . În acest mod se parcurge tot drumul optim care pleacă din j . Vom utiliza un tablou $S[j]$ care va înregistra numai primul vârf (Secundul) de pe drumul optim care pleacă din j , iar întregul drum va fi $(j, S[j], S[S[j]], \dots, n)$ (reprezentat ca listă de vârfuri). Determinarea funcției ValOpt presupune parcurgerea digrafului pe etaje, începând cu ultimul

etaj $V_{p+1} = n$. Considerăm numai cazul optim = max. Pentru o descriere uniformă a algoritmului, se presupune că inițial are loc:

$$\text{ValOpt}[j] = -\infty \quad , j = 0, \dots, n-2$$

unde $-\infty$ va fi reprezentat de o constantă MinusInf, mai mică decât orice $c[i, j]$. Plecând de la ipoteza că pentru vârful k de pe nivelul i valoarea optimă $\text{ValOpt}[k]$ a fost determinată, se actualizează $\text{ValOpt}[j]$ pentru toate vârfurile j de pe nivelul $i-1$ cu proprietatea că există arcul (j, k) . Actualizarea constă în verificarea condiției $\text{ValOpt}[j] < c[j, k] + \text{ValOpt}[k]$ și în cazul când este adevărată, noua valoare a lui $\text{ValOpt}[j]$ este $c[j, k] + \text{ValOpt}[k]$ iar valoarea variabilei $D[j]$ este k . Valoarea $c[j, k]$ este memorată de variabila $q->c$, unde $q->\text{varf} = j$ iar q este o variabilă referință (pointer) cu care se parcurge lista vârfurilor incidente spre interior în k . După completarea tablourilor ValOpt și S , se determină drumul optim. Valoarea drumului optim este $\text{ValOpt}[0]$.

```

function alocRes(G, ValOpt, D)
    for j ← 0 to G.n-2 do
        ValOpt[j] ← -∞
    ValOpt[G.n-1] ← 0
    for k ← G.n-1 downto 1 do
        q ← G.a[k]
        while (q ≠ NULL) do
            j ← q->varf
            if (ValOpt[j] < ValOpt[k] + q->c)
                then ValOpt[j] ← ValOpt[k] + q->c
                    S[j] ← k
            q ← q->succ
    D[0] ← 0
    D[p] ← n-1
    for i ← 1 to p-1 do
        D[i] ← S[D[i-1]]
    return ValOpt[0]
end

```

Evaluarea algoritmului Fiecare arc este cercetat o singură dată, astfel că determinarea celor două tablouri ValOpt și S necesită $O(m)$ timp, unde m este numărul de arce. Deoarece determinarea drumului se face în timpul $\Theta(p)$, rezultă că algoritmul **alocRes** necesită $O(m)$ timp.

11.4 Drumurile cele mai scurte între oricare două vârfuri ale unui digraf

11.4.1 Descrierea problemei

Această problemă este o generalizare a problemei determinării drumului optim într-un digraf etajat:

Se consideră $G = (V, A)$ un digraf cu $V = \{0, \dots, n - 1\}$ și $\ell : A \rightarrow \mathcal{R}$ o funcție de etichetare a arcelor. Perechea (G, ℓ) se mai numește *digraf ponderat*. Notăm ℓ_{ij} în loc de $\ell((i, j))$ și numim ℓ_{ij} *lungimea* ($=$ costul) arcului (i, j) . *Lungimea* ($=$ costul) *unui drum* este suma lungimilor arcelor ce compun drumul. Problema constă în a determina, pentru orice două vârfuri i, j , un drum de lungime minimă de la vârful i la vârful j (când există).

11.4.2 Modelul matematic

Pentru o prezentare uniformă a metodei, extindem funcția ℓ la $\ell : V \times V \rightarrow \mathcal{R}$, punând $\ell_{ij} = \infty$ pentru acele perechi de vârfuri distincte cu $(i, j) \notin E$ și $\ell_{ii} = 0$ pentru orice $i = 0, \dots, n - 1$.

Drept stare definim $\text{DM2VD}(X)$ (Drum Minim între oricare două vârfuri ale unui Digraf) ca fiind subproblema corespunzătoare determinării drumurilor de lungime minimă cu vârfuri intermediare din mulțimea $X \subseteq V$. Evident, $\text{DM2VD}(V)$ este chiar problema inițială. Notăm cu ℓ_{ij}^X lungimea drumului minim de la i la j construit cu vârfuri intermediare din X . Dacă $X = \emptyset$, atunci $\ell_{ij}^\emptyset = \ell_{ij}$. Considerăm decizia optimă care transformă starea $\text{DM2VD}(X \cup \{k\})$ în $\text{DM2VD}(X)$. Presupunem că (G, ℓ) este un digraf ponderat fără circuite negative. Fie ρ un drum optim de la i la j ce conține vârfuri intermediare din mulțimea $X \cup \{k\}$. Avem $\text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$, unde $\text{lung}(\rho)$ este lungimea drumului ρ . Dacă vârful k nu aparține lui ρ , atunci politica obținerii lui ρ corespunde de asemenea și stării $\text{DM2VD}(X)$ și, aplicând principiul de optim, obținem:

$$\ell_{ij}^X = \text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$$

În cazul în care k aparține drumului ρ , notăm cu ρ_1 subdrumul lui ρ de la i la k și cu ρ_2 subdrumul de la k la j . Aceste două subdrumuri au vârfuri intermediare numai din X . Conform principiului de optim, politica optimă corespunzătoare stării $\text{DM2VD}(X)$ este subpolitica a politicii optime corespunzătoare stării $\text{DM2VD}(X \cup \{k\})$. Rezultă că ρ_1 și ρ_2 sunt optime în $\text{DM2VD}(X)$. De aici rezultă:

$$\ell_{ij}^{X \cup \{k\}} = \text{lung}(\rho) = \text{lung}(\rho_1) + \text{lung}(\rho_2) = \ell_{ik}^X + \ell_{kj}^X$$

Acum, ecuația funcțională analitică pentru valorile optime ℓ_{ij}^X are următoarea formă:

$$\ell_{ij}^{X \cup \{k\}} = \min\{\ell_{ij}^X, \ell_{ik}^X + \ell_{kj}^X\} \quad (11.3)$$

Corolar 11.1. Dacă (G, ℓ) nu are circuite de lungime negativă, atunci au loc următoarele relații:

- $\ell_{kk}^{X \cup \{k\}} = 0$
- $\ell_{ik}^{X \cup \{k\}} = \ell_{ik}^X$
- $\ell_{kj}^{X \cup \{k\}} = \ell_{kj}^X$

pentru orice $i, j, k \in V$.

Calculul valorilor optime rezultă din rezolvarea subproblemelor

$$\text{DM2VD}(\emptyset), \text{DM2VD}(\{0\}), \text{DM2VD}(\{0, 1\}), \dots, \text{DM2VD}(\{0, 1, \dots, n-1\}) = \text{DM2VD}(V)$$

Convenim să notăm ℓ_{ij}^k în loc de $\ell_{ij}^{\{0, \dots, k-1\}}$. Pe baza corolarului 11.1 rezultă că valorile optime pot fi memorate într-un același tablou. Maniera de determinare a acestora este asemănătoare cu cea utilizată la determinarea matricei existenței drumurilor de către algoritmul lui Warshall.

Pe baza ecuațiilor 11.3, proprietatea de substructură optimă se caracterizează prin: un drum optim de la i la j include drumurile optime de la i la k și de la k la j , pentru orice vârf intermediu k al său. Astfel că drumurile optime din $\text{DM2VD}(X \cup \{k\})$ pot fi determinate utilizând drumurile minime din $\text{DM2VD}(X)$. În continuare considerăm numai cazurile $X = \{0, 1, \dots, k-1\}$. Ca și în cazul digrafurilor etajate, determinarea drumurilor optime poate fi făcută cu ajutorul unor tablouri $P^k = (P_{ij}^k)$, dar care de această dată au o semnificație diferită: P_{ij}^k este vârful penultimului vârf de pe drumul optim de la i la j . Pentru $k=0$ avem $P_{ij}^0 = i$ dacă $\langle i, j \rangle \in E$ și $P_{ij}^0 = 0$, în celealte cazuri. Decizia k determină P^k odată cu determinarea matricei $\ell^k = (\ell_{ij}^k)$. Dacă $\ell_{ik}^{k-1} + \ell_{kj}^{k-1} < \ell_{ij}^{k-1}$, atunci drumul optim de la i la j este format din concatenarea drumului optim de la i la k cu drumul optim de la k la j și deci penultimul vârf de pe drumul de la i la j coincide cu penultimul vârf de pe drumul de la k la j ; $P_{ij}^k = P_{kj}^{k-1}$. În caz contrar, avem $P_{ij}^k = P_{ij}^{k-1}$. Cu ajutorul matricei P_{ij}^n pot fi determinate drumurile optime: ultimul vârf pe drumul de la i la j este $j_t = j$, penultimul vârf este $j_{t-1} = P_{ij}^n$, antipenultimul este $j_{t-2} = P_{ij_{t-1}}^n$, și.m.d. În acest mod, toate drumurile pot fi memorate utilizând numai $O(n^2)$ spațiu.

Aplicarea programării dinamice pentru problema determinării drumurilor minime este descrisă de schema procedurală drmin:

```

procedure drmin(G, ℓ, P)
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
            ℓij0 = { 0 , i = j
                      ℓij , ⟨i, j⟩ ∈ A
                      ∞ , altfel
            Pij0 = { i , i ≠ j, ⟨i, j⟩ ∈ A
                      0 , altfel
    for k ← 0 to n-1 do
        for i ← 0 to n-1 do
            for j ← 0 to n-1 do
                ℓijk = min{ℓijk-1, ℓikk-1 + ℓkjk-1}
                Pijk = { Pijk-1 , ℓijk = ℓijk-1
                            Pkjk-1 , ℓijk = ℓikk-1 + ℓkjk-1
    end

```

11.4.3 Implementare

Presupunem că digraful $G = (V, A)$ este reprezentat prin matricea de adiacență, pe care convenim să o notăm aici cu $G.L$ (este ușor de văzut că matricea ponderilor include și reprezentarea lui A). Datorită corolarului 11.1, matricele ℓ^k și ℓ^{k+1} pot fi memorate de același tablou bidimensional $G.L$. Este ușor de văzut că matricea $G.L$ include și reprezentarea lui A . Simbolul ∞ este reprezentat de o constantă `plusInf` cu valoare foarte mare. Dacă digraful are circuite negative, atunci acest lucru poate fi depistat: dacă la un moment dat se obține $G.L[i, i] < 0$, pentru un i oarecare, atunci există un circuit de lungime negativă care trece prin i . Funcția `drmin` întoarce valoarea `true` dacă digraful ponderat reprezentat de matricea $G.L$ nu are circuite negative și în acest caz $G.L$ va conține la ieșire lungimile drumurilor minime între oricare două varfuri, iar `G.P` reprezentarea acestor drumuri.

```

procedure drmin(G, P)
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
            if ((i ≠ j) and (L[i, j] ≠ plusInf))
                then P[i, j] ← i
            else P[i, j] ← 0
    for k ← 0 to n-1 do
        for i ← 0 to n-1 do
            for j ← 1 to n do
                if ((L[i, k] = PlusInf) or (L[k, j] = PlusInf))
                    then temp ← plusInf
                else temp ← L[i, k]+L[k, j]
                if (temp < L[i, j])
                    then L[i, j] ← temp
                        P[i, j] ← P[k, j]
                if ((i = j) and (L[i, j] < 0))
                    then throw '(di)graful are circuite negative'
    end

```

Evaluare. Se verifică ușor că execuția algoritmului `drmin` necesită $O(n^3)$ timp și utilizează $O(n^2)$ spațiu.

Observație. Algoritmul descris de `drmin` este cunoscut în literatură sub numele de algoritmul Floyd-Warshall. O formă restrictivă a problemei drumurilor minime se obține prin precizarea varfului de start. Algoritmii cei mai cunoscuți care rezolvă această formă restrictivă sunt Dijkstra (exercițiul 9.5) și Bellman-Ford (exercițiul 11.2).

sfobs

11.5 Problema rucsacului II (varianta discretă)

11.5.1 Descrierea problemei

Considerăm următoarea versiune modificată a problemei rucsacului:

Se consideră un rucsac de capacitate $M \in \mathbb{Z}_+$ și n obiecte $1, \dots, n$ de dimensiuni (greutăți) $w_1, \dots, w_n \in \mathbb{Z}_+$. Un obiect i este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus deloc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvență x_1, \dots, x_n cu $x_i \in \{0, 1\}$ și $\sum_i x_i \cdot w_i \leq M$. Ca și în cazul continuu, introducerea obiectului i în rucsac aduce profitul $p_i \in \mathbb{Z}$, iar profitul total este $\sum_{i=1}^n x_i p_i$. Problema constă în a determina o alegere (x_1, \dots, x_n) care să aducă un profit maxim.

Deci, singura deosebire față de varianta continuă studiată la metoda greedy constă în condiția $x_i \in \{0, 1\}$, în loc de $x_i \in [0, 1]$.

Exercițiul 11.5.1. Considerăm o variantă mai „veselă” a problemei: un hoț intră într-un magazin unde sunt n obiecte de dimensiuni și valori diferite. În sacul hoțului încap numai o parte dintre aceste obiecte. Hoțul trebuie să decidă într-un timp foarte scurt ce obiecte pune în sac astfel încât să aibă un „profit” cât mai mare. Fiind un bun algoritmist, acesta aplică un algoritm greedy.

1. Arătați că indiferent de criteriul de alegere locală, nu întotdeauna soluția aleasă este și cea optimă.
2. Am spus că hoțul era un bun algoritmist. De ce a ales el strategia greedy și nu oricare altă metodă care dă soluția optimă? (Deși un răspuns riguros argumentat poate fi dat după citirea acestei secțiuni, încercați să-l anticipați.)

11.5.2 Modelul matematic

Formulată ca problemă de optim, problema rucsacului, varianta discretă, este:

- funcția obiectiv:

$$\max \sum_{i=1}^n x_i \cdot p_i$$

- restricții:

$$\begin{aligned} \sum_{i=1}^n x_i \cdot w_i &\leq M \\ x_i &\in \{0, 1\}, i = 1, \dots, n \\ w_i &\in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \dots, n \\ M &\in \mathbb{Z} \end{aligned}$$

O stare este notată cu $RUCSAC(j, X)$ și reprezintă următoarea problemă, care este o generalizare a celei inițiale:

- funcția obiectiv:

$$\max \sum_{i=1}^j x_i \cdot p_i$$

- restricții:

$$\begin{aligned} \sum_{i=1}^j x_i \cdot w_i &\leq X \\ x_i &\in \{0, 1\}, i = 1, \dots, j \\ w_i &\in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \dots, j \\ X &\in \mathbb{Z} \end{aligned}$$

Cu $f_j(X)$ notăm valoarea optimă pentru instanța RUCSAC(j, X). Dacă $j = 0$ și $X \geq 0$, atunci $f_j(X) = 0$. Presupunem $j > 0$. Considerăm decizia optimă prin care starea RUCSAC(j, X) este transformată în RUCSAC($j - 1, ?$). Notăm cu (x_1, \dots, x_j) alegerea care dă valoarea optimă $f_j(X)$. Dacă $x_j = 0$ (obiectul j nu este pus în rucsac), atunci, conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea RUCSAC($X, j - 1$) și de aici $f_j(X) = f_{j-1}(X)$. Dacă $x_j = 1$ (obiectul j este pus în rucsac), atunci, din nou conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea RUCSAC($j - 1, X - w_j$) și, de aici, $f_j(X) = f_{j-1}(X - w_j) + p_j$. Combinând relațiile de mai sus obținem:

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ și } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X - w_j) + p_j\}, & \text{dacă } j > 0 \text{ și } X \geq 0 \end{cases} \quad (11.4)$$

Am considerat $f_j(X) = -\infty$, dacă $X < 0$.

Utilizând 11.4, proprietatea de substructură optimă se caracterizează astfel: soluția optimă (x_1, \dots, x_j) a problemei RUCSAC(j, X) include soluția optimă (x_1, \dots, x_{j-1}) a subproblemei RUCSAC($j - 1, X - x_j w_j$). Astfel, soluția optimă pentru RUCSAC(j, X) se poate obține utilizând soluțiile optime pentru subproblemele RUCSAC(i, Y) cu $1 \leq i < j, 0 \leq Y \leq X$. Notăm că 11.4 implică o recursie în cascadă și deci numărul de subprobleme de rezolvat este $O(2^n)$. De aceea, în continuare ne ocupăm de calculul și memorarea eficientă a valorilor optime pentru subprobleme. Mai întâi considerăm următorul exemplu.

Exemplu. Fie $M = 10, n = 3$ și greutățile și profiturile date de următorul tabel:

	i	1	2	3
w_i		3	5	6
p_i		10	30	20

Valorile optime pentru subprobleme sunt calculate cu ajutorul relațiilor 11.4 și pot fi memorate într-un tablou bidimensional astfel:

X	0	1	2	3	4	5	6	7	8	9	10
f_0	0	0	0	0	0	0	0	0	0	0	0
f_1	0	0	0	10	10	10	10	10	10	10	10
f_2	0	0	0	10	10	30	30	30	40	40	40
f_3	0	0	0	10	10	30	30	30	40	40	40

Tabloul de mai sus este calculat linie cu linie: pentru a calcula valorile de pe o linie sunt consultate numai valorile de pe linia precedentă. Algoritmul de calcul este foarte simplu. De exemplu, $f_2(8) = \max\{f_1(8), f_1(8 - 5) + 30\} = \max\{10, 40\} = 40$. Valoarea optimă a funcției obiectiv este $f_3(10) = 40$. Soluția care dă această valoare se determină în modul următor: se testează dacă $f_2(10) = f_3(10)$. Răspunsul este afirmativ și, de aici, rezultă $x_3 = 0$, i.e. obiectul 3 nu este pus în rucsac. În continuare se testează dacă $f_1(10) = f_2(10)$. De data aceasta, răspunsul este negativ. Rezultă că $f_2(10) = f_1(10 - 5) + 30 = f_1(5) + 30$ și, de aici, $x_2 = 1$, i.e., obiectul 2 este pus în rucsac. În continuare se testează dacă $f_0(5) = f_1(5)$. Si de

data aceasta răspunsul este negativ și deci $x_1 = 1$. Dinamica valorilor comparate este sugerată în tablou prin scrierea acestora în dreptunghiuri.

Tabloul de mai sus are dimensiunea $n \cdot m$ (au fost ignorate prima linie și prima coloană). Dacă $m = O(2^n)$ rezultă că atât complexitatea spațiu, cât și cea timp sunt exponențiale. Privind tabloul de mai sus observăm că există multe valori care se repetă. În continuare ne punem problema memorării mai compacte a acestui tablou. Construim graficele funcțiilor f_0, f_1, f_2 și f_3 pentru exemplul de mai sus. Avem:

$$f_0(X) = \begin{cases} -\infty & , X < 0 \\ 0 & , X \geq 0 \end{cases}$$

Notăm cu g_0 funcția dată de:

$$g_0(X) = f_0(X - w_1) + p_1 = \begin{cases} -\infty & , X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Graficele funcțiilor f_0 și g_0 sunt reprezentate în figura 11.3.

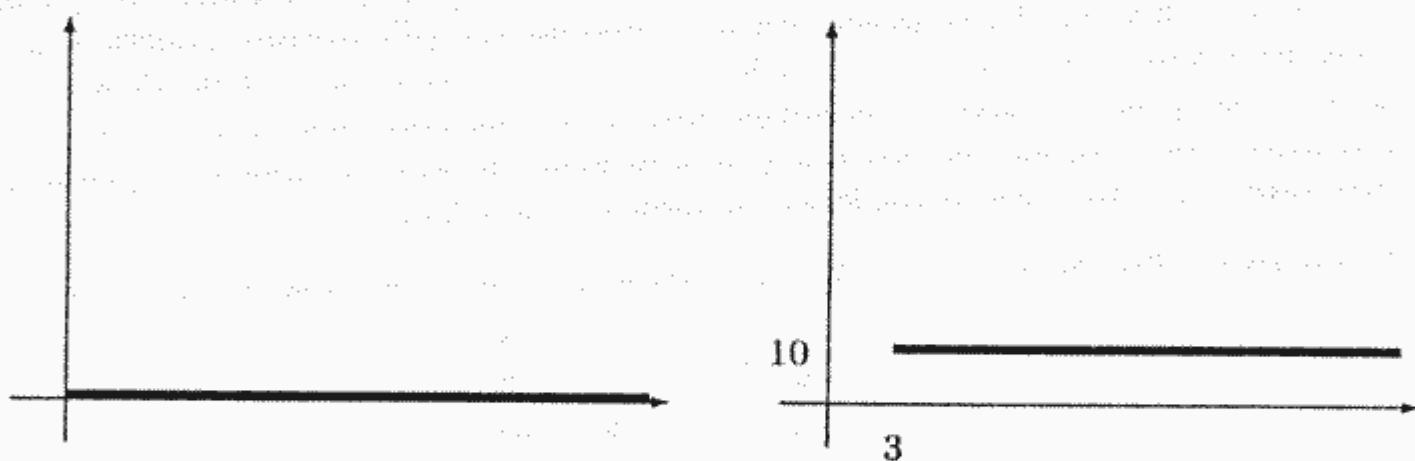


Figura 11.3: Funcțiile f_0 și g_0

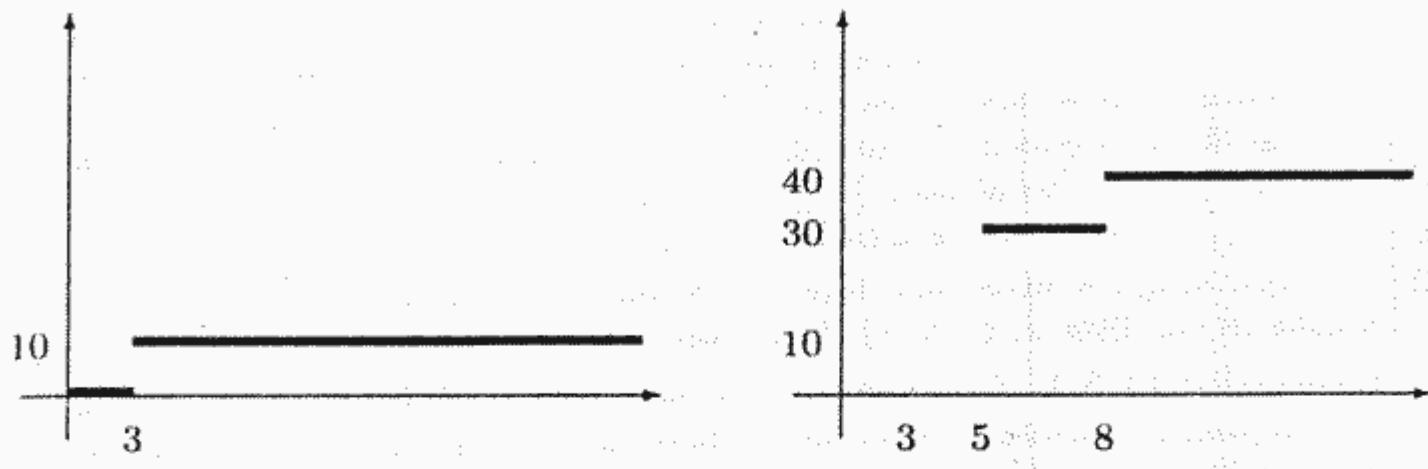
Funcția f_1 se calculează prin:

$$f_1(X) = \max\{f_0(X), g_0(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Notăm cu g_1 funcția dată de:

$$g_1(X) = f_1(X - w_2) + p_2 = \begin{cases} -\infty & , X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

Graficele funcțiilor f_1 și g_1 sunt reprezentate în figura 11.4.

Figura 11.4: Funcțiile f_1 și g_1

Funcția f_2 se calculează prin:

$$f_2(X) = \max\{f_1(X), g_1(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

În continuare, notăm cu g_2 funcția dată prin:

$$g_2(X) = f_2(X - w_3) + p_3 = \begin{cases} -\infty & , X < 6 \\ 20 & , 6 \leq X < 9 \\ 30 & , 9 \leq X < 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

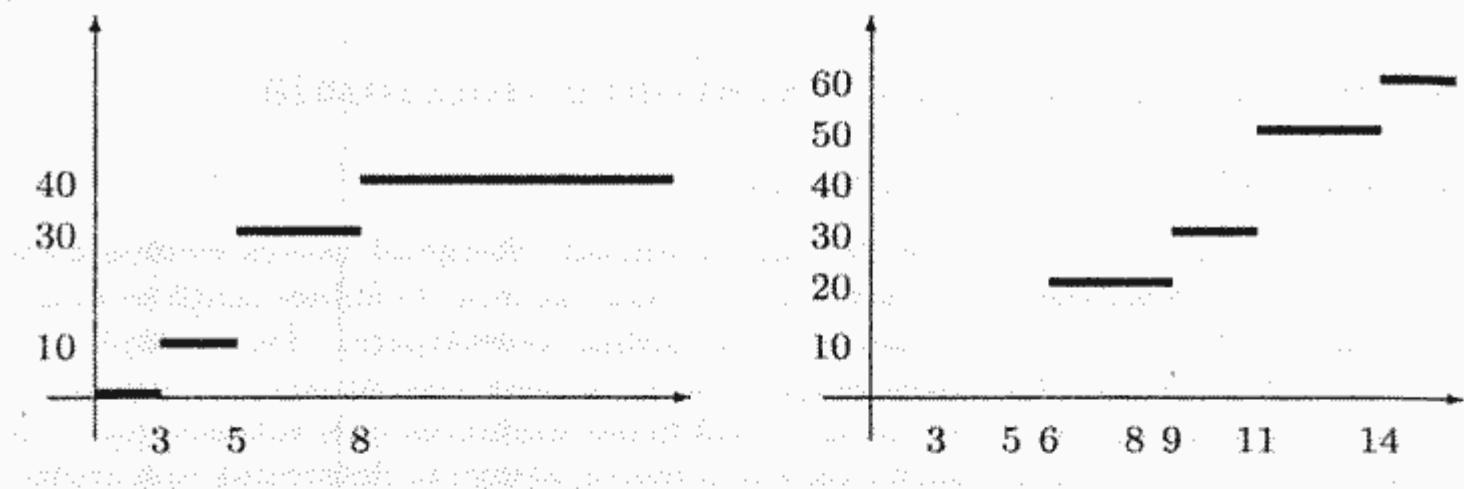
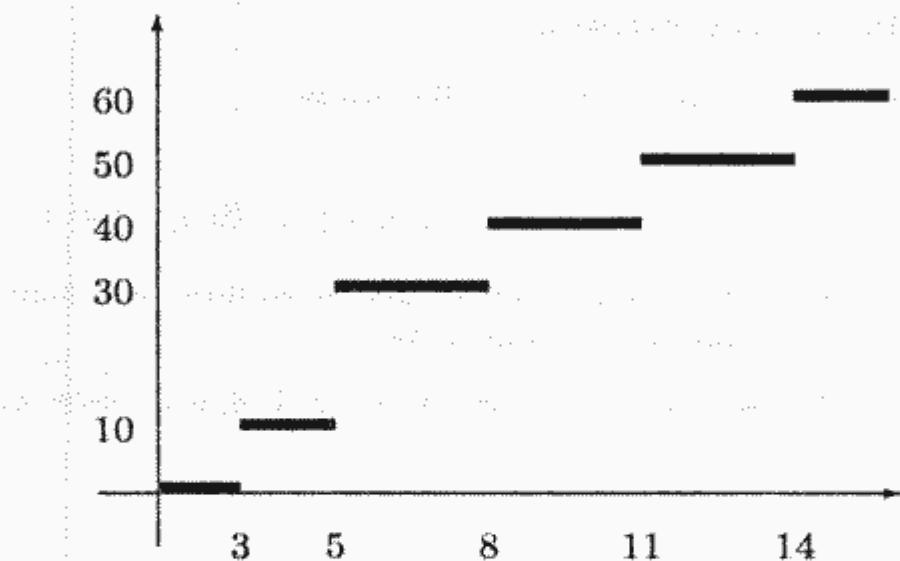
Graficele funcțiilor f_2 și g_2 sunt reprezentate în figura 11.5.

Funcția f_3 se calculează prin:

$$f_3(X) = \max\{f_2(X), g_2(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 < X \leq 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

Graficul funcției f_3 este reprezentat în figura 11.6.

Se remarcă faptul că funcțiile f_i și g_i sunt funcții în scară. Graficele acestor funcții pot fi reprezentate prin mulțimi finite din puncte din plan. De exemplu,

Figura 11.5: Funcțiile f_2 și g_2 Figura 11.6: Funcția f_3

graficul funcției f_2 este reprezentat prin mulțimea $\{(0,0), (3,10), (5,30), (8,40)\}$. O mulțime care reprezintă o funcție în scară conține acele puncte în care funcția face salturi. Graficul funcției g_i se obține din graficul funcției f_i printr-o translație, iar graficul funcției f_{i+1} se obține prin interclasarea graficelor funcțiilor f_i și g_i .

În general, fiecare f_i este complet specificat de o mulțime $S_i = \{(X_j, Y_j) \mid j = 0, \dots, r\}$, unde $Y_j = f_i(X_j)$. Presupunem $X_1 < \dots < X_r$. Analog, funcțiile g_i sunt reprezentate prin mulțimile $T_i = \{(X + w_i, Y + p_i) \mid (X, Y) \in S_i\}$. Notăm $T_i = \tau(S_i)$ și $S_{i+1} = \mu(S_i, T_i)$. Mulțimea S_{i+1} se obține din S_i și T_i prin interclasare. Operația de interclasare se realizează într-un mod asemănător cu cel de la interclasarea a două linii ale orizontului. Se consideră o variabilă L care ia valoarea 1 dacă graficul lui f_{i+1} coincide cu cel al lui f_i și cu 2 dacă el coincide cu cel al lui g_i . Deoarece $(0,0)$ aparține graficului rezultat, considerăm $L = 1$, $j = 1$ și $k = 1$. Presupunând că la un pas al interclasării se compară $(X_j, Y_j) \in S_i$ cu $(X_k, Y_k) \in T_i$, atunci:

- dacă $L = 1$:
 - dacă $X_j < X_k$, atunci se adaugă (X_j, Y_j) în S_{i+1} și se incrementează j ;

- dacă $X_j = X_k$:
 - * dacă $Y_j > Y_k$, atunci se adaugă (X_j, Y_j) în S_{i+1} și se incrementează j și k ;
 - * dacă $Y_j < Y_k$, atunci se adaugă (X_k, Y_k) în S_{i+1} , $L = 2$ și se incrementează j și k ;
- dacă $X_j > X_k$, atunci, dacă $Y_k > Y_j$, se adaugă (X_k, Y_k) în S_{i+1} , $L = 2$ și se incrementează k ;
- dacă $L = 2$:
 - dacă $X_j < X_k$, atunci, dacă $Y_j > Y_k$, se adaugă (X_j, Y_j) în S_{i+1} , $L = 1$ și se incrementează j ;
 - dacă $X_j = X_k$:
 - * dacă $Y_j < Y_k$, atunci se adaugă (X_k, Y_k) în S_{i+1} și se incrementează j și k ;
 - * dacă $Y_j > Y_k$, atunci se adaugă (X_j, Y_j) în S_{i+1} , $L = 1$ și se incrementează j și k ;
 - dacă $X_j > X_k$, atunci se adaugă (X_k, Y_k) în S_{i+1} și se incrementează k ;

Notăm cu `interclGrafice(Si, Ti)` funcția care determină S_{i+1} conform algoritmului de mai sus. Rămâne de extras soluția optimă din S_n . Considerăm mai întâi cazul din exemplul de mai sus.

Exemplu (continuare).

- Se caută în $S_n = S_3$ perechea (X_j, Y_j) cu cel mai mare X_j pentru care $X_j \leq M$. Obținem $(X_j, Y_j) = (8, 40)$. Deoarece $(8, 40) \in S_3$ și $(8, 40) \in S_2$ rezultă $f_{\text{optim}}(M) = f_{\text{optim}}(8) = f_3(8) = f_2(8)$ și deci $x_3 = 0$. Perechea (X_j, Y_j) rămâne neschimbată.
- Pentru că $(X_j, Y_j) = (8, 40)$ este în S_2 și nu este în S_1 , rezultă că $f_{\text{optim}}(8) = f_1(8 - w_2) + p_2$ și deci $x_2 = 1$. În continuare se ia $(X_j, Y_j) = (X_j - w_2, Y_j - p_2) = (8 - 5, 40 - 30) = (3, 10)$.
- Pentru că $(X_j, Y_j) = (3, 10)$ este în S_1 și nu este în S_0 , rezultă că $f_{\text{optim}}(3) = f_1(3 - w_1) + p_1$ și deci $x_1 = 1$.

Metoda poate fi descrisă pentru cazul general:

- Inițial se determină perechea $(X_j, Y_j) \in S_n$ cu cel mai mare X_j pentru care $X_j \leq M$. Valoarea Y_j constituie încărcarea optimă a rucsacului, i.e., valoarea funcției obiectiv din problema inițială.
- Pentru $i = n - 1, \dots, 0$:
 - dacă (X_j, Y_j) este în S_i , atunci $f_{i+1}(X_j) = f_i(X_j) = Y_j$ și se face $x_{i+1} = 0$ (obiectul $i + 1$ nu este ales);
 - dacă (X_j, Y_j) nu este în S_i , atunci $f_{i+1}(X_j) = f_i(X_j - w_{i+1}) + p_{i+1} = Y_j$ și se face $x_{i+1} = 1$ (obiectul $i + 1$ este ales), $X_j = X_j - w_{i+1}$ și $Y_j = Y_j - p_{i+1}$.

11.5.3 Implementare

Descrierea algoritmică completă a metodei prezentate anterior este următoarea:

```

procedure rucsac_II(n, w, p, valOpt, x)
    S0 ← {(0, 0)}
    T0 ← {(w1, p1)}
    for i ← 1 to n
        Si(X) ← interclGrafice(Si-1, Ti-1)
        Ti ← {(X + wi, Y + pi) | (X, Y) ∈ Si}
    determină (Xj, Yj) cu Xj = max{Xi | (Xi, Yi) ∈ Sn, Xi ≤ M}
    for i ← n-1 downto 1 do
        if (Xj, Yj) ∈ Si
            then xi+1 ← 0
            else xi+1 ← 1
                Xj ← Xj - wi+1
                Yj ← Yj - pi+1
    end

```

Evaluare. Ne propunem să determinăm timpul de execuție și spațiul utilizat de algoritmul rucsac_II. Notăm $m = \sum_{i=0}^n \#S_i$. Deoarece $\#T_i = \#S_i$ rezultă că $\#S_{i+1} \leq 2 \cdot \#S_i$ și de aici $\sum_i \#S_i \leq \sum_i 2^i = 2^n - 1$. Calculul lui S_i din S_{i-1} necesită timpul $\Theta(\#S_{i-1})$ și de aici calculul lui S_n necesită timpul $\sum_i \Theta(\#S_i) = O(2^n)$. Deoarece profiturile p_i sunt numere întregi, pentru orice $(X, Y) \in S_i$, Y este întreg și $Y \leq \sum_{j \leq i} p_j$. Analog, pentru că dimensiunile w_i sunt numere întregi, pentru $(X, Y) \in S_i$, X este întreg și $X \leq \sum_{j \leq i} w_j$. Deoarece perechile (X, Y) cu $X > M$ nu interesează, ele pot să nu fie incluse în mulțimile S_i . De aici, rezultă că numărul maxim de perechi (X, Y) distincte din S_i satisface relațiile:

$$\#S_i \leq 1 + \sum_{j=1}^i w_j \quad \text{și} \quad \#S_i \leq M$$

care implică

$$\#S_i \leq 1 + \min\left\{\sum_{j=1}^i w_j, M\right\}.$$

Relația de mai sus permite o estimare mai precisă a spațiului necesar pentru memorarea mulțimilor S_i în cazul unor probleme concrete. În ceea ce privește timpul, făcând calculele, rezultă că algoritmul are complexitatea timp

$$O(\min(2^n, n \sum_{i=1}^n p_i, nM)).$$

11.6 Subsecvență crescătoare maximală

11.6.1 Descrierea problemei

Fie $a = (a_1, \dots, a_n)$ o secvență de numere întregi. Stergând câteva elemente din a se obține o nouă secvență, pe care o numim *subsecvență*. O subsecvență se numește *crescătoare* dacă elementele sale sunt în ordine crescătoare. De exemplu, dacă $a = (7, 3, 8, 5, 13)$, atunci subsecvența $(7, 3, 5)$ nu este crescătoare, iar subsecvența $(7, 8, 13)$ este crescătoare. Problema constă în determinarea unei subsecvențe crescătoare de lungime maximă. Dacă există mai multe, atunci se determină una arbitrară dintre ele.

11.6.2 Modelul matematic

Se atașează problemei un digraf $G = (V, E)$, unde

$$\begin{aligned} V &= \{0, 1, \dots, n\} \\ A &= \{(i, j) \mid a[i] \leq a[j], 1 \leq i < j \leq n\} \cup \{(0, i) \mid i = 1, \dots, n\} \end{aligned}$$

Determinarea unei subsecvențe crescătoare maxime este echivalentă cu determinarea unui drum de lungime maximă în G .

Exemplu. Digraful asociat secvenței $a = (3, 1, 4, 6, 3)$ este cel reprezentat în figura 11.7.

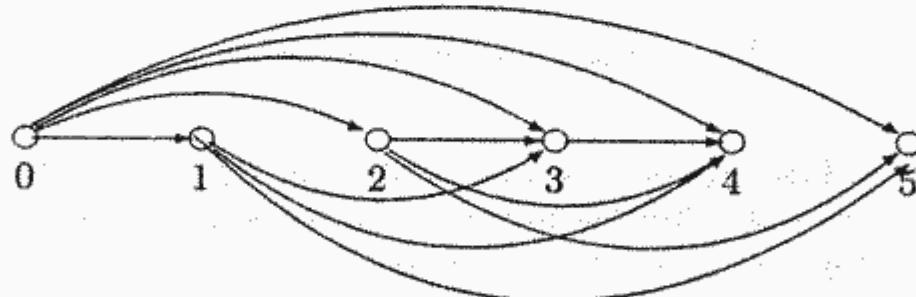


Figura 11.7: Digraful asociat secvenței $a = (3, 1, 4, 6, 3)$

slex

Se consideră următoarea matrice de costuri pe arce (ponderi):

$$c[i, j] = \begin{cases} 1 & , \text{dacă } (i < j \text{ și } a_i \leq a_j) \text{ sau } (i = 0), \\ -\infty & , \text{altfel.} \end{cases}$$

Notăm cu $\text{SCM}(i)$ subproblema determinării lungimii celui mai lung drum care se termină în vârful i , echivalent, subproblema determinării subsecvenței crescătoare

maximale ce are pe ultimul loc pe $a[i]$. Fie $L[i]$ valoarea optimă pentru $SCM(i)$. Dacă $i = 0$, atunci $L[0] = 0$. Presupunem $i > 0$. Fie j sursa ultimului arc de pe drumul care dă valoarea optimă $L[i]$. Evident, $j < i$. Aplicând principiul de optim, politica optimă corespunzătoare stării $SCM'(j)$ este subpolitica a politicii optime corespunzătoare stării $SCM'(i)$. Rezultă $L[i] = L[j] + c[j, i]$. Deci $L[i]$ satisfacă următoarea relație de recurență:

$$L[i] = \begin{cases} 0 & , \text{ dacă } i = 0, \\ \max_{j < i} L[j] + c[j, i] & , \text{ dacă } i > 0. \end{cases} \quad (11.5)$$

Pentru a determina o secvență maximală din tabelul valorilor optime, se procedează într-un mod asemănător cu determinarea drumului optim din matricea ce memorează penultimul vârf de pe fiecare drum optim. Aici nu este nevoie de memorarea penultimului vârf, pentru că el poate fi determinat din tabelul valorilor optime. Se procedează în modul următor:

- se determină j cu $L[j] = \max_i L[i]$;
- $k \leftarrow L[j]; s[k] \leftarrow a[j]$;
- următorul proces se repetă până când k devine 1:
 - se determină j_1 cu $j_1 < j$, $a[j_1] \leq a[j]$ și $L[j_1] = L[j] - 1$;
 - $k \leftarrow k - 1; j \leftarrow j_1; s[k] \leftarrow a[j]$.

11.6.3 Implementare

Pentru implementare nu este necesară construirea efectivă a digrafului. Acesta a fost utilizat numai pentru găsirea și verificarea corectitudinii algoritmului.

```

procedure scm(a, n, sm, lg)
  L[1] ← 1
  for i ← 2 to n do
    k ← 0
    for j ← 1 to i-1 do
      if ((a[j] ≤ a[i]) and (L[j] > k))
        then k ← L[j]
    L[i] ← k+1
    lg ← L[1]
    j ← 1
    for i ← 2 to n do
      if (L[i] > lg)
        then lg ← L[i]
        j ← i
    sm[lg] ← a[j]
    for i ← lg-1 downto 1 do
      k ← j-1
      while ((L[k]+1 ≠ L[j]) or (a[k] > a[j])) do
        k ← k-1
      sm[i] ← a[k]
      j ← k
  end

```

Evaluare. Prima parte, care determină lungimile subsecvențelor crescătoare maxime care se termină în i , pentru $i = 1, \dots, n$, se realizează în timpul $O(n^2)$. A doua etapă, care determină secvența optimă utilizând tabelul lungimilor optime, necesită timpul $O(n)$. Rezultă că întreg algoritmul are complexitatea timp $O(n^2)$. Memorarea secvenței inițiale, subsecvenței crescătoare maxime și a lungimilor utilizează $O(n)$ spațiu.

Observație. Performanța timp a programului DetSCM2 poate fi îmbunătățită prin utilizarea suplimentară a unui tablou p și completând acest tablou în prima parte punând $p[j] = k$, unde k dă maximul. Acest tablou poate fi utilizat la determinarea subsecvenței crescătoare maxime în timpul $O(n)$. Utilizarea tabloului p este similară cu cea a tabloului bidimensional p de la drumurile minime. sfobs

11.7 Distanța între siruri

11.7.1 Descrierea problemei

Se consideră două siruri $\alpha = a_1 \dots a_n$ și $\beta = b_1 \dots b_m$ formate cu litere dintr-un alfabet A . Asupra sirului α se pot face următoarele operații:

- *Stergere*: $S(i)$ șterge litera de pe poziția i ;
- *Inserare*: $I(i, c)$ inserează litera c pe poziția i ;
- *Modificare*: $M(i, c)$ înlocuiește litera de pe poziția i cu c .

Problema constă în determinarea unei secvențe de operații de lungime minimă care transformă pe α în β .

Exemplu. Fie $\alpha = carnet$, $\beta = paleta$. O secvență de transformări este $carnet \rightarrow parnet \rightarrow palnet \rightarrow palet \rightarrow paleta$. Transformările efectuate sunt $M(1, p)$, $M(3, l)$, $S(4)$ și $I(6, a)$. Există o altă secvență mai scurtă? sfex

11.7.2 Modelul matematic

Mai întâi dovedim câteva proprietăți ale secvențelor optime de transformări care duc pe α în β .

Lema 11.1. Fie $s = (\dots, T(i, \bullet), T'(j, \bullet), \dots)$ o secvență optimă (de lungime minimă) care transformă pe α în β . Atunci există k, ℓ , astfel încât secvența $s' = (\dots, T'(k, \bullet), T(\ell, \bullet), \dots)$, obținută din s prin interschimbarea celor două operații T și T' și în rest rămânând neschimbată, este de asemenea o secvență optimă care transformă pe α în β .

Demonstrație. Distingem următoarele cazuri.

1. $T(i, \bullet) = S(i)$, $T'(j, \bullet) = S(j)$. Nu contează ordinea în care sunt șterse două litere. k și ℓ sunt pozițiile de ștergere actualizate.
2. $T(i, \bullet) = S(i)$, $T'(j, \bullet) = I(j, c)$. Dacă $i > j$, atunci $T'(k, \bullet) = I(j, c)$ și $T(\ell, \bullet) = S(i+1)$. Dacă $i \leq j$, atunci $T'(k, \bullet) = I(j+1, c)$ și $T(\ell, \bullet) = S(i)$.
3. $T(i, \bullet) = S(i)$, $T'(j, \bullet) = M(j, c)$. Analog cazului 2.

4. $T(i, \bullet) = I(i, c), T'(j, \bullet) = S(j)$. Dacă $i \neq j$, atunci se procedează ca la 2. Dacă $i = j$, atunci, datorită faptului că execuția succesivă a operațiilor $I(i, c), S(i)$ lasă neschimbăt sirul, rezultă că secvența de operații nu ar fi optimă.
5. $T(i, \bullet) = I(i, c), T'(j, \bullet) = M(j, c)$. Analog cu 2.
6. $T(i, \bullet) = I(i, c), T'(j, \bullet) = I(j, c)$. Nu are importanță ordinea în care sunt inserate două litere. k și ℓ sunt pozițiile de inserare actualizate.
- Celelalte cazuri, când $T(i, \bullet) = M(i, c)$ se tratează în mod asemănător. sfdem

Corolar 11.2. Există o secvență optimă care efectuează transformările:

$$\alpha = \alpha_0 \mapsto \alpha_1 \mapsto \dots \mapsto \alpha_t = \beta$$

atunci pentru orice i , $|\alpha_i| \leq |\beta|$, unde prin $|_\cdot|$ am notat lungimea sirului $__$.

Demonstrație. Primele transformări care se fac sunt ștergerile. sfdem

Notăm cu $d(\alpha, \beta)$ lungimea unei secvențe optime.

Lema 11.2. Are loc:

- i) $d(\alpha, \alpha) = 0$;
- ii) $d(\alpha, \beta) = d(\beta, \alpha)$;
- iii) $d(\alpha, \gamma) \leq d(\alpha, \beta) + d(\beta, \gamma)$.

Demonstrație. i) este evidentă. Proprietatea ii) rezultă din faptul că orice operație $T : \alpha \rightarrow \beta$ are inversă $T^{-1} : \beta \rightarrow \alpha$:

$$\begin{aligned} S^{-1}(i) &= I(i, a_i) \\ I^{-1}(i, c) &= S(i) \\ M^{-1}(i, c) &= M(i, a_i) \end{aligned}$$

iii) rezultă din condiția de minim. sfdem

Observație. Lema de mai sus arată că $d(\alpha, \beta)$ este o metrică. De aici și numele problemei. Această metrică mai este numită și *distanță Levenshtein*, după numele matematicianului rus care a proiectat algoritmul în 1965. sfobs

În continuare aplicăm metoda programării dinamice pentru determinarea secvenței optime. Definim drept stare subproblemă $DS(\alpha_i, \beta_j)$ corespunzătoare transformării subșirului $\alpha_i = a_1 \dots a_i$ în $\beta_j = b_1 \dots b_j$ și prin $d[i, j]$ valoarea optimă $d(\alpha_i, \beta_j)$. Dacă $i = 0$, α_0 este sirul vid și β_j se obține prin j inserări: deci $d[0, j] = j$. Dacă $j = 0$, atunci β_j se obține prin i ștergeri și avem $d[i, 0] = i$. Presupunem $i, j > 0$. Considerăm decizia optimă prin care starea $DS(a_1 \dots a_i, b_1 \dots b_j)$ este transformată într-o stare $DS(a_1 \dots a_{i'}, b_1 \dots b_{j'})$ cu $(i' < i$ și $j' \leq j)$ sau $(i' \leq i$ și $j' < j)$.

Distingem următoarele situații:

1. Dacă $a_i = b_j$, atunci $i' = i - 1, j' = j - 1$ și, aplicând principiul de optim, obținem $d[i, j] = d[i - 1, j - 1]$.
2. $DS(a_1, \dots, a_i, b_1, \dots, b_j)$ se obține prin ștergere. Rezultă $i' = i - 1, j' = j$ și, aplicând principiul de optim, obținem $d[i, j] = d[i - 1, j] + 1$.

3. $DS(a_1, \dots, a_i, b_1, \dots, b_j)$ se obține prin inserare. Avem $i' = i, j' = j + 1$ și, aplicând principiul de optim, obținem $d[i, j] = d[i, j - 1] + 1$. Din corolarul lemei precedente rezultă că această operație poate fi realizată numai dacă $i < j$.
4. $DS(a_1, \dots, a_i, b_1, \dots, b_j)$ se obține prin modificare. Avem $i' = i - 1, j' = j - 1$ și, aplicând principiul de optim, obținem $d[i, j] = d[i - 1, j - 1] + 1$.

Deoarece $d[i, j]$ trebuie să fie minimă, rezultă:

$$d[i, j] = \min\{d[i - 1, j] + 1, d[i - 1, j - 1] + \delta, d[i, j - 1] + 1\}$$

unde

$$\delta = \begin{cases} 0 & , \text{dacă } a_i = b_j \\ 1 & , \text{dacă } a_i \neq b_j \end{cases}$$

În continuare vom utiliza valorile $d[i, j]$ la determinarea secvenței optime. Procedeul este asemănător cu cel de la subsecvența maximală, numai că acum se caută într-un tablou bidimensional. Notăm cu L lista care înregistrează transformările soluției optime. Se procedează în modul următor:

- inițial se consideră $i \leftarrow n; j \leftarrow n$; și $L \leftarrow ()$ (lista vidă);
- următorul proces se repetă până când i și j devin 0:
 - dacă $d[i, j] = d[i - 1, j - 1]$, atunci L rămâne neschimbată și se face $i \leftarrow i - 1; j \leftarrow j - 1$;
 - altfel, dacă $d[i, j] = d[i - 1, j - 1] + 1$, atunci se face $L \leftarrow M(i, b_j) @ L$ și $i \leftarrow i - 1; j \leftarrow j - 1$;
 - altfel, dacă $d[i, j] = d[i - 1, j] + 1$, atunci se face $L \leftarrow S(i) @ L$ și $i \leftarrow i - 1$;
 - altfel, dacă $d[i, j] = d[i, j - 1] + 1$, atunci se face $L \leftarrow I(i, b_j) @ L$ și $j \leftarrow j - 1$.

Procedura care determină distanțele este:

```

procedure distSir(a, b, n)
  for j ← 1 to n do
    d[0, j] ← j
  for i ← 1 to n do
    d[i, 0] ← i
  for i ← 1 to n do
    for j ← 1 to n do
      δ ← if (a[i]=b[j]) then 0 else 1
      d[i, j] ← min{d[i - 1, j] + 1, d[i - 1, j - 1] + δ, d[i, j - 1] + 1}
  L ← listaVida()
  i ← n
  j ← n
  repeat
    if (d[i, j] = d[i-1,j-1])
      then i ← i-1
           j ← j-1
    else if (d[i, j] = d[i-1,j-1]+1)

```

```

        then adLaInc(L, ('M', i, b[j]))
            i ← i-1
            j ← j-1
        else if (d[i,j]=d[i-1,j]+1)
            then adLaInc(L, ('S', i))
                i ← i-1
            else adLaInc(L, ('I', i, b[j]))
                j ← j-1
    until ((i = 0) and (j = 0))
end

```

11.7.3 Implementare

Valorile $d[i, j]$ sunt memorate într-un tablou bidimensional și ordinea de calcul este linie cu linie, începând cu linia 1. Secvența de transformări va fi memorată într-o listă liniară abstractă.

Notăm că utilizarea programului de mai sus necesită o implementare a tipului abstract **Lista**.

Evaluarea algoritmului este simplă și conduce la următorul rezultat:

Teorema 11.1. Determinarea distanței optime și a secvenței optime care transformă un sir α într-un sir β , $|\alpha| = |\beta| = n$, se poate face în timpul $O(n^2)$.

11.8 Arbori binari de căutare optimali

11.8.1 Problema arborelui binar de căutare optimal

Fie $A = (a_1, a_2, \dots, a_n)$ o secvență de chei sortată crescător din care se construiește un arbore binar de căutare. Se consideră că operația de căutare se execută cu anumite frecvențe. Anume:

Notăm cu p_i probabilitatea de a fi căutat elementul $a_i, i = 1, \dots, n$ și cu P secvența (p_1, p_2, \dots, p_n) . De asemenea, notăm cu q_i probabilitatea de a fi căutat un element x cu proprietatea $a_i < x < a_{i+1}, i = 0, \dots, n$, unde am presupus $a_0 = -\infty, a_{n+1} = +\infty$. Secvența (q_0, q_1, \dots, q_n) va fi notată cu Q .

În aceste condiții avem:

$\sum_{i=1}^n p_i$ – probabilitatea căutărilor terminate cu succes;

$\sum_{i=0}^n q_i$ – probabilitatea căutărilor terminate fără succes

și

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1.$$

Pentru P și Q date, arborele binar de căutare optimal este cel pentru care operația de căutare oferă timp mediu minim.

11.8.2 Modelul matematic

Pentru a pune în evidență timpul mediu se consideră arborele binar de căutare completat cu o serie de noduri pendante corespunzătoare intervalelor de insucces. În figura 11.8, se prezintă un arbore binar de căutare și arborele completat cu nodurile pendante.

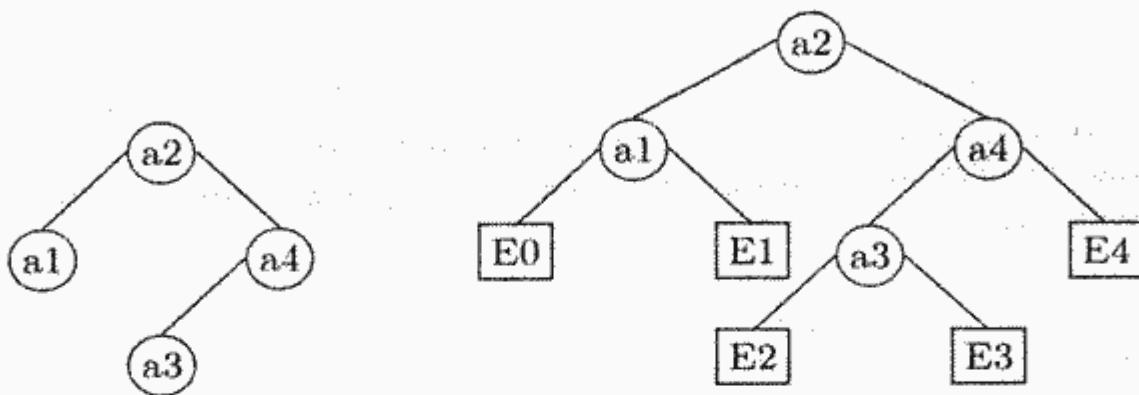


Figura 11.8: Arbore binar de căutare completat cu noduri pendante

Nodul E_i corespunde tuturor căutărilor fără succes pentru valori cuprinse în intervalul (a_i, a_{i+1}) . Timpul unei operații de căutare pentru un element x este dat de adâncimea nodului de valoare x sau de adâncimea pseudonodului corespunzător intervalului care-l conține pe x .

Costul unui arbore binar de căutare se definește ca timpul mediu al unei căutări pentru o valoare x , adică:

$$\text{cost}(T) = \sum_{i=1}^n p_i * \text{nivel}(a_i) + \sum_{i=0}^n q_i * (\text{nivel}(E_i) - 1) \quad (11.8.1)$$

Prin $\text{nivel}(a)$ se notează nivelul nodului a și reprezintă numărul de comparații efectuate de funcția de căutare pe drumul de la rădăcină până la nodul a . Ponderând acest număr cu probabilitatea de a fi căutat nodul a și efectuând suma pentru toate nodurile se obține timpul mediu de căutare cu succes a unui nod. În mod similar, se calculează timpul mediu de căutare pentru insucces, cu observația că din $\text{nivel}(E_i)$ se scade valoarea 1, deoarece decizia de apartenență la un interval nu se face la nivelul pseudonodului, ci la nivelul părintelui său.

Dată fiind secvența de valori A , se pot construi o mulțime de arbori binari de căutare cu cheile nodurilor din A . Un arbore binar de căutare optimal este arborele de cost minim. Construcția unui astfel de arbore are la bază modelul programării dinamice. Programarea dinamică reduce numărul de încercări eliminând o serie de secvențe care nu pot fi optimale și aceasta apelând la principiul optimalității:

Din punctul de vedere al programării dinamice, construirea unui arbore constă într-un sir de decizii privind nodul care se alege ca rădăcină la fiecare pas. Fie $c_{i,j}$ costul construirii unui arbore binar de căutare optimal care are vârfurile din secvența $(a_{i+1}, a_{i+2}, \dots, a_j)$ și pseudovârfurile din secvența $(e_i, e_{i+1}, \dots, e_j)$.

Lema 11.3. Matricea $C = (c_{i,j})_{i,j=0,\dots,n}$ poate fi calculată cu ajutorul recurenței

$$c_{i,j} = \begin{cases} \min_{i+1 \leq k \leq j} \{c_{i,k-1} + c_{k,j} + w_{i,j}\}, & i \neq j \\ 0, & i = j. \end{cases} \text{ unde } w_{i,j} = q_i + \sum_{l=i+1}^j (p_l + q_l) \quad 11.8.2$$

Demonstrație. Fie $(a_1, a_2, \dots, a_r, \dots, a_n)$ secvența de noduri sortată crescător. Presupunem că în primul pas se alege ca rădăcină nodul a_r .

Aplicând principiul optimalității, subarborele stâng L este optimal și construit cu secvențele $(a_1, a_2, \dots, a_{r-1})$ și $(e_0, e_1, \dots, e_{r-1})$:

$$\text{cost}(L) = \sum_{i=1}^{r-1} p_i * \text{nivel}(a_i) + \sum_{i=0}^{r-1} q_i * (\text{nivel}(e_i) - 1)$$

Analog, subarborele drept R este optimal și construit cu secvențele $(a_{r+1}, a_{r+2}, \dots, a_n)$ și $(e_r, e_{r+1}, \dots, e_n)$:

$$\text{cost}(R) = \sum_{i=r+1}^n p_i * \text{nivel}(a_i) + \sum_{i=r}^n q_i * (\text{nivel}(e_i) - 1)$$

Valorile `nivel()` sunt considerate în subarborii L, R .

Costul asociat arborelui T devine:

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + \sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^n p_i + \sum_{i=r}^n q_i.$$

Sumele $\sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^n p_i + \sum_{i=r}^n q_i$ reprezintă valorile suplimentare care apar datorită faptului că T introduce un nivel nou.

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + q_0 + \sum_{i=1}^{r-1} (p_i + q_i) + q_r + \sum_{i=r+1}^n (p_i + q_i)$$

Dacă se utilizează notația $w_{i,j} = q_i + \sum_{l=i+1}^j (p_l + q_l)$, rezultă

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + w_{0,r-1} + w_{r,n} = \text{cost}(L) + \text{cost}(R) + w_{0,n}$$

În urma aplicării definiției lui $c_{i,j}$ se obține:

$$\text{cost}(T) = c_{0,n}, \text{ cost}(L) = c_{0,r-1} \text{ and } \text{cost}(R) = c_{r,n}$$

$$c_{0,n} = c_{0,r-1} + c_{r,n} + w_{0,n} = \min_{1 \leq k \leq n} \{c_{0,k-1} + c_{k,n} + w_{0,n}\}$$

Prin generalizare rezultă relația (11.8.2)

sfdem

Recurența (11.8.2) este utilizată pentru calcularea matricei de valori optime, care la rândul ei, va fi la baza construirii unui arbore binar de căutare optimă.

Teorema 11.2. *Arborele binar de căutare construit pe baza matricei C este optimă.*

Demonstrație. Relația (11.8.2) asigură faptul că $c_{0,n}$ este minimum între toate costurile arborilor binari de căutare construși cu secvența (a_1, a_2, \dots, a_n) . Astfel, arborele binar de căutare construit pe baza matricei C are cost minim deci este optimă.

sfdem

11.8.3 Implementare

Matricea C se poate calcula aplicând relația de recurență (11.8.2) pentru $i - j = 1$, apoi $j - i = 2, \dots, j - i = n$.

Pentru a pregăti informațiile necesare construirii efective a arborelui binar de căutare optimă, vor fi calculate și valorile $r_{i,j}$, unde $r_{i,j}$ semnifică indicele nodului rădăcină al subarborelui optimă format din secvența $(a_{i+1}, a_{i+2}, \dots, a_j)$.

Valorile inițiale sunt $c_{i,i} = 0, w_{i,i} = q_i, 0 \leq i \leq n$. În plus, se folosește relația $w_{i,j} = p_j + q_j + w_{i,j-1}$.

Algoritmul de calcul al arborelui optimă este următorul:

```

procedure abc_opt(c,r,w,p,q,n)
  for i=1 to n-1 do
    c_{i,i} ← 0; r_{i,i} ← 0; w_{i,i} ← q_i
    c_{i,i+1} ← q_i+p_{i+1}+q_{i+1}; r_{i,i+1} ← i+1; w_{i,i+1} ← q_i+p_{i+1}+q_{i+1}
    w_{n,n} ← q_n; r_{n,n} ← 0; c_{n,n} ← 0;
  for d=2 to n do
    for i=0 to n-d do
      j ← i+d
      w_{i,j} ← w_{i,j-1}+p_j+q_j
      fie k indicele pentru care se obține valoarea minimă pentru
        {c_{i,h-1}+c_{h,j}; h ∈ I=[i+1,j]} xxx
      c_{i,j} ← w_{i,j}+c_{i,k-1}+c_{k,j}
      r_{i,j} ← k
    end
  
```

Evaluare. Se observă ușor că timpul de execuție este $O(n^3)$.

11.9 Prezentarea formală a paradigmelor

11.9.1 Modelul matematic

Problemele ale căror soluții se pot obține prin programarea dinamică sunt probleme de optim. Un prototip de astfel de problemă este următorul:

Să se determine:

$$\text{optim } R(x_0, \dots, x_{m-1}) \quad (11.6)$$

după x_0, \dots, x_{m-1} , în condițiile în care acestea satisfac restricții de forma:

$$g(x_0, \dots, x_{m-1}) ? 0 \quad (11.7)$$

unde $? \in \{<, \leq, =, \geq, >\}$.

Prin *optim* înțelegem *min* sau *max*, iar ecuația 11.6 se mai numește și *funcție obiectiv*. Un alt prototip este furnizat de digrafurile ponderate, unde $R(x_0, \dots, x_{m-1})$ exprimă suma ponderilor arcelor x_0, \dots, x_{m-1} , iar restricțiile impun ca x_0, \dots, x_{m-1} să fie drum sau circuit cu anumite proprietăți.

Paradigma programării dinamice propune găsirea valorii optime prin luarea unui sir de decizii (d_1, \dots, d_n) , numit și *politică*, unde decizia d_i transformă starea (problemei) s_{i-1} în starea s_i , aplicând *principiul de optim* [BD62]:

Secvența de decizii optime (politica optimă) care corespunde stării s_0 are proprietatea că după luarea primei decizii, care transformă starea s_0 în starea s_1 , secvența de decizii (politica) rămasă este optimă pentru starea s_1 .

Prin stare a problemei înțelegem, de fapt, o subproblemă. Unei stări s îi asociem o valoare z și definim $f(z)$, astfel încât, dacă starea s corespunde problemei inițiale, atunci:

$$f(z) = \text{optim } R(x_0, \dots, x_{m-1})$$

Principiul de optim conduce la obținerea unei ecuații funcționale de forma [BD62]:

$$f(z) = \underset{y}{\text{optim}} [H(z, y, f(T(z, y)))] \quad (11.8)$$

unde:

- s și s' sunt două stări cu proprietatea că una se obține din cealaltă aplicând decizia d ;
- z este valoarea asociată stării s ;
- $T(z, y)$ calculează valoarea stării s' , iar
- H exprimă algoritmul de calcul al valorii $f(z)$ dat de decizia d .

Relația de mai sus poate fi interpretată astfel: dintre toate deciziile care se pot lua în starea s (sau care conduc la starea s), se alege una care dă valoarea optimă în condițiile în care politica aplicată în continuare (sau până atunci) este și ea optimă.

Relația 11.8 poate fi dovedită utilizând inducția și reducerea la absurd. Deoarece este posibil ca principiul de optim să nu aibă loc pentru anumite formulări, este necesară verificarea sa pentru problema supusă rezolvării. Rezolvarea ecuațiilor recurente 11.8 conduce la determinarea unui sir de decizii ce în final constituie politica optimă prin care se determină valoarea funcției obiectiv.

11.9.2 Implementare

Principul de optim implică proprietatea de substructură optimă a soluției, care afirmă că *soluția optimă a problemei include soluțiile optime ale subproblemelor sale*. Această proprietate este utilizată la găsirea soluțiilor corespunzătoare stărilor optime. În general, programele care implementează modelul dat de programarea dinamică au două părți:

1. În prima parte, se determină valorile optime date de sirul de decizii optime, prin rezolvarea ecuațiilor 11.8. Presupunem că pentru exemplul rețelei de numere valorile sunt memorate într-un tablou unidimensional f .
2. În partea a doua se construiesc soluțiile (valorile x_i care dă optimul) corespunzătoare stărilor optime pe baza valorilor calculate în prima parte, utilizând proprietatea de substructură optimă.

Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime. Dacă în procesul de descompunere problemă \rightarrow subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de câte ori apare. Este mult mai convenabil ca valorile optime corespunzătoare subproblemelor să fie memorate într-un tablou și apoi combinate pe baza ecuațiilor 11.8 pentru a obține valoarea optimă a unei supraprobleme. În acest fel, orice subproblemă este rezolvată o singură dată (aici este una dintre diferențele dintre programarea dinamică și strategia *divide-et-impera* unde o aceeași subproblemă poate fi calculată de mai multe ori), iar determinarea valorilor optime se face de la subproblemele mai mici la cele mai mari (*bottom-up*). Prin memorarea valorilor optime într-un tablou, regăsirea oricărei dintre ele se face în timpul $O(1)$ ceea ce conduce la un timp scurt de calcul pentru optimul unei supraprobleme. Toate aceste observații sunt strâns legate de metoda de derecursivare din subsecțiunea 3.4. Pentru exemplul cu numerele aşezate în rețea valorile corespunzătoare drumurilor optime vor fi memorate într-un tablou unidimensional și ordinea de calcul va fi de la baza triunghiului spre vârf, adică în ordinea descrescătoare a indicilor.

Complexitatea algoritmului care calculează valorile optime depinde direct de tipul de recursivitate implicat de recurențele rezultate prin aplicarea principiului de optim. Dacă rezultă o recursivitate liniară atunci știm de la tehniciile de derecursivare (subsecțiunea 3) că valorile optime ale subproblemelor pot fi calculate în timp liniar și memorate într-un tablou unidimensional. În cazul recursiei în cascadă lucrurile sunt mai complicate. Dacă adâncimea arborelui corespunzător apelurilor recursive este n atunci rezultă un număr de 2^n de subprobleme de rezolvat. În unele cazuri, o redefinire a noțiunii de stare poate conduce la obținerea unei recursii liniare obținându-se astfel o reducere drastică a complexității – de la exponențial la polinomial. De asemenea, o reducere la complexitate polinomială (de cele mai multe ori pătratică) se obține atunci când se poate aplica metoda de derecursivare cu memorarea rezultatelor pentru subprobleme în tabele (subsecțiunea 3.4). Pentru problemele dificile (de exemplu, cele NP-complete) astfel de posibilități nu există. Pentru aceste cazuri se caută un mod de memorare cât mai compactă a valorilor optime pentru subprobleme și metode cât mai eficiente pentru calculul acestora astfel încât, pentru instanțele de dimensiuni rezonabile, determinarea soluției să se facă într-un timp acceptabil.

11.10 Exerciții

Exercițiul 11.1. Se consideră următoarea variantă a determinării LEP minime:

Se consideră date o listă de numere întregi $x^0 = (x_1^0, \dots, x_n^0)$ și un alt număr întreg M^0 . Acestea constituie starea inițială a problemei. O decizie corespunzătoare stării $((x_1, \dots, x_k), M)$ constă în extragerea a două numere x_i și x_j din lista $x = (x_1, \dots, x_k)$, introducerea în locul lor în listă a sumei $x_i + x_j$ și de asemenea adăugarea acestei sume la M . Se obține o nouă stare $\langle x', M' \rangle$ unde $x' = x \setminus (x_i, x_j) \cup (x_i + x_j)$ și $M' = M + x_i + x_j$. Problema constă în determinarea unei secvențe de decizii care conduce la starea finală $\langle (x_1^0 + \dots + x_n^0), M \rangle$ cu M minim.

Notăm cu $\text{LEPOPT}(\langle x, M \rangle)$ valoarea sumei din starea finală dată de soluția optimă.

1. Să se arate că¹:

$$\text{LEPOPT}(\langle x, M \rangle) = \min \{ \text{LEPOPT}(\langle x \setminus (u, v) \cup (u + v), M + (u + v) \rangle) \mid u, v \text{ apar ca elemente distințe în } x \} \quad (11.9)$$

2. Utilizând 11.9 să se calculeze $\text{LEPOPT}(\langle (5, 8, 3, 11), 0 \rangle)$. Valorile stărilor intermedii vor fi memorate într-un tabel. Apoi să se deducă din acest tabel secvența de decizii optime.
3. Să se proiecteze un algoritm bazat pe paradigma programării dinamice care determină soluția optimă.
4. Să se arate că stările date de algoritmul greedy satisfac:

$$\text{LEPOPT}(\langle x, M \rangle) = \text{LEPOPT}(\langle x \setminus (u, v) \cup (u + v), M + (u + v) \rangle) \quad (11.10)$$

unde $u = \min x$ și $v = \min(x \setminus (u))$.

5. Explicați diferențele și asemănările dintre algoritmul greedy și cel dat de programarea dinamică.

Exercițiul 11.2. Următorul algoritm, cunoscut sub numele de algoritm Bellman-Ford, determină drumurile minime într-un digraf ponderat $D = ((V, A), \ell)$ care pleacă dintr-un vârf i_0 dat. Pentru fiecare vârf i , $d[i]$ va fi lungimea drumului minim de la i_0 la i și $p[i]$ va fi predecesorul lui i pe drumul minim de la i_0 la i .

```

procedure BellmanFord(D, i0, d, p)
    for i ← 1 to n do
        p[i] ← 0
        d[i] ← ∞
    d[i0] ← 0
    for k ← 1 to n-1 do
        for each  $(i, j) \in A$  do
            if  $(d[j] > d[i] + \ell[i, j])$ 

```

¹Aici privim listele ca reprezentări ale multimapelor. Într-o multimapă U , un element u poate apărea de mai multe ori. Astfel, o multimapă U peste S poate fi definită ca o funcție $U : S \rightarrow \mathbb{N}$, unde $U(a)$ reprezintă de câte ori apare elementul $a \in S$ în U . Operațiile peste multimapă se definesc în mod natural. De exemplu, $a \in U \iff U(a) > 0$, $(U \cup V) : S \rightarrow \mathbb{N}$ este dată prin $(U \cup V)(a) = U(a) + V(a)$ etc.

```

        then  $d[j] \leftarrow d[i] + \ell[i, j]$ 
         $p[j] \leftarrow i;$ 
    for each  $\langle i, j \rangle \in A$  do
        if ( $d[j] > d[i] + \ell[i, j]$ )
            then throw 'există drum de lungime negativă'
    end

```

Se cere:

1. Să se definească noțiunea de stare (subproblemă) pentru problema drumurilor minime cu sursă precizată.
2. Să se aplique principiul de optim pentru a obține relația de recurență.
3. Să se precizeze ordinea în care sunt rezolvate subproblemele.
4. Să se arate că, dacă digraful D nu are circuite negative, atunci algoritmul Bellman-Ford determină corect drumurile minime care pleacă din i_0 .
5. Să se determine complexitatea algoritmului Bellman-Ford.

Exercițiul 11.3. (*Înmulțirea optimă a unui sir de matrici*.) Se consideră un sir (A_1, \dots, A_n) de matrici, unde A_i este de dimensiune $p_i \times p_{i+1}$. Se dorește calcularea produsului $A_1 A_2 \cdots A_n$, utilizând un subprogram de înmulțire a două matrici. Datorită asociativității, există mai multe posibilități de înmulțire a matricelor două câte două. Numărul posibilităților este egal cu numărul expresiilor complet parantezate. De exemplu, pentru $n = 4$ există cinci posibilități. Înmulțirea a două matrici de dimensiuni $p \times q$ și respectiv $q \times r$ necesită pqr înmulțiri. Problema constă în determinarea unei ordini de înmulțire a matricelor două câte două (echivalent, a unei expresii complet parantezate) care dă numărul minim de înmulțiri.

1. Pentru $1 \leq i \leq j \leq n$, se notează cu $m[i, j]$ numărul minim de înmulțiri necesare pentru a calcula $A_i \cdots A_j$. Să se arate că:

$$m[i, j] = \begin{cases} 0 & , \text{dacă } i = j \\ \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + p_i p_{k+1} p_{j+1} \} & , \text{dacă } i < j \end{cases}$$

2. Să se scrie un program care determină valorile $m[i, j]$. Care este complexitatea algoritmului descris de program?
3. Să se proiecteze o structură de date pentru reprezentarea unei ordini de înmulțire (unei expresii complet parantezate).
4. Să se scrie un program care, având la intrare valorile calculate la punctul 2, determină o ordine de înmulțire optimă.
5. Să se modifice algoritmul de la 2 astfel încât să determine simultan valorile $m[i, j]$, dar și ordinea de înmulțire optimă.

Exercițiul 11.4. Considerăm noțiunea de *subsecvență* în sensul definiției din 11.6.1.

1. Să se arate că problema calculului celei mai lungi subsecvențe comune a două secvențe $x = (x_1, \dots, x_m)$ și $y = (y_1, \dots, y_n)$ date este echivalentă cu determinarea distanței dintre siruri calculată considerând numai operațiile de inserare și ștergere.

2. Să se proiecteze un algoritm pentru determinarea celei mai lungi subsecvențe comune utilizând paradigma programare dinamică.

Exercițiul 11.5. [CLR93] Fie P un poligon convex în plan. O triangularizare a poligonului P este o mulțime T de diagonale ale lui P care împarte interiorul poligonului în triunghiuri cu interioarele disjuncte (de aici rezultă că diagonalele nu se intersectează). Laturile unui triunghi sunt laturi sau diagonale complete ale poligonului. Peste mulțimea tuturor triunghiurilor ce participă la cel puțin o triangularizare se consideră dată o funcție de ponderi w . Ca exemplu, se poate considera drept pondere a unui triunghi suma lungimilor laturilor sale. Să se scrie un program care să determine o triangularizare pentru care suma ponderilor este minimă.

Indicație. Presupunem $P = v_1 \dots v_n$ (v_i sunt vârfurile poligonului). Unei triangularizări îi putem asocia un arbore binar definit recursiv astfel: rădăcina arborelui este latura $v_1 v_n$. Fie $v_1 v_n v_i$ triunghiul din triangularizare care are pe $v_1 v_n$ ca latură. Dacă $v_1 v_i$ este diagonală, atunci subarborele aflat la stânga va fi cel corespunzător subpoligonului mărginit de $v_1 v_i$, aflat în partea opusă triunghiului și cu rădăcina $v_1 v_i$. Dacă $v_1 v_i$ este latură a poligonului (i.e., $i = 2$), atunci subarborele aflat la stânga va fi format numai din nodul $v_1 v_i$. Analog, dacă $v_n v_i$ este diagonală, atunci subarborele aflat la dreapta va fi cel corespunzător subpoligonului mărginit de $v_n v_i$, aflat în partea opusă triunghiului și cu rădăcina $v_n v_i$. Dacă $v_n v_i$ este latură a poligonului (i.e., $i = n - 1$), atunci subarborele aflat la dreapta va fi format numai din nodul $v_n v_i$. Un exemplu de construire a arborelui este arătat în figura 11.9. În continuare se raționează la fel ca la înmulțirea optimă a matricelor (exercițiul 11.3).

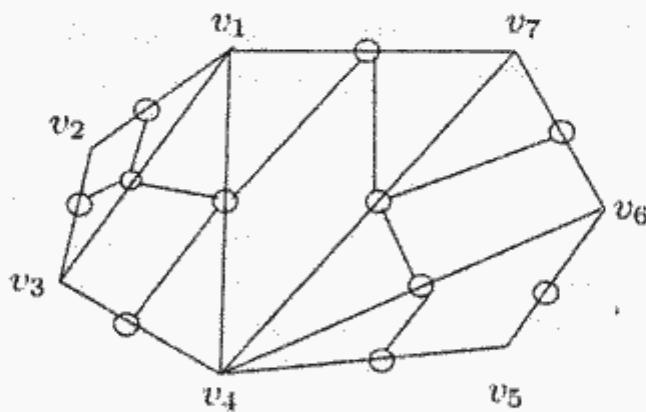


Figura 11.9: Triangularizare și arborele binar atașat

Exercițiul 11.6. Fie p_1, \dots, p_n n puncte în plan. Presupunem $p_i = (x_i, y_i)$ cu $x_i \neq x_j$, dacă $i \neq j$. Un *traseu monoton* este un drum care unește cel mai din stânga punct cu cel mai din dreapta și este format din segmente ce unesc puncte din $\{p_1, \dots, p_n\}$ și pot fi parcuse toate în același sens (de exemplu, de la stânga la dreapta). Un *tur bitonic* este format din reuniunea a două trasee monotone care conectează toate cele n puncte. Un exemplu de tur bitonic este dat în figura 11.10. Să se scrie un program care determină un tur bitonic de lungime minimă.

Indicație. Se vor explora punctele de la stânga la dreapta, menținându-se cele două trasee optime.

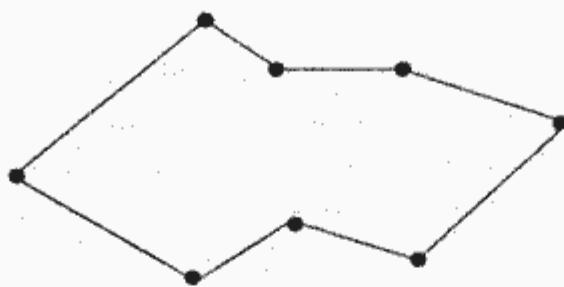


Figura 11.10: Tur bitonic

Exercițiu 11.7. [CLR93] Se consideră problema scrierii la imprimantă într-un mod estetic a unui paragraf. Textul de intrare este o secvență de n cuvinte w_1, \dots, w_n de lungimi ℓ_1, \dots, ℓ_n , respectiv. Lungimile cuvintelor sunt măsurate în caractere. Se dorește scrierea acestui paragraf într-un mod cât mai estetic, astfel încât fiecare linie conține cel mult M caractere. Criteriul prin care se măsoară „estetica” textului este următorul. Dacă o linie conține cuvintele de la w_i la w_j , atunci numărul de spații de la sfârșitul liniei este $M - j + i - \sum_{k=i}^j \ell_k$. Un text scris estetic minimizează suma, după toate liniile exceptând ultima, a cuburilor numerelor de spații extra de la fiecare sfârșit de linie. Să se scrie un program care scrie la imprimantă paragraful dat respectând acest criteriu de esteticitate descris mai sus. Se presupune că $M \leq 80$ și că un paragraf are cel mult 20 de linii.

Exercițiu 11.8. [CLR93] Când un terminal „intelligent” modifică o linie a unui text, înlocuind sirul „sursă” $x[1..m]$ cu sirul „destinație” $y[1..n]$, există câteva operații de bază prin care realizează această modificare:

1. un caracter al textului-sursă poate fi șters (**delete** c), înlocuit (**replace** c by c'), sau copiat (**copy** c) în textul-destinație;
2. un caracter este inserat în textul-destinație (**insert** c);
3. două caractere vecine din textul-sursă sunt interschimbate în timpul copierii în textul-destinație (**twiddle** cc' into $c'c$);
4. după efectuarea tuturor operațiilor de tipul celor de mai sus, astfel încât textul-destinație este complet, sufixul textului-sursă este șters (**kill** ...).

Considerăm ca exemplu modificarea cuvântului **algorithm** în **altruistic**:

Operație	text sursă	Text destinație
copy a	lgorithm	a
copy l	gorithm	al
replace g by t	orithm	alt
delete o	rithm	alt
copy r	ithm	altr
insert u	ithm	altru
insert i	ithm	altrui
insert s	ithm	altruis
twiddle it into ti	hm	altruisti
insert c	hm	altruistic
kill hm		altruistic

Fiecare operație $\neq \text{kill}$ are asociat un cost. Se presupune, de exemplu, că operația de înlocuire are costul mai mic decât suma costurilor operațiilor de stergere și de inserare. Costul unei operații de modificare este suma costurilor operațiilor individuale. Să se scrie un program care, având la intrare textele-sursă și destinație, determină secvența de operații care modifică sursa în destinație cu un cost minim.

Exercițiul 11.9. [CLR93] Profesorul McKenzie este consultat de președintele companiei A&B Company pentru a planifica o petrecere cu angajații companiei. Relația șef-subaltern din companie poate fi reprezentată printr-un arbore în care rădăcina este președintele companiei. La angajare, în urma unui interviu, fiecărui angajat i s-a atribuit un coeficient de „jovialitate”, care este un număr real. Pentru a face petrecerea mai nostimă, s-a decis să nu participe nici o perche formată dintr-un angajat și șeful lui direct.

1. Să se scrie un program care-l ajută pe profesor să planifice petrecerea astfel încât coeficientul total de jovialitate (= suma coeficienților individuali ai persoanelor care participă la petrecere) să fie maxim.
2. Să se modifice programul de la 1, astfel încât președintele companiei să participe la petrecerea companiei sale.

11.11 Referințe bibliografice

În scrierea acestui capitol am fost cel mai mult influențat de [CLR93, CLR00, HS84]. Mai pot fi consultate [LG86, Sed88, Sah98].

Capitolul 12

Backtracking și branch-and-bound

Backtracking și branch-and-bound sunt două strategii care îmbunătățesc căutarea exhaustivă. Un algoritm de căutare exhaustivă este definit după următoarea schemă: 1) se definește spațiul soluțiilor candidat (fezabile) U ; 2) cu ajutorul unui algoritm de enumerare se selectează acele soluții candidat care sunt soluții ale problemei. În contrast cu căutarea exhaustivă, paradigmele backtracking și branch-and-bound propun o căutare sistematică în spațiul soluțiilor. Ambele paradigmă au la bază următoarele ingrediente:

1. Spațiul soluțiilor este organizat ca un arbore cu rădăcină astfel încât există o corespondență bijectivă între vârfurile de pe frontieră arborelui și soluțiile candidat. Soluția candidat este descrisă de drumul de la rădăcină la vârful de pe frontieră corespunzător. Un exemplu de organizare a spațiului soluțiilor ca arbore este reprezentat în figura 12.1.
2. Căutarea sistematică se realizează cu ajutorul unui algoritm de explorare sistematică a acestui arbore.
3. Vâfurile arborelui sunt clasificate astfel:
 - a) un vârf *viabil* este un vârf pentru care sunt sănse să se găsească o soluție a problemei explorând subarborele cu rădăcina în acel vârf;
 - b) un vârf *activ (live)* este un vârf care a fost vizitat cel puțin o dată de algoritmul de explorare și urmează să mai fie vizitat cel puțin încă o dată;
 - c) un vârf activ după ce a fost vizitat ultima dată de algoritmul de explorare devine *inactiv (death)*.
4. Sunt explorate numai subarborii cu rădăcini viabile. În felul acesta se evită procesarea inutilă a subarborilor pentru care suntem siguri că nu conțin soluții.
5. Cele două paradigmă, backtracking și branch-and-bound, diferă doar prin modul în care explorează lista vâfurilor viabile din arbore.

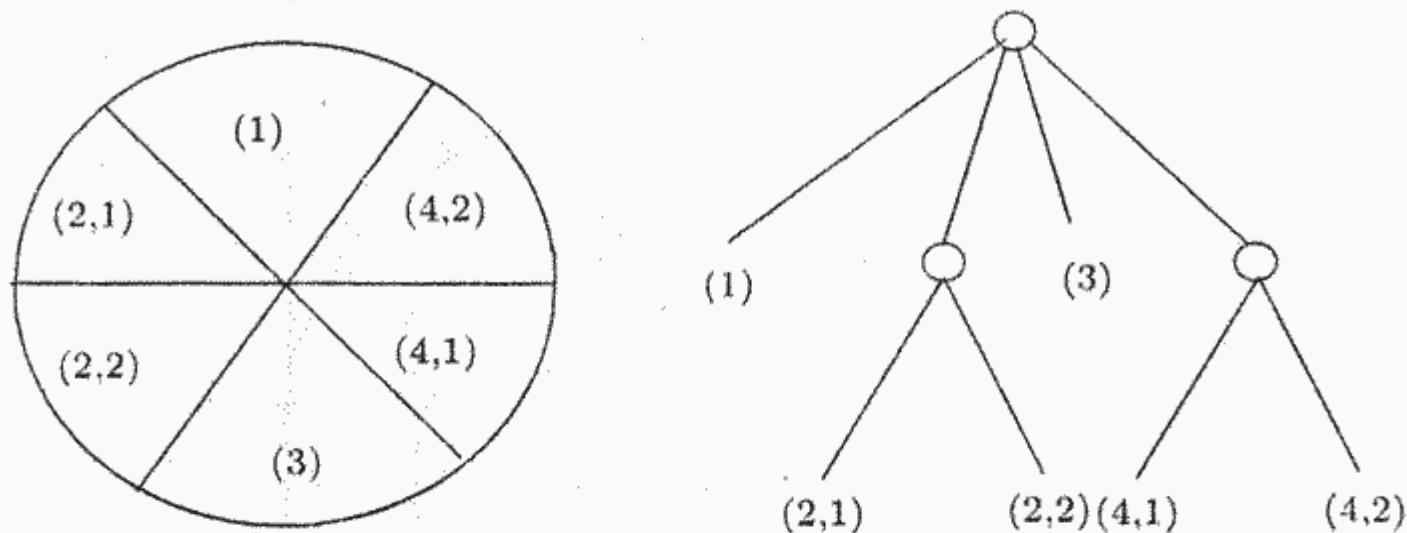


Figura 12.1: Spațiului soluțiilor ca arbore

12.1 Organizarea spațiului soluțiilor candidat

Metodele *backtracking* și *branch-and-bound* rezolvă o problemă prin căutarea sistematică a soluției în spațiul soluțiilor candidat. Conceptual, această căutare folosește o organizare a spațiului soluțiilor candidat sub forma unui arbore. Multimea elementelor produsului cartezian, submulțimile unei mulțimi, mulțimea permutărilor unei mulțimi și drumurile într-un graf sunt spațiile de căutare cele mai utilizate. Din acest motiv vom prezenta în secțiunile care urmează organizarea acestora sub formă de arbore.

12.1.1 Produsul cartezian

Considerăm mai întâi cazul în care toate elementele produsului cartezian au aceeași lungime. Fără să restrângem generalitatea, putem considera problema enumerării elementelor produsului cartezian $\{0, \dots, m-1\}^n = \{0, \dots, m-1\} \times \dots \times \{0, \dots, m-1\}$ (de n ori). Dimensiunea spațiului soluțiilor candidat este m^n . Mulțimea $\{0, \dots, m-1\}^n$ poate fi reprezentată printr-un arbore cu n nivele în care fiecare vârf intern are exact m succesiuni, iar vârfurile de pe frontieră corespund elementelor mulțimii. De exemplu, arborele corespunzător cazului $m = 2$ și $n = 3$ este reprezentat grafic în figura 12.2.

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

```
procedure enumProdCart(n, m)
    k ← 0
    x[0] ← -1
```

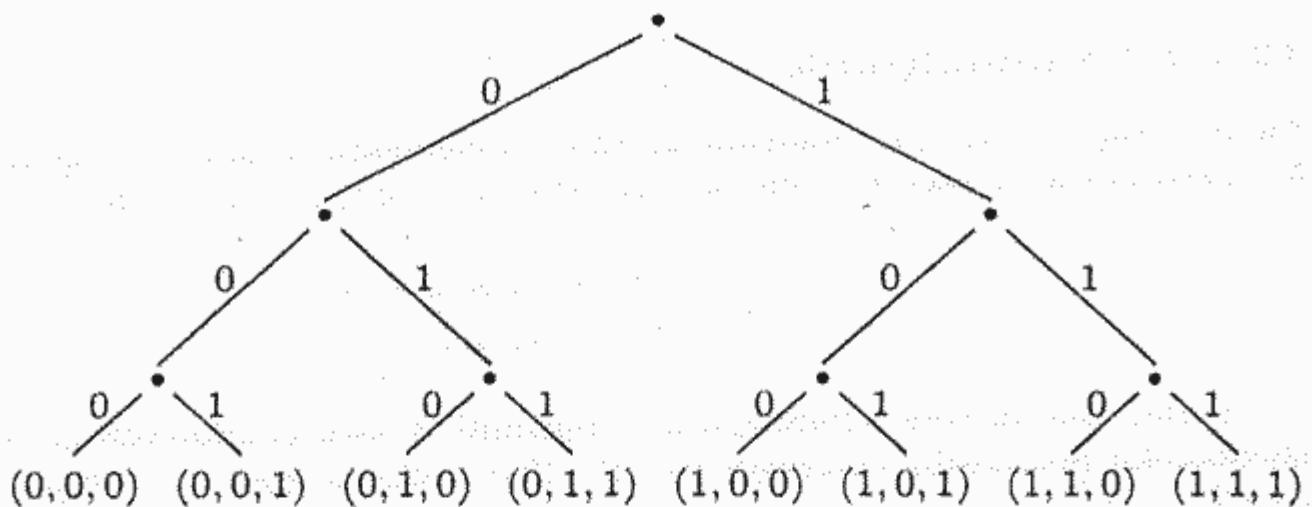


Figura 12.2: Arborele produsului cartezian

```

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 enumProdCart este următoarea:

```

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

```

Enumerarea tuturor elementelor produsului cartezian cu varianta recursivă se face executând apelul:

```
enumProdCartRec(x, 0)
```

Considerăm acum cazul când mulțimile care definesc produsul cartezian nu au același cardinal. Presupunem din nou, fără să restrângem generalitatea, că se dorește enumerarea elementelor produsului cartezian $\{0, \dots, m_1 - 1\} \times \dots \times \{0, \dots, m_n - 1\}$. Programul de enumerare se obține prin înlocuirea în programul corespunzător de mai sus a variabilei simple m cu componenta tablou $m[k]$, unde $m[k] = m_k$.

12.1.2 Submulțimile

Fie $A = \{a_0, \dots, a_{n-1}\}$ o mulțime finită cu n elemente. O submulțime $S \subseteq A$ poate fi reprezentată de vectorul său caracteristic $(x[i] \mid 0 \leq i < n)$, unde

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

Acum, enumerarea submulțimilor mulțimii A este echivalentă cu enumerarea elementelor produsului cartezian $\{0, 1\}^n$. Dimensiunea spațiului soluțiilor candidat este 2^n .

12.1.3 Permutările

Notăm cu S_n mulțimea permutărilor mulțimii $\{0, 1, \dots, n-1\}$. Dimensiunea spațiului soluțiilor candidat este $n! = 1 \cdot 2 \cdots n$.

12.1.3.1 Varianta I

Spațiul soluțiilor este reprezentat ca un arbore în care orice drum de la rădăcină la frontieră descrie o permutare. Arboarele pentru S_3 este reprezentat în figura 12.3. Deoarece orice permutare din S_n este un element al produsului cartezian $\{0, 1, \dots, n-1\}^n$ ce satisfac anumite restricții, rezultă că putem utiliza algoritmul backtracking descris în secțiunea 12.1.1. Un vârf $x[k]$ este *viabil*, dacă satisfac următoarea proprietate:

$$0 \leq x[k] \leq n-1 \wedge (\forall i) 0 \leq i < k \Rightarrow x[i] \neq x[k]$$

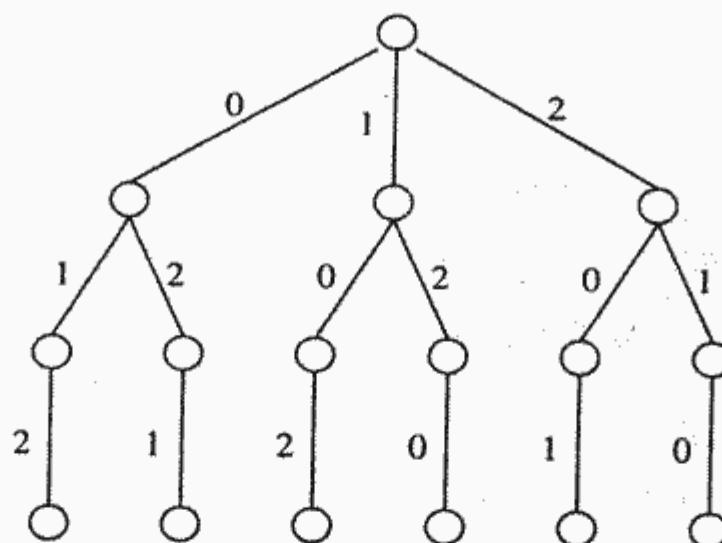
Verificarea condiției de mai sus este destul de costisitoare, $O(k)$. Deoarece această condiție este verificată pentru toate valorile candidat pentru $x[k]$, rezultă că timpul cel mai nefavorabil pentru determinarea unui $x[k]$ viabil este $O(k^2)$. Performanțele algoritmului pot fi îmbunătățite dacă se consideră un tablou suplimentar $(y[j] \mid 0 \leq j < n)$ cu semnificația:

$$y[j] = \begin{cases} 1, & \text{dacă există } i \text{ cu proprietatea } x[i] = j, \\ 0, & \text{altfel.} \end{cases}$$

Căutarea următorului $x[k]$ viabil se face incrementând $x[k]$ până când are loc condiția $y[x[k]] = 0$, după care $y[x[k]]$ devine 1. Decrementarea lui k este, de asemenea, precedată de atribuirea valorii 0 lui $y[x[k]]$.

12.1.3.2 Varianta II

Se bazează pe o altă reprezentare arborescentă a permutărilor. Cazurile practice în care această enumerare este utilizată ca bază pentru strategia backtracking sunt mai rare, dar enumerarea simplă a permutărilor este mai eficientă.

Figura 12.3: Permutări – varianta I ($n = 3$)

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

$$\begin{aligned} S_1 &= \{(0)\} \\ S_n &= \{(i_0, \dots, i_{n-2}, n-1), (i_0, \dots, n-1, i_{n-2}), \dots, (n-1, i_0, \dots, i_{n-2}) \mid \\ &\quad (i_0, \dots, i_{n-2}) \in S_{n-1}\} \end{aligned}$$

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

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

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

```

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

```

Enumerarea tuturor celor $n!$ permutări se realizează prin execuția următoarelor două instrucțiuni:

```
p[0] ← 0
enumPermRec(p, 0)
```

Enumerarea nerecursivă. Metodei recursive i se poate atașa un arbore ca în figura 12.4. 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. Este normal să fie aşa pentru că programul de enumerare a permutărilor realizează același lucru: parcurgerea mai întâi în adâncime a arborelui din figura 12.4. Deci și varianta nerecursivă a algoritmului de generare a permutărilor poate fi obținut din varianta nerecursivă a algoritmului DFS. Locul tabloului p din DFS este luat de o funcție $f(k, i, p)$ care pentru o permutare $(p[0], \dots, p[k-1])$ (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 părinte, 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 enumPerm(n)
    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
```

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

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

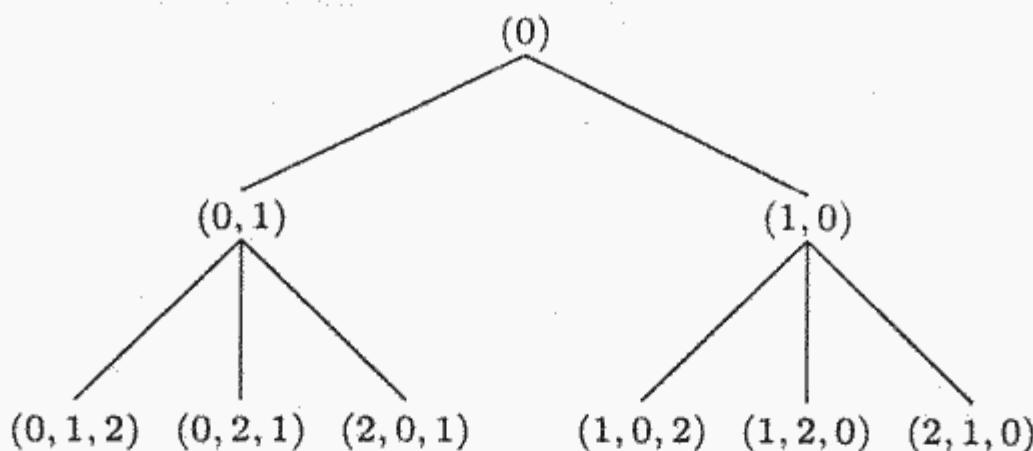


Figura 12.4: Arborele permutărilor generat de metoda recursivă

Exercițiul 12.1.1. Să se scrie un subprogram care, pentru numerele naturale i și n date, determină a i -a permutare (în ordinea lexicografică) din S_n .

Observație. Algoritmul de mai sus poate fi îmbunătățit din punctul de vedere al timpului de execuție. Mai întâi să notăm faptul că orice algoritm de enumerare a permutărilor are timpul de execuție $O(n!) = O(n^n)$. Ideea este de a găsi un algoritm care să efectueze cât mai puține operații pentru determinarea permutării succesoare. Execuția a $c' \cdot n!$ operații în loc de $c \cdot n!$, $c' < c$, semnifică, de fapt, o reducere cu $(c - c') \cdot n!$ a timpului de execuție. Un astfel de algoritm este obținut după cum urmează. În arborele din figura 12.4 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 transpoziție de poziții vecine. Dacă se reușește generarea permutărilor direct în această ordine, fără a simula parcurgerea arborelui, atunci se obține un program care generează permutările cu număr minim de operații. Regula generală prin care se obține această ordine este următoarea (figura 12.5):

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

Evitarea simulării parcurgerii arborelui se realizează prin utilizarea unui vector de „direcții”, $d = (d[k] | 0 \leq k < n)$, cu următoarea semnificație:

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

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

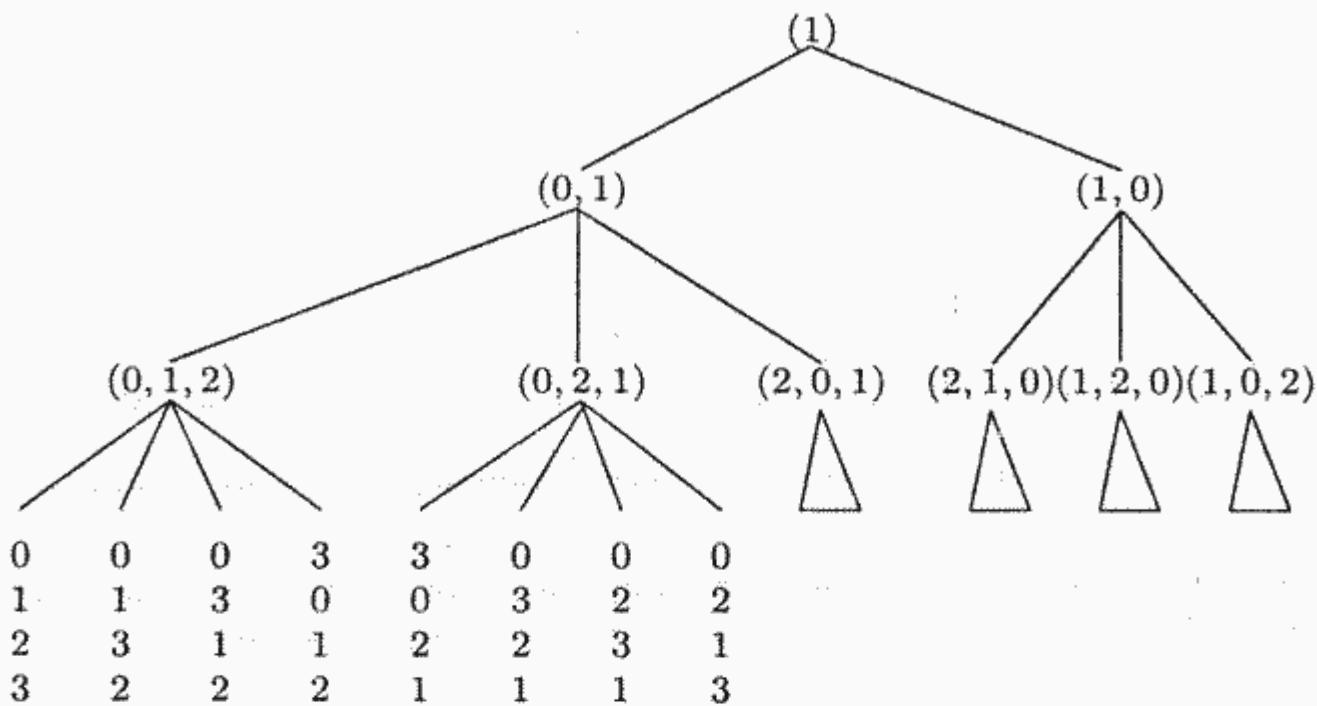


Figura 12.5: Arborele permuatărilor modificat

12.1.4 Drumurile într-un (di)graf

În multe probleme practice, spațiul soluțiilor candidat este dat de drumurile unui (di)graf. Arborele reprezentând drumurile care pleacă dintr-un vîrf fixat i_0 poate fi obținut prin inserarea liniei m: în algoritmul DFS

- varianta nerecursivă:

```

procedure drumuriDigraf(D, i0, viziteaza())
    for i ← 0 to D.n-1 do
        p[i] ← D.a[i].prim
        SB ← stivaVida()
        push(SB, i0)
        viziteaza(i0)
        while (not esteVida(SB)) do
            i ← top(SB)
            if (p[i] = NULL)
                then pop(SB)
                    /* se reia procesul de explorare a vecinilor */
            m:   p[i] ← D.a[i].prim
                  else j ← p[i]->elt
                      p[i] ← p[i]->succ
                      if j ∈ SB
                          then viziteaza(j)
                              push(SB, j)
    end

```

- varianta recursivă:

```

procedure drumuriDigrafRec(i);
    viziteaza(i)

```

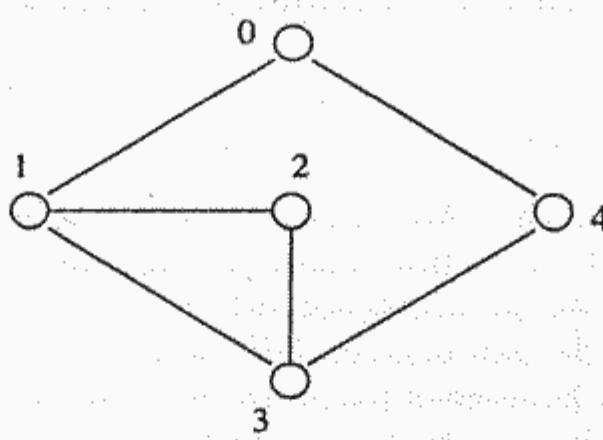


Figura 12.6: Graf

```

push(SB, i)
S[i] ← 1
for fiecare vârf j adiacent cu i do
    if (j ∈ SB) then drumuriDigrafRec(j)
pop(SB)
end
  
```

Dacă presupunem că graful din figura 12.6 este reprezentat ca digraf de liste de adiacență:

$$\begin{aligned}
 0 &\rightarrow (1, 4) \\
 1 &\rightarrow (0, 2, 3) \\
 2 &\rightarrow (1, 3) \\
 3 &\rightarrow (1, 2, 4) \\
 4 &\rightarrow (0, 3)
 \end{aligned}$$

atunci arborele care reprezintă toate drumurile ce pleacă din 0 este cel reprezentat în figura 12.7.

Un caz particular îl constituie labirinturile. De exemplu, labirintul din figura 12.8a poate fi reprezentat prin matricea:

$$a = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

care, la rândul ei, este reprezentată de graful din figura 12.8b.

Parcurea tuturor drumurilor din labirint care pleacă din $(0, 0)$ nu presupune construcția efectivă a grafului, ci utilizarea directă a matricei a . Convenim că în timpul parcurgerii să punem $a[i, j] = 2$ dacă și numai dacă (i, j) este în stivă (elementele egale cu 2 din matrice vor descrie stiva SB). Presupunem că listele

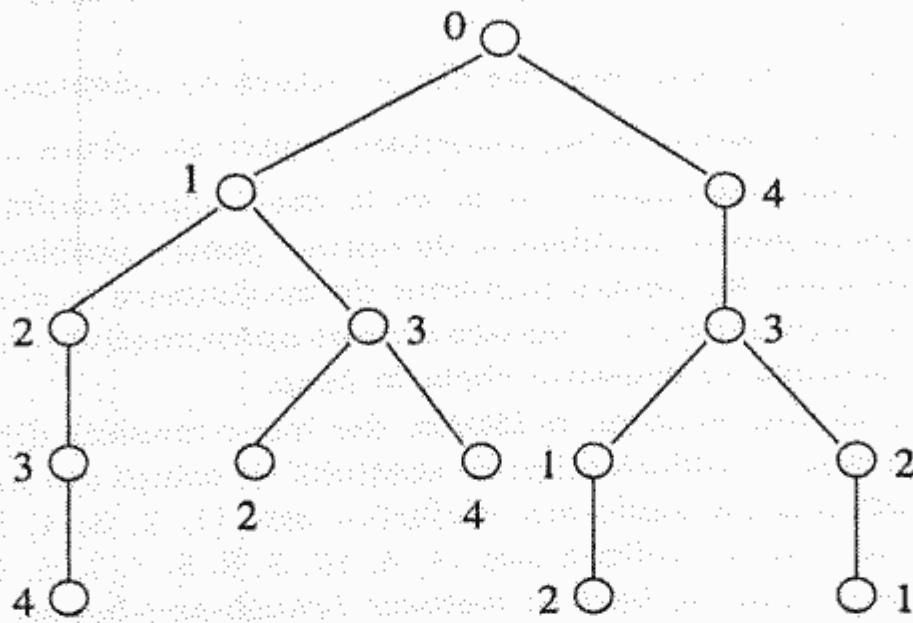


Figura 12.7: Arborele drumurilor din graful 12.6

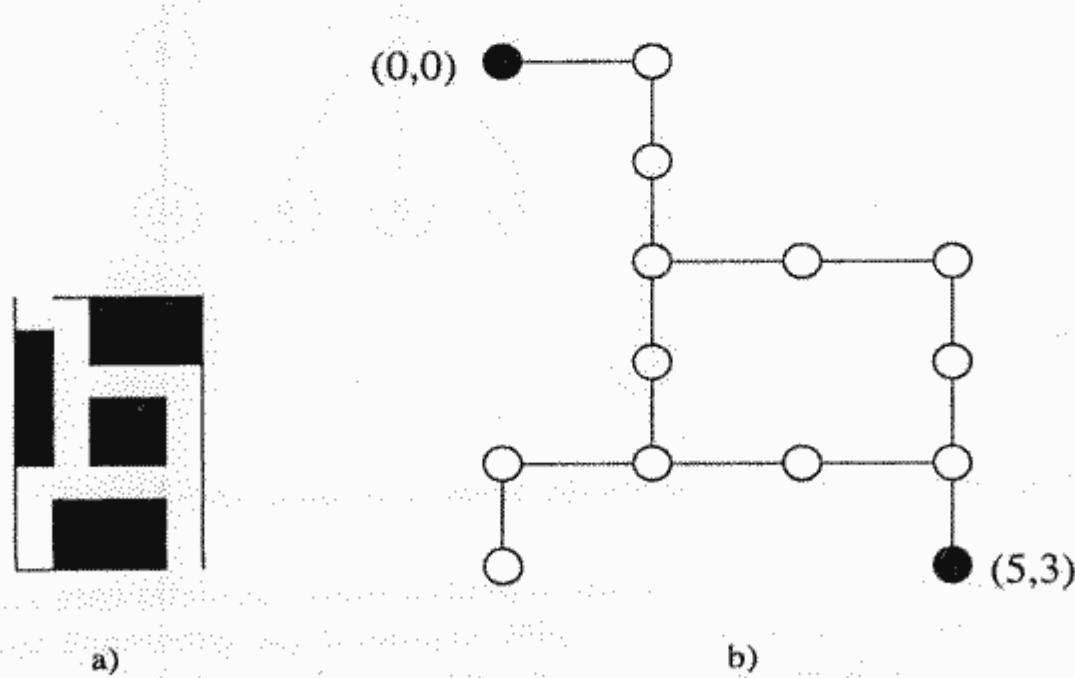


Figura 12.8: Labirint

de adiacență sunt reprezentate în ordinea parcurgerii vecinilor în sensul arcelor de ceasornic începând cu vecinul din dreapta. Cel mai simplu este să modificăm varianta recursivă, înlocuind bucla `for each` cu apelurile corespunzătoare vecinilor:

```

procedure drumuriLabirint(i,j);
    viziteaza(i,j)
    a[i,j] ← 2
    if (j < n-1 and a[i,j+1] = 0) then drumuriLabirint(i, j+1)
    if (i < m-1 and a[i+1,j] = 0) then drumuriLabirint(i+1, j)
    if (j > 0 and a[i,j-1] = 0) then drumuriLabirint(i, j-1)
    if (i > 0 and a[i-1,j] = 0) then drumuriLabirint(i-1, j)
    a[i,j] ← 0
end

```

12.2 Backtracking

Presupunem că o soluție candidat este soluție a problemei dacă satisfacă o anumită condiție, notată cu ST (eSTe), ce poate fi testată în timp polinomial. Putem privi ST ca o funcție $ST : \mathbb{U} \rightarrow Boolean$. Paradigma backtracking se bazează pe următoarele reguli:

1. se definește o funcție criteriu prin care se stabilește dacă un vârf este viabil sau nu;
2. arborele este explorat prin algoritmul DFS;
3. fie $x^k = (x_0, \dots, x_k)$ secvența care descrie drumul de la rădăcina la vârful curent;
4. dacă vârful curent este pe frontieră, atunci se verifică dacă x^k este soluție;
5. dacă vârful curent nu este pe frontieră, se alege următorul succesor viabil (dacă există).

Dacă $x^k = (x_0, \dots, x_k)$ este secvența care descrie drumul de la rădăcina la vârful curent, se notează cu $T(x^k)$ mulțimea tuturor valorilor posibile pentru x_{k+1} , astfel încât secvența $x^{k+1} = (x_0, x_1, \dots, x_k, x_{k+1})$ descrie de asemenea un drum de la rădăcină către o frunză.

Modelul general de algoritm backtracking în varianta nerecursivă are descrierea următoare:

```

procedure backtracking(n)
    k ← 0
    while (k ≥ 0) do
        if ( $\exists y \in T(x^k)$  neîncercat and viabil(y))
            then  $x_{k+1} \leftarrow y$ 
                  if  $ST(x^{k+1})$ 
                      then scrie( $x^{k+1}$ )
                      else k ← k+1
            else k ← k-1
    end

```

Varianta recursivă poate fi descrisă astfel:

```

procedure backtrackingRec( $x^k$ )
     $k \leftarrow 0$ 
    for each  $y \in T(x^k)$  neîncercat and viabil( $y$ ) do
         $x_{k+1} \leftarrow y$ 
        if  $ST(x^{k+1})$ 
            then scrie( $x^{k+1}$ )
        else backtrackingRec( $x^{k+1}$ )
    end

```

Dacă spațiul soluțiilor candidat este produsul cartezian $\{0, 1, \dots, m-1\}^n$, atunci algoritmii backtracking au următoarea descriere generală:

1. varianta nerecursivă:

```

procedure backtrack( $n, m$ )
     $k \leftarrow 0$ 
     $x[0] \leftarrow -1$ 
    while ( $k \geq 0$ ) do
        if ( $x[k] < m-1$ )
            then repeat
                 $x[k] \leftarrow x[k]+1$ 
            until(viabil( $x, k$ ) or ( $x[k]=m-1$ ))
        if (viabil( $x, k$ ))
            then if (( $k = n-1$ ) and  $ST(x)$ )
                then scrieElement( $x, n$ )
            else  $k \leftarrow k+1$ 
                 $x[k] \leftarrow -1$ 
        else  $k \leftarrow k-1$ 
    end

```

2. varianta recursivă:

```

procedure backtrackRec( $x, k$ )
    for  $j \leftarrow 0$  to  $m-1$  do
         $x[k] \leftarrow j$ 
        if (( $k = n-1$ ) and  $ST(x)$ )
            then scrieElement( $x, n$ )
        else if (viabil( $x, k$ ))
            then backtrackRec( $x, k+1$ )
    end

```

12.3 Branch-and-bound

Presupunem că spațiul soluțiilor candidat (fezabile) este S și, ca și în cazul paradigmelor *backtracking*, este organizat ca un arbore. Rădăcina arborelui corespunde problemei inițiale (căutării în S), iar fiecare vârf intern corespunde unei subprobleme a problemei inițiale. *Branch-and-bound* dezvoltă acest arbore conform următoarelor reguli:

1. Fiecare vârf viabil este explorat o singură dată.

2. Când un vârf viabil este explorat, toți fiile viabili sunt generați și memorati într-o structură de date de așteptare pe care o notăm cu A . Inițial, A include doar rădăcina.
3. Următorul vârf explorat este ales din structura de așteptare A .

Există trei moduri clasice de implementare pentru structura de așteptare:

- *Coadă*. Arborele este explorat la fel ca în cazul algoritmului BFS.
- *Heap*. Se aplică pentru problemele de optim: pentru probleme de minimizare se alege min-heap, iar pentru probleme de maximizare se alege max-heap. Se asociază o funcție *predictor* (valoare de cost aproximativă) pentru fiecare vârf și valoarea acesteia va constitui cheia în heap.
- *Stivă*. Arborele este explorat în lățime, dar într-o manieră diferită de BFS. De exemplu, dacă se consideră arborele spațiului de soluții din figura 12.9, atunci în cazul cozii ordinea explorării vârfurilor este $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$, iar în cazul stivei ordinea este $(1, 4, 10, 3, 9, 8, 7, 2, 6, 11, 5)$. Deși pare paradoxal, principiul de bază al metodei branch-and-bound este respectat și în cazul stivei: când un vârf viabil este explorat, toți fiile viabili sunt generați și memorati. Astfel, în momentul în care este explorat vârful 1, vârfurile 2, 3 și 4 sunt generate și memorate în stivă. Următorul vârf explorat este 4, situat în vârful stivei. Procesul continuă cu acest vârf.

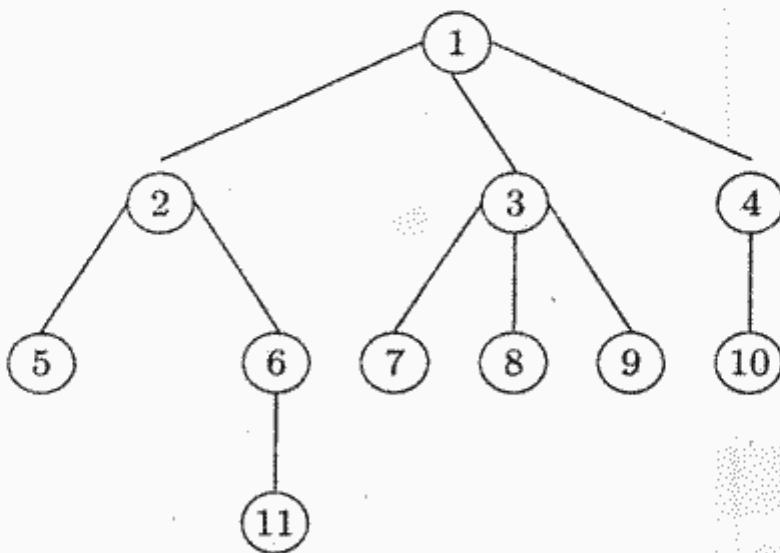


Figura 12.9: Arbore de soluții pentru *branch-and-bound*

Branch-and-bound este termenul generic pentru toate tehniciile care au în comun faptul că vârful curent explorat este complet expandat (se generează toți fiile și se memorează în structura de așteptare), după care se alege pentru explorare, după o strategie oarecare, un nou vârf. Procedura de mai sus se descrie algoritmic astfel:

```

procedure BranchAndBound()
  fie r rădăcina
  A ← {r}
  while A ≠ ∅ do
    selectează v din A
  
```

```

A ← A \ {v}
if v este pe frontieră
    then return solutie(v)
    else fie  $v_1, \dots, v_k$  fiile viabili ai lui v
        A ← A ∪ { $v_i \mid 1 \leq i \leq k$ }
        throw 'Nu există soluție.'
end

```

12.3.1 Branch-and-bound pentru probleme de optim

Se aplică problemelor de optim pentru care nu există algoritmi greedy sau bazați pe programare dinamică. În general, sunt probleme NP-dificile (a se vedea secțiunea 13.3). Se poate aplica și problemelor care nu sunt de optim, dar care pot fi formulate ca probleme de optim prin adăugarea unei funcții de cost care să selecteze soluția (soluțiile) dorite; un astfel de exemplu este dat în secțiunea 12.8. Reamintim că fiecare vârf v descrie o *soluție potențială* x^k (drumul de la rădăcina la v); prin $c(v)$ notăm valoarea $c(x^k)$. Multimea soluțiilor potențiale o include pe cea a soluțiilor candidat; presupunem că funcția c este definită peste soluțiile potențiale.

12.3.2 Branch-and-bound cu cost minim

Branch-and-bound cu cost minim poate fi descrisă pe scurt astfel:

1. Se consideră o *funcție de predicție* c^* definită pentru fiecare vârf v din arbore, care trebuie să satisfacă următoarele proprietăți:
 - a) $c^*(v) \leq c(v)$ pentru fiecare vârf v ;
 - b) dacă v este pe frontieră, atunci $c^*(v) = c(v)$;
 - c) dacă w este fiu al lui v , atunci $c^*(v) \leq c^*(w)$; prin tranzitivitate, obținem $c^*(v) \leq c^*(w)$ pentru orice descendenter w al lui v .
2. Structură de aşteptare A este un min-heap pentru ca la fiecare pas să fie ales elementul cu $c^*(v)$ minim.

Procedura de mai sus se descrie algoritmic astfel, unde operațiile peste A trebuie citite ca operații peste un min-heap:

```

procedure BBCostMin()
    fie r rădăcina
    calculează  $c^*(r)$ 
    A ← {(r,  $c^*(r)$ )}
    while (A ≠ ∅) do
        selectează v din A cu  $c^*(v)$  minim
        A ← A \ {(v,  $c^*(v)$ )}
        if v este pe frontieră
            then return solutie(v)
        else fie  $v_1, \dots, v_k$  fiile viabili ai lui v
            calculează  $c^*(v_1), \dots, c^*(v_k)$ 
            A ← A ∪ {( $v_i, c^*(v_i)$ ) |  $1 \leq i \leq k$ }
            throw 'Nu există soluție.'
    end

```

Structura standard a funcției c^* este

$$c^*(v) = f(h(v)) + g^*(v)$$

unde:

- $h(v)$ este costul drumului de la rădăcină la v (de exemplu, lungimea drumului);
- f este o funcție crescătoare;
- $g^*(v)$ este costul subestimat al obținerii unei soluții pornind din v ; $g^*(v)$ va constitui componenta optimistă a costului $c^*(v)$.

Calculul lui $g^*(v)$ nu este întotdeauna simplu. Nu se poate da o formulă generală pentru această funcție.

Teorema 12.1. *Dacă v este vârful calculat de BBCostMin(r, c^*), atunci drumul de la r la v reprezintă soluția de cost minim, adică pentru orice alt vârf s de pe frontieră, avem $c(s) \geq c(v)$.*

Demonstrație. Din proprietățile lui c^* rezultă că pentru orice vârf s de pe frontieră, $c^*(s) = c(s)$ și pentru orice descendant w al lui v , $c^*(v) \leq c^*(w)$. Vârful v este primul vârf de pe frontieră extras din structura de așteptare A , deci celelalte vârfuri de pe frontieră se află în structura de așteptare A sau sunt descendenți ai unor vârfuri din această structură. Rezultă:

- $c(v) = c^*(v) \leq c^*(w)$, pentru orice vârf w din structura de așteptare,
- $c^*(w) \leq c^*(s) = c(s)$, pentru orice vârf s de pe frontieră, descendant al unui vârf w din structura de așteptare.

În consecință, pentru orice vârf s de pe frontieră, $c(s) \geq c(v)$.

sfdem

12.3.3 Branch-and-bound cu cost minim și mărginire

Strategia branch-and-bound cu cost minim și mărginire este similară cu branch-and-bound cu cost minim și poate fi descrisă astfel:

1. Se consideră *funcția de predicție* c^* utilizată de strategia cost minim și o funcție de predicție suplimentară u , cu următoarele proprietăți:
 - a) $c(v) \leq u(v)$ pentru fiecare vârf v ;
 - b) dacă v este pe frontieră, atunci $c(v) = u(v)$;
 - c) dacă w este fiu al lui v , atunci $u(w) \leq u(v)$; prin tranzitivitate, obținem $u(w) \leq u(v)$ pentru orice descendant w al lui v .
2. Structură de așteptare A este de asemenea un min-heap pentru ca la fiecare pas să fie ales tot elementul cu $c^*(v)$ minim.

$u(v)$ este o funcție compusă $f(h(v)) + k^*(v)$, unde f și h sunt definite la fel ca în cazul funcției c^* , iar $k^*(v)$ exprimă o estimare supraevaluată (pesimistă) a costului obținerii unei soluții pornind din v .

Se pune problema determinării unui vârf s de pe frontieră, pentru care $u(s) = \min\{u(v)\}$.

Observație. Pentru un vârf de pe frontieră, $c^*(s) = c(s) = u(s)$. Deci, problema determinării lui s este echivalentă cu problema determinării unui vârf de pe frontieră de cost minim.

sfobs

Algoritmul branch-and-bound cu cost minim și mărginire este următorul:

```

procedure BBCostMin&Marg()
    fie r rădăcina
    calculează  $c^*(r)$ 
     $A \leftarrow \{(r, c^*(r))\}$ 
     $s \leftarrow \emptyset$ 
    while  $A \neq \emptyset$  do
        selectează  $v$  din  $A$  cu  $c^*(v)$  minim
         $A \leftarrow A \setminus \{(v, c^*(v))\}$ 
        if ( $c^*(v) \geq u_{min}$ )
            then break
        else for each  $w$  fiu al lui  $v$ 
            calculează  $c^*(w)$ 
            if ( $c^*(w) < u_{min}$ )
                 $A \leftarrow A \cup \{(w, c^*(w))\}$ 
                if ( $w$  este pe frontieră)
                    then  $u_{min} \leftarrow c^*(w)$ 
                     $s \leftarrow w$ 
                else calculează  $u(w)$ 
                     $u_{min} \leftarrow \min\{u_{min}, u(w) + \epsilon\}$ 
        if  $s \neq \emptyset$ 
            then return solutie(s)
        else throw 'Nu există soluție.'
    end

```

Lema 12.1. Vârfurile w , care schimbă valoarea lui u_{min} și nu sunt pe frontieră, nu pot fi ultimele ce realizează această modificare.

Demonstrație. Din relațiile $c^*(w) \leq u(w) < u(w) + \epsilon = u_{min}$ rezultă faptul că w este un posibil viitor vârf selectat. Aceasta înseamnă că urmează și alte iterații. Pentru orice descendenter d al lui w , $c^*(d) \leq u(d) \leq u(w) < u(w) + \epsilon = u_{min}$. Astfel, toți descendenții lui w vor fi introdusi în structura de așteptare, deoarece u_{min} rămâne neschimbat până la terminarea execuției algoritmului. Printre descendenții lui w se află și vârfurile de pe frontieră arborelui cu rădăcina w . În concluzie, până la terminarea algoritmului, va fi inserat în structura de așteptare un vârf de pe frontieră fără ca acesta să schimbe valoarea u_{min} , deoarece w este ultimul care a făcut aceasta.

sfdem

Teorema 12.2. Dacă s este vârful calculat de BBCostMin&Marg(), atunci acesta este un vârf de pe frontieră pentru care $u(s) = \min\{u(v)\}$.

Demonstrație.

- Deoarece $u(v) \geq c(v) \geq c^*(v)$, rezultă că dacă $c^*(v) \geq u_{min}$, atunci și $u(v) \geq u_{min}$. Deci, este suficient să se testeze $c^*(v) < u_{min}$ pentru a decide dacă vârful v va fi sau nu va fi inserat în structura de așteptare. Aceasta conduce însă la inserarea în structura de așteptare a unor vârfuri care pot avea $u(v) \geq u_{min}$, deoarece $c^*(v) < u_{min}$ nu asigură $u(v) < u_{min}$.

2. Atunci când un vîrf de pe frontieră este inserat în structura de aşteptare, acesta va avea costul $c^*(v) = c(v) = u(v)$ inferior lui u_{min} , deci costul său va defini noua valoare a lui u_{min} .
3. Vârfurile v care schimbă valoarea lui u_{min} și nu sunt vârfuri de pe frontieră, nu pot fi ultimele care realizează această modificare (lema 12.1).

Din 1-3 rezultă că ultimul vîrf s de pe frontieră reținut va fi și ultimul care micșorează u_{min} , iar alte vârfuri care pot micșora u_{min} nu mai există. Astfel, s satisface $u(s) = u = \min\{u(v)\}$. sfdem

12.4 Colorarea grafurilor

12.4.1 Descrierea problemei

Se consideră un (di)graf $G = (V, E)$ cu $V = \{1, \dots, n\}$ și m culori numerotate de la 1 la m . O *colorare* a (di)grafului este o funcție de la mulțimea vârfurilor V la mulțimea culorilor $\{1, \dots, m\}$ cu proprietatea că oricare muchie are extremitățile colorate diferit. Problema constă în generarea tuturor colorărilor posibile.

Problema este inspirată de la colorarea hărților. Fiecare hartă poate fi transformată într-un graf planar în modul următor: fiecare regiune a hărții este reprezentată printr-un vîrf al grafului, iar două vârfuri sunt adiacente dacă regiunile corespunzătoare au o frontieră comună. De exemplu, harta din figura 12.10a este reprezentată de graful din figura 12.10b.

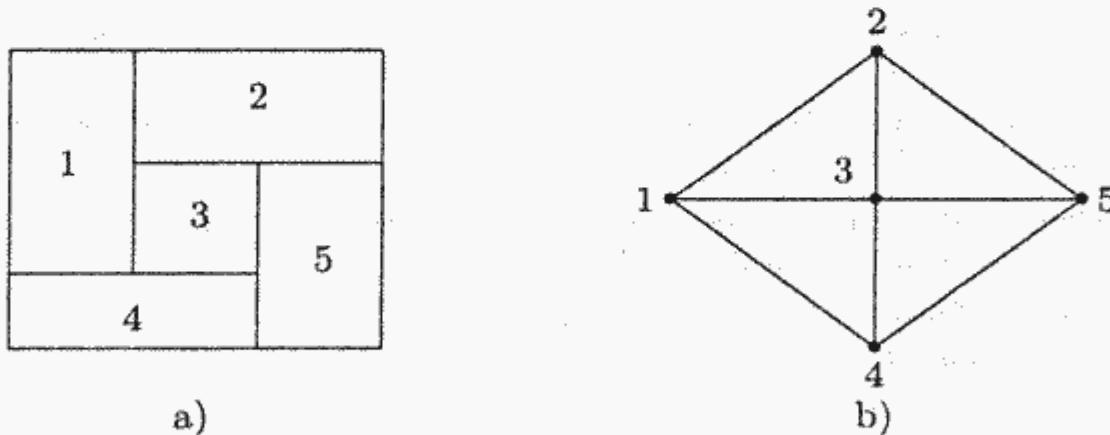
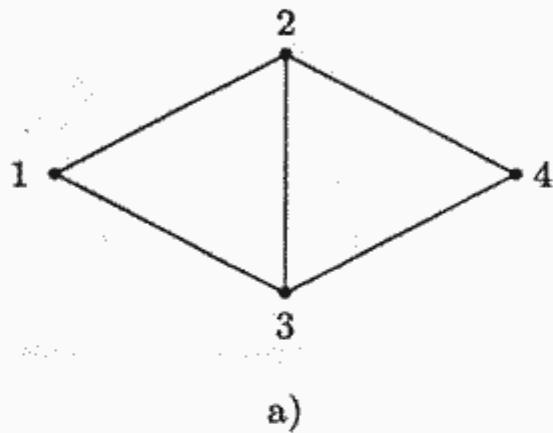


Figura 12.10: Relația de vecinătate dintr-o hartă reprezentată printr-un graf

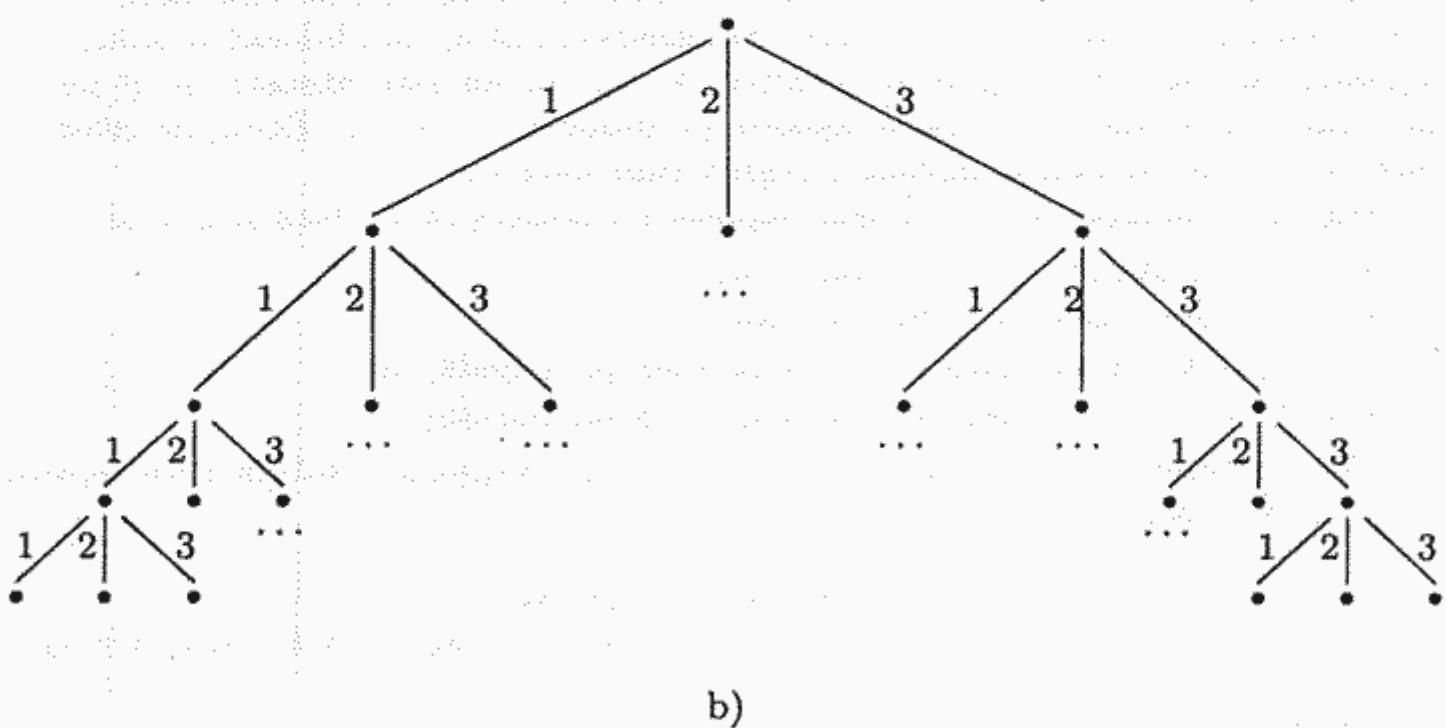
Observație. Pentru mulți ani s-a cunoscut faptul că cinci culori sunt suficiente pentru colorarea unei hărți, dar nu s-a găsit nici o hartă care să necesite mai mult de patru culori. După mai bine de o sută de ani (problema a fost formulată pentru prima dată în 1852 de către un student londonez, Francis Guthrie), s-a putut demonstra (Appel și Haken, 1976) că patru culori sunt suficiente pentru colorarea unei hărți. Demonstrația acestui rezultat include verificarea unei proprietăți de reductibilitate a grafurilor cu ajutorul calculatorului, fapt care a condus la „discuții aprinse” în ceea ce privește corectitudinea. sfobs

12.4.2 Modelul matematic

Convenim să notăm o soluție a problemei printr-un vector (x_1, \dots, x_n) unde x_i reprezintă culoarea vârfului i . De asemenea, presupunem că G este graf. Astfel, spațiul soluțiilor potențiale este $\{1, \dots, m\}^n$ și poate fi reprezentat printr-un arbore cu n nivele în care fiecare nod interior are exact m succesi. De exemplu, graful din figura 12.11a are spațiul soluțiilor potențiale reprezentat de arborele din figura 12.11b.



a)



b)

Figura 12.11: Colorarea grafurilor – spațiul soluțiilor potențiale

Condiția „nu există muchii cu extremitățile de aceeași culoare”, prin care se stabilește dacă un vector $(x_1, \dots, x_n) \in \{1, \dots, m\}^n$ este colorare, se exprimă prin:

$$\forall i, j \in V : \{i, j\} \in E \Rightarrow x_i \neq x_j \quad (12.1)$$

Criteriul de stabilire a unui candidat (de mărginire) se obține restricționând 12.1 la soluția potențială parțială (x_1, \dots, x_k) :

$$\forall j : (1 \leq j \leq k-1) \wedge (\{j, k\} \in E \Rightarrow x_j \neq x_k) \quad (12.2)$$

Utilizând 12.2 ca funcție de tăiere, arborele din exemplul de mai sus este transformat în arborele parțial din figura 12.12. Orice vârf de pe frontieră acestui arbore parțial este vârf soluție.

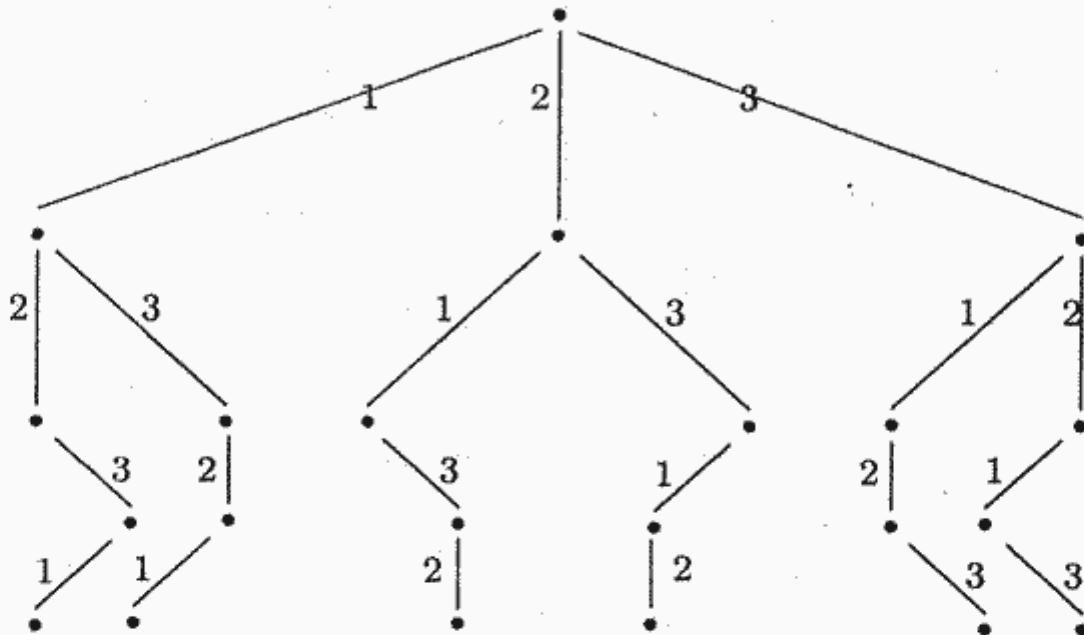


Figura 12.12: Colorarea grafurilor – arbore parțial

Ca algoritm de enumerare, utilizăm subprogramul de generare a elementelor produsului cartezian. Întrucât, de regulă, algoritmul de enumerare este determinat de structura de date aleasă pentru reprezentarea soluțiilor, rezultă că două sunt elementele ce caracterizează un algoritm backtracking: reprezentarea soluțiilor și criteriul de mărginire.

12.4.3 Implementare

Pentru implementare considerăm grafurile reprezentate prin matricele de adiacență.

```

funcția C(x, k)
    candidat ← true
    for j ← 1 to k-1 do
        if ((G.a[j,k]=1) and (x[j]=x[k]))
            then candidat ← false
    return candidat
end
  
```

12.5 Problema celor n regine

12.5.1 Descrierea problemei

Se consideră o tablă de șah de dimensiune $n \times n$ și n regine. Problema constă în așezarea pe tabla de șah a celor n regine, astfel încât să nu se captureze una pe alta. Dacă există mai multe asemenea așezări, atunci se vor determina toate.

	1	2	3	4
1				
2				
3				
4				

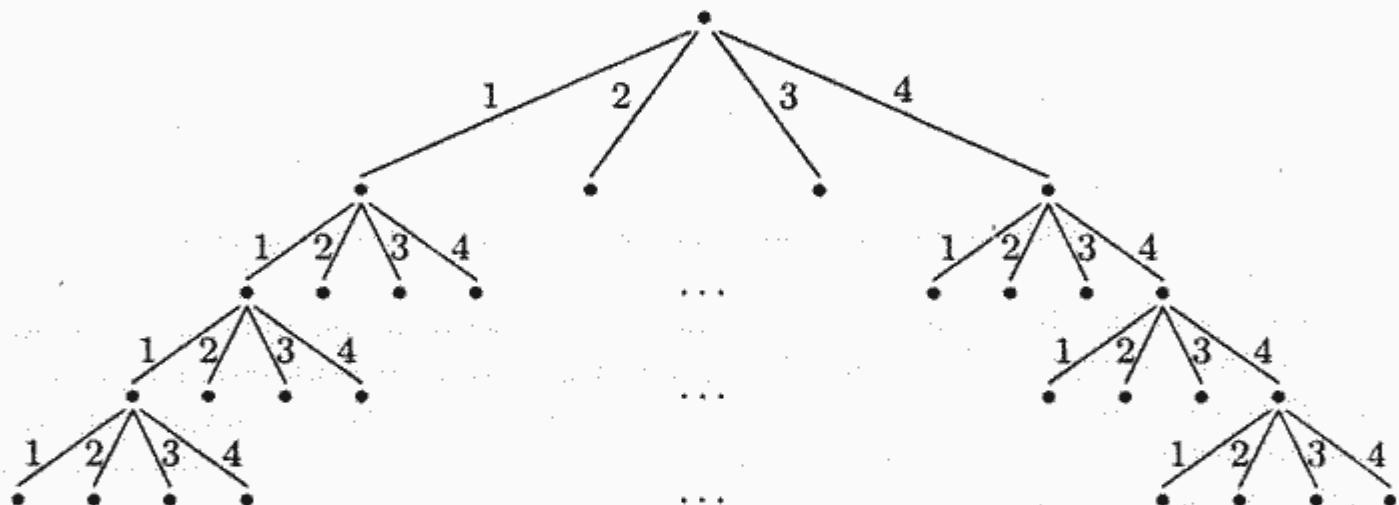
Figura 12.13: Tabla de săh 4×4 

Figura 12.14: 4 regine – reprezentarea spațiului soluțiilor potențiale

12.5.2 Modelul matematic

Pozitiiile de pe tabla de sah sunt identificate prin perechi (i, j) cu $1 \leq i, j \leq n$. De exemplu, pentru $n = 4$, tabla este cea reprezentata in figura 12.13. O solutie poate fi reprezentata printr-o functie $Q : \{1, \dots, n\} \times \{1, \dots, n\} \rightarrow \{\text{false}, \text{true}\}$ cu semnificația: $Q[i, j] = \text{true}$ dacă și numai dacă pe poziția (i, j) se află o regină. Se observă direct că spațiul soluțiilor potențiale conține 2^{n^2} elemente. O reducere substanțială a acestuia se obține dacă se reprezintă soluțiile printr-o funcție $s : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ cu semnificația: $s[i] = j \Leftrightarrow Q[i, j] = \text{true}$. În acest caz, spațiul soluțiilor potențiale conține n^n elemente. De exemplu, pentru $n = 8$ avem $2^{n^2} = 2^{64}$, iar $8^8 = 2^{24}$. Așadar alegerea reprezentării pentru soluțiile potențiale este foarte importantă pentru metoda backtracking: o alegere bună poate conduce la reducerea substanțială a dimensiunii spațiului acestor soluții. Cele n^n elemente ale spațiului soluțiilor potențiale pot fi reprezentate printr-un arbore cu n nivele în care fiecare vârf are exact n succesiuni imediați. Pentru cazul $n = 4$, arborele este cel reprezentat în figura 12.14.

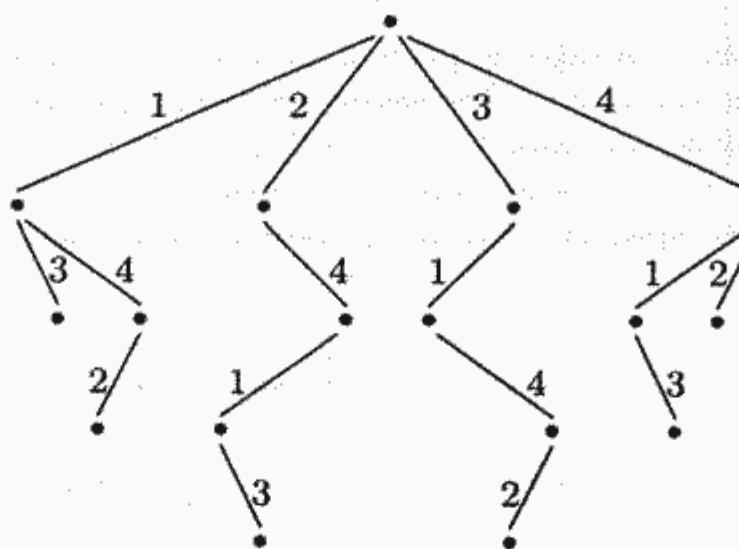


Figura 12.15: 4 regine – arborele parțial

În continuare vom căuta criteriul de mărginire prin care să eliminăm acei subarbore care nu conțin vârfuri-soluție. Să observăm că s_k candidează la soluție, dacă așezând cea de-a k -a regină pe poziția (k, s_k) , ea nu este atacată de și nu atacă nici una dintre cele $k - 1$ regine așezate pe liniile $1, \dots, k - 1$. Ecuațiile celor patru direcții pe care se poate deplasa o regină sunt date de:

regina de poziția (i, j) se poate deplasa pe poziția (k, ℓ) dacă și numai dacă:

- $i = k$ (deplasare pe orizontală) sau
- $j = \ell$ (deplasare pe verticală) sau
- $i - j = k - \ell$ (deplasare pe o diagonală principală) sau
- $i + j = k + \ell$ (deplasare pe o diagonală secundară).

Considerând $(i, j) = (i, s_i)$ și $(k, \ell) = (k, s_k)$ obținem condiția care reprezintă criteriul de mărginire:

$$C(s_1, \dots, s_k) = \forall i (1 \leq i \leq k-1) : \neg Q[i, s_k] \wedge \neg (i + s_i = s_k + k \vee i - s_i = k - s_k)$$

Sau, ținând cont de:

$$s_k + k = i + s_i \Rightarrow s_i = s_k + k - i \text{ și } s_k - k = s_i - i \Rightarrow s_i = s_k - k + i$$

obținem forma echivalentă:

$$C(s_1, \dots, s_k) = \forall i (1 \leq i \leq k-1) : \neg Q[i, s_k] \wedge \neg Q[i, s_k + k - i] \wedge \neg Q[i, s_k - k + i]$$

Pentru a ne face o idee cât de mult taie C din arborele spațiului soluțiilor potențiale, prezentăm arborele parțial obținut în urma aplicării criteriului de mărginire pentru cazul $n = 4$ în figura 12.15.

O funcție $s : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ este în fapt un element din $\{1, 2, \dots, n\}^n$ și deci ca algoritm de enumerare poate fi folosită oricare dintre variantele de generare a elementelor produsului cartezian.

12.6 Submulțime de sumă dată

12.6.1 Descrierea problemei

Se consideră o mulțime A cu n elemente, fiecare element $a \in A$ având o dimensiune $s(a) \in \mathbb{Z}_+$ și un număr întreg pozitiv M . Problema constă în determinarea tuturor submulțimilor $A' \subseteq A$ cu proprietatea $\sum_{a \in A'} s(a) = M$.

12.6.2 Modelul matematic

Presupunem $A = \{1, \dots, n\}$ și $s(i) = w_i, 1 \leq i \leq n$. Pentru reprezentarea soluțiilor avem două posibilități.

1. Prin vectori care să conțină elementele care compun soluția. Această reprezentare are dezavantajul că trebuie utilizat un algoritm de enumerare a vectorilor de lungime variabilă. De asemenea testarea condiției $a \in A \setminus A'$? nu mai poate fi realizată în timpul $O(1)$ dacă nu se utilizează spațiu de memorie suplimentar.
2. Prin vectori de lungime n , (x_1, \dots, x_n) cu $x_i \in \{0, 1\}$ având semnificația: $x_i = 1$ dacă și numai dacă w_i aparține soluției (vectorii caracteristici).

Exemplu. Fie $n = 4$, $(w_1, w_2, w_3, w_4) = (4, 7, 11, 14)$ și $M = 25$. Există următoarele soluții:

- $(4, 7, 14)$ care mai poate fi reprezentată prin $(1, 2, 4)$ sau $(1, 1, 0, 1)$ și
- $(11, 14)$ care mai poate fi reprezentată prin $(3, 4)$ sau $(0, 0, 1, 1)$.

sfex

Noi vom opta pentru ultima variantă, deoarece vectorii au lungime fixă. Remarcăm faptul că spațiul soluțiilor conține 2^n posibilități (elementele mulțimii $\{0, 1\}^n$) și poate fi reprezentat printr-un arbore binar. Ca algoritm de enumerare vom utiliza GenProdCart. Așa cum am mai văzut, acesta generează soluțiile potențiale prin partităionarea mulțimii A în două: o parte $\{1, \dots, k\}$ care a fost luată în considerare pentru a stabili candidații la soluție și a doua parte $\{k + 1, \dots, n\}$ ce urmează a fi luată în considerare. Cele două părți trebuie să satisfacă următoarele două inegalități:

- suma parțială dată de prima parte (adică de candidații aleși) să nu depășească M :

$$\sum_{i=1}^k x_i \cdot w_i \leq M \quad (12.3)$$

- ceea ce rămâne să fie suficient pentru a forma suma M :

$$\sum_{i=1}^k x_i \cdot w_i + \sum_{i=k+1}^n w_i \geq M \quad (12.4)$$

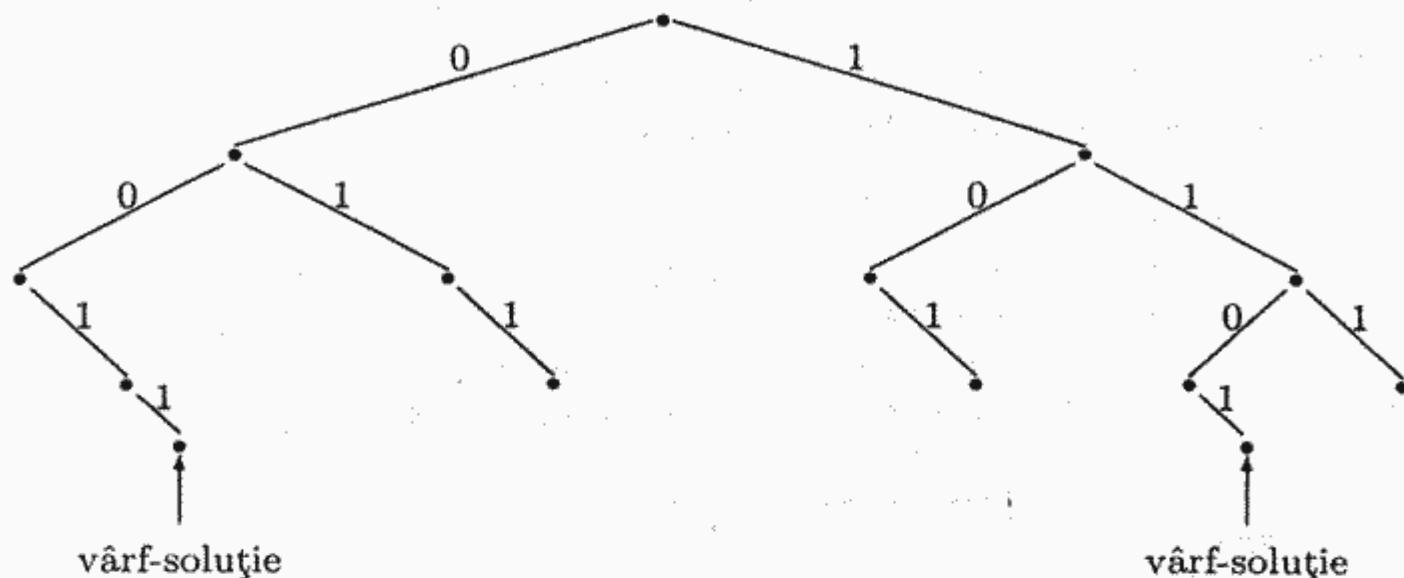


Figura 12.16: Arbore parțial pentru submulțime de sumă dată

Cele două inegalități pot constitui criteriu de mărginire. Cu acest criteriu de tăiere, arborele parțial rezultat pentru exemplul anterior este cel reprezentat în figura 12.16.

De remarcat că acest criteriu nu elimină toți subarborii care nu conțin vârfuri-soluție, dar elimină foarte mulți, restrângând astfel spațiul de căutare. Atingerea unui vârf pe frontieră presupune imediat determinarea unei soluții: suma $\sum_{i=k+1}^n w_i$ este zero (deoarece $k = n$) și dubla inegalitate data de relațiile 12.3 și 12.4 implică $\sum_{i=1}^n w_i = M$.

Observație. Dacă se utilizează drept criteriu de mărginire numai inegalitatea 12.3, atunci atingerea unui vârf pe frontieră în arborele parțial nu presupune neapărat și obținerea unei soluții. Mai trebuie verificat dacă suma submulțimii alese este exact M . sfobs

12.6.3 Optimizare

O varianta optimizată a algoritmului *backtracking* pentru problema submulțimilor de sumă dată se obține astfel:

- se consideră w_1, w_2, \dots, w_n în ordine crescătoare (fără a restrânge generalitatea);
- pentru cazul în care suma parțială dată de candidații aleși este strict mai mică decât M ($\sum_{i=1}^k x_i w_i < M$), se introduce un criteriu de mărginire suplimentar:

$$\sum_{i=1}^k x_i w_i + w_{k+1} \leq M.$$

Presupunem valorile x_1, \dots, x_{k-1} calculate. Notăm cu s suma parțială corespunzătoare valorilor x_1, \dots, x_{k-1} ($s = \sum_{i=1}^{k-1} x_i w_i$) și cu r suma $\sum_{i=k}^n w_i$. Presupunem $w_1 \leq M$ și $\sum_{i=1}^n w_i \geq M$.

Se obține astfel următorul algoritm *backtracking*:

```

procedure submultimiOpt(s, k, r)
    xk ← 1
    if (s+wk=M)
        then scrie(xk) /* xk=(x1, x2, ..., xk) */
    else if (s+wk+wk+1 ≤ M)
        then submultimiOpt(s+wk, k+1, r-wk)
            if ((s+r-wk ≥ M) and (s+wk+1 ≤ M))
                then xk ← 0
                submultimiOpt(s, k+1, r-wk)
    end

```

Apelul inițial este $\text{submultimiOpt}\left(0, 1, \sum_{i=1}^n w_i\right)$, iar $w_i \leq M$ și $0 + \sum_{i=1}^n w_i \geq M$. Astfel, la intrarea în execuție sunt asigurate condițiile $s + w_k \leq M$ și $s + r \geq M$. Aceste condiții sunt asigurate și la apelul recursiv: înainte de primul apel recursiv, $\text{submultimiOpt}(s + w_k, k + 1, r - w_k)$, nu e nevoie să se mai verifice dacă $\sum_{i=1}^k x_i w_i + \sum_{i=k+1}^n w_i \geq M$, deoarece $s + r > M$ și $x_k = 1$.

Nu se verifică explicit nici $k > n$. Inițial, $s = 0 < M$ și $s + r \geq M$. Rezultă $r \neq 0$, deci k nu poate depăși n . De asemenea, în linia „if $(s + w_k + w_{k+1} \leq M)$ ”, deoarece $s + w_k < M$, rezultă $r \neq w_k$, deci $k + 1 \leq n$.

12.7 Problema rucsacului II (continuare)

12.7.1 Descrierea problemei

Reamintim problema rucsacului în varianta discretă, descrisă la capitolul programare dinamică:

Se consideră n obiecte, $1, \dots, n$, de dimensiuni (greutăți) $w_1, \dots, w_n \in \mathbb{Z}_+$ și un rucsac de capacitate $M \in \mathbb{Z}_+$. Un obiect i sau este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus de loc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvență x_1, \dots, x_n cu $x_i \in \{0, 1\}$ și $\sum_i x_i \cdot w_i \leq M$. Introducerea obiectului i în rucsac aduce profitul $p_i \in \mathbb{Z}$, iar profitul total este $\sum_{i=1}^n x_i p_i$. Problema constă în a determina o alegere (x_1, \dots, x_n) care să aducă un profit maxim.

12.7.2 Modelul matematic

Convenim să notăm cu (y_1, \dots, y_n) soluția curentă. Presupunem că au fost selectate valorile $y_i, 1 \leq i \leq k$, și urmează să determinăm dacă y_k poate candida la o soluție optimă. Obținem un criteriu de mărginire bun dacă dispunem de o margine superioară pentru valoarea celei mai bune soluții ce se poate obține pentru alegerea făcută până la momentul considerat. Această margine poate fi obținută dacă relaxăm problema în modul următor:

- funcția obiectiv:

$$\max \sum_{i=1}^n y_i \cdot p_i$$

- restricții:

$$\begin{aligned}y_i &\in \{0, 1\}, 1 \leq i \leq k \\0 &\leq y_i \leq 1, k+1 \leq i \leq n \\\sum_{i=1}^n y_i \cdot w_i &\leq M\end{aligned}$$

Reamintim că valorile pentru y_i cu $1 \leq i \leq k$ sunt deja atribuite. Se rezolvă problema relaxată utilizând algoritmul greedy, obținându-se o margine superioară f_{sup} (profitul maxim care se poate realiza cu (y_1, \dots, y_k)). Fie $x_i \in \{0, 1\}$, $1 \leq i \leq n$, o soluție arbitrară. De fapt, vom vedea că $(x_i)_{1 \leq i \leq n}$ va reprezenta cea mai bună soluție determinată până la momentul curent. Dacă:

$$\sum_{i=1}^n x_i \cdot p_i \geq f_{sup}$$

atunci, evident, (y_1, \dots, y_k) nu poate candida la o soluție optimă și trebuie făcută o altă alegere pentru primele k componente. Această nouă alegere se face în maniera următoare:

- se determină cel mai mare k' cu $k' \leq k$ și $y'_k = 1$ (k' este ultimul obiect introdus în rucsac);
- se face $y'_k = 0$ (se scoate ultimul obiect introdus) și se testează dacă $(y_1, \dots, y_{k'})$ poate candida la o soluție optimă.

Acum, ordinea atribuirii de valori pentru y_k este următoarea: întâi se consideră y_k egal cu 1 (se încearcă mai întâi introducerea obiectului k în rucsac). Se testează dacă această alegere poate conduce la o soluție mai bună. Dacă da, atunci se determină cea mai bună soluție pentru alegerea făcută până în acel moment și se actualizează. Dacă această alegere nu poate conduce la o soluție mai bună decât cea întâlnită până în acel moment, atunci se atribuie lui y_k valoarea zero (obiectul k nu va fi introdus în rucsac).

Criteriul de mărginire se bazează pe cunoașterea unei soluții (x_1, \dots, x_n) care să reprezinte optimul peste soluțiile potențiale explorate. Valorile inițiale pentru x_i pot fi alese cu un algoritm greedy: se alege obiectul ce aduce un profit maxim peste multimea obiectelor nealese. Presupunem $\frac{p_1}{w_1} \geq \dots \geq \frac{p_n}{w_n}$.

Determinarea soluției inițiale (x_1, \dots, x_n) coincide cu parcurgerea primelor n vârfuri din explorarea DFS a arborelui corespunzător spațiului soluțiilor potențiale. Pentru a nu mai calcula încă o dată această soluție, se consideră inițial $(y_1, \dots, y_n) = (x_1, \dots, x_n)$.

12.7.3 Implementare

12.7.3.1 Backtracking

Vom considera un subprogram care determină soluția inițială:

```

procedure detSolInit(ob, n, M, k, y, pinit, cinit)
    cinit ← 0
    pinit ← 0
    i ← 1
    while (i ≤ n) do
        cinit ← cinit+ob[i].w
        if (cinit ≤ M)
            then ob[i].x ← 1
            y[i] ← 1
            pinit ← pinit+ob[i].p
            i ← i+1
        else k ← i-1
            cinit ← cinit-ob[i].w
            i ← n+1
    end

```

Procedura care calculează problema relaxată are o descriere asemănătoare:

```

function fSup(ob, n, M, k, ck, pk)
    ptemp ← pk
    ctemp ← ck
    for i ← k+1 to n do
        ctemp ← ctemp+ob[i].w
        if (ctemp < M)
            then ptemp ← ptemp+ob[i].p
        else ptemp ← ptemp+(1-(ctemp-M)/ob[i].w)*ob[i].p
    return ptemp
end

```

Algoritmul backtracking care rezolvă problema rucsacului este descris de următorul program:

```

procedure rucsacIIBack(ob, n, M, copt, popt)
    detSolInit(ob, n, M, k, y, ptemp, ctemp)
    popt ← ptemp
    while (k ≥ 0) do
        while (fsup(ob, n, k, M, ctemp, ptemp) ≤ popt) do
            while ((k ≥ 0) and (y[k] = 0)) do
                k ← k-1
                if (k > 0)
                    then y[k] ← 0
                ctemp ← ctemp-ob[k].w
                ptemp ← ptemp-ob[k].p
            while ((k < n) and (ob[k+1].w+ctemp ≤ M)) do
                k ← k+1
                y[k] ← 1
                ctemp ← ctemp+ob[k].w
                ptemp ← ptemp+ob[k].p

```

```

if (k=n)
    then popt ← ptemp
        ctemp ← ctemp
        for i ← 1 to n do
            ob[i].x ← y[i]
        else k ← k+1
            y[k] ← 0
end

```

Exemplu. Presupunem $n = 4$ și greutățile și profiturile date de următorul tabel:

i	1	2	3	4
p_i	40	50	55	70
w_i	30	40	45	60

Spațiul soluțiilor potențiale este reprezentat de arborele din figura 12.17. Pentru vârfurile de pe frontieră sunt trecute greutățile și profiturile corespunzătoare soluțiilor date de soluțiile potențiale date de aceste vârfuri. Considerăm două cazuri.

1. $M = 100$. Arborele parțial parcurs de algoritmul backtracking este cel din figura 12.18. Calculele corespunzătoare vârfurilor A, B, C, D , alese la întâmplare, sunt:

$$A: f_{sup} = 40 + 55 + \frac{25}{60} \cdot 70 > p_{opt} = 90$$

$$B: f_{sup} = 40 + 70 = 110 > p_{opt} = 95$$

$$C: f_{sup} = 50 + 55 + \frac{15}{60} \cdot 70 > p_{opt} = 110$$

$$D: f_{sup} = 55 + \frac{55}{60} \cdot 70 = \frac{330 + 385}{6} = \frac{715}{6} < p_{opt} = 120$$

2. $M = 80$. Arborele parțial parcurs de algoritmul backtracking este cel din figura 12.19. Calculele corespunzătoare vârfurilor A, B, C, D sunt:

$$A: f_{sup} = 40 + 55 + \frac{5}{60} \cdot 70 > p_{opt} = 95$$

$$B: f_{sup} = 40 + \frac{50}{60} \cdot 70 = \frac{240 + 350}{6} = \frac{590}{6} > p_{opt} = 95$$

$$C: f_{sup} = 50 + \frac{40}{45} \cdot 85 = \frac{450 + 40}{4} = \frac{890}{9} > p_{opt} = 95$$

$$D: f_{sup} = 55 + \frac{35}{60} \cdot 70 = \frac{330 + 245}{6} = \frac{575}{7} < p_{opt} = 95$$

sfex

12.7.3.2 Branch and bound cu cost minim și mărginire

Problema rucsacului implică maximizarea funcției profit. Pentru a fi posibilă aplicarea strategiei cu cost minim și mărginire, funcția obiectiv $\max \sum_{i=1}^n x_i \cdot p_i$ este transformată în $\min \sum_{i=1}^n x_i \cdot (-p_i)$.

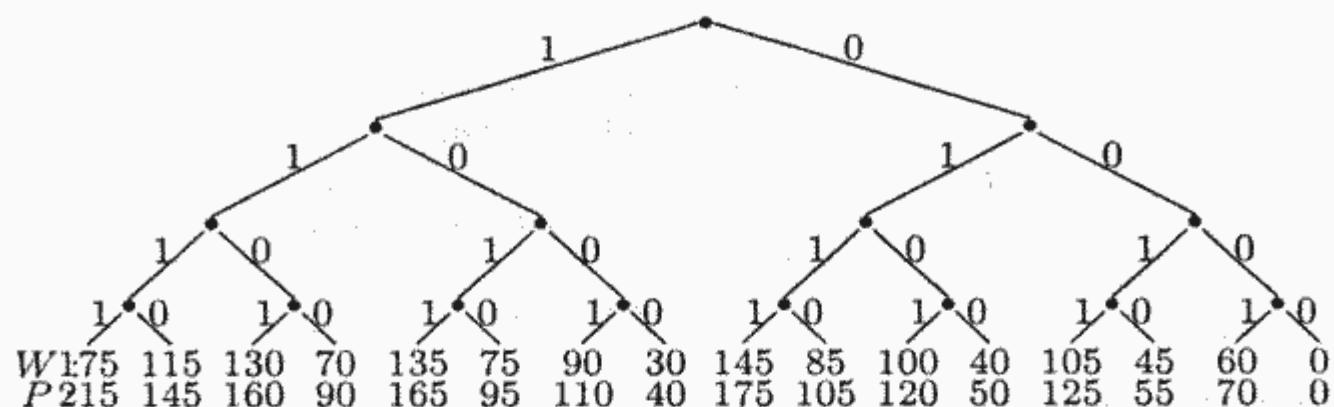
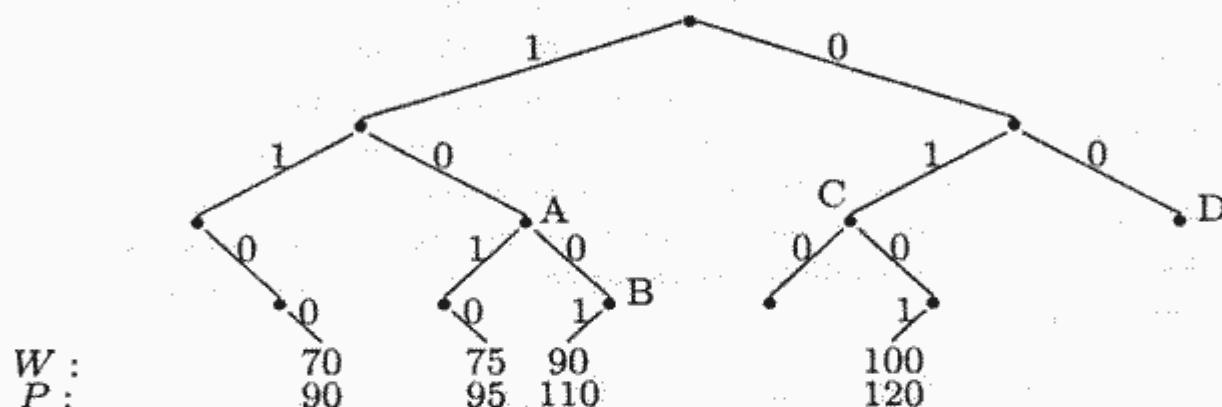


Figura 12.17: Problema rucsacului – arborele soluțiilor potențiale

Figura 12.18: Problema rucsacului – arborele parțial pentru $M = 100$

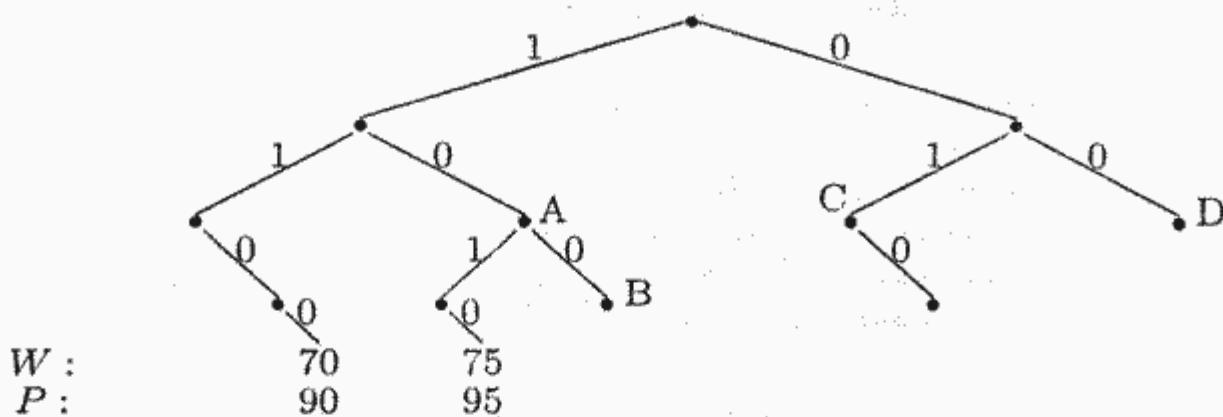
În aceste condiții, pentru un vârf v , $c^*(v)$ se definește ca *–profitul maxim posibil* pentru v și $u(v)$ ca *–profitul sigur*.

Considerăm obiectele ordonate astfel încât $\frac{p_1}{w_1} \geq \dots \geq \frac{p_n}{w_n}$. Fie v un vârf de pe nivelul k ; valorile x_1, x_2, \dots, x_{k-1} sunt deja calculate. Notăm cu c_r capacitatea rămasă și cu p_o profitul deja obținut. Presupunem că obiectele k, \dots, n nu au fost încă analizate. În aceste condiții, valorile $c^*(v)$ și $u(v)$ pot fi calculate cu procedura următoare:

```

procedure margini(p, w, cr, po, n, k, c*, u)
  u ← -po; cap ← cr;
  for i ← k to n do
    if (cap ≥ w[i])
      then cap ← cap-w[i] /* obiectul i poate fi încărcat în rucsac */
            u ← u-p[i]
    else c* ← u - cap * (p[i]/w[i])
        for j ← i+1 to n do
          if (cap ≥ w[j])
            then cap ← cap-w[j]/* obiectul j are loc în rucsac */
                  u ← u-p[j]
    return
  c* ← u /* toate obiectele k, k + 1, ..., n pot fi încărcate în rucsac */
end

```

Figura 12.19: Problema rucasacului: arborele parțial pentru $M = 80$

Observație. Dacă $x_k = 1$, adică $\text{cap} \geq w(k)$, atunci valorile c^* și u calculează în urma apelurilor:

`margini(p, w, cr, po, n, k, c*, u)`

și

`margini(p, w, cr - w[i], po + p[k], n, k + 1, c*, u)`

sunt identice.

sfobs

Pentru rezolvarea problemei prin metoda *branch-and-bound* cu cost minim și mărginire, se poate aplica algoritmul standard `BBCostMin&Marg()`, care să folosească valorile $c^*(v)$ și $u(v)$ calculate cu procedura `margini`.

12.8 Perspico

12.8.1 Descrierea problemei

Considerăm următoarea variantă simplificată a jocului *Perspico*. Fie o rețea formată din 3×3 pătrate, numite *locații*. În primele opt locații se găsesc opt piese etichetate cu litere de la A la H, de dimensiuni potrivite astfel încât să poată fi deplasate pe orizontală sau verticală atunci când există o locație vecină liberă (a se vedea figura 12.20). Numim această așezare a pieselor *configurație finală* și o notăm cu C_f . O *mutare* constă în deplasarea unei piese în locația liberă, atunci când este posibil.

Problema este următoarea:

1. dată o configurație C , să se decidă dacă există o listă de mutări care să permită obținerea lui C_f din C ;
2. în cazul în care există, să se determine o astfel de listă de mutări.

12.8.2 Modelul matematic

Spațiul soluțiilor conține $9!$ combinații (dacă nu se verifică ciclitățile). În plus, interesează modul de obținere a soluțiilor, adică drumul de la configurația C la

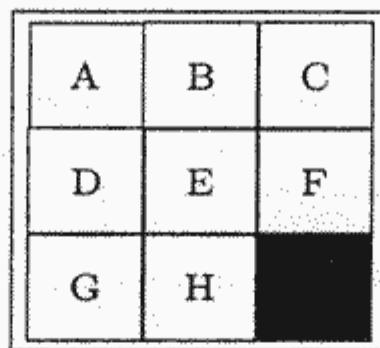
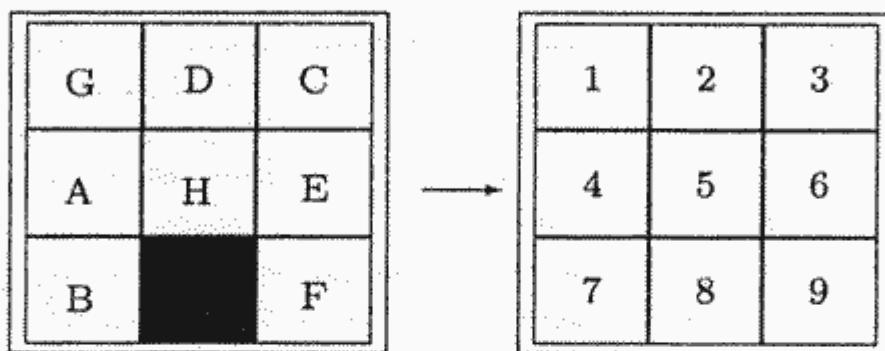


Figura 12.20: Jocul Perspico

Figura 12.21: Configurația definită prin funcția *poz*

configurația C_f .

Dată o configurație C , se poate decide dacă există o listă de mutări care să permită obținerea lui C_f din C . Dată o configurație C , se definește funcția injectivă:

$$poz_C : \{A, B, C, \dots, L\} \rightarrow \{1, 2, 3, \dots, 8, 9\}$$

unde $poz_C(X) = i$ semnifică faptul că piesa notată cu X se află pe poziția i . Locația liberă a fost asociată cu piesa simbolică L . În figura 12.21, $poz_C(D) = 2$, $poz_C(L) = 8$.

Pentru o configurație C și pentru fiecare piesă X se definește

$$less_C(X) = \#\{Y | A \leq Y \leq L, Y < X, poz_C(Y) > poz_C(X)\}.$$

Peste mulțimea $\{A, B, C, \dots, H, L\}$, se consideră ordinea alfabetică. În exemplul anterior avem $less_C(H) = 3$, $less_C(L) = 1$.

Teorema 12.3. Fie C o configurație, astfel încât $i, j \in \{1, 2, 3\}$ desemnează linia și coloana unde este plasată locația liberă, piesa L , și

$$l(C) = \begin{cases} 0, & \text{dacă } i + j \text{ este par;} \\ 1, & \text{dacă } i + j \text{ este impar.} \end{cases}$$

Atunci nu există un sir de transformări până la configurația finală C_f dacă:

$$S(C) = \sum_{X=A}^L less_C(X) + l(C) \text{ este număr impar.}$$

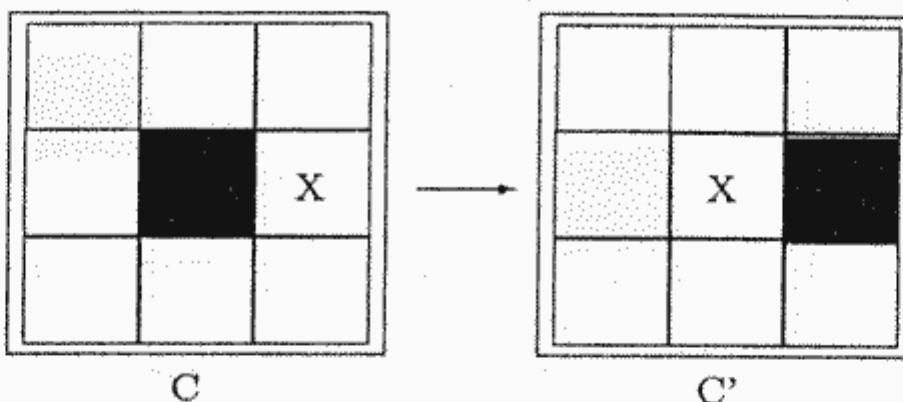


Figura 12.22: Locația liberă se deplasează la dreapta

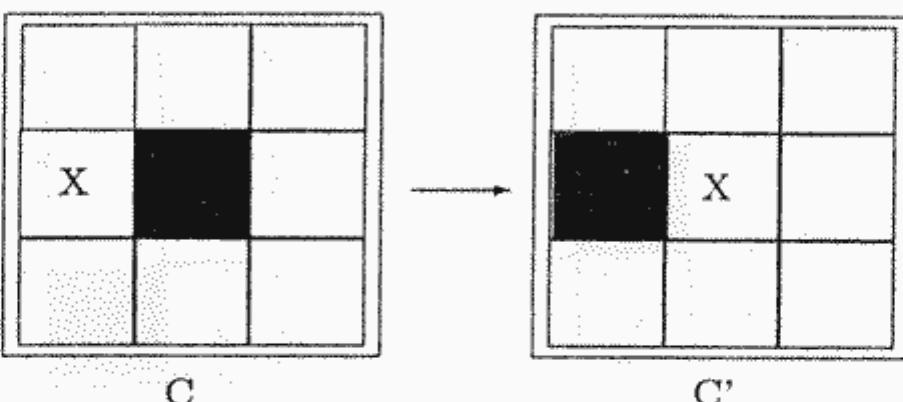


Figura 12.23: Locația liberă se deplasează la stânga

Demonstrație. Pentru configurația finală C_f : $S(C_f) = 0 + 0$, care este evident par. Fie o configurație C pentru care $S(C) =$ este număr par și C' obținută din C prin una din mutările permise.

- Cazul 1. Locația liberă se deplasează la dreapta (figura 12.22). Avem: $l(C') = (l(C) + 1) \bmod 2$, $\text{less}_{C'}(L) = \text{less}_C(L) - 1$, $\text{less}_{C'}(X) = \text{less}_C(X)$, $(\forall)X \neq L$. Rezultă $S(C')$ număr par.
- Cazul 2. Locația liberă se deplasează la stânga (figura 12.23). Similar cu cazul 1.
- Cazul 3. Locația liberă se deplasează în sus (fig. 12.24). Avem:
 - $l(C') = (l(C) - 1) \bmod 2$;
 - $\text{less}_{C'}(L) = \text{less}_C(L) + 3$;
 - X schimbă locul cu L . Dacă $X < Y$, atunci $\text{less}_{C'}(Y) = \text{less}_C(Y) + 1$ și $\text{less}_{C'}(X) = \text{less}_C(X)$; dacă $X > Y$, atunci $\text{less}_{C'}(X) = \text{less}_C(X) - 1$ și $\text{less}_{C'}(Y) = \text{less}_C(Y)$. Analiza de mai sus este valabilă și pentru Z , deci la $S(C)$ se adună un număr par (-2, 0 sau 2). Combinând cu 3.1, 3.3 și 3.3, rezultă că $S(C')$ rămâne par.
- Cazul 4. Locația liberă se deplasează în jos (figura 12.25):
Simetric cu cazul 3.

Deoarece prin transformările $\rightarrow \uparrow \downarrow \leftarrow$ nu se schimbă paritatea valorii $S(C)$ și cum $S(C_f)$ este 0 (par), rezultă că nu există un sir de transformări până la configurația finală C_f , dacă $S(C)$ este număr impar.

sfdem

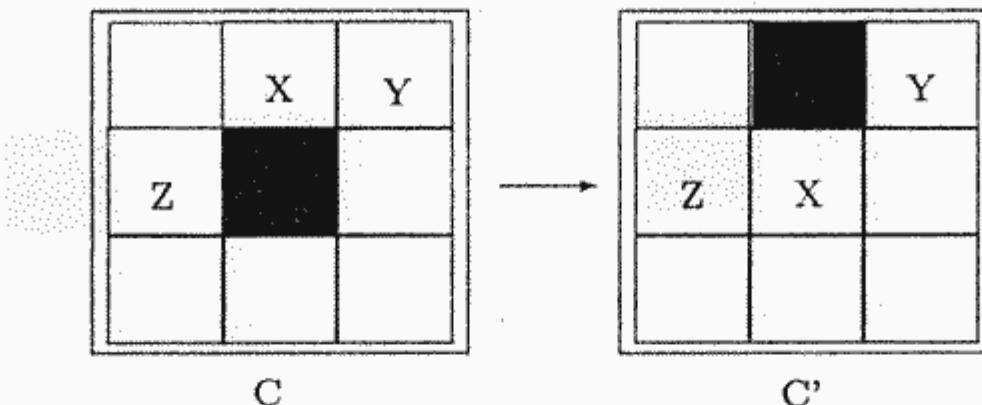


Figura 12.24: Locația liberă se deplasează în sus

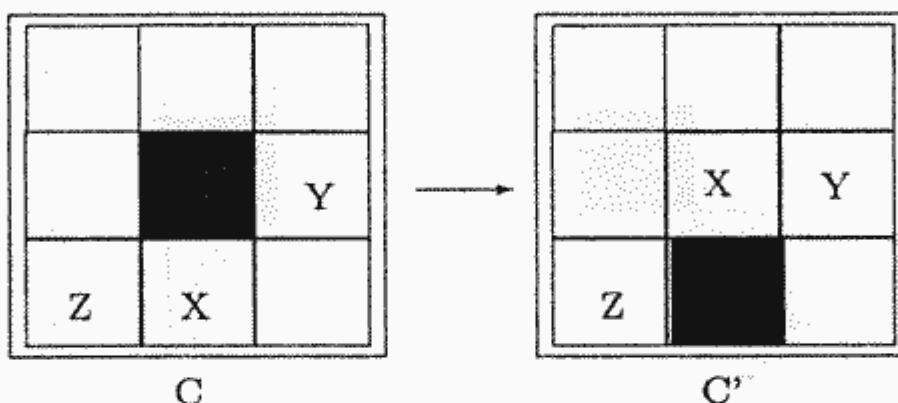


Figura 12.25: Locația liberă se deplasează în jos

12.8.3 Implementare

Pentru a rezolva problema determinării listei de mutări care să permită obținerea lui C_f din C , dacă această există, poate fi aplicat algoritmul *branch-and-bound* cu cost minim.

În acest caz $c^*(x) = f(x) + g^*(x)$, unde: $f(x) = \text{nivel}(x)$, iar $g^*(x) = \text{numărul pieselor care nu sunt pe pozițiile definite de } C_f$.

12.9 Exerciții

Exercițiul 12.1. Să se proiecteze un algoritm backtracking pentru problema din exercițiul 9.2 (*Schimbarea banilor*).

Exercițiul 12.2. (*Problema labirintului*) Se consideră un tablou bidimensional cu elemente 0 și 1 reprezentând un labirint: 1 blochează calea, iar 0 semnifică o poziție deschisă. Să se scrie un algoritm backtracking care încearcă să traseze un drum de la poziția $(1, 1)$ la poziția (n, n) .

Exercițiul 12.3. [HS84] (*Mărcile poștale*) Se consideră n denumiri de mărci poștale și se presupune că nu se permit mai mult de m mărci poștale pe o scrisoare. Să se scrie un algoritm backtracking care determină cel mai mare domeniu (multime de numere consecutive) de valori poștale ce pot fi puse pe o scrisoare și toate combinațiile posibile de denumiri care realizează domeniul.

Exercițiu 12.4. [HS84] (*Conecțarea tranzistorilor*) Se consideră:

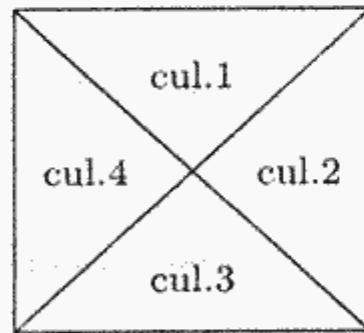
- n componente electrice (tranzistori) ce urmează a fi plasăți pe o placă de circuite pe care sunt marcate n poziții;
- o matrice de conectare CON cu $CON[i, j]$ reprezentând numărul de conectări ce trebuie făcute între componentele i și j ;
- o matrice de distanțe $DIST$ cu $DIST[r, s]$ reprezentând distanța de la poziția r la poziția s a plăcii de circuite;
- costul unei conectări este $\sum_{i,j} CON[i, j] \cdot DIST[i, j]$.

Să se scrie un algoritm backtracking care determină o plasare a componentelor pe pozițiile plăcii, astfel încât costul de conectare să fie minim.

Exercițiu 12.5. [HS84] (*Procesare paralelă*) Există n programe ce urmează a fi executate de k procesoare care lucrează în paralel. Timpul necesar executării programului i este t_i . Scrieți un algoritm backtracking care să determine ce programe să fie executate de fiecare procesor și în ce ordine, astfel încât timpul după ce se termină ultimul program să fie minim.

Exercițiu 12.6. (*Mutarea calului*) Se consideră o tablă de sah $n \times n$ pe care se plasează arbitrar un cal pe poziția (x, y) . Să se determine $n^2 - 1$ mutări ale calului, astfel încât fiecare poziție a tablei să fie vizitată de către acesta exact o dată. Există asemenea secvențe de mutări pentru orice n ?

Exercițiu 12.7. [MS91] (*Mozaic*) O plăcuță de mozaic are formă pătrată de dimensiune 1 și este divizată în patru părți colorate diferit. Fiecare parte colorată este unic determinată de o latură și centrul pătratului.



Există m tipuri de plăcuțe, i.e., m aranjamente coloristice diferite. De fiecare tip există t_i plăcuțe, astfel încât $\sum_{i=1}^m t_i = n^2$. Să se scrie un algoritm backtracking care să determine o amplasare a plăcuțelor într-un pătrat de latura n , astfel încât ori de câte ori două plăcuțe au o latură comună să existe aceeași culoare de o parte și de alta a laturii comune. Să se considere și cazul când pe o plăcuță pot să fie mai puțin de patru culori distincte.

Exercițiu 12.8. [MS91] (*Galeriile de artă*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri, astfel încât reuniunea câmpurilor interne de vizibilitate acoperă tot interiorul poligonului.

Exercițiu 12.9. [MS91] (*Fortăreața*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri, astfel încât reuniunea câmpurilor externe de vizibilitate acoperă tot exteriorul poligonului.

Exercițiul 12.10. [MS91] (*Curtea închisorii*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri, astfel încât reuniunea câmpurilor interne și externe de vizibilitate acoperă tot planul.

Exercițiul 12.11. Să se arate că orice număr natural $n > 2$ se poate scrie ca o sumă de numere prime. Să se scrie un program care, pentru un număr natural $n > 2$ dat, determină o secvență de lungime minimă de numere prime a căror sumă este egală cu n .

12.10 Referințe bibliografice

Acvest capitol are la bază [HS84, Sah98, LG86]. În [Luc93] este descris un algoritm backtracking pentru rezolvarea ecuațiilor diofantice liniare.

Capitolul 13

Probleme NP-complete

13.1 Algoritmi nedeterminiști

Activitatea unui algoritm nedeterminist se desfășoară în două etape: într-o primă etapă „se ghicește” o anumită structură S și în etapa a doua se verifică dacă S satisfacă o condiție de rezolvare a problemei. Putem adăuga „puteri magice de ghicire” unui limbaj de programare adăugându-i o funcție de formă:

`choice(M)` – care întoarce un element din mulțimea M .

Pentru a ști dacă verificarea s-a terminat cu succes sau nu, se adăugă și două instrucțiuni de terminare:

`success` – care semnalează terminarea verificării (și a algoritmului) cu succes, și

`failure` – care semnalează terminarea verificării (și a algoritmului) fără succes.

Această definiție a algoritmilor nedeterminiști este strâns legată de rezolvarea problemelor de decizie, ce constituie forma standard utilizată de teoria calculabilității. Alegerea pare oarecum surprinzătoare având în vedere că foarte multe probleme studiate sunt probleme de optimizare. Cum pot fi formalizate problemele ca probleme de decizie? Prezentăm ca exemplu problema rucsacului, varianta discretă:

Rucsac 0/1

Instanță. O mulțime O (obiectele), o mărime $s(o) \in \mathbb{Z}_+$ și o valoare $v(o) \in \mathbb{Z}_+$ pentru fiecare obiect $o \in O$, o restricție $M \in \mathbb{Z}_+$ și un scop $K \in \mathbb{Z}_+$.

Întrebare. Există o submulțime $O' \subseteq O$ cu proprietatea

$$\sum_{o \in O'} s(o) \leq M \text{ și } \sum_{o \in O'} v(o) \geq K?$$

Se poate dovedi că problema de optim poate fi redusă polinomial la problema de decizie și reciproc (vom vedea în secțiunea următoare ce înseamnă exact acest lucru). În acest fel, cele două probleme sunt echivalente din punctul de vedere al calculabilității.

Un algoritm nedeterminist care rezolvă problema de decizie a rucsacului este următorul:

```

procedure rucsacIIBack(0, s, v, n, M, K, x)
    /* etapa de ghicire */
    for fiecare  $o \in O$  do
         $x[o] \leftarrow \text{choice}(0,1)$ 
    /* etapa de verificare */
    sGhicit  $\leftarrow 0$ 
    vGhicit  $\leftarrow 0$ 
    for fiecare  $o \in O$  do
        sGhicit  $\leftarrow sGhicit + s[o]*x[o]$ 
        vGhicit  $\leftarrow vGhicit + v[o]*x[o]$ 
    if ((sGhicit  $\leq M$ ) and (vGhicit  $\geq K$ ))
        then success
        else failure
    end

```

Este ușor de văzut că algoritmul are complexitatea $O(n)$. Așadar, puterea de a ghici reduce drastic complexitatea. Acest aspect va fi discutat mai pe larg în secțiunea 13.3.

13.2 Clasele \mathbb{P} și \mathbb{NP}

Notăm cu \mathbb{P} clasa problemelor P pentru care există un algoritm determinist care rezolvă P în timp polinomial și cu \mathbb{NP} clasa problemelor P pentru care există un algoritm nedeterminist care rezolvă P în timp polinomial. Evident, are loc:

$$\mathbb{P} \subseteq \mathbb{NP}$$

Există foarte multe motive pentru a crede că inclusiunea $\mathbb{P} \subseteq \mathbb{NP}$ este strictă, i.e., $\mathbb{P} \subset \mathbb{NP}$. Algoritmii nedeterminiști constituie un instrument mult mai puternic decât cei determiniști și până acum nu s-a putut găsi o metodă prin care algoritmi nedeterminiști să poată fi convertiți în algoritmi determiniști în timp polinomial. Cel mai puternic rezultat care s-a putut dovedi este următorul:

Teorema 13.1. *Dacă o problemă P aparține clasei \mathbb{NP} , atunci există polinoamele $p(n)$ și $q(n)$, astfel încât P poate fi rezolvată de un algoritm determinist în timpul $O(p(n)2^{q(n)})$.*

Demonstrație. Fără să restrângem generalitatea, presupunem că funcția `choice` alege aleator din mulțimea $\{0, 1\}$ și că algoritmul nedeterminist „ghicește” structura prin $q(n)$ apeluri ale lui `choice`. Mai presupunem că verificarea structurii se face în timpul $O(p(n))$. Algoritmul determinist va genera și verifica toate cele $2^{q(n)}$ structuri. Evident, această explorare necesită timpul $O(p(n)2^{q(n)})$. sfdem

Pentru a arăta că o anumită problemă este într-o clasă dată, este necesară găsirea unui algoritm care să rezolve problema și să îndeplinească cerințele din definiția clasei. Dovedirea neapartenenței este mai dificilă și se face pe o cale indirectă. O asemenea metodă de demonstrare este reducerea.

Definiția 13.1. Fie P și Q două probleme și $g(n)$ un polinom. Spunem că P este transformată în timpul $O(g(n))$ în problema Q dacă există o funcție t astfel încât:

1. t are complexitatea timp $O(g(n))$;
2. t transformă o instanță $p \in P$ într-o instanță $t(p) \in Q$;
3. pentru orice instanță $p \in P$, p și $t(p)$ au același răspuns (valoare de adevăr).

Notăm $P \propto_{g(n)} Q$ și citim P se reduce polinomial la Q . Dacă precizarea polinomului $g(n)$ nu interesează, atunci notăm numai $P \propto Q$.

Are loc următorul rezultat:

Teorema 13.2. a) Dacă P are complexitatea timp $\Omega(f(n))$ și $P \propto_{g(n)} Q$, atunci Q are complexitatea timp $\Omega(f(n) - g(n))$.

b) Dacă Q are complexitatea $O(f(n))$ și $P \propto_{g(n)} Q$, atunci P are complexitatea $O(f(n) + g(n))$.

Demonstrație. Fie B un algoritm care rezolvă Q . Următorul algoritm, notat cu A , rezolvă P :

1. calculează $t(p)$;
2. apelează B pentru intrarea $t(p)$.

a) Deoarece P are complexitatea $\Omega(f(n))$, rezultă că există $c > 0$ astfel încât $T_A(n) \geq c \cdot f(n)$. Dar:

$$T_A(n) = c \cdot g(n) + T_B(n)$$

de unde:

$$T_B(n) \geq c \cdot f(n) - c' \cdot g(n)$$

care arată că:

$$T_B(n) = \Omega(f(n) - g(n))$$

b) Deoarece Q are complexitatea $O(f(n))$, rezultă $T_B \leq c \cdot f(n)$. Avem:

$$T_A(n) \leq c \cdot f(n) + c' \cdot g(n) = O(f(n) + g(n)).$$

Toate inegalitățile de mai sus au loc pentru $n \geq n_0$, unde n_0 este o constantă convenabil aleasă. sfdem

Iată cum poate fi folosită reducerea la demonstrarea neapartenenței la \mathbb{P} : dacă $P \notin \mathbb{P}$ și $P \propto Q$, atunci $Q \notin \mathbb{P}$. De asemenea, reducerea poate fi utilizată și pentru a dovedi apartenența: dacă $Q \in \mathbb{P}$ și $P \propto Q$, atunci $P \in \mathbb{P}$.

Deci pentru a putea a arăta că există inclusiunea strictă $\mathbb{P} \subset \text{NP}$, este necesar să găsim o problemă care este în NP și nu este în \mathbb{P} . Pentru acest rol ar putea candida acele probleme P cu proprietatea că pentru orice altă problemă $Q \in \text{NP}$ are loc $Q \propto P$. Această observație justifică următoarea definiție.

Definiția 13.2. a) Problema P este NP -difícilă dacă $Q \propto P$ pentru orice $Q \in \text{NP}$.

b) Problema P este NP -completă dacă $P \in \text{NP}$ și P este NP -difícilă.

Prima problemă care a fost dovedită a fi NP -completă este cunoscută sub numele de SATISFIABILITATE (pe scurt *SAT*).

Definiția 13.3. Problema SAT

Fie $X = \{x_0, \dots, x_{n-1}\}$ o mulțime de variabile. Un literal este o variabilă x sau negația sa \bar{x} . O atribuire este o funcție $\alpha : X \rightarrow \{0, 1\}$. Atribuirea α se extinde la literale astfel:

$$\alpha(u) = \begin{cases} \alpha(x) & \text{dacă } u = x \in X, \\ \neg\alpha(x) & \text{dacă } u = \bar{x}, x \in X. \end{cases}$$

O clauză este o mulțime finită c de literale. Clauza c este satisfăcută de atribuirea α dacă $\alpha(u) = 1$ pentru cel puțin un $u \in c$. Problema satisfiabilității constă în a determina dacă, pentru o secvență de cluze $C = (c_0, \dots, c_{q-1})$, există o atribuire α care satisface orice clauză care apare în C .

Observație. SAT poate fi reformulată astfel: dată o formulă din calculul propozițional în formă normală conjunctivă, să se decidă dacă există o atribuire pentru variabile pentru care formula este adevărată. În alte lucrări, forma normală conjunctivă este înlocuită cu forma normală disjunctivă. Evident, cele două probleme sunt echivalente: există algoritmi polinomiali care transformă o formă normală în cealaltă.

sfobs

Următoarea teoremă î se datorează lui Steven Cook (1971).

Teorema 13.3. SAT este NP-completă.

Demonstrație. Mai întâi dovedim că $SAT \in NP$. Un algoritm nedeterminist care rezolvă SAT este construit schematic după cum urmează:

1. ghicește o atribuire α ;
2. evaluează cluzele c_i , $i = 0, \dots, q - 1$;
3. dacă α satisface orice clauză c_i , $i = 0, \dots, q - 1$, atunci algoritmul se oprește cu succes;
4. altfel, se oprește cu eșec.

Acum vom arăta că pentru orice problemă $P \in NP$ avem $P \leq SAT$. Fie $P \in NP$. Există un algoritm nedeterminist A care rezolvă în timp polinomial P . Fără să restrângem generalitatea, facem asupra lui A următoarele presupuneri:

- lucrează numai cu numere naturale;
- o locație de memorie poate memora numere oricât de mari; un număr x este reprezentat pe $O(\log x)$ biți (cifre binare 0 și 1);
- instrucțiunile sunt etichetate cu $0, 1, 2, \dots$ (precizăm că A are un număr finit de instrucțiuni);
- pentru orice intrare de mărime n :
 - utilizează $p(n)$ locații, unde $p(n)$ este un polinom;
 - notăm aceste locații cu $1, 2, \dots, p(n) - 1$;
 - există un polinom $q(n)$ cu proprietatea că mărimea reprezentărilor numerelor memorate în locații nu depășește $q(n)$.

Problema P poate fi reformulată astfel: dată o intrare x de lungime n pentru A , să se decidă dacă există un calcul pentru care instrucțiunea din configurația finală este success. Vom descrie schematic un algoritm care transformă în timp polinomial mulțimea calculelor posibile pentru o intrare dată într-o formulă logică din calculul propozițional. Formulele vor fi construite cu ajutorul următoarelor variabile booleene:

1. $B_{i,j,t}$ reprezintă valoarea bitului j din reprezentarea numărului memorat în locația i la momentul t ($0 \leq i \leq p(n) - 1, 0 \leq j \leq q(n) - 1, 0 \leq t \leq T_A(n)$);
2. $S_{k,t}$ arată dacă instrucțiunea de etichetă k este cea care este executată la momentul t .

Formula logică atașată unei intrări x , este de forma $F(A, x) = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6$ unde:

- F_1 reprezintă starea inițială;
- F_2 reprezintă faptul că instrucțiunea de etichetă 1 este prima instrucțiune care se execută;
- F_3 reprezintă faptul că după t pași există exact o instrucțiune care se poate executa, $t = 0, 1, \dots, T_A(n) - 1$;
- F_4 reprezintă calculul de etichete pentru instrucțiuni;
- F_5 reprezintă schimbările stărilor memoriei;
- F_6 spune dacă la momentul $T_A(n)$ s-a executat instrucțiunea **success** sau nu.

Nu vom da modurile de construire a fiecărei subformule pentru cazul general, dar vom considera un exemplu. Presupunem pentru moment că A este următorul algoritm:

```
procedure A(x)
  1: if (x > 2)
  2: then y ← x*x
  3: else y ← x*x*x
  4: success
end
```

Notăm cu 0, 1, 2 locațiile de memorie care memorează pe 2, x și respectiv y . Presupunem că toate numerele sunt reprezentate binar. Rezultă: $n = \log x$, $p(n) = 3$ (polinom constant), $q(n) = 3n$ și $T_A(n) = 3^1$. Pentru a face lucrurile și mai precise, presupunem $x = 3$ care implică $n = 2$. Subformulele de mai sus sunt calculate după cum urmează:

1. Inițial, locația 0 memorează 10 (reprezentarea lui 2), iar locația 1 pe 11 (reprezentarea lui x):

$$F_1 = B_{0,0,0} \wedge \overline{B}_{0,1,0} \wedge B_{1,0,0} \wedge B_{1,1,0}$$

$$2. F_2 = S_{1,0} \wedge \overline{S}_{2,0} \wedge \overline{S}_{3,0} \wedge \overline{S}_{4,0}.$$

3. $F_3 = G_0 \wedge G_1 \wedge G_2$, unde G_t spune că la momentul t se execută exact o instrucțiune. Putem lua:

$$G_t = G_{1,t} \oplus G_{2,t} \oplus G_{3,t} \oplus G_{4,t}$$

unde $G_{k,t}$ spune că la momentul t se execută instrucțiunea k . De exemplu $G_{2,t} = \overline{S}_{1,t} \wedge S_{2,t} \wedge \overline{S}_{3,t} \wedge \overline{S}_{4,t}$.

¹ Calculul complexității timp este artificial, deoarece, în general, timpul necesar efectuării unei înmulțiri depinde de mărimea operanzilor. Pentru a simplifica prezentarea, am presupus că orice instrucțiune se realizează într-o unitate de timp.

4. $F_4 = H_0 \wedge H_1 \wedge H_2$, unde H_t spune cum se calculează adresa de instrucțiune la momentul t . Putem considera:

$$H_t = H_{1,t} \wedge H_{2,t} \wedge H_{3,t} \wedge H_{4,t}$$

cu $H_{k,t}$ reprezentând modul în care se calculează adresa dacă la momentul t se execută instrucțiunea de adresă k . De exemplu, $H_{2,t} = \overline{S}_{2,t} \vee S_{4,t+1}$.

5. $F_5 = K_0 \wedge K_1 \wedge K_2$, unde K_t arată cum se schimbă memoria la momentul t . Schimbarea memoriei depinde de instrucțiunea care se execută la momentul t :

$$K_t = K_{1,t} \wedge K_{2,t} \wedge K_{3,t} \wedge K_{4,t}$$

unde $K_{k,t}$ reprezintă modul în care se schimbă memoria dacă la momentul t se execută instrucțiunea k . De exemplu, faptul că instrucțiunea 1 lasă memoria neschimbată se exprimă prin formula:

$$K_{1,t} = \bigwedge_{i=0}^2 \bigwedge_{j=0}^{5-i} ((B_{i,j,t} \wedge B_{i,j,t+1}) \vee (\overline{B}_{i,j,t} \wedge \overline{B}_{i,j,t+1}))$$

unde $2 = p(n) - 1$, $5 = q(n) - 1$.

6. $F_6 = S_{4,3}$.

Cititorul este invitat să verifice că toate cele șase subformule pot fi determinante pentru cazul general.

Pentru o intrare x de lungime n , există un calcul al lui A care se termină cu succes numai dacă există o atribuire de valori booleene pentru variabilele $B(i, j, t)$ și $S(k, t)$ care satisface $F(A, x)$. Transformarea formulei $F(A, x)$ în formă normală conjunctivă se face cu unul dintre algoritmii cunoscuți.

sfdem

Utilizând această teoremă și tehnică reducerii, s-a putut dovedi că multe probleme sunt NP-complete. Vom mai discuta despre aceste probleme în capitolul 13.

13.3 Probleme NP-complete

Reamintim că o problemă P este NP-completă dacă:

- este în NP, i.e., există un algoritm nedeterminist care rezolvă P în timp polinomial (echivalent, există un algoritm determinist care rezolvă P în timp exponențial);
- este NP-dificilă, i.e., orice altă problemă Q din NP se reduce polinomial la P .

Avem $\mathbb{P} = \text{NP}$ dacă există un algoritm determinist care rezolvă în timp polinomial o problemă NP-completă. Până astăzi nu a fost găsit un asemenea algoritm. De fapt, toți cercetătorii din informatică teoretică consideră că egalitatea nu are loc, i.e., există inclusiunea strictă $\mathbb{P} \subset \text{NP}$ și, de aceea, pare o nebunie orice încercare de găsire a unui asemenea algoritm.

Din păcate, multe probleme practice s-au dovedit a fi NP-complete. Există probleme NP-complete în calculul numeric, în geometrie, în algebră, în procesarea

grafurilor, în procesarea sirurilor etc. De aceea, un bun proiectant de algoritmi trebuie să înțeleagă bine elementele de bază ale NP-completitudinii. Aceste elemente includ formalizarea abstractă a unei probleme, instrumente cu care se poate dovedi că o anumită problemă este NP-completă și ce trebuie făcut după ce s-a dovedit că o problemă este NP-completă.

13.3.1 Cum se poate dovedi NP-completitudinea

În [GJ79] sunt propuse următoarele tehnici prin care se poate dovedi că o problemă P este NP-completă:

Reducerea. Se arată că $P \in \text{NP}$ prin proiectarea unui algoritm nedeterminist polinomial care rezolvă P . Pentru a arăta că P este NP-dificilă se utilizează reducerea. Se știe că problema Q este NP-completă (deci NP-dificilă) și se arată că $Q \leq P$. Metoda poate fi reprezentată schematic prin:

$$(Q \text{ NP-completă}, Q \leq P) \Rightarrow P \text{ NP-completă}$$

Considerăm ca exemplu problema 3SAT care se obține din SAT impunând restricția că fiecare clauză să fie formată exact din trei literale. Faptul că 3SAT $\in \text{NP}$ rezultă din observația că orice algoritm care rezolvă SAT rezolvă de asemenea 3SAT. Vom arăta că SAT \leq 3SAT descriind modul în care se construiește o instanță a problemei 3SAT plecând de la o instanță a problemei SAT. Prezentăm, pentru câteva cazuri particulare, cum o clauză oarecare c poate fi scrisă cu ajutorul clauzelor cu exact trei literale.

1. $c = u_1$. Considerăm $c' = (u_1 \vee y_1 \vee y_2) \wedge (u_1 \vee y_1 \vee \bar{y}_2) \wedge (u_1 \vee \bar{y}_1 \vee y_2) \wedge (u_1 \vee \bar{y}_1 \vee \bar{y}_2)$, unde y_1 și y_2 sunt două variabile noi.
2. $c = u_1 \vee u_2$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_1 \vee u_2 \vee \bar{y}_1)$, unde y_1 este o variabilă nouă.
3. $c = u_1 \vee u_2 \vee u_3 \vee u_4$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee u_4 \vee \bar{y}_1)$, unde y_1 este o variabilă nouă.
4. $c = u_1 \vee u_2 \vee u_3 \vee u_4 \vee u_5$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee \bar{y}_1 \vee y_2) \wedge (u_4 \vee u_5 \vee \bar{y}_2)$, unde y_1 și y_2 sunt variabile noi.

În toate cazurile, c este satisfiabilă dacă și numai dacă c' este satisfiabilă. Extensia se referă la atribuirea de valori pentru variabilele noi y_i . Cititorul este invitat să găsească singur regula generală de construcție a clauzei c' și să arate că această construcție este făcută în timp polinomial.

Un alt exemplu îl constituie problema V -acoperirii într-un graf. O V -acoperire în graful $G = \langle V, E \rangle$ este o submulțime de vârfuri $V' \subseteq V$ cu proprietatea că fiecare muchie din E are o extremitate în V' , i.e., $(\forall \{i, j\} \in E) i \in V' \vee j \in V'$. Problema V -acoperirii este:

V -acoperire (VA)

Instanță. Un graf $G = \langle V, E \rangle$ și $k \in \mathbb{Z}_+$.

Intrebare. Există o V -acoperire V' cu $\#V' \leq k$?

Se poate arăta că 3SAT \propto VA. De exemplu, graful G construit pentru $C = (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_4) = c_1 \wedge c_2$ este reprezentat în figura 13.1. Considerăm $k = 8 = n + 2m$. Orice submulțime $V' \subseteq V$ definește o atribuire $\alpha_{V'}$ dată prin: $\alpha_{V'}(x_i) = \text{true} \iff x_i \in V'$ și $\alpha_{V'}(\bar{x}_j) = \text{false} \iff \bar{x}_j \in V'$. Dacă V' este o V -acoperire, rezultă $\{i, j\} \in E$ dacă și numai dacă $i \in V' \vee j \in V'$. Dacă $\#V' \leq 8$, atunci orice clauză c_i are numai două vârfuri în V' . Deci există un al treilea vârf care nu este în V' . Dar din acest vârf există o muchie la un x_j sau un \bar{x}_j , care apare în c_i și deci $\alpha_{V'}(x_j) = \text{true}$ sau $\alpha_{V'}(\bar{x}_j) = \text{true}$. În final, obținem că orice V -acoperire V'' cu $\#V'' \leq 8$ definește o atribuire care satisface C .

Exercițiul 13.1. Să se arate că VA este în NP.

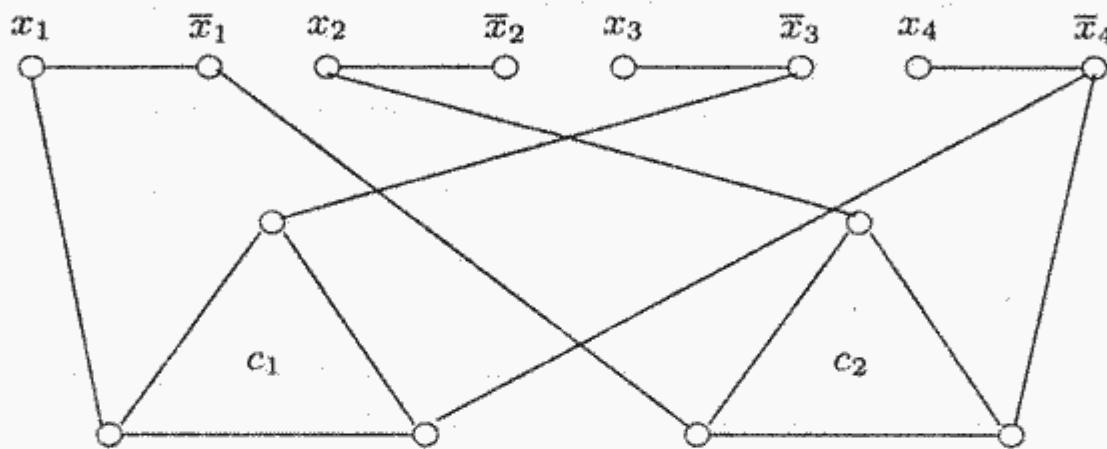


Figura 13.1: Graf asociat unei mulțimi de clauze

Restricția. Se cunoaște că problema Q este NP-completă. Se arată că P este NP-completă dovedind că Q este un caz special al problemei P , i.e., orice instanță a lui Q se obține dintr-o instanță a lui P prin adăugarea de noi restricții. Metoda poate fi reprezentată schematic prin:

$$(Q \text{ NP-completă}, Q \text{ caz special al lui } P) \Rightarrow P \text{ NP-completă}$$

Considerăm următoarele două probleme:

Circuit Hamiltonian într-un digraf (CHD)

Instanță. Un digraf $D = \langle V, A \rangle$.

Întrebare. Conține D un circuit hamiltonian (circuit care trece prin toate vâfurile din V)?

Circuit Hamiltonian într-un graf (CHG)

Instanță. Un graf $G = \langle V, E \rangle$.

Întrebare. Conține G un circuit hamiltonian?

Se știe că CHG este NP-completă (se poate arăta, de exemplu, că VA \propto CHG [GJ79]). Restricționând CHD la acele digrafuri $D = \langle V, A \rangle$ cu proprietatea că pentru orice două vârfuri $i \neq j$ există perechea de arce $\langle i, j \rangle, \langle j, i \rangle \in A$, obținem o problemă echivalentă cu CHG. De aici, rezultă că CHD este NP-completă.

Înlocuirea locală. Se bazează pe reducere. Se alege o problemă NP-completă și se pun în evidență câteva aspecte ale instanțelor. Acestea formează o colecție de unități de bază. Înlocuind fiecare unitate de bază cu o structură echivalentă, se obține o instanță a problemei P (despre care dorim să arătăm că este NP-completă). Un exemplu de aplicare a acestei tehnici îl constituie reducerea SAT \propto 3SAT.

Proiectarea componentelor. și aceasta se bazează pe reducere. Constituenții unei instanțe a problemei P sunt utilizati pentru a proiecta anumite „ componente” care apoi sunt combinate pentru a crea o instanță a unei probleme NP-complete cunoscute. Reducerea 3SAT \propto VA este un exemplu de aplicare a acestei tehnici. Constituenții unei instanțe 3SAT – clauzele și atribuirile care satisfac clauzele – au fost utilizate pentru proiectarea componentelor – graf și alegerea unei V -acoperiri.

13.3.2 Despre euristică

Ce trebuie să facem atunci când avem de rezolvat o problemă practică despre care am dovedit că este NP-completă? Din păcate nu putem da răspunsuri care să se constituie în rețete „sigure”. Pentru problemele de optimizare, *algoritmii de aproximare* (euristicile) constituie o soluție aplicată cu succes. Algoritmii de aproximare se bazează pe ideea „mai bine o soluție aproape optimă obținută în timp polinomial (deci util) decât soluția exactă obținută într-un timp foarte mare, deci inefectiv”. Cei mai mulți algoritmi de aproximare se bazează pe metoda greedy. Considerăm ca exemplu *problema comis-voiajorului*:

Comis-voiajor (CV)

Instanță. Un graf ponderat $\langle(V, E), c\rangle$ cu $c(i, j) \in \mathbb{Z}_+$ pentru orice muchie $\{i, j\} \in E$ și $K \in \mathbb{Z}_+$.

Întrebare. Există un circuit hamiltonian de cost $\leq K$?

Exercițiul 13.2. Să se arate că CV este NP-completă.

Cititorul a observat deja că problema de optimizare corespunzătoare lui CV constă în determinarea unui circuit hamiltonian de cost minim. Numele problemei vine de la următoarea modelare: un comis-voiajor trebuie să treacă prin n orașe (vârfurile grafului). O muchie în graf corespunde existenței unui drum între orașele din extremități, iar costul muchiei reprezintă lungimea drumului. Problema constă în determinarea unui tur al celor n orașe de lungime minimă (pentru ca efortul comis-voiajorului să fie minim). În cazul în care are loc o relație de tip „inegalitatea triunghiului”:

$$c(i, j) + c(j, k) \geq c(i, k)$$

se poate da o schemă de aproximare pentru CV bazată pe construcția unui arbore de cost minim (prin algoritmul lui Kruskal sau al lui Prim, exercițiul 9.9) și parcurgerea în preordine a acestuia.

Atunci când problemele au dimensiuni rezonabile, se poate aplica metoda programării dinamice sau backtracking. De exemplu, pentru problema rucsacului, varianta discretă, algoritmul backtracking are o eficiență satisfăcătoare. Dacă valoarea

M , care dă capacitatea rucsacului este mică, atunci se poate aplica cu succes și algoritmul construit prin metoda programării dinamice. De fapt, dacă M este reprezentat unar, atunci algoritmul are complexitate polinomială. Complexitatea timp este un polinom în lungimea reprezentării unare a datelor de intrare. Un asemenea algoritm este numit *algoritm cu complexitate pseudopolinomială*.

13.3.3 Câteva probleme NP-complete importante

În această secțiune prezentăm o listă cu câteva probleme NP-complete. Câteva dintre ele au fost studiate deja în capitolele anterioare. Pentru o listă mai extinsă se poate consulta [GJ79].

13.3.3.1 Mulțimi

Submulțimea de sumă dată

Instanță. O mulțime A , o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ și $K \in \mathbb{Z}_+$.

Întrebare. Există o submulțime $A' \subseteq A$ pentru care suma mărimilor elementelor din A' să fie exact K ?

Submulțimea de produs dată

Instanță. O mulțime A , o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ și $K \in \mathbb{Z}_+$.

Întrebare. Există o submulțime $A' \subseteq A$ pentru care produsul mărimilor elementelor din A' să fie exact K ?

13.3.3.2 Planificare

Planificarea procesoarelor

Instanță. O mulțime P de programe, un număr m de procesoare, un timp de execuție $t(p)$, pentru fiecare $p \in P$, și un termen D .

Întrebare. Există o planificare a procesoarelor pentru P astfel încât orice program să fie executat înainte de termenul D ?

13.3.3.3 Logică

Satisfiabilitate (SAT)

Instanță. O mulțime X de variabile și o mulțime C de clauze peste X .

Întrebare. Există o atribuire $\alpha : X \rightarrow \text{Boolean}$ care să satisfacă C ?

3-Satisfiabilitate (3SAT)

Instanță. O mulțime X de variabile și o mulțime C de clauze peste X astfel încât orice clauză $c \in C$ are $|c| = 3$.

Întrebare. Există o atribuire $\alpha : X \rightarrow \text{Boolean}$ care să satisfacă C ?

13.3.3.4 Algebră

Congruențe pătratice

Instanță. Întregii pozitivi a, b, c .

Întrebare. Există un întreg pozitiv $x < c$ astfel încât $x^2 \equiv a \pmod{b}$?

Ecuări diofantice pătratice

Instanță. Întregii pozitivi a, b, c .

Întrebare. Există $x, y \in \mathbb{Z}_+$ astfel încât $ax^2 + by = c$?

13.3.3.5 Programare matematică

Programare întreagă

Instanță. O mulțime X de perechi (x, b) , unde $x = (x_1, \dots, x_m), x_i, b \in \mathbb{Z}$, o secvență $c = (c_1, \dots, c_m)$ și un întreg K .

Întrebare. Există $y = (y_1, \dots, y_m)$ astfel încât:

1. $\sum_i x_i y_i \leq b$ pentru orice $(x, b) \in X$;
2. $\sum_i c_i y_i \geq K$?

Rucsac 0/1

Instanță. O mulțime O (obiectele), o mărime $w(o) \in \mathbb{Z}_+$ și o valoare $v(o) \in \mathbb{Z}_+$ pentru fiecare obiecte $o \in O$, o restricție $M \in \mathbb{Z}_+$ și un scop $K \in \mathbb{Z}_+$.

Întrebare. Există o submulțime $O' \subseteq O$ astfel încât

$$\sum_{o \in O'} w(o) \leq M \text{ și } \sum_{o \in O'} v(o) \geq K?$$

13.3.3.6 Grafuri

K -Colorare

Instanță. Un graf $G = \langle V, E \rangle$ și $k \in \mathbb{Z}_+$.

Întrebare. Există o colorare cu k culori a grafului G ?

V -acoperire (VA)

Instanță. Un graf $G = \langle V, E \rangle$ și $k \in \mathbb{Z}_+$.

Întrebare. Există o V -acoperire V' cu $\#V' \leq k$?

Circuit Hamiltonian într-un digraf (CHD)

Instanță. Un digraf $D = \langle V, A \rangle$.

Întrebare. Conține D un circuit hamiltonian (circuit care trece prin toate vârfurile din V)?

Circuit Hamiltonian într-un graf (CHG)

Instanță. Un graf $G = \langle V, E \rangle$.

Întrebare. Conține G un circuit hamiltonian?

Comis-voiajor (CV)

Instanță. Un graf ponderat $\langle \langle V, E \rangle, c \rangle$ cu $c(i, j) \in \mathbb{Z}_+$ pentru orice muchie $\{i, j\} \in E$ și $K \in \mathbb{Z}_+$.

Întrebare. Există un circuit hamiltonian de cost $\leq K$?

Cel mai lung drum

Instanță. Un graf ponderat $G = \langle V, E \rangle$ cu ponderele $\ell : E \rightarrow \mathbb{Z}_+$, un întreg pozitiv K și două vârfuri distincte i și j .

Întrebare. Există în G un drum de la i la j de lungime $\geq K$?

13.4 Exerciții

Exercițiu 13.3. Să se proiecteze un algoritm cu complexitatea timp pseudopolynomială care rezolvă problema determinării unei submulțimi de sumă dată.

Exercițiu 13.4. [CLR93] Se consideră următorul algoritm greedy ca algoritm de aproximare pentru V -acoperire: pasul de alegere greedy selectează vârful cu cel mai mare grad și elimină toate muchiile incidente în el. Pentru o instanță dată, notăm cu C soluția dată de algoritmul de aproximare și cu C^* soluția optimă.

1. Să se găsească un exemplu pentru care $C \neq C^*$.
2. Să se găsească un exemplu pentru care $\frac{\#C}{\#C^*} > 2$.

Exercițiu 13.5. Să se proiecteze un algoritm eficient care determină o V -acoperire optimă pentru un arbore în timp liniar.

13.5 Referințe bibliografice

Pentru o tratare mai detaliată a subiectului recomandăm [GJ79, Mel84b, LG86, Koz92, AHU74, HS84, Giu04]. O trecere în revistă a algoritmilor de aproximare, în special pentru probleme de combinatorică și de programare liniară, poate fi găsită în [Vaz04]. Pentru o abordare sistematică a problemelor dificile, incluzând algoritmi aleatorii, poate fi consultată [Hro04]. O paletă mai largă de euristică se găsește în [MF04].

Bibliografie

- [AHU74] A.V. Aho, J.E. Hopcroft și J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
- [AA78] S. Alagic și M.A. Arbib. *The Design of Well-Structured and Correct Programs*. Springer-Verlag, 1978.
- [AS84] S. Arikawa și S. Shiraishi. „Pattern Matching Machines for Replacing Several Character Strings”, *Bulletin of Informatics and Cybernetics*, volumul 21, pp. 101-112, martie 1984.
- [Baa78] Sarah Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, Massachusetts, 1978.
- [BD62] R.E. Bellman și S.E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.
- [BG00] S. Baase și A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, Massachusetts, 2000.
- [CB02] M. Craus și C. Bărsan. *Structuri de date și algoritmi*. Editura „Gh. Asachi”, Iași, 2002.
- [CLR93] T.H. Cormen, C.E. Leiserson și R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1993.
- [CLR00] T.H. Cormen, C.E. Leiserson și R.L. Rivest. *Introducere în algoritmi*. Computer Libris Agora, 2000.
- [Coh90] E. Cohen. *Programming in the 1990s: An Introduction to the Calculation of the Programs*. Springer-Verlag, 1990.
- [Cro92] Cornelius Croitoru. *Tehnici de bază în optimizarea combinatorie*. Editura Universității „Al.I. Cuza”, Iași, 1992.
- [Cro02] Cornelius Croitoru. *Introducere în proiectarea algoritmilor paraleli*. Editura Matrix Rom, București, 2002.
- [Giu04] C.A. Giu male. *Introducere în ANALIZA ALGORITMILOR. Teorie și aplicație*. Polirom, 2004.
- [GJ79] M.R. Garey și D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman and Company, San Francisco, 1979.
- [Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [HE87] G.H. Hale și R.J. Easton. *Applied Data structures Using Pascal*. D.C.Heath and Company, 1987.
- [Hei96] G. Heileman. *Data Structures, Algorithms and Object-Oriented Programming*. McGraw-Hill, 1996.

- [Hro04] J. Hromkovic. *Algorithmics for Hard Problems*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.
- [HS84] E. Horowitz și S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1984.
- [HSAF93] E. Horowitz, S. Sahni și S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, 1993.
- [Kao08] M.-Y. Kao. *Encyclopedia of Algorithms*. Springer, 2008.
- [KGGK94] V. Kumar, A. Grama, A. Gupta și G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.
- [Knu74] D.E. Knuth. *Algoritmi fundamentali*, volumul 1: *Tratat de programarea calculatoarelor*. Editura Tehnică, București, 1974.
- [Knu76] D.E. Knuth. *Sortare și căutare*, volumul 3: *Tratat de programarea calculatoarelor*. Editura Tehnică, București, 1976.
- [Koz92] D.C. Kozen. *The Design and Analysis of Algorithms*. Springer Verlag, 1992.
- [Kro79] L.I. Kronsjo. *Algorithms: Their Complexity and Efficiency*. John Wiley & Sons, 1979.
- [LG86] L. Livovschi și H. Georgescu. *Sinteza și analiza algoritmilor*. Editura Științifică și Enciclopedică, București, 1986.
- [Luc93] D. Lucanu. *Programarea algoritmilor: Tehnici elementare*. Editura Universității „Al.I. Cuza”, Iași, 1993.
- [Luc96] D. Lucanu. *Bazele proiectării programelor și algoritmilor*. Curs multiplicat la Editura Universității „Al.I. Cuza”, Iași, Romania, 1996, volumul 1: *Modele de calcul*, volumul 2: *Tehnici de programare*, volumul 3: *Proiectarea algoritmilor*.
- [Man89] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, 1989.
- [MB00] R. Miller și L. Boxer. *Algorithms Sequential & Parallel: A Unified Approach*. Prentice Hall, New Jersey, 2000.
- [Mel79] K. Melhorn. Dynamic data structures. În J.W. de Bakker și J. van Leeuwen (ed.), *Foundations of Computer Science III*, volumul 1. Mathematical Centre Tracts, Amsterdam, 1979.
- [Mel84a] K. Melhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer Verlag, 1984.
- [Mel84b] K. Melhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer Verlag, 1984.
- [MF04] Z. Michalewicz și D.B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2004.
- [MS91] B.M.E. Moret și H.D. Shapiro. *Algorithms from P to NP: Design and Efficiency*. The Benjamin/Cummings Publishing Company, Inc., 1991.
- [Sah98] S. Sahni. *Data Structures and Algorithms in C++*. WCB McGraw-Hill, 1998.
- [Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1988.
- [Vaz04] V.V. Vazirani. *Approximation Algorithms*. Springer, 2004.

- [Wei92] M.A. Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin/Cummings Publishing Company, Inc., 1992.
- [Wan80] M. Wand. *Induction, Recursion and Programming*. North Holland, 1980.
- [Woo84] D. Wood. *Paradigms and Programming with Pascal*. Computer Science Press, 1984.



Index

- 3SAT, 349, 352
- algoritm, 13
 - Strassen, 274
 - complexitate pseudo-polinomială –, 352
 - reducere polinomială, 345
 - timpul de execuție –, 31
 - timpul mediu de execuție –, 34
- analizaă amortizată, 104
 - metoda agregării, 104
 - metoda conturilor, 106
 - metoda potențialului, 106
- arbore, 86
 - 2-3, 171
 - AVL, 161
 - B, 179
 - bicolor, 168
 - binar complet, 79
 - cu rădăcină, 86
 - de decizie
 - pentru căutare, 150
 - pentru sortare, 135
 - lungimea externă a unui –, 151
 - lungimea internă a unui –, 151
 - digital (trie), 190
 - echilibrat, 161
 - ordonat, 86
 - ponderat, 228
- arbore binar, 71
 - dimensiunea unui –, 71
 - frontiera unui –, 71
 - înălțimea unui –, 71
- arbore binar de căutare
 - orientat pe frontieră, 158
 - orientat pe noduri interne, 157
- arbore parțial
 - BFS, 98
 - DFS, 96
- arbore ponderat pe frontieră,
 - vezi arbore ponderat, 228
- arce, 85
- axioma de accesibilitate,
 - vezi sistem accesibil, 246
- backtracking, 309, 319
- Bellman-Ford
 - algoritmul –, 285, 304
- Boyer-Moore, algoritmul –, 205
- branch and bound
 - cu cost minim, 322
 - cu mărginire, 323
- branch-and-bound, 309, 320
- Bubble Sort,
 - vezi sortare prin interschimbare, 124
- Bucket Sort,
 - vezi sortare prin distribuire, 141
- bucătă într-un graf, 83
- căutare
 - în mulțimi total ordonate
 - Fibonacci, 150
 - binară, 150
 - divide et impera, 149
 - liniară, 150
 - prin interpolare, 154
 - arbore de decizie, 150
 - aspect dinamic, 147
 - aspect static, 147
 - binară, 33
 - peste siruri, 201
- ciclu într-un graf, 84
- circuit într-un graf, 84

- clasă de arbori stabilă, 161
- clauză, 346
 - satisfacerea unei –, 346
- coada, 62
- coadă cu priorități, 79
- coliziune,
 - vezi* dispersie, 185
- configurație, 26
- configurație finală, 26
- configurație inițială, 26
- criteriu de alegere locală (greedy), 246
- diametrul unui arbore, 274
- digraf, 85
 - etajat, 280
 - etichetat, 85
 - ponderat, 85
 - tare conex, 85
 - tipul abstract –, 88
 - parcugerea sistematică a –, 94
- Dijkstra
 - algoritmul –, 250, 285
- dispersie, 184
 - cu înlănuire, 186
 - cu adresare deschisă, 188
- distribuire prin segmentare,
 - vezi* sortare prin –, 141
- divide-et-impera, 253
- drum închis într-un graf, 84
- drum intr-un graf, 84
- drum minim intr-un digraf, 282
- enumerare
 - a permutărilor
 - nerecursivă, 314
 - recursivă, 313
- euristică, 351
- execuția programelor, 24
- explorarea BFS,
 - vezi* parcugerea sistematică a digrafurilor, 97
- explorarea DFS,
 - vezi* parcugerea sistematică a digrafurilor, 95
- expresie regulată, 208
- FFT,
 - vezi* transformata Fourier rapidă, 271
- Fibonacci
 - căutare –, 150
 - funcția lui –, 117
- Floyd-Warshall
 - algoritmul –, 285
- Fourier,
 - vezi* transformata –, 270
- funcție
 - de dispersie, 185
- funcție obiectiv, 302
- graf, 83
 - conex, 84
 - etichetat, 85
 - ponderat, 85
 - reprezentarea unui – ca digraf, 89
 - tipul abstract –, 86
- graf general (pseudograf), 83
- greedy, 225
- hash,
 - vezi* dispersie, 184
- Heap Sort
 - vezi* sortarea prin selecție, 131
- interpolare,
 - vezi* căutare în mulțimi total ordonate, 154
- Knuth-Morris-Pratt, algoritmul –, 202
- Kruskal, algoritmul lui –, 244
- limbaj algoritmice, 13
- lista circulară, 55
- lista generalizată, 66
- lista liniară, 41
- lista liniră, - ordonată 51
- liste de adiacență, 92
 - dinamice, 92
 - tablouri, 98
- literal, 346
- lungime externă,
 - vezi* arbore de decizie, 151

- lungime externă ponderată,
 vezi arbore ponderat, 228
- lungime internă,
 vezi arbore de decizie, 151
- mărimea unei instanțe, 30
- matrice de adiacență, 89
- matroid, 247
 - ponderat, 247
- max-heap, 79
- maxim
 - intr-un tablou, 32
- memorie, 14
- Merge Sort,
 - vezi sortare prin interclasare*, 255
- mers într-un graf, 84
- metoda bulelor,
 vezi sortare prin interschimbare, 124
- metoda par-impar,
 vezi sortare prin interschimbare, 126
- modelul arborilor de decizie pentru căutare, 150
- modelul arborilor de decizie pentru sortare, 135
- muchie (în graf), 83
- muchie incidentă, 83
- multime
 - parțial ordonată,
 vezi sortare topologică, 144
- multigraf, 83
- pădure, 101
- Par-Impar Sort,
 - vezi sortare prin interschimbare*, 126
- paradigme de proiectare a algoritmilor, 217
- parcurgerea în inordine nerecursivă, 109
- Patricia, 196
- pattern generalizat, 208
- politică (optimă), 302
- principiul de optim, 302
- problemă
 - NP-completă, 345, 348
 - NP-dificilă, 345, 348
 - instantă a unei –, 30
- problemă, 28
 - decidabilă, 29
 - nedecidabilă, 29
 - nerezolvabilă, 29
 - parțial decidabilă, 29
 - rezolvabilă, 29
 - semidecidabilă, 29
- problema
 - celor n regine, 327
 - 3-satisfiabilității, 352
 - K-colorării, 353
 - V-acoperirii, 350, 353
 - închisorii, 342
 - înmulțirii optime a unui sir de matrici, 305
 - înmulțirii polinoamelor, 274
 - alocării optime a fișierelor, 248
 - alocării resurselor, 279
 - arborelui parțial de cost minim, 244
 - celebrității, 218
 - circuitului hamiltonian, 350, 353
 - colorării grafurilor, 325
 - comisului voiajor, 351, 353
 - conectării tranzistorilor, 341
 - congruenței pătratice, 353
 - determinării celui de-al n -lea număr Fibonacci, 274
 - determinării minimului și maximului simultan, 274
 - distanței între siruri, 295
 - dominoului,
 vezi problema mozaicului, 341
 - drumului de lungime maximă într-un graf, 353
 - ecuațiilor diofantice pătratice, 353
 - evaluării polinoamelor, 274
 - fortăreței, 342
 - galeriilor de artă, 341
 - instructorului de schi, 242
 - interclasării optime, 234
 - labirintului, 340

- liniei orizontului, 264
- lungimii externe ponderate optime, 228
- mărcilor poștale, 340
- memorării programelor, 225
- memorării programelor II, 249
- mozaicului, 341
- mutării calului de săh, 341
- opririi, 29
- permutării maximale, 219
- planificării procesoarelor, 352
- procesării paralele, 341
- programării întregi, 353
- rucsacului, varianta continuă, 236
- rucsacului, varianta discretă, 285, 332
- satisfiabilității, 352
- schimbării banilor, 249, 340
- secvențializării optime a activităților, 239
- selecției, 262
- submulțimii de produs dată, 352
- submulțimii de sumă dată, 330, 352
- subsecvenței crescătoare maxime, 293
- triangularizării optime a unui poligon, 306
- turului bitonic minim, 306
- Rucsac 0/1, 343, 353
- program, 14
- programare dinamică, 278, 301
- proprietatea de ereditate,
 vezi matroid, 247
- proprietatea de alegere locală, 227
- proprietatea de interschimbare,
 vezi matroid, 247
- pseudograf, 83
- Quick Sort,
 vezi sortare rapidă, 258
- Rabin-Karp, algoritmul -, 206
- Radix Sort,
 vezi sortare prin distribuire, 139
- recursie, 113
- în cascadă, 116
- liniară, 113
- reprezentarea expresiilor aritmetice, 78
- SAT, 346, 352
- secvență *h*-ordonată, 129
- selecție naivă,
 vezi sortare prin selecție, 131
- selecție sistematică,
 vezi sortare prin selecție, 131
- selecția sistematică, 131
- sistem accesibil, 246
- sortare
 - a cuvintelor, 138
 - cu micșorarea incrementului, 128
 - externă, 123
 - externă, 234
 - internă, 123
 - metoda inserției, 32
 - prin distribuire, 138
 - prin inserție, 127
 - prin interclasare, 255
 - prin interschimbare, 124
 - prin numărare, 142
 - prin selecție, 130
 - rapidă, 258
 - topologică, 144
- sortare prin distribuire, 138
- stabilitate a algoritmilor de sortare, 123
- stiva, 57
- structură
 - dinamică înlănțuită, 22
 - dinamică dublu înlănțuită, 24
 - dinamică simplu înlănțuită, 24
 - statică, 22
- subgraf, 83
 - induc, 83
 - parțial, 83
- substructură optimă, 278, 303
- swap, 19
- tabelă de dispersie, 185
- tabelă de simboluri,
 vezi dispersie, 184
- tablou, 20
 - 1-dimensional, 20

- 2-dimensional, 21
- cu auto-organizare, 148
- tehnici de derecursivare, 109
- tip de date, 13
 - abstract, 13
 - domeniul unui -, 13
- transformata Fourier, 270
 - rapidă, 271
- transformata Fourier discretă, 270
- union-find, structura -, 103
- vârf (în graf), 83
- vârfuri adiacente, 83
- vârfuri pendante,
 - vezi* arbore de decizie, 150
- variabilă, 14
 - dinamică, 15

