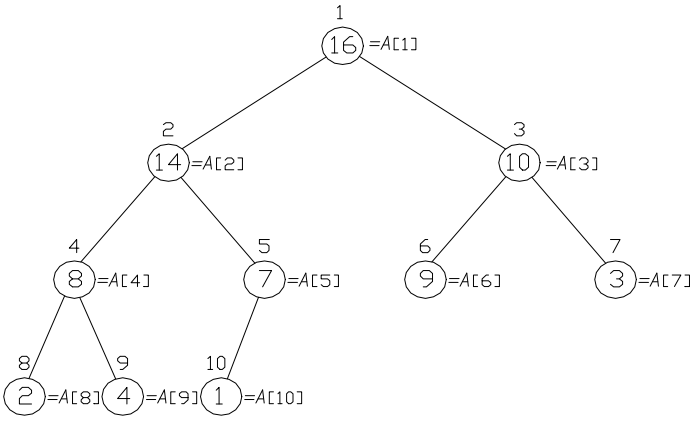
Structuri arborescente (cont.)

HEAP-URI

O structură de tip *heap* (***descendent***) este un **arbore binar aproape complet**, cu următoarea proprietate, numită *proprietate de heap*: **cheia fiecărui nod este mai mare sau egală decât cheia fiecărui fiu al său**. O structură de tip *heap ascendent* este un heap în care fiecare nod are asociată o cheie mai mică sau egală decât cheia fiecărui fiu al său

În continuare se va considera tipul de date heap descendent.



Structura de date heap (binar) poate fi privit şi ca un vector care poate fi vizualizat sub forma unui arbore binar aproape complet (figura 1.1). Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate nodurilor. Arborele este plin, exceptând eventual nivelul inferior, care **este plin de la stânga la dreapta doar până la un anumit loc**.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
A:	**16**	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

Figura 1.1 Un heap reprezentat sub forma unui vector şi sub forma unui arbore. Numerele înscrise în cercurile reprezentând nodurile arborelui sunt valorile ataşate nodurilor, iar cele scrise lângă cercuri sunt indicii elementelor corespunzătoare din vector.

Un **vector *A* care reprezintă un heap** are două atribute: ***lungime*(*A*), care reprezintă numărul elementelor din vector** şi *dimensiune_heap*(*A*) care reprezintă numărul elementelor heap-ului memorat în vectorul *A*. Astfel, chiar dacă *A*[1],…, *A*[*lungime*(*A*)] conţine în fiecare element al său date valide, este posibil ca elementele următoare elementului *A*[*dimensiune_heap*(*A*)], unde *dimensiune_heap*(*A*) ≤ *lungime*(*A*), să nu aparţină heap-ului. **Rădăcina arborelui este *A*[1]. Dat fiind un indice *i*, corespunzător unui nod, se pot determina uşor indicii părintelui acestuia PĂRINTE(i), al fiului STÂNGA(i) şi al fiului DREAPTA(i).**

$$\text{PĂRINTE}(i) = \begin{cases} [i/2], \text{ dacă } i > 1 \\ \text{nu există, dacă } i = 1 \end{cases}$$

$$\text{STÂNGA}(i) = \begin{cases} 2i, \text{ dacă } 2i \le \text{lungime}(A) \\ \text{nu există, dacă } 2i > \text{lungime}(A) \end{cases} \quad \text{(fiul stânga)}$$

$$\text{DREAPTA}(i) = \begin{cases} 2i+1, \text{ dacă } 2i+1 \le \text{lungime}(A) \\ \text{nu există, dacă } 2i+1 > \text{lungime}(A) \end{cases} \quad \text{(fiul dreapta)}$$

sau

PĂRINTE(i)
    returnează $\lfloor i/2 \rfloor$
STÂNGA(i)
    returnează $2i$
DREAPTA(i)
    returnează $2i+1$

În cele mai multe cazuri, funcţia STÂNGA poate calcula valoarea **2*i*** cu o singură instrucţiune, **translatând reprezentarea binară a lui *i* la stânga cu o poziţie binară**. Similar, funcţia DREAPTA poate determina rapid valoarea **2*i* + 1**, **translatând reprezentarea binară a lui *i* la stânga cu o poziţie binară, iar bitul nou intrat pe poziţia binară cea mai nesemnificativă va fi 1**. În funcţia **PĂRINTE valoarea se va calcula prin translatarea cu o poziţie binară la dreapta a reprezentării binare a lui *i***.

Pentru orice nod *i*, diferit de rădăcină, ***proprietate de heap*** se poate scrie:

$$A[\text{PĂRINTE}(i)] \ge A[i] \qquad (1.1)$$

adică valoarea ataşată nodului este mai mică sau egală cu valoarea asociată părintelui său. Astfel cel mai mare element din heap este păstrat în rădăcină, iar valorile nodurilor oricărui subarbore al unui nod sunt mai mici sau egale cu valoarea nodului respectiv.

Reconstituirea proprietăţii de heap

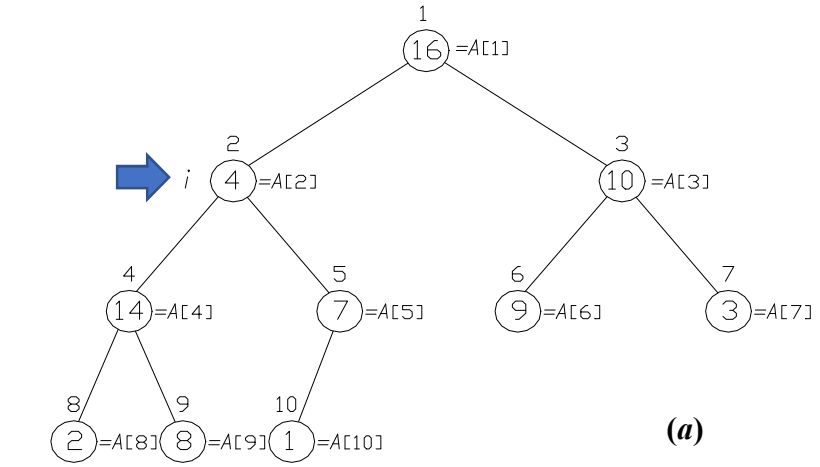
Funcţia RECONSTITUIE_HEAP este un subprogram important în prelucrarea heap-urilor. Datele de intrare ale acesteia sunt un vector *A* şi un indice *i* din vector. Atunci când se apelează RECONSTITUIE_HEAP, **se presupune că subarborii, având ca rădăcini nodurile STÂNGA(*i*), respectiv DREAPTA(*i*), sunt heap-uri.** Dar, cum elementul ***A*[*i*] poate fi mai mic decât descendenţii săi, acesta nu respectă proprietatea de heap** (1.1).

Sarcina funcţia RECONSTITUIE_HEAP este de a "scufunda" în heap valoarea $A[i]$, astfel încât subarborele care are în rădăcină valoarea elementului de indice *i*, să devină un heap.

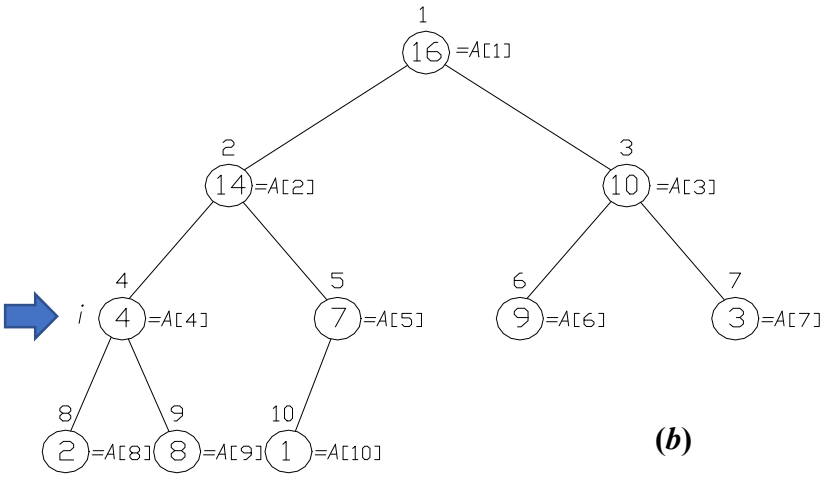
RECONSTITUIE_HEAP (*A*, *i*)
Pas 1. *l* ← STÂNGA(*i*)
Pas 2. *r* ← DREAPTA(*i*)
Pas 3. dacă *l* ≤ *dimensiune_heap*(*A*) şi ***A*[*l*] > *A*[*i*] atunci *maxim* ← *l***
altfel ***maxim* ← *i***
Pas 4. dacă *r* ≤ *dimesiun_heap*(*A*) şi ***A*[*r*] > *A*[*maxim*] atunci *maxim* ← *r***
Pas 5. dacă *maxim* ≠ *i* atunci schimbă ***A*[*i*] ↔ *A*[*maxim*]**
altfel stop procedură.
Pas 6. RECONSTITUIE_HEAP (*A*, *maxim*)

La fiecare pas **se determină cel mai mare element dintre *A*[*i*], *A*[STÂNGA(*i*)] şi *A*[DREAPTA(*i*)],** iar indicele său se păstrează în variabila ***maxim***. **Dacă *A*[*i*] este cel mai mare, atunci subarborele având ca rădăcină nodul *i* este un heap** şi funcţia se termină. **În caz contrar, cel mai mare element este unul dintre cei doi descendenţi şi *A*[*i*] este interschimbat cu *A***. Astfel, nodul *i* şi descendenţii săi satisfac proprietatea de heap. Nodul *maxim* are acum valoarea iniţială a lui *A*[*i*], deci este posibil ca subarborele de rădăcină *maxim* să nu îndeplinească proprietatea de heap. Rezultă că funcţia RECONSTITUIE_HEAP trebuie apelată recursiv din nou pentru acest subarbore.

Exemplu: În figura 1.2 este ilustrat efectul funcţia RECONSTITUIE_HEAP.



(*a*)



(*b*)

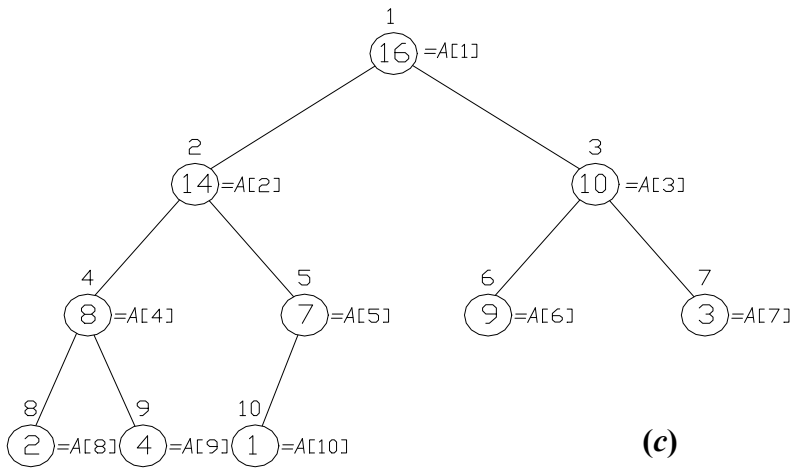


Figura 1.2

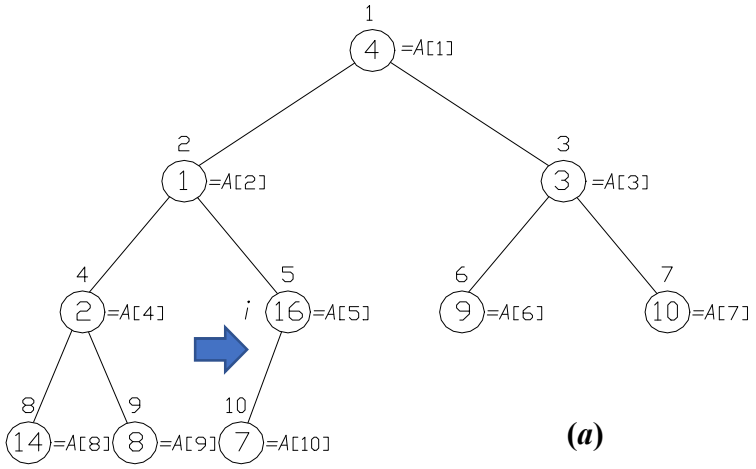
- Efectul procedurii RECONSTITUIE_HEAP(*A*,2), unde *dimensiune_heap* (*A*) = 10 (**a**)
- Configuraţia iniţială a heap-ului, unde *A*[2] (pentru nodul *i*=2) nu respectă proprietatea de heap, deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilită pentru nodul 2 în (**b**) prin interschimbarea lui *A*[2] cu *A*[4], ceea ce anulează proprietatea de heap pentru nodul 4.
- Apelul recursiv al procedurii RECONSTITUIE_HEAP(*A*, 4) poziţionează valoarea lui *i* pe 4. După interschimbarea lui *A*[4] cu *A*[9], aşa cum se vrea în (**c**), nodul ajunge la locul său şi apelul recursiv RECONSTITUIE_HEAP(*A*, 9) nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

Construirea unui heap

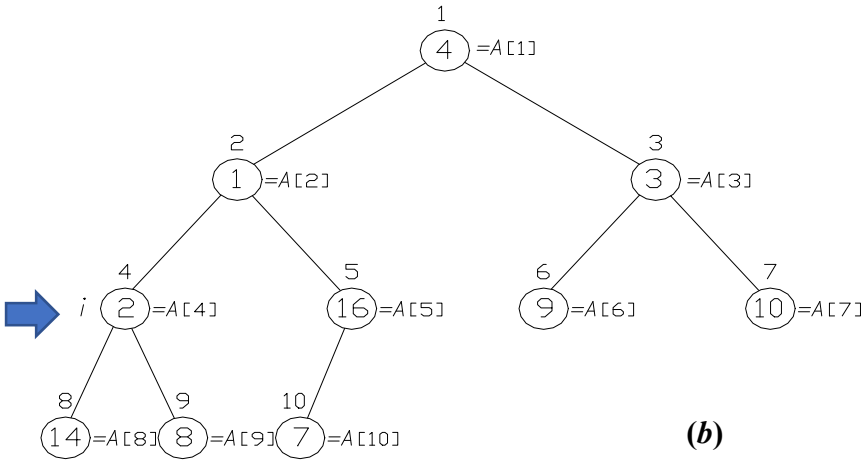
Procedura RECONSTITUIE_HEAP poate fi utilizată "de jos în sus" pentru **transformarea vectorului *A*[1], …, *A*[*n*] în heap**, unde *n* = *lungime*(*A*). Deoarece toate elementele subşirului ***A*[[*n*/2]+1], …, *A*[*n*] sunt frunze**, acestea pot fi considerate ca fiind heap-uri formate din câte un element. Astfel, procedura CONSTRUIEŞTE_HEAP trebuie să traverseze doar restul elementelor şi să execute procedura RECONSTITUIE_HEAP pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor asigură ca subarborii, având ca rădăcină descendenţi ai nodului *i* să formeze heap-uri înainte ca RECONSTITUIE_HEAP să fie executat pentru aceste noduri.

CONSTRUIEŞTE_HEAP (*A*)
Pas 1. *dimensiune_heap*(*A*) ← *lungime*(*A*)
Pas 2. pentru ***i* ← *lungime*(*A*) / 2, …, 1** execută pasul 3 (**de jos în sus** în arbore)
Pas 3. **RECONSTITUIE_HEAP(*A*, *i*)**

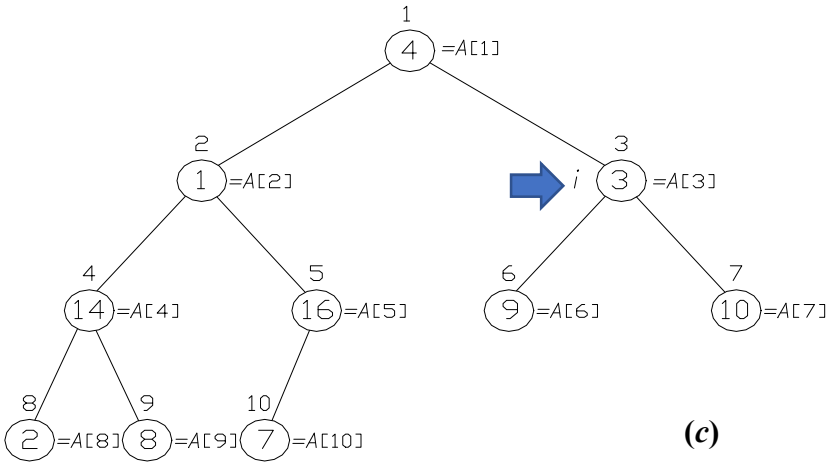
Figura 1.3 ilustrează modul de funcţionare al procedurii CONSTRUIEŞTE_HEAP.


1
4 =A[1]
2
1 =A[2]
3
3 =A[3]
4
2 =A[4]
i
5
16 =A[5]
6
9 =A[6]
7
10 =A[7]
8
14 =A[8]
9
8 =A[9]
10
7 =A[10]

(*a*)


1
4 =A[1]
2
1 =A[2]
3
3 =A[3]
i
4
2 =A[4]
5
16 =A[5]
6
9 =A[6]
7
10 =A[7]
8
14 =A[8]
9
8 =A[9]
10
7 =A[10]

(*b*)


1
4 =A[1]
2
1 =A[2]
i
3
3 =A[3]
4
14 =A[4]
5
16 =A[5]
6
9 =A[6]
7
10 =A[7]
8
2 =A[8]
9
8 =A[9]
10
7 =A[10]

(*c*)

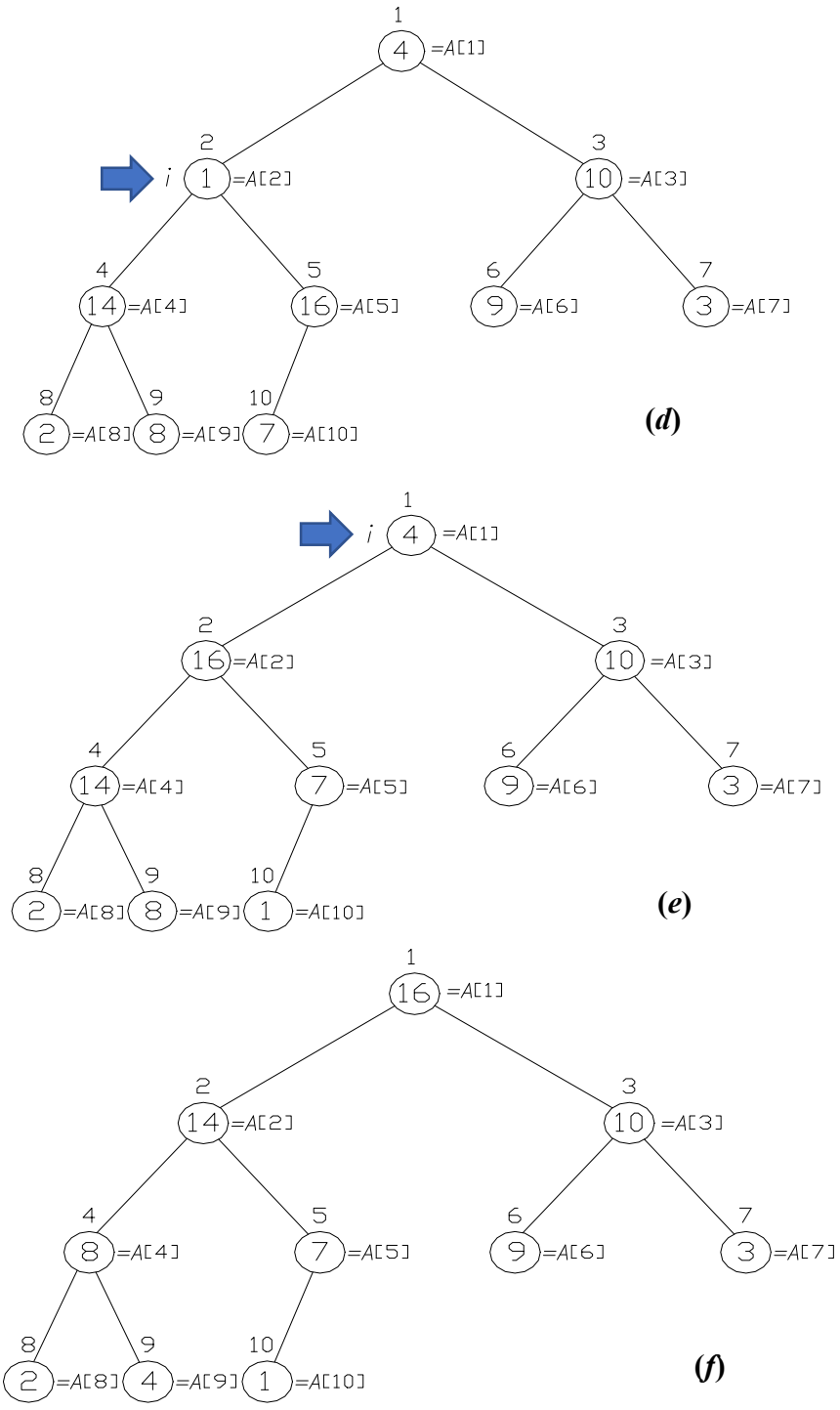


Figura 1.3 Modul de execuţie a procedurii CONSTRUIEŞTE_HEAP. În figură se vizualizează structuri de date în starea lor anterioară apelului procedurii RECONSTITUIE_HEAP (linia 3 din procedura CONSTRUIEŞTE_HEAP). **(a)** Se consideră un vector A având 10 elemente şi arborele binar corespunzător. După cum se vede în figură, variabila de control i a ciclului, în momentul apelului RECONSTITUIE_HEAP(A,i), indică nodul 5. **(b)** reprezintă rezultatul; variabila de control i a ciclului acum indică nodul 4. **(c) – (e)** vizualizează iteraţiile succesive ale ciclului "pentru" din CONSTRUIEŞTE_HEAP. Se observă că, atunci când se apelează procedura RECONSTITUIE_HEAP pentru un nod dat, subarborii acestui nod sunt deja heap-uri. **(f)** reprezintă heap-ul final al procedurii CONSTRUIEŞTE_HEAP.

Arbori echilibraţi. Arbori AVL.

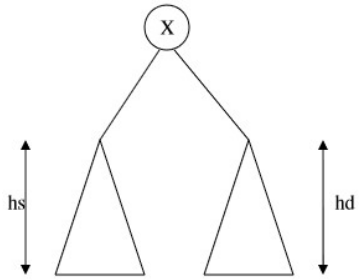
Arborii de căutare sunt adecvaţi situaţiilor în care structura arborilor nu este modificată şi se folosesc numai pentru căutări ale informaţiei memorate în nodurile structurii. Acest lucru, deoarece în cazurile în care structura arborelui ar fi modificată prin inserarea sau eliminarea unui nod, criteriul de optim la căutare nu ar mai fi respectat deci arborele ar trebui reorganizat pentru respectarea criteriului de optim.

O astfel de reorganizare este complicată şi necesită consum de timp mai ales când apar frecvent operaţii de eliminare sau inserare de noduri.

Din acest motiv se preferă un alt tip de arbori: arbori echilibraţi. Această structură asigură un timp de căutare bun şi evită degenerarea structurii.

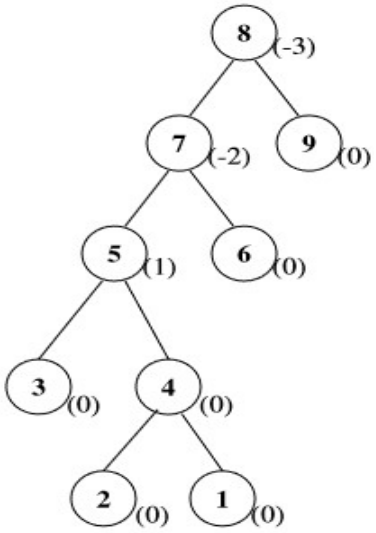
Structura acestor arbori poate fi modificată la operaţiile de inserare sau eliminare a unui nod relativ uşor.

Definiţie: Un arbore este ***echilibrat*** (în înălţime) dacă şi numai **dacă înălţimile celor doi subarbori ai săi diferă cu cel mult o unitate** (pentru un nod i dintr-un arbore A putem scrie $-1 \leq \lg(S^S(i)) - \lg(S^D(i)) \leq 1$, unde $\lg(S)$ este înălţimea subarborelui S).

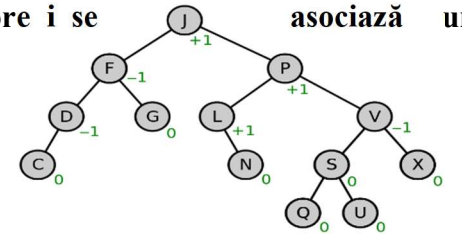


Definiţie: **Un arbore binar de căutare echilibrat se numeşte *arbore AVL*** (după iniţialele celor care l-au descoperit: Adelsohn, Velskii şi Landis).

De exemplu, arborele de mai jos nu este AVL (lângă fiecare nod, în paranteză apare diferența dintre înălțimile subarborelui stând și drept.



Se notează cu e **diferența dintre înălțimea subarborelui stâng și înălțimea subarborelui drept** $\lg(S^S(i))- \lg(S^D(i))=e$, iar **fiecărui nod *i* din arbore i se asociază un factor de reechilibrare**:



$$F = \begin{cases} +, & e = -1 \\ \cdot, & e = 0 \\ -, & e = 1 \end{cases}$$

atunci cele trei variante de arbori echilibraţi vor fi prezentate în figurile următoare.

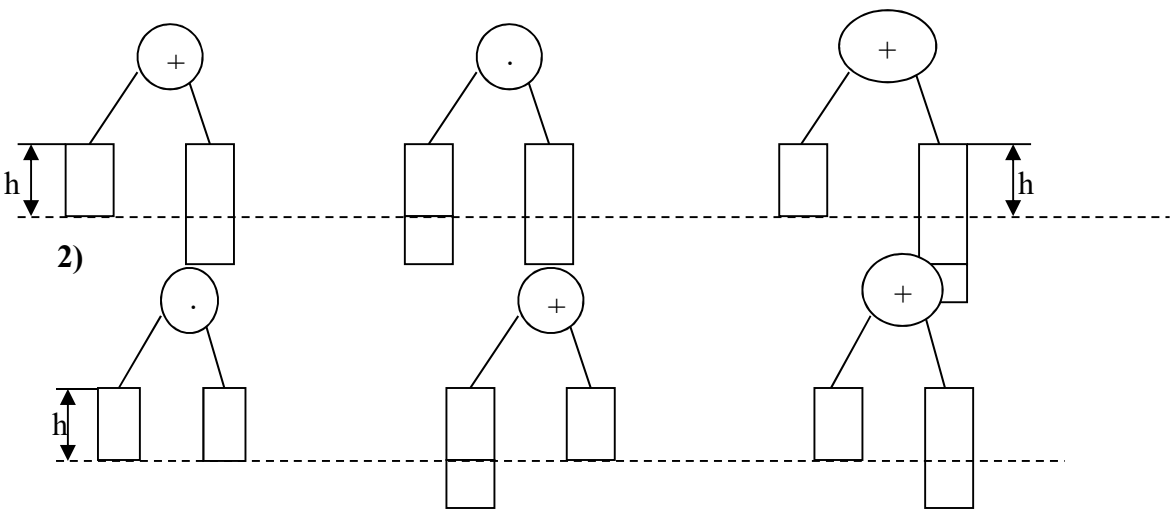
Se poate observa că inserarea sau eliminarea unui nod dintr-un arbore echilibrat poate strica echilibrul acestuia şi deci este necesară reechilibrarea arborelui.

La prima vedere ar părea că sunt necesare transformări la nivelul fiecărui nod; de fapt, procesul de reechilibrare este destul de simplu şi eficace.

Avem următoarele situaţii:

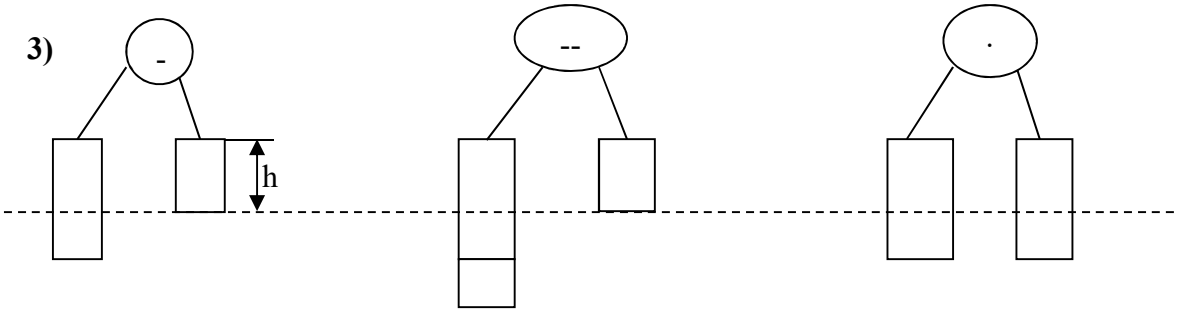
înainte de inserare — după inserare b) în S^S — după inserare c) în S^D

1)



2)

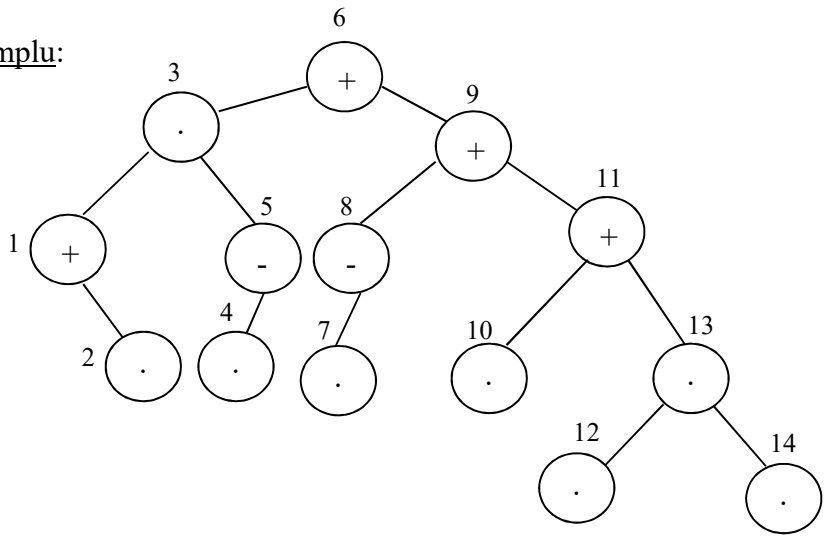
3)



Asupra unui arbore echilibrat se pot efectua următoarele operaţii:

a) căutarea unui nod cu o cheie dată;
b) inserarea unui nod cu o cheie dată;
c) ştergerea unui nod cu o cheie dată.

Exemplu:



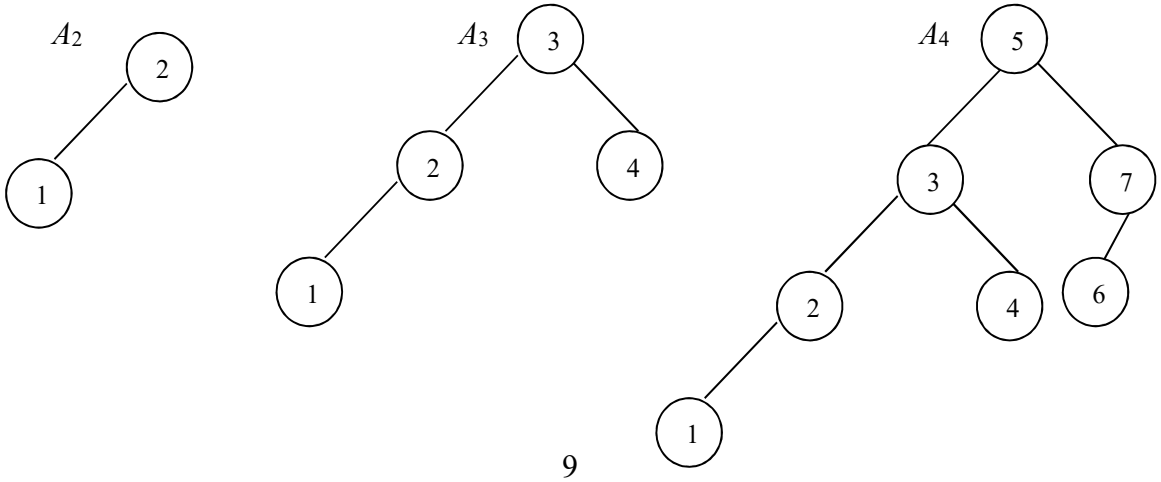
Pentru a găsi înălţimea maximă h a tuturor arborilor echilibraţi cu n noduri se consideră o anumită înălţime h' şi se încearcă construirea arborelui echilibrat având numărul minim de noduri.

Această metodă este recomandabilă deoarece, ca şi la arborele cu h minim, valoarea dorită a lui h nu poate fi atinsă numai pentru anumite valori specifice ale lui n. Vom nota arborele AVL de înălţime h cu A_h.

Rezultă că A_0 este arborele vid, A_1 arborele cu un nod, etc.

Pentru a construi arborele A_h cu h>1, se va porni de la rădăcină, căreia i se vor ataşa 2 subarbori care la rândul lor vor avea un număr minim de noduri. Aceşti 2 subarbori sunt la rândul lor arbori echilibraţi. Evident unul din subarbori trebuie să aibă înălţimea h-1, iar celuilalt subarbore îi este permisă o înălţime cu o unitate mai mică (h-2).

Exemplu:



Propoziţie: **Numărul minim de noduri ale unui arbore AVL de înălţime dată *h* satisface relaţia de recurenţă:**

$$\mathbf{NAVL_{min}(\mathit{h}) = NAVL_{min}(\mathit{h}\text{-}1) + NAVL_{min}(\mathit{h}\text{-}2) + 1}$$

Demonstraţie: Relaţia se demonstrează prin inducţie după *h*.

Evident relaţia este adevărată pentru h=2 şi h=3 ($NAVL_{min}(0)=0$, $NAVL_{min}(1)=1$, $NAVL_{min}(2)=2$, $NAVL_{min}(3)=4$).

Să presupunem că la un moment dat au fost construiţi arbori AVL de înălţime strict mai mică decât *h* şi care îndeplinesc relaţia de recurenţă din enunţ. Pentru a construi un arbore AVL de înălţime *h* se adaugă ca subarbori ai rădăcinii (notată *R*) arborii AVL de înălţime *h*-2, respectiv *h*-1, construiţi anterior. Evident, acest arbore are numărul de noduri:

$N=1+N(S^S(R))+ N(S^D(R))$.

Se observă că dacă din acest arbore se înlătură oricare dintre noduri., el nu mai satisface proprietatea de echilibru AVL (fie unul din subarborii rădăcinii nu mai este arbore AVL, fie se obţine un arbore de înălţime *h*-1).

Observaţie: Relaţia de recurenţă din propoziţia anterioară este similară relaţiei de recurenţă pentru determinarea numerelor lui Fibonacci:

$F_0=1$, $F_1=1$ şi $F_n= F_{n-1}+ F_{n-2}$, pentru $n\geq 2$.

Din acest motiv, **arborii AVL se mai numesc şi arbori Fibonacci**.

Definiţie: Un ***arbore Fibonacci*** se defineşte astfel:

a) **arborele vid este arborele Fibonacci de înălţime 0;**
b) **arborele cu un singur nod este arborele Fibonacci de înălţime 1;**
c) **dacă arborii A_{h-1}, A_{h-2} sunt arbori de înălţime *h*-1 şi *h*-2 atunci arborele $A=(A_{h-1}, x, A_{h-2})$ va fi un arbore Fibonacci de înălţime *h*;**
d) nici un alt arbore nu este un arbore Fibonacci.

Observaţie: Numărul de noduri din arborele Fibonacci, A_h este definit de următoarea relaţie de recurenţă:

- numărul de noduri pentru A_0, $N_0=0$;
- pentru A_1, $N_1=1$;
- $N_h=N_{h-1}+1+ N_{h-2}$;

În această relaţie prin N_i s-au notat numărul de noduri valabile pentru cazul cel mai nefavorabil de înălţime. În acest caz, pentru un arbore cu *n* noduri: $h_n=1.44*\log_2(n+2)-0.328$.

Observaţie: Timpul maxim de căutare va depinde de înălţimea arborelui respectiv.

Pentru arborii echilibraţi s-a demonstrat următoarea teoremă:

Teoremă: (Adelson-Velski-Landis)

Înălţimea h_n a unui arbore echilibrat cu *n* noduri respectă relaţia:

$$\mathbf{\log_2(n+1) \leq h_n \leq 1.44*\log_2(n+2)-0.328}$$

Demonstraţie: Cazul cel mai favorabil este cel al unui arbore perfect echilibrat (arborele în care toate nodurile neterminale au câte 2 descendenţi direcţi). Într-un astfel de caz, numărul de noduri dintr-

un arbore de înălţime h va fi dat de $n=1+2+2^2+...+2^{h-1}=2^h-1$, de unde $h_n=\log_2(n+1)$; pentru a determina valoarea maximă pe care poate să o ia h_n va trebui să construim arborele de înălţime h_n care are cel mai mic număr de noduri posibile. În general, pentru construirea acestui arbore A_h se aplică strategia legării subarborilor A_{h-1} şi A_{h-2} cu condiţia ${}^hA_{h-1}={}^hA_{h-2}+1$.

Aceştia sunt tot arbori Fibonacci. Numărul de noduri dintr-un arbore Fibonacci este dat de relaţia: $1.44*\log_2(n+2) \geq h_n$, deci înălţimea maximă a unui arbore echilibrat cu n noduri este, de fapt înălţimea unui arbore Fibonacci cu n noduri(sau cu un număr de noduri n' care este primul întreg superior lui n).

Rezultă deci că: $h_n \leq 1.44*\log_2(n+2)-0.328$.

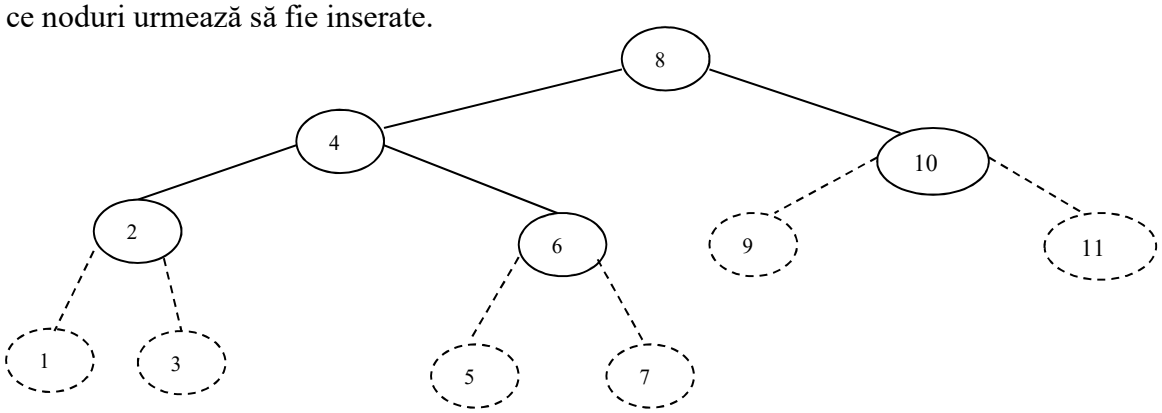
Tehnica inserţiei nodurilor în arbori echilibraţi

Dându-se un arbore cu rădăcina R, notăm cu S subarborele stâng şi cu D subarborele drept. În cazul inserţiei unui nod se vor distinge 3 situaţii. Se presupune că nodul se inserează în S determinând creşterea cu o unitate a înălţimii acestui subarbore.

Avem:

a) Dacă $h_S = h_D$ în urma inserţiei subarborii S şi D nu vor mai avea înălţimi egale, dar nu se violează echilibrul.
b) Dacă $h_S < h_D$ după inserţie cei 2 subarbori devin de înălţimi egale.
c) Dacă $h_S > h_D$ după inserţie echilibrul e violat, deci arborele trebuie reechilibrat.

Exemplu: Se consideră un arbore echilibrat, dat prin linie continuă; prin linie punctată se arată ce noduri urmează să fie inserate.



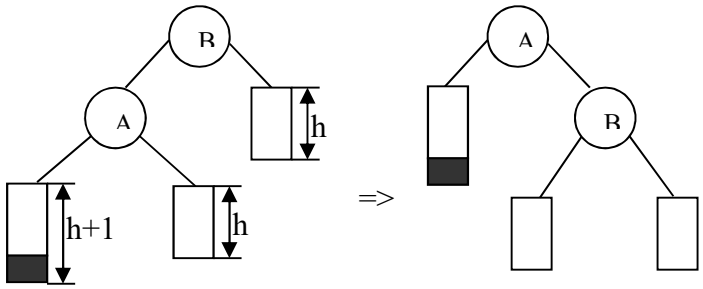
Dacă se inserează nodul 9 sau 11 operaţia va fi fără reechilibrare. Inserţiile unuia din nodurile 1, 3, 5, 7 necesită reechilibrarea arborelui. O analiză a situaţiilor posibile ce rezultă în urma inserţiei unui nod evidenţiază faptul că există numai două configuraţii ce necesită tratamente speciale.

Celelalte situaţii pot fi reduse la cele două din considerente de simetrie.

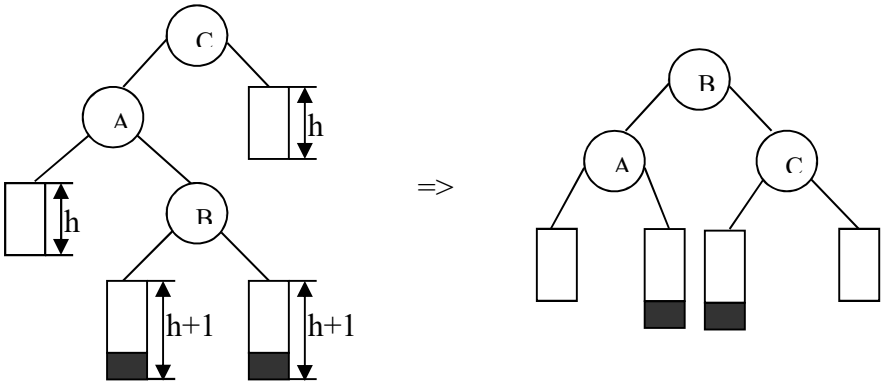
Prima situaţie se referă la inserţia nodului 1 sau 3 în arborele reprezentat cu linie continuă. A doua situaţie se referă la inserţia nodului 5 sau 7 în arborele reprezentat cu linie punctată.

Cele două situaţii pot fi prezentate, în general, în figurile următoare în care dreptunghiurile reprezintă subarbori, iar elementele adăugate prin inserţie apar haşurate. În aceste figuri se indică şi transformările pentru obţinerea arborelui echilibrat. Se subliniază că singurele mişcări permise sunt cele pe verticală.

Poziţiile relative pe orizontală ale nodurilor precizate şi ale subarborilor trebuie să rămână nemodificate.



Caz 1



Caz 2

Un algoritm pentru inserarea unui nod într-un arbore echilibrat şi pentru reechilibrare, depinde de maniera în care este memorată informaţia referitoare la situaţia echilibrului arborelui. O soluţie extremă este aceea în care această informaţie este conţinută în mod implicit în însăşi structura arborelui. În acest caz, factorul de echilibru al unui nod afectat de inserţie trebuie determinat de fiecare dată, lucru ce duce la o manoperă excesivă.

Altă soluţie este aceea prin care se atribuie fiecărui nod un factor explicit de echilibru, adică se poate da următoarea descriere de tip:

```
struct nod
{
 int cheie;
 int contor;
```

```
nod *st,*dr;
int ech; //-1,0,1
};
```

În continuare, factorul de echilibru al unui nod va fi dat de diferenţa între înălţimea subarborelui drept şi cea a subarborelui stâng. Pornind de la structura de nod dată, operaţia de inserare a unui nod se desfăşoară în trei etape:

1) se parcurge arborele binar pentru a verifica factorul de echilibru dacă nu cumva cheia există deja;
2) se inserează noul nod şi se determină factorul de echilibru;
3) se revine pe drumul de căutare şi se verifică factorul de echilibru pentru fiecare nod.

Observaţie: Deşi algoritmul prezentat necesită unele verificări redundante (odată echilibrul stabilit nu mai este necesară verificarea factorului de echilibru pentru strămoşii nodului); pentru moment se va face apel la ea fiind uşor de implementat.

Procedurile de căutare şi inserţie date pot fi modificate astfel încât să se asigure ”revenirea de-a lungul drumului de căutare”. Informaţia ce trebuie transmisă în fiecare pas este cea referitoare la modificarea înălţimii subarborelui în care s-a făcut inserţia. Astfel, în lista de parametrii ai procedurii se introduce un parametru variabil de tip boolean notat *H*, a cărui valoare “true” semnifică creşterea înălţimii subarborelui.

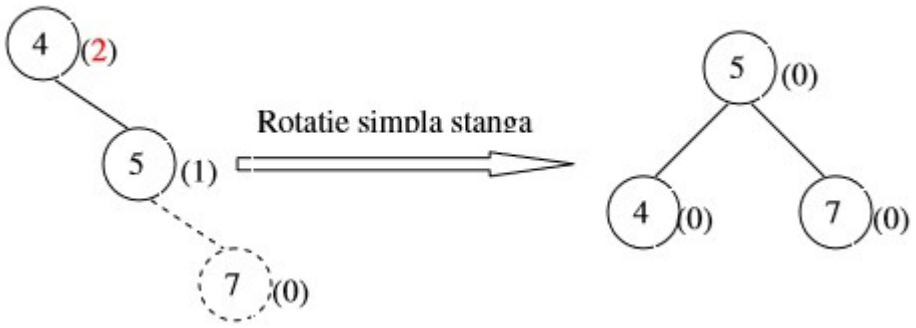
Presupunem că procedura de inserţie se reîntoarce din subarborele stâng la un nod p cu indicaţia că înălţimea sa a crescut. Se pot distinge trei situaţii referitoare la înălţimea subarborelui înaintea inserţiei:

1) $h_S > h_D$ şi p->ech=1; echilibrul anterior referitor la nodul p^ a fost rezolvat;
2) $h_S = h_D$ şi p->ech=0; greutatea arborelui va fi înclinată spre stânga;
3) $h_S < h_D$ şi p->ech=-1; este necesară reechilibrarea; inspectarea factorului de echilibru al rădăcinii subarborelui stâng şi dat de şi p->ech conduce la stabilirea cazului 1 şi 2 din ultima figură.

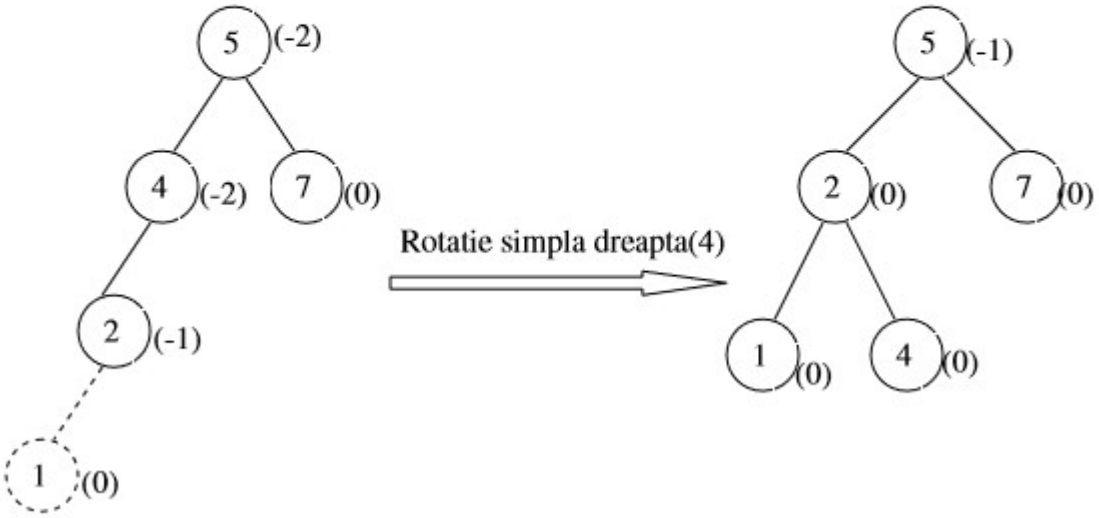
Dacă acest nod are la rândul său înălţimea subarborelui său stâng mai mare decât înălţimea subarborelui drept vom fi în cazul 1, altfel suntem în cazul 2. În această situaţie nu poate să apară un subarbore stâng a cărui rădăcină are un factor de echilibru nul. Operaţia de reechilibrare constă dintr-o secvenţă de reasignări de pointeri. De fapt, pointerii sunt schimbaţi ciclic rezultând fie o simplă fie o dublă rotaţie a două sau trei noduri implicate. În plus, pe lângă rotirea pointerilor, factorii de echilibru respectivi sunt reajustaţi.

Tehnica **inserţiei nodurilor** într-un arbore echilibrat poate fi prezentată pentru cazul creării unui arbore.

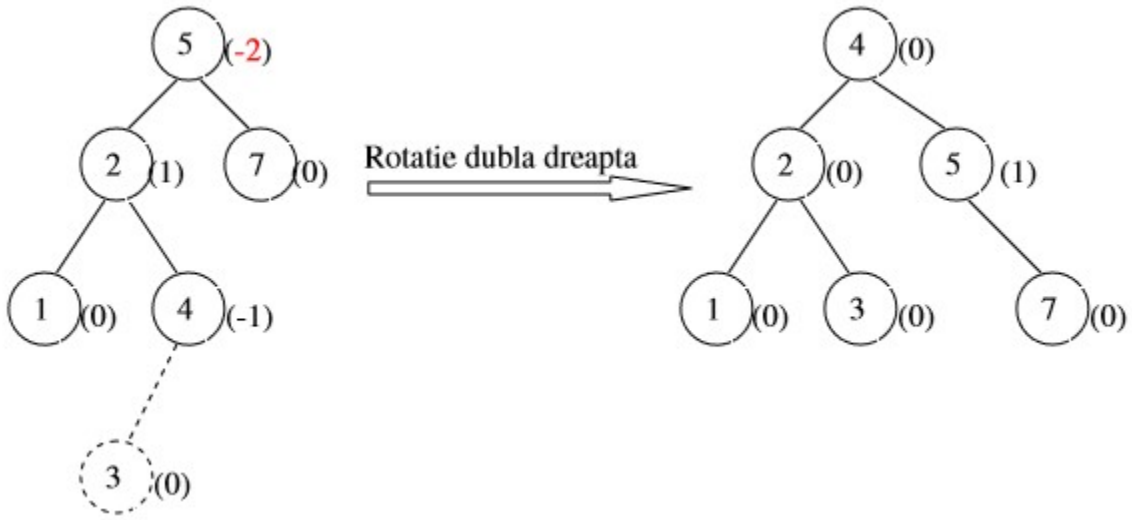
Exemplu: Se va considera secventa de chei 4,5,7,2,1,3,6 care se insereaza intr-un arbore AVL initial vid. Evolutia arborelui si echilibrarile sunt:
-nodurile cu cheile 4,5 se vor insera ca si la arborii binari de cautare
-in mod normal nodul cu cheia 7 ar fi inserat in dreapta nodului cu eticheta 5. In acest caz arborele este dezechilibrat.



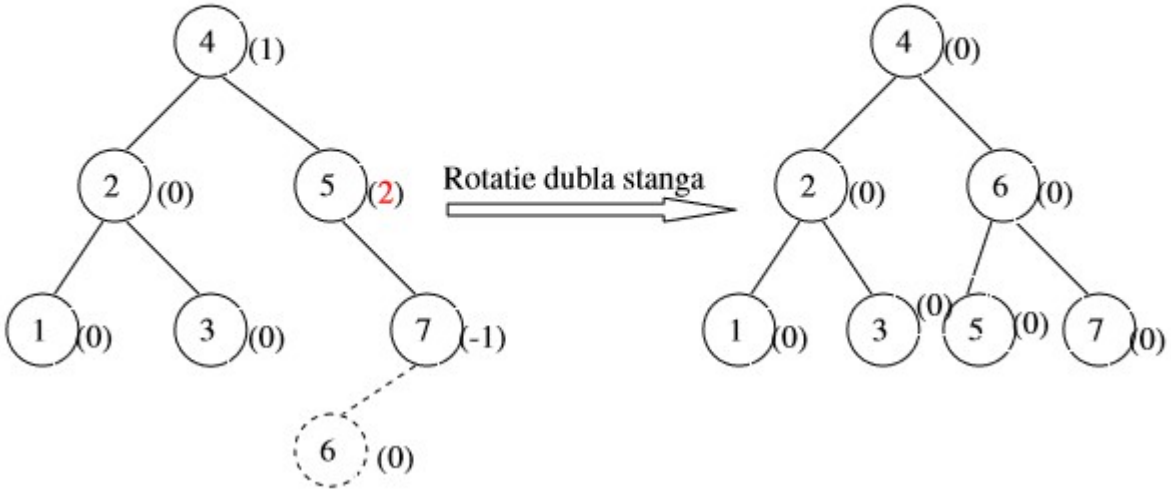
-nodul cu cheia 2 se va insera in stanga nodului cu cheia 4 fara ca arborele sa se dezechilibreze
-nodul cu cheia 1 se va insera in stanga nodului cu cheia 2, dar in acest caz arborele este dezechilibrat:



-dupa inserarea nodului 3 arborele se va dezechilibra din nou:

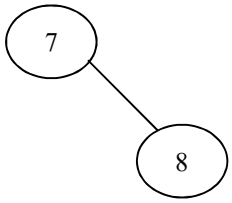


-nodul cu cheia 6 s-ar insera in stanga nodului cu cheia 7, dar in acest caz arborele ar fi dezechilibrat:



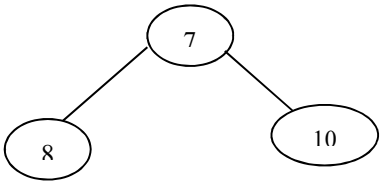
Exemplu: Tehnica **inserţiei nodurilor** într-un arbore echilibrat poate fi prezentată pentru cazul creării unui arbore. **Se consideră un arbore cu două noduri prezentat în figura a).**

a)



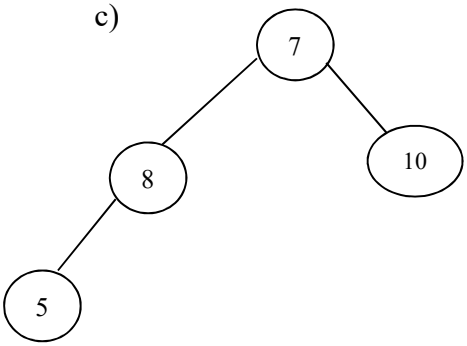
Se doreşte acum inserţia nodului cu cheia 10. Dacă am insera în acest arbore binar s-ar produce o dezechilibrare. Pentru echilibrarea arborelui se apelează la algoritmul rotire simplă dreapta, obţinându-se arborele din figura b).

b)

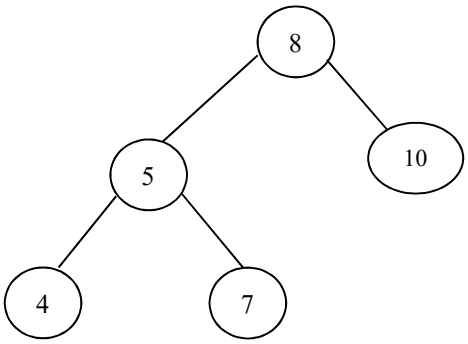


Se doreşte acum inserarea nodurilor 5 şi 4. **Inserarea cheii 5 nu strică echilibrul. Inserarea cheii 4 conduce la dezechilibrarea subarborelui cu rădăcina 7. Echilibrarea se poate face printr-o rotire simplă stânga, obţinându-se arborii din figurile c) şi d).**

c)

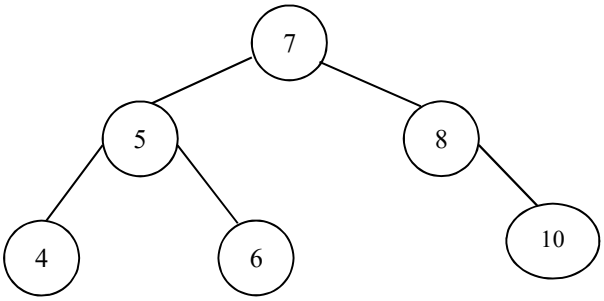


d)



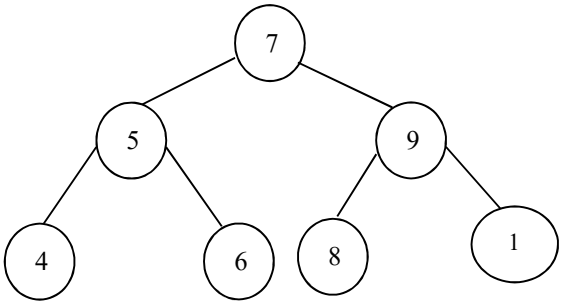
Se doreşte inserarea unui nod cu cheia 6; se va produce din nou dezechilibrul, iar pentru echilibrare se apelează la rotaţia dublă stângă, rezultând arborele din figura e).

e)



Inserarea nodului cu cheia 9 conduce la necesitatea aplicării cazului rotaţie dublă dreapta, obţinându-se arborele din figura f).

f)



Ţinând cont de cele discutate, se poate da o procedură care realizează inserţia cu echilibrare a unui nod în arborele echilibrat.

```
void Inserare(int &x, nod *&p, int &h)
{
 nod *p1,*p2;
 if (p==NULL)//cuvantul nu este in arbore => insertie
  {
   p=new nod;
   p->cheie=x; p->contor=1; p->st=p->dr=NULL; p->ech=0;
  }
  else
   if (x<p->cheie)
    {Inserare(x,p->st,h);
     if (h==1) //ramura din stanga a crescut in inaltime
      {
       switch (p->ech)
       {
       case +1: p->ech=0; h=0; break;
       case 0: p->ech=-1; break;
       case -1: //reechilibrare
         p1=p->st;
         if (p1->ech==-1)
          {p->st=p1->dr;
           p1->dr=p;
           p->ech=0;
           p=p1;
          }
         else //caz 2 stanga
          { p2=p1->dr; p1->dr=p2->st; p2->st=p1;
            p->st=p2->dr; p2->dr=p;
            if (p2->ech==-1) p->ech=1;
            else p->ech=0;
            if (p2->ech==1) p->ech=-1;
            else p->ech=0;
            p=p2;
          }
          p->ech=0;
          h=0;
      }
      }
    }
    else
    if (x>p->cheie)
    {Inserare(x,p->dr,h);
     if (h==1) //ramura din stanga a crescut in inaltime
      {
       switch (p->ech)
       {
       case -1: p->ech=0; h=0; break;
       case 0: p->ech=-1;break;
```

```
        case 1: //reechilibrare
          p1=p->dr;
          if (p1->ech==-1)//caz 1 dreapta
           {p->dr=p1->st;
            p1->st=p;
            p->ech=0;
            p=p1;
           }
          else //caz 2 stanga
           { p2=p1->st; p1->st=p2->dr; p2->dr=p1;
             p->dr=p2->st; p2->st=p;
             if (p2->ech==1) p->ech=-1;
             else p->ech=0;
             if (p2->ech==-1) p->ech=1;
             else p->ech=0;
             p=p2;
           }
           p->ech=0;
           h=0;
      }
      }
    }
   else p->contor++;
}
```

În legătură cu performanţele operaţiei de inserţie a unui nod într-un arbore echilibrat, se observă două probleme:

a) dacă toate cele $n!$ permutări pentru cele n chei cu aceeaşi probabilitate se pune problema care este înălţimea probabilă a arborelui echilibrat care se construieşte;
b) care este probabilitatea ca inserţia unui nod să necesite reechilibrarea.

Teste empitice ale înlălţimii arborilor generaţi conform algoritmului dat conduc la o valoare pentru $h=\log_2(n)+0.25$. Aceasta înseamnă că în practica utilizării arborilor echilibraţi, ei se comportă la fel de bine ca şi arborii perfecţi echilibraţi, însă mai uşor de reechilibrat.

Testele empirice sugerează de asemenea că în medie reechilibrarea este necesară aproximativ la 2 inserţii.

Observaţie: Complexitatea operaţiei de echilibrare sugerează faptul că arborii echilibraţi trebuie utilizaţi mai ales în operaţiile de căutare a informaţiei.

Suprimarea nodurilor în arborii echilibraţi

În cazul arborilor echilibraţi operaţia de suprimare este mai complexă decât operaţia de inserţie a unui nod.

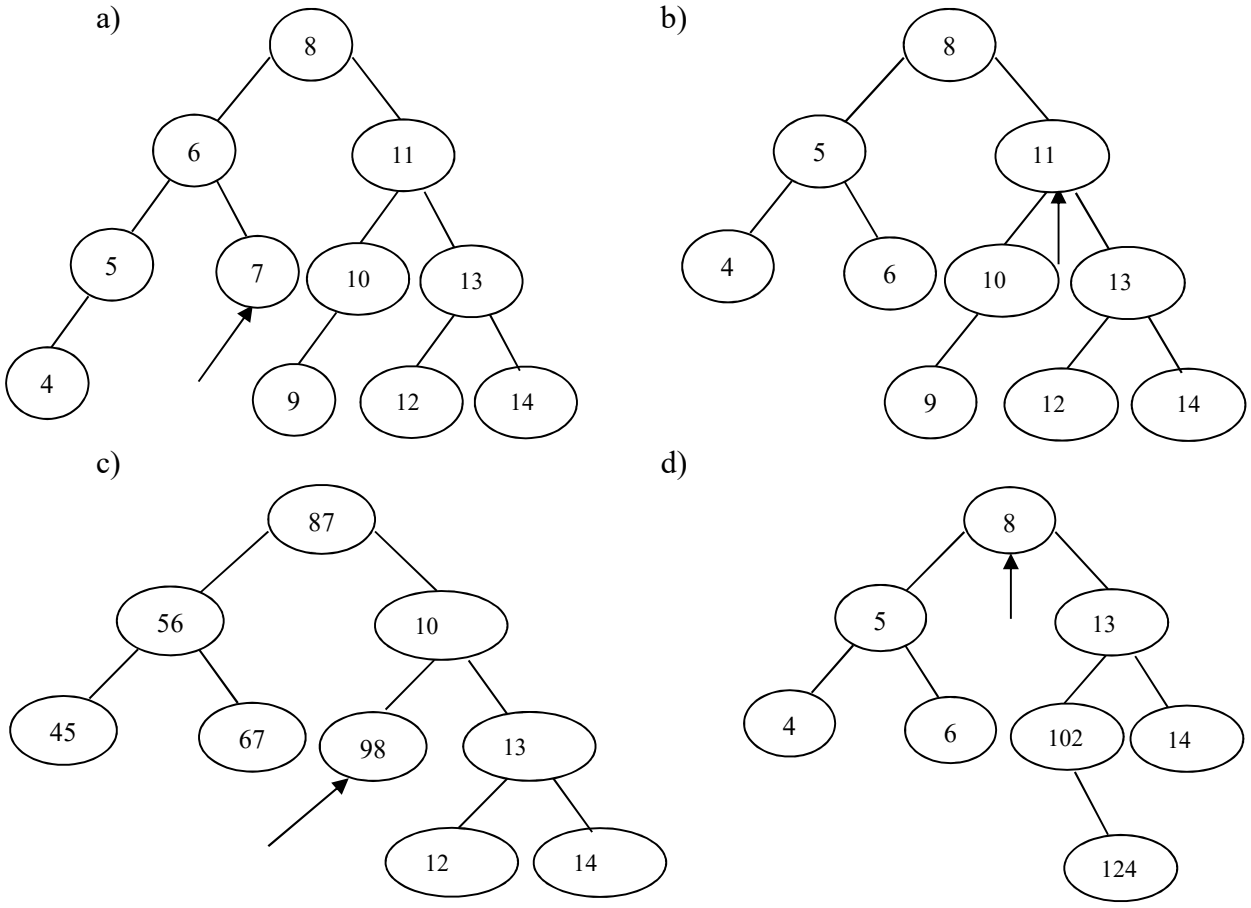
În principiu, operaţia de reechilibrare rămâne aceeaşi, reducându-se una sau două rotaţii stângă sau dreaptă. Tehnica de suprimare a unui nod dintr-un arbore dintr-un arbore echilibrat este asemănătoare cu tehnica de suprimare a unui nod prezentată la arborii de căutare. Apar probleme

noi în cazul în care nodul ce se suprimă are 2 subarbori ca descendenţi. În acest caz, nodul poate fi înlocuit cu cel mai din dreapta nod al subarborelui său stâng.

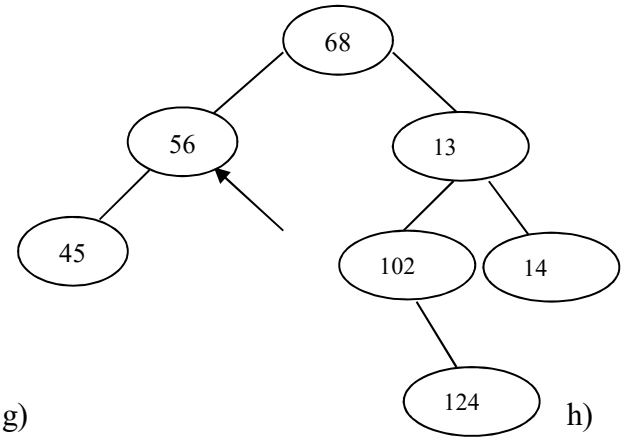
Exemplu: Se consideră un arbore prezentat în figura a) şi se doreşte să se suprime pe rând nodurile cu cheile 7, 11, 9, 8, 5, 4, 10. Rezultă arborii din figurile b), c), d), e), f), g), h).

Suprimarea nodului cu cheia 7 conduce la subarborele dezechilibrat cu rădăcina 6; reechilibrarea acestuia presupune o rotaţie simplă stângă. Reechilibrarea devine din nou necesară după suprimarea nodului cu cheia 9. De data aceasta subarborele cu rădăcina 10 este reechilibrat printr-o rotaţie simplă dreapta. Deşi nodul cu cheia 5 are un singur descendent, suprimarea sa presupune o reechilibrare printr-o rotaţie dublă dreapta. Cazul suprimării nodului cu cheia 10 presupune înainte de reechilibrare, înlocuirea acestui nod cu cel mai din dreapta element al subarborelui său stâng.

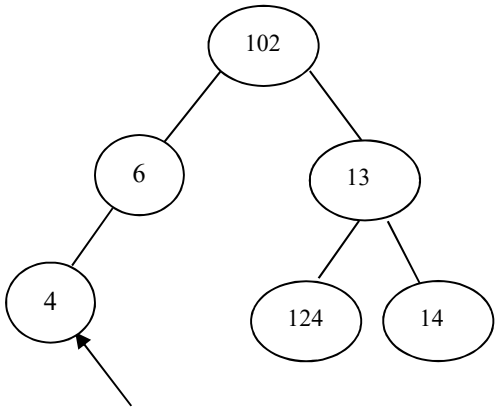
Observaţie: Diferenţa esenţială dintre operaţia de inserţie şi operaţia de suprimare, în cazul arborilor echilibraţi este următoarea: dacă în urma unei inserţii, reechilibrarea se realizează prin una sau două rotaţii antrenând 2 sau 3 noduri, operaţia de suprimare poate necesita o rotaţie a fiecărui nod situat pe drumul de căutare a nodului cu cheia de suprimare.



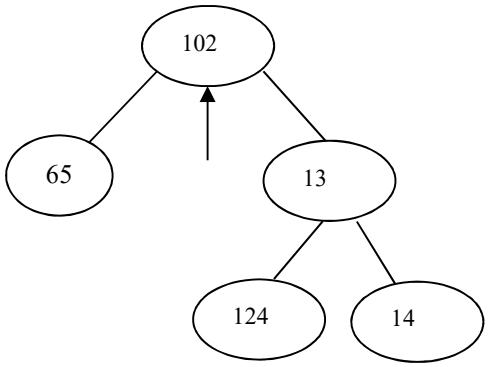
e)



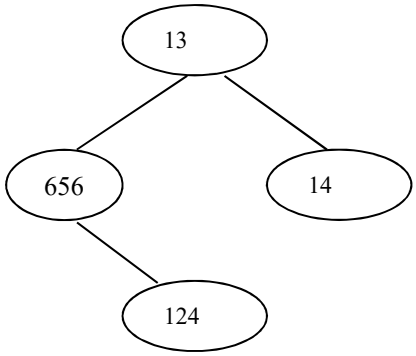
f)



g)



h)



În cazul arborilor Fibonacci suprimarea nodului său cel mai din dreapta necesită număr maxim de rotaţii; acesta este cazul cel mai nefavorabil de suprimare dintr-un arbore echilibrat.

Testele experimentale au arătat faptul că, dacă în cazul inserţiei reechilibrarea era necesară aproape la fiecare a 2-a inserţie, în cazul suprimării acest lucru se întâmplă la aproximativ a 5-a suprimare.

Pentru realizarea efectivă a eliminării unui nod cu cheia *k* se face apel la algoritmul descris de procedura elimină nod ai cărei parametrii sunt:

- *n* = referinţă la nodul ce trebuie eliminat;
- *fe* = indicator de modificare a factorului de echilibru, care va fi propagat spre rădăcina arborelui.

Atâta timp cât *fe*<>0 va fi apelată procedura de reechilibrare a arborelui. Operaţia de reechilibrare se încheie în momentul în care se ajunge la *fe*=0 sau se ajunge la rădăcina arborelui.

Echilibrarea subarborelui dominat de nodul *n*, necesară după eliminarea unui nod este realizată de procedura echilibrează subarbore cu rădăcina *n*.

Funcţiile C++ ce realizează aceste operaţii sunt:

```
void Echilibru1(nod*&p, int&h)
{
 nod *p1,*p2;
 int e1,e2;
 //h=true, ramura stanga a devenit mai mica
```

```
 switch (p->ech)
  {
   case -1: p->ech=0; break;
   case 0: p->ech=-1; h=0; break;
   case 1: //reechilibrare
    p1=p->dr; e1=p1->ech;
    if (e1>=0)
     {//cazul 1 dreapta
      p->dr=p1->st; p1->st=p;
      if (e1>0) { p->ech=1; p1->ech=-1; h=0; }
      else p->ech=p1->ech=0;
      p=p1;
     }
    else
     {//cazul 2 dreapta
      p2=p1->st; e2=p2->ech;
      p1->st=p2->dr; p2->dr=p1;
      p2->st=p;
      if (e2==1) p->ech=-1;
      else p->ech=0;
      if (e2==-1) p->ech=1;
      else p->ech=0;
      p=p2; p2->ech=0;
     }
  }
}

void Echilibru2(nod*&p, int&h)
{
 nod *p1,*p2;
 int e1,e2;
 //h=true, ramura dreapta cea mai mica
 switch (p->ech)
  {case 1: p->ech=0; break;
   case 0: p->ech=-1; h=0; break;
   case -1: //reechilibrare
    p1=p->st; e1=p1->ech;
    if (e1<=0)
     {//cazul 1 stanga
      p->st=p1->dr; p1->dr=p;
      if (e1==0) { p->ech=-1; p1->ech=1; h=0; }
      else p->ech=p1->ech=0;
      p=p1;
     }
    else
     {//cazul 2 dreapta
      p2=p1->dr; e2=p2->ech;
      p1->dr=p2->st; p2->st=p1;
      p2->st=p1;
      p->st=p2->dr; p2->dr=p;
      if (e2==-1) p->ech=1;
```

```
      else p->ech=0;
      if (e2==1) p->ech=-1;
      else p->ech=0;
      p=p2; p2->ech=0;
     }
  }
}

void Suprima(nod*&r, int &h)
{//h=false
 if (r->dr!=NULL)
  {Suprima(r->dr,h);
   if (h==1) Echilibru2(r,h);
  }
 else
  {q->cheie=r->cheie;
   q->contor=r->contor;
   r=r->st;
   h=0;
  }
}

void SuprimEchilibrat(int x, nod *&p, int &h)
{
 if (p==NULL) {cout<<"cheia nu este in arbore"; h=0;}
 else
  if (x<p->cheie)
   {SuprimEchilibrat(x,p->st,h);
    if (h==1) Echilibru1(p,h);
   }
  else
    if (x>p->cheie)
     {SuprimEchilibrat(x,p->dr,h);
      if (h==1) Echilibru2(p,h);
     }
    else
     {//suprima p
      q=p;
      if (q->dr==NULL) {p=q->st; h=1;}
      else
       if (q->st==NULL) {p=q->dr; h=1;}
       else
      {Suprima(q->st,h);
       if (h==1) Echilibru1(p,h);
      }
     }
}
```