8.3. Arbori binari ordonaţi

8.3.1. Definiţii

- Structura **arbore binar** poate fi utilizată pentru a reprezenta în mod convenabil o mulţime de elemente, în care elementele se regăsesc după o **cheie unică**.
    - Se **presupune** că avem o mulţime de n noduri definite ca articole, fiecare având câte o cheie care este număr întreg.
    - Dacă cele n articole se **organizează** într-o structură **listă liniară**, căutarea unei chei necesită în medie $n/2$ comparaţii.
    - După cum se va vedea în continuare, **organizarea** celor n articole într-o **structură arbore binar convenabilă**, reduce numărul de căutări la maximum $\log_2 n$.
    - Acest lucru devine posibil utilizând structura **arbore binar ordonat.**
- Prin **arbore binar ordonat** se înţelege un **arbore binar** care are proprietatea că, parcurgând nodurile sale **în inordine**, secvenţa cheilor este **monoton crescătoare**.
- Un **arbore binar ordonat** se bucură de următoarea **proprietate**:
    - Dacă n este un **nod oarecare** al arborelui, având cheia c, **atunci:**
        - **Toate** nodurile din **subarborele stâng** a lui n au cheile mai **mici** sau egale cu c.
        - **Toate** nodurile din **subarborele drept** al lui n au chei mai **mari** sau egale cu c.
- De aici rezultă un procedeu foarte simplu de **căutare**:
    - Începând cu rădăcina, se trece la fiul **stâng** sau la fiul **drept**, după cum cheia căutată este mai **mică** sau mai **mare** decât cea a nodului curent.
- Numărul **comparaţiilor de chei** efectuate în cadrul acestui procedeu este cel mult egal cu **înălţimea arborelui**.
- Din acest motiv aceşti arbori sunt cunoscuţi şi sub denumirea de **arbori binari de căutare** (“**Binary Search Trees**”).
- În general înălţimea unui arbore **nu** este determinată de **numărul** nodurilor sale.
    - Spre exemplu cu cele 9 noduri precizate în fig.8.3.1.a se poate construi atât arborele ordonat (a) de înălţime 4 cât şi arborele ordonat (b) de înălţime 6.

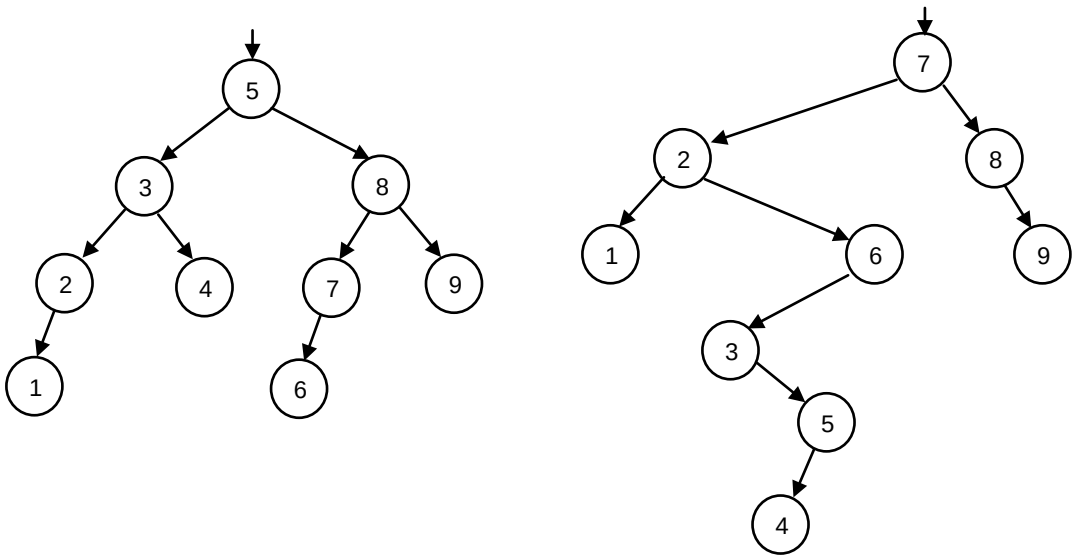


Fig.8.3.1.a. Arbori binari ordonaţi de diferite înălţimi

- Este simplu de observat că un arbore are înălţimea **minimă** dacă **fiecare** nivel al său conţine **numărul maxim de noduri**, cu excepţia posibilă a ultimului nivel.

- Deoarece **numărul maxim de noduri** al nivelului `i` este 2^{i-1}, rezultă că **înălţimea minimă** a unui arbore binar cu `n` noduri este:

$$h_{\min} = \lceil \log_2 (n+1) \rceil$$

- Prin aceasta se justifică şi afirmaţia că o **căutare** într-un **arbore binar ordonat** necesită aproximativ **$\log_2 n$ comparaţii de chei.**
-
    - Se precizează însă, că această afirmaţie este valabilă în **ipoteza** că nodurile s-au organizat într-o **structură arbore binar ordonat de înălţime minimă.**

- Dacă această condiţie **nu** este satisfăcută, **eficienţa** procesului de căutare poate fi mult redusă, în cazul cel mai defavorabil arborele degenerând într-o structură **listă liniară.**

- Aceasta se întâmplă când subarborele drept (sau stâng) al tuturor nodurilor este **vid**, caz în care înălţimea arborelui devine egală cu `n`, iar căutarea **nu** este mai eficientă decât căutarea într-o **listă liniară** (O *(n)*).

8.3.2. Tipul de date abstract arbore binar ordonat

- Într-o manieră similară celei în care au fost definite **tipurile de date abstracte** pe parcursul acestui curs, şi în cazul **arborilor binari ordonaţi** se poate defini un astfel de **TDA**.

- Acesta presupune desigur:

    - (1) Definirea **modelului matematic** asociat.

    - (2) Precizarea **setului de operatori**.

- Ca şi pentru celelalte structuri studiate şi în acest caz este greu de definit un set de operatori **general** valabil.

- Din mulţimea seturilor posibile se propune setul prezentat în [8.3.2.a].

TDA Arbore Binar Ordonat (ABO)

Modelul matematic: este un arbore binar, fiecare nod având asociată o cheie specifică. Pentru fiecare nod al arborelui este valabilă următoarea proprietate: cheia nodului respectiv este mai mare decât cheia oricărui nod al subarborelui său stâng şi mai mică decât cheia oricărui nod al subarborelui său drept.

Notatii:

TipCheie - tipul cheii asociate structurii nodului
TipElement - tipul asociat structurii unui nod
RefTipNod - referinţa la un nod al structurii
TipABO - tipul arbore binar ordonat

```
TipABO b;
TipCheie x,k;
TipElement e;
p: RefTipNod;
```

[8.3.2.a]

Operatori:

1. ***Creaza****(TipABO b)* - procedură care crează arborele binar vid *b*;

2. *RefTipNod* ***Cauta****(TipCheie x, TipABO b)* - operator funcţie care caută în arborele *b* un nod având cheia identică cu *x* returnând referinţa la nodul în cauză respectiv indicatorul vid dacă un astfel de nod nu există;

3. ***Actualizeaza****(TipElement e, TipABO b)* - caută nodul din arborele *b* care are aceeaşi cheie cu nodul *e* şi îi modifică conţinutul memorând pe *e* în acest nod. Dacă un astfel de nod nu există, operatorul nu realizează nici o acţiune;

4. ***Insereaza****(TipElement e, TipABO b)* - inserează elementul *e* în arborele *b* astfel încât acesta să rămână un ABO;

5. ***SuprimaMin****(TipABO b, TipElement e)* - extrage nodul cu cheia minimă din arborele cu rădacina *b* şi îl returnează în *e*. În urma suprimării arborele *b* rămâne un ABO;

6. ***Suprima****(TipCheie x, TipABO b)* - suprimă nodul cu cheia *x* din arborele *b*, astfel încât arborele să rămână ordonat. Dacă nu există un astfel de nod, procedura nu realizeză nimic.

8.3.3. Tehnici de căutare în arbori binari ordonaţi

- Fie b o referinţă care indică rădăcina unui **arbore binar ordonat**, ale cărui noduri au structura definită în [8.3.3.a] şi

- Fie x un număr întreg dat.

- În aceste condiţii funcţia **Cauta(x,b)** precizată în secvenţa [8.3.3.b] execută căutarea acelui nod aparţinând arborelui b care are cheia egală cu x.

    - Căutarea se realizează în conformitate cu procedeul descris în paragraful anterior.

    - Funcţia **Cauta** returnează valoarea NIL dacă **nu** găseşte nici un nod cu cheia x, altminteri valoarea ei este egală cu pointerul care indică acest nod.

```
-----------------------------------------------------------------
/*Structura de date Arbore Binar Ordonat*/

typedef struct tip_nod
          {
               //diferite campuri
               char cheie;
               struct tip_nod* stang;     /*[8.3.3.a]*/
               struct tip_nod* drept;
          };

typedef tip_nod * ref_tip_nod;
-----------------------------------------------------------------
/*Căutare în ABO (Varianta iterativă) - varianta pseudocod*/

ref_tip_nod Cauta(tip_cheie x, ref_tip_nod b)

/*caută iterativ în arborele binar b nodul cu cheia x şi
returnează referinţa la acest nod, sau NULL daca nu îl
găseşte*/

    boolean gasit=false;
    cat_timp(b<>NULL) AND (NOT gasit)          [8.3.3.b]
     | daca(b->cheie=x) gasit=true;
     |   altfel
     |     daca(x<b->cheie)
     |         b=b->stang;
     |       altfel
     |         b=b->drept;
     | □
    returneaza b;
   /*cauta*/
-----------------------------------------------------------------
/*Căutare în ABO (Varianta iterativă) - implementare C*/

ref_tip_nod Cauta(tip_cheie x, ref_tip_nod b)

{
    boolean gasit;
```

```
    gasit=false;
    while ((b!=NULL) && (!gasit))              /*[8.3.3.b]*/
      {
        if (b->cheie==x)
            gasit=true;
          else
            if (x<b->cheie)
                b=b->stang;
              else
                b=b->drept;
      }
    return b;
 } {Cauta}
```

- Acelaşi proces de căutare poate fi implementat şi în **variantă recursivă** ţinând cont de faptul ca **arborele binar** este definit ca şi o **structură de date recursivă**.

- **Varianta recursivă** a căutării apare în secvenţa [8.3.3.c].

    - Se face însă precizarea că această implementare este **mai puţin performantă** deoarece principial căutarea în arborii binari ordonaţi este o operaţie **pur secvenţială** care **nu** necesită memorarea drumului parcurs.

/*Căutare în ABO (Varianta recursivă pseudocod)*/

```
ref_tip_nod CautaRecursiv(tip_cheie x, ref_tip_nod b)

/*caută recursiv în arborele binar b nodul cu cheia x şi
returnează referinţa la acest nod, sau NULL daca nu îl
găseşte*/

    daca (b==NULL) returneaza b;
      altfel
        daca (x<b->cheie) b=CautaRecursiv(x,b->stang);
          altfel                                  /*8.3.3.c*/
            daca (x>b->cheie) b=CautaRecursiv(x,b->drept);
              altfel
                returneaza b;
  /*CautaRecursiv*/
```

/*Căutare în ABO (Varianta recursivă)- implementare C*/

```
ref_tip_nod CautaRecursiv(tip_cheie x, ref_tip_nod b)

/*caută recursiv în arborele binar b nodul cu cheia x şi
returnează referinţa la acest nod, sau NULL daca nu îl
găseşte*/

{
    if (b==NULL) THEN
        return b;
      else
       if (x<b->cheie)
           b=CautaRecursiv(x,b->stang);       /*[8.3.3.c]*/
```

```
          else
            if (x>b->cheie)
                b=CautaRecursiv(x,b->drept);
              else
                return b;
 } {CautaRecursiv}
```

- **Tehnica de căutare** poate fi simplificată dacă se aplică **metoda fanionului**.
  - În cazul arborilor această metodă presupune completarea structurii cu un **nod fictiv**, nodul fanion, care este indicat de un pointer notat cu `f`.
  - Se modifică în continuare structura arborelui, **modificând** toate referinţele egale cu NIL astfel încât să-l indice pe `f`.
  - Spre exemplu, arborele binar ordonat din figura 8.3.1.a. stânga va avea structura din figura 8.3.3.a. Procedura de căutare propriu-zisă apare în secvenţa 8.3.3.d.

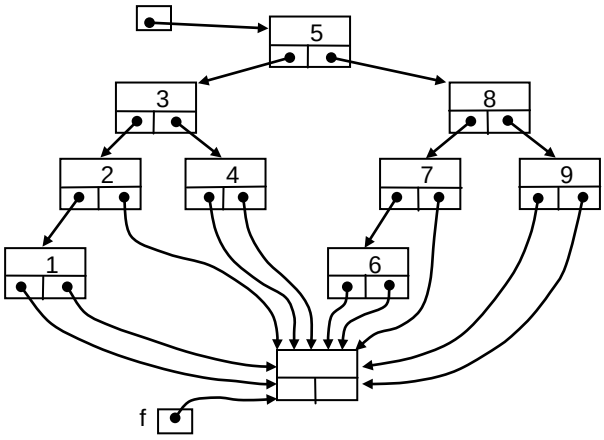


Fig.8.3.3.a. Arbore binar ordonat modificat

```
/*Căutare in ABO  utilizând tehnica fanionului*/

 ref_tip_nod Cauta1(tip_cheie x, ref_tip_nod b)
  {
    f->cheie=x;   /*ref_tip_nod f este parametru global*/
    while (b->cheie!=x)
      if (x<b->cheie)
          b=b->stang;                          /*8.3.3.d*/
        else
          b=b->drept;
    return b;
  } /*Cauta1*/
```

- Înainte de demararea procesului de căutare propriu-zisă, se asignează cheia fanionului `f` cu `x`.
  - În procesul căutării, nodul cu cheia `x` se găseşte acum cu certitudine.

- Dacă acest nod este fanionul `f` atunci în arborele iniţial **nu** există un nod cu cheia `x`.
- În caz contrar, nodul găsit este cel căutat.

- Se observă însă că structura din fig.8.3.3.a **nu** mai este **din punct de vedere formal** un **arbore**, dar ea se poate utiliza în reprezentarea unei **structuri arbore** în procesul de căutare.

- Se remarcă **simplificarea** condiţiei în instrucţiunea **`while`**, element care conferă o performanţă superioară acestei metode.

8.3.4. Inserţia nodurilor în ABO. Crearea arborilor binari ordonaţi

- În cadrul acestui paragraf se tratează:
  - (1) **Inserţia nodurilor** într-un **arbore binar ordonat**.
  - (2) Problema **construcţiei unui arbore binar ordonat**, pornind de la o mulţime dată de noduri.

- Procesul de creare al unui ABO constă în **inserţia** câte unui nod într-un arbore binar ordonat care iniţial este vid.
  - Problema care se pune este aceea de a executa inserţia de o asemenea manieră încât arborele să rămână **ordonat** şi după adăugarea noului nod.
  - Acesta se realizează **traversând** arborele începând cu rădăcina şi selectând fiul **stâng** sau fiul **drept**, după cum cheia de inserat este mai **mică** sau mai **mare** decât cheia nodului parcurs.
  - Aceasta proces se **repetă** până când se ajunge la un pointer NULL.
  - În continuare inserţia se realizează modificând acest pointer astfel încât să indice noul nod.
- Se precizează că inserţia noului nod **trebuie** realizată chiar dacă arborele conţine deja un nod cu cheia egală cu cea nouă.
  - În acest caz, dacă se ajunge la un nod cu cheia egală cu cea de inserat, se procedează ca şi cum aceasta din urmă ar fi **mai mare**, deci se trece la fiul **drept** al nodului curent.
  - În felul acesta la parcurgerea în **inordine** a arborelui binar ordonat se obţine o metodă de **sortare stabilă** (Vol.1 &3.1).

- În fig.8.3.4.a se prezintă inserţia unei noi chei cu numărul 8 în structura existentă de arbore ordonat.
  - La parcurgerea în inordine a acestui arbore, se observă că cele două chei egale sunt parcurse în ordinea în care au fost inserate.

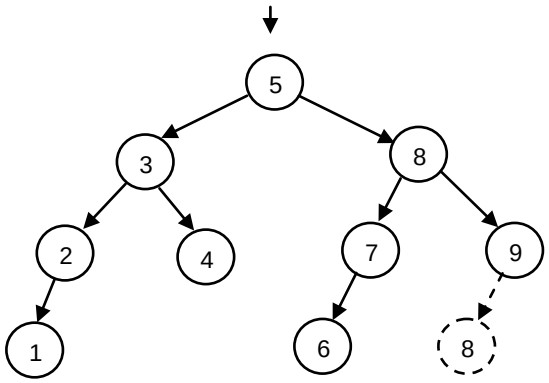


Fig.8.3.4.a. Inserţia unui nod nou cu o cheie existentă

- În continuare se prezintă o **procedură recursivă** pentru inserţia unui nod într-un **arbore binar ordonat**, astfel încât acesta să rămână ordonat.

- Se precizează că iniţial, arborele poate fi vid.

- Structura arbore la care se vor face referiri este cea precizată în secvenţa [8.3.3.a].

- Procedura **`Insereaza`** realizează inserţia unui nod cu cheia `x` într-un arbore binar ordonat [8.3.4.a].

    - Se precizează că `x` este un număr întreg reprezentând cheia nodului de inserat şi `b` un pointer care indică rădăcina arborelui

```
-------------------------------------------------------------
/*Inserţia unui nod într-un arbore binar ordonat (Varianta
pseudocod)*/

void insereaza(tip_cheie x, ref_tip_nod b)

/*inserează nodul x în arborele binar ordonat cu radăcina b*/
    daca(b!=NULL)
         daca(x<b->cheie)  /*parcurgere arbore binar*/
             insereaza(x,b->stang);
           altfel
             insereaza(x,b->drept);
       altfel   /*b este NULL s-a găsit locul de inserţie*/
           | /*insertie nod nou*/
           | b=aloca_memorie(tip_nod); /*alocare nod nou în b*/
           | b->cheie=x; b->stang=NULL; b->drept=NULL;
           | □
  /*insereaza1*/
-------------------------------------------------------------
```

- În secvenţa [8.3.4.b] se prezintă un fragment de **program principal** care utilizează procedura de mai sus în vederea creării unui arbore binar ordonat.

    - Se presupune că:

        - Toate cheile sunt diferite de zero.
        - Cheile se citesc de la dispozitivul de intrare.

- Secvenţa de chei se încheie cu o cheie fictivă egală cu zero pe post de terminator.

```
-------------------------------------------------------------
/*Construcţia unui arbore binar ordonat - varianta
pseudocod*/

ref_tip_nod radacina;
tip_cheie c;

   radacina=NULL;
   citeste(c);                                    /*8.3.4.b*/
   cat_timp (c!=0)
     | insereaza(c,radacina);
     | citeste(c);
     □
-------------------------------------------------------------
```

- Desigur, în crearea arborilor binari ordonaţi se poate utiliza **metoda fanionului** prezentată în paragraful anterior, caz în care trebuie realizate unele **modificări** în codul procedurii `Inserează`.

- În continuare se descrie o **variantă nerecursivă** a procedurii **`Inserează.`**

  - În cadrul acestei variante se disting două părţi şi anume:

    - (1) **Parcurgerea** arborelui pentru găsirea locului unde trebuie inserat noul nod.

    - (2) **Inserţia** propriu-zisă.

- Prima parte se implementează cu ajutorul a **doi pointeri** `q1` şi `q2`, urmând un algoritm similar celui utilizat la liste (tehnica celor doi pointeri - Vol.1 &6.4.2).

  - Cei doi pointeri indică mereu două noduri "**consecutive**" ale arborelui:

    - `q2^` este nodul curent (iniţial rădăcina).

    - `q1^` este fiul său stâng sau fiul drept, după cum `x`, cheia care se caută, este mai mică respectiv mai mare decât cheia nodului curent indicat de `q2`.

  - Pointerii avansează din nod în nod de-a lungul arborelui, până când pointerul `q1` devine NULL, moment în care se realizează inserţia propriu-zisă.

  - Se precizează că este nevoie şi de o variabilă întreagă `d`, pentru a preciza dacă nodul nou trebuie inserat ca fiu stâng sau ca fiu drept al lui `q2^`.

    - Această variabilă se asignează în timpul parcurgerii arborelui şi se testează în cadrul inserţiei propriu-zise [Wi76].

- Spre deosebire de **varianta recursivă** în care traseul parcurs este **memorat implicit** de către mecanismul de implementare al recursivităţii cu ajutorul unei **stive**, în acest caz **nu** este nevoie de stivă întrucât **nu** trebuie să se revină în arbore decât cu un singur nivel (pentru a realiza înlănţuirea), motiv pentru care sunt suficienţi **doi pointeri consecutivi** (fig.8.3.4.b (b)).

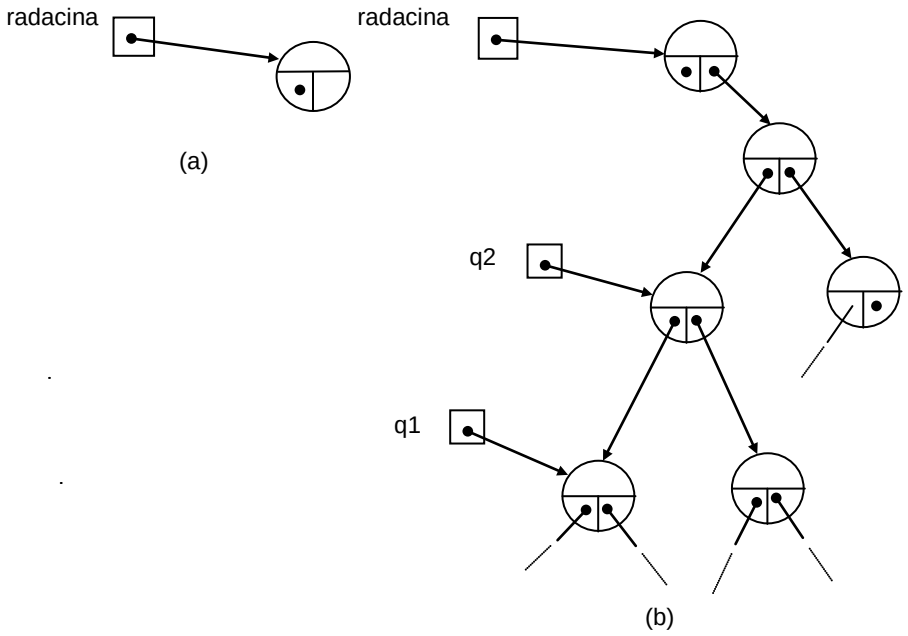


Fig.8.3.4.b. Arbori binari ordonaţi. Tehnica celor doi pointeri

- Procedura care realizează inserţia unui nod într-un **arbore binar ordonat** în manieră nerecursivă apare în secvenţa [8.3.4.c].

```
-----------------------------------------------------------------
/*Inserţia în ABO (Implementare nerecursivă) - varianta
pseudocod)*/

subprogram InsereazaNerecursiv(tip_cheie x, ref_tip_nod b);

/*inserează un nod cu cheia x în arborele binar ordonat b -
varianta iterativă*/

  ref_tip_nod q1,q2;
  int d;

    q2=b;
    q1=(*q2).drept;          /*q1=q2->drept;*/
    d=1;
    /*parcurgere arbore binar*/
    cat_timp(q1!=NULL)
      |   q2=q1;
      |   daca(x<(*q1).cheie)
      |      |    q1=(*q1).stang;
      |      | d=-1;
      |      □
      |  altfel
      |      | q1=(*q1).drept;                    /*8.3.4.c*/
      |      | d=1;
      |      □
      | □ /*terminare parcurgere arbore binar*/
    /*inserţie nod nou pe poziţia lui q1*/
    q1=aloca_memorie(tip_nod); /*creare nod nou*/
    (*q1).cheie=x; (*q1).stang=NULL; (*q1).drept=NULL;

    /*completare înlănţuire părinte*/
    daca (d<0)          /*daca (x<q2^cheie)  ...*/
```

```
        (*q2).stang=q1; /*este fiu stâng*/
      altfel
        (*q2).drept=q1; /*este fiu drept*/
/*InsereazaNerecursiv*/
-------------------------------------------------------------
```

- Este uşor de văzut că această procedură funcţionează corect **numai** dacă arborele are **cel puţin un nod.**
- Din acest motiv în implementarea structurii arborelui se utilizează **tehnica nodului fictiv**.
- Astfel iniţial arborele va conţine un **nod fictiv** a cărui înlănţuire pe dreapta indică primul **nod efectiv** al arborelui.
  - În această accepţiune **arborele binar vid** arată ca şi în figura 8.3.4.b.(a).
  - Drept consecinţă cei doi pointeri vor putea fi poziţionaţi în mod corespunzător chiar şi pentru arborele vid: `q2` indică nodul fictiv iar `q1` este NULL.
- De asemenea se face precizarea că se poate renunţa la variabila `d`.
  - Faptul că noul nod trebuie inserat ca fiu stâng sau ca fiu drept al lui `q2` se poate stabili comparând cheia lui `q2` cu cheia de inserat `x`.
  - Acest procedeu este sugerat ca şi comentariu în secvenţa [8.3.4.c.]
- Cu privire la crearea arborilor binari ordonaţi se poate menţiona faptul că **înălţimea** arborilor obţinuţi prin procedurile prezentate, depinde de **ordinea** în care se furnizează iniţial cheile.
  - Astfel, dacă spre exemplu secvenţa cheilor iniţiale este 5, 3, 8, 2, 4, 7, 9, 1, 6, atunci se obţine arborele din figura 8.3.1.a stânga, având o înălţime minimă.
  - Dacă aceleaşi chei se furnizează în ordinea 7, 2, 8, 1, 6, 9, 3, 5, 4 atunci rezultă arborele mai puţin avantajos din aceeaşi figura dreapta.

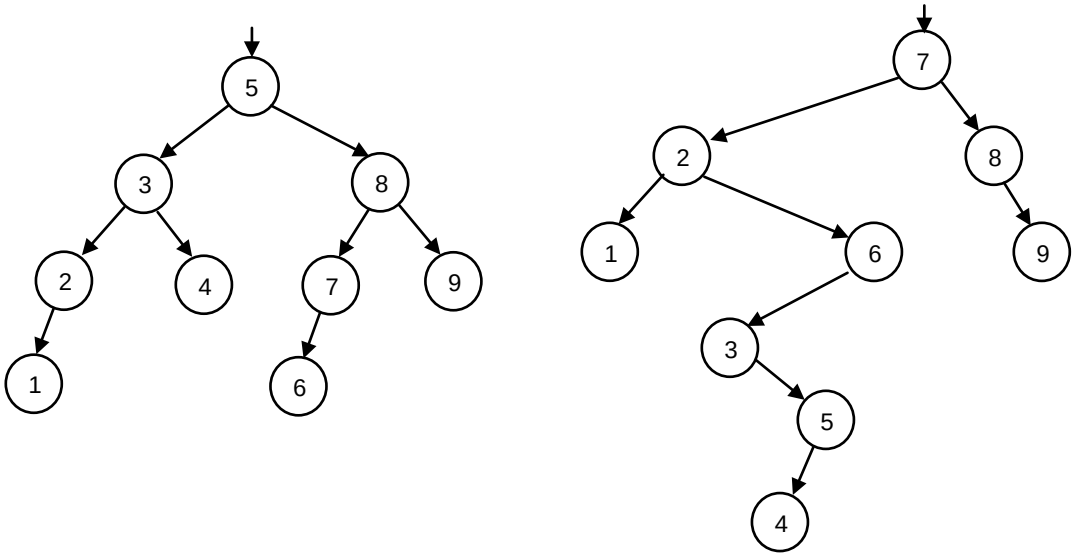


Fig.8.3.1.a. Arbori binari ordonaţi (reluare)

- În cazul cel mai **defavorabil**, arborele poate degenera în **listă liniară**, lucru care se întâmplă în cazul în care cheile sunt furnizate în vederea inserţiei în **secvenţă ordonată crescător** respectiv **descrescător** (fig.8.3.4.c.(a),(b)).

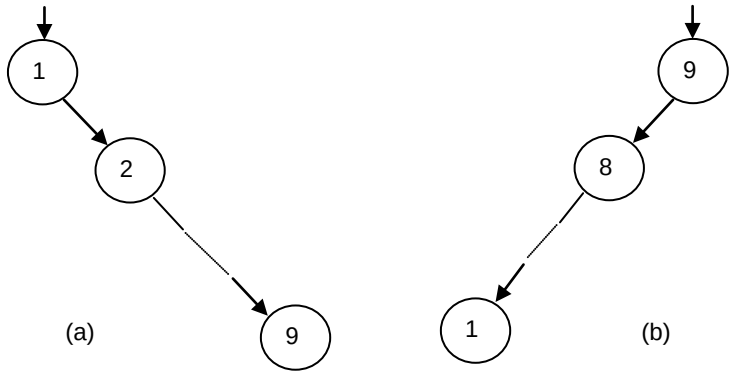


Fig.8.4.3.c. Arbori binari ordonaţi degeneraţi în liste liniare.

- Este evident faptul că în astfel de situaţii performanţa căutării scade catastrofal fiind practic egală cu cea a căutării într-o listă **liniară ordonată**.

- Din fericire, probabilitatea ca să apară astfel de situaţii este destul de redusă, fenomen ce va fi analizat mai târziu în cadrul acestui capitol.

8.3.5. Suprimarea nodurilor în arbori binari ordonaţi

- Se consideră o structură **arbore binar ordonat** şi o cheie precizată x.
- Se cere să se **suprime** din structura arbore nodul având cheia x.
    - Pentru aceasta, în prealabil se **caută** dacă există un nod cu o astfel de cheie.
    - Dacă **nu**, suprimarea s-a încheiat şi se emite eventual un mesaj de eroare.
    - În caz contrar se execută suprimarea propriu-zisă, de o asemenea manieră încât arborele să rămână **ordonat** şi după terminarea ei.
- Se disting două cazuri, după cum nodul care trebuie suprimat are:
    - (1) **Cel mult un fiu.**
    - (2) **Doi fii.**
- (1) **Primul caz** se rezolvă conform figurii 8.3.5 (a,b,c) în care se prezintă cele trei variante posibile.

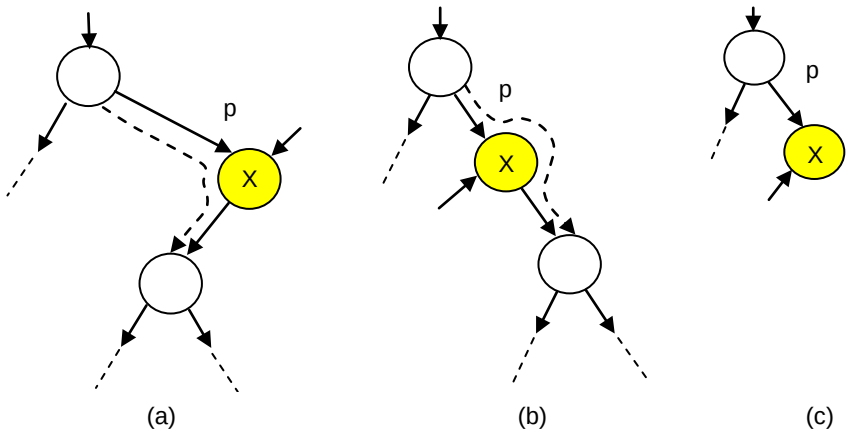


Fig.8.3.5.a. Suprimarea unui nod într-un ABO. Cazul 1.

- **Regula generală** care se deduce în acest caz este următoarea:
  - Fie p câmpul referintă aparţinând **tatălui** nodului x, referinţă care indică nodul x.
  - Valoarea lui p se **modifică** astfel încât acesta să indice **unicul fiu** al lui x (dacă un astfel de fiu există - fig. 8.3.5.a (a),(b)) sau în caz contrar p devine NULL (fig.8.3.5.a (c)).
- Fragmentul de cod care apare în continuare ilustrează acest procedeu [8.3.5.a].

```
-------------------------------------------------------------
/*Suprimarea unui nod într-un ABO. Cazul 1: nodul de suprimat
are un singur sau niciun fiu – varianta pseudocod*/

q=p; /*p indică nodul de suprimat*/
daca(q->drept=NULL)
    p=q->stang;
  altfel                                          /*8.3.5.a*/
    daca(q->stang=NULL)
      p=q->drept;
-------------------------------------------------------------
```

- Ca **exemplu**, se prezintă în continuare implementarea operatorului **SuprimaMin** care suprimă şi în acelaşi timp returnează **cel mai mic element** al unui arbore binar ordonat (secvenţa [8.3.5.b]).
  - **Cel mai mic** element al unui arbore binar ordonat, este **cel mai din stânga nod** al arborelui, nod la care se ajunge înaintând mereu spre stânga pornind de la rădacină.
  - Primul nod care **nu** are înlănţuire spre stanga (**nu** are fiu stâng) este nodul căutat.
  - Suprimarea lui este imediată în baza procedeului precizat mai sus.

```
-------------------------------------------------------------
/*Operatorul SuprimaMin în ABO - varianta pseudocod*/

ref_tip_nod SuprimaMin(ref_tip_nod b)

    ref_tip_nod temp;

    daca(b!=NULL)
      daca(b->stang!=NULL)                    [8.3.5.b]
          temp=SuprimaMin(b->stang);
        altfel /*b->stang este null*/
          |temp=b;
          |b=b->drept;   /*suprimare*/
          |return temp;
          |□
/*SuprimaMin*/
-------------------------------------------------------------
```

- Într-o manieră similară se poate implementa operatorul **`SuprimaMax`.**

    - Acesta realizează suprimarea celui mai mare nod al arborelui, care este evident nodul situat cel mai la **dreapta** în arbore.

- (2) **Cel de-al doilea caz**, în care nodul cu cheia `x` are doi fii se rezolvă astfel:

    - (1) Se caută **predecesorul** nodului de suprimat `x` în ordonarea în **inordine** a arborelui.

        - Fie acesta `y`. Se demonstrează că nodul `y` există şi că el **nu** are fiu drept.

    - (2) Se modifică nodul `x`, asignând toate câmpurile sale, cu excepţia câmpurilor `stâng` şi `drept` cu câmpurile corespunzătoare ale lui `y`.

        - În acest moment în structura arbore, nodul `y` se găseşte în dublu exemplar: în locul său iniţial şi în locul fostului nod `x`.

    - (3) Se suprimă nodul `y` iniţial, conform fragmentului [8.3.5.a] deoarece nodul nu are fiu drept.

- Cu privire la nodul `y`, se poate demonstra că el se detectează după următoarea **metodă**:

    - Se construieşte o secvenţă de noduri care începe cu fiul **stâng** al lui `x`, după care se alege drept succesor al fiecărui nod, fiul său **drept**.

    - Primul nod al secvenţei care **nu** are fiu drept este `y` (fig.8.3.5.b).

        - Este de fapt **cel mai mare nod** al subarborelui stâng al arborelui binar care are rădăcina `x`.

.

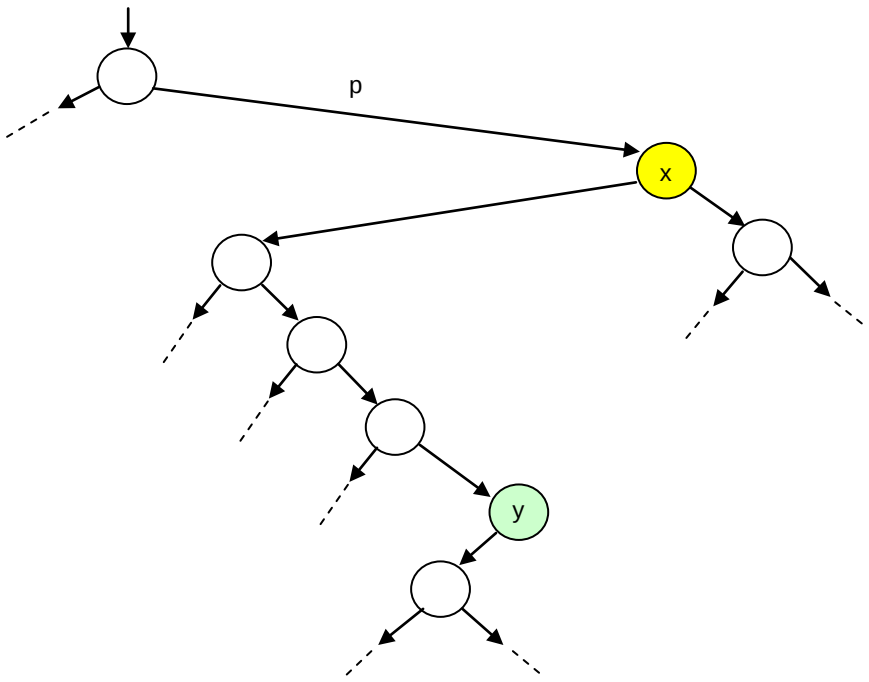


Fig. 8.3.5.b. Suprimarea unui nod într-un ABO. Cazul 2.

- Procedura care realizează **suprimarea** unui nod într-o **structură arbore binar ordonat** apare în secvenţa [8.3.5.c].

  - Procedura locală **SuprimaPred**, caută **predecesorul în inordine** al nodului x, realizând suprimarea acestuia conform metodei descrise (are cel mult un fiu).

  - După cum se observă, procedura **SuprimaPred** se utilizează numai în situaţia în care nodul x are doi fii.

```
/*Suprimarea unui nod într-un ABO. Cazul 2: nodul de suprimat
are doi fii – varianta pseudocod*/

ref_tip_nod q; /*global*/

void SuprimaPred(ref_tip_nod r)
 /*caută şi suprimă predecesorul nodului indicat de r*/

  daca(r->drept!=NULL)
      SuprimaPred(r->drept);
    altfel /*
      | q->cheie=r->cheie; /*mută conţinutul lui r în q*/
      | q=r; /*salvare adresa nod r*/
      | r=r->stang; /*suprimă nodul r*/
      | □
 /*SuprimaPred*/

void Suprima(tip_cheie x, ref_tip_nod b)
 /*caută şi suprimă nodul cu cheia x din arborele binar
   ordonat b*/

    daca(b=NULL) /*nodul nu a fost gasit*/
        afiseaza('nodul nu se gaseste');       [8.3.5.c]
      altfel
        daca(x<b->cheie) /*cautare cheie de suprimat*/
            Suprima(x,b->stang);
          altfel
            daca(x>b->cheie)
                Suprima(x,b^.drept);
              altfel /*nodul s-a gasit*/
                | q=b;
                | daca(q->drept=NULL)
                |     /*nodul nu are fiu drept*/
                |     b=q->stang;  /*suprimare*/
                |   altfel
                |     daca q->stang=NULL
                |         /*nodul nu are fiu stang*/
                |         b=q^.drept; /*suprimare*/
                |       altfel /*nodul are 2 fii*/
                |         SuprimaPred(q->stang);
                | □
  /*Suprima*/
```

- Procedura **SuprimaPred**:

  - Găseşte pointerul r care indică nodul având cea mai mare cheie, dintre cheile subarborelui stâng, al arborelui care are drept rădăcină nodul cu cheia x.

- Înlocuieşte câmpurile nodului cu cheia `x`, indicat de pointerul `q`, cu câmpurile nodului indicat de `r` (cu excepţia înlănţuirilor).

- Suprimă nodul indicat de `r`, acesta din urmă având un singur fiu (sau niciunul).

- Pentru a ilustra comportarea acestei proceduri în fig.8.3.5.c se prezintă:

  - O structură arbore binar ordonat (a).

  - Din care se suprimă în mod succesiv nodurile având cheile 7, 8, 4, şi 6 (fig.8.3.5.c (b-e)).

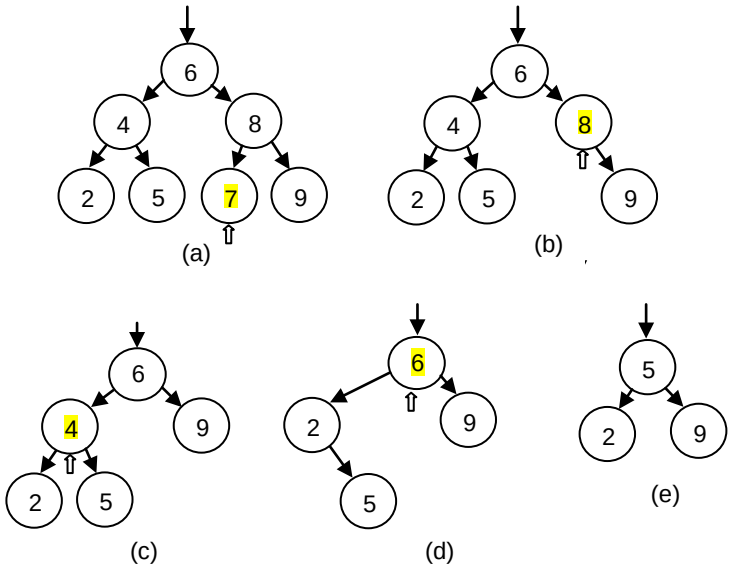


Fig. 8.3.5.c. Suprimarea nodurilor într-o structură arbore binar ordonat

- Există şi o altă soluţie de a rezolva suprimarea în cazul în care nodul `x` are doi fii şi anume:

  - (1) Se caută **succesorul** nodului `x` în ordonarea în inordine a cheilor arborelui. Se demonstrează că el există şi ca **nu** are fiu stâng.

  - (2) Pentru suprimare se procedează analog ca şi în cazul anterior, cu deosebirea că totul se realizează **simetric** (în oglindă).

  - În acest caz, de fapt se caută nodul cu cea mai mică cheie a subarborelui drept al arborelui care-l are pe `x` drept rădăcină.

- Pentru o mai bună înţelegere a celor prezentate se reaminteşte o **proprietate** a arborilor binari ordonaţi:

  - Proiecţia pe abscisă a nodurilor unui arbore binar, conduce la ordonarea lor în **inordine**.

  - În cazul **arborilor binari ordonaţi** se obţine de fapt **secvenţa ordonată crescător** a cheilor arborelui.

8.3.6. Analiza căutării în arbori binari ordonaţi

- În general, în activitatea de programare se manifestă o anumită suspiciune faţă de căutarea şi inserţia nodurilor într-o structură **arbore binar ordonat**.

- Această suspiciune este motivată de faptul că programatorul în general **nu** are controlul creşterii arborelui şi ca atare **nu** poate anticipa cu suficientă precizie forma acestuia.

    - După cum s-a precizat, efortul de căutare al unei chei variază între $O(log_2\ n)$ pentru **arborele binar perfect echilibrat** (de înălţime minimă) şi $O(n)$ pentru **arborele binar ordonat degenerat într-o listă liniară**.

    - Cele două situaţii reprezintă extremele situaţiilor reale iar probabilitatea ca ele să apară este în general redusă [Wi76].

- **Cazul general** care va fi analizat în continuare, este următorul:

    - Se consideră `n` chei.
    - Cele `n` chei pot fi permutate în `n!` moduri.
    - Întrucât forma unui **arbore binar ordonat** depinde de ordinea în care sunt inserate cheile în arbore, **fiecare permutare** conduce la un arbore binar distinct.
    - Se cere să se determine **lungimea medie a_n a drumului de căutare**, corespunzător tuturor celor `n!` arbori binari cu `n` noduri, care pot fi generaţi pornind de la cele `n!` permutări ale celor `n` chei originale.
    - Se consideră că cele `n` chei sunt **distincte** având valorile `1,2,...,n` şi se presupune că sosesc în ordine aleatoare cu o **distribuţie normală** a **probabilităţii** de apariţie.
    - În acest context **lungimea medie a drumului de căutare** într-un **arbore binar** cu `n` noduri, notată a_n , se defineşte ca fiind o **sumă** de `n` termeni, fiecare termen referindu-se la un nod al arborelui şi fiind egal cu **produsul** dintre **nivelul nodului** respectiv (adică **lungimea drumului** la nodul respectiv) şi **probabilitatea** sa de acces.
    - Dacă se presupune că **toate nodurile** sunt **în mod egal căutate** (au probabilitatea de acces `1/n`), atunci formal **lungimea medie a drumului de căutare a_n** apare în [8.3.6.a] unde p_i este lungimea drumului la nodul `i` (nivelul nodului `i`).

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \qquad [8.3.6.a]$$

- La crearea arborelui, probabilitatea ca cheia `i` să devină **prima** cheie şi deci implicit rădăcina arborelui este `1/n`.

- În consecinţă, subarborele stâng va conţine `i-1` noduri iar subarborele drept `n-i` noduri (fig.8.3.6.a).

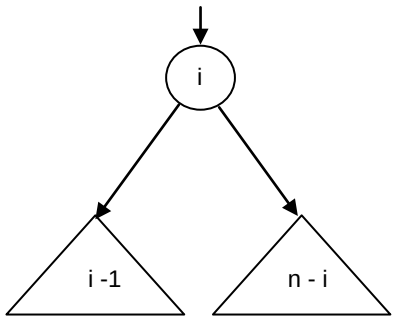


Fig.8.3.6.a. Distribuţia numerică a cheilor într-un ABO

- Fie a_{i-1} lungimea medie a drumului de căutare pentru subarborele stâng, respectiv a_{n-i} lungimea medie a drumului de căutare pentru subarborele drept, presupunînd din nou că toate permutările celor `n-1` chei rămase sunt egal posibile.

- În aceste condiţii în arborele din figura 8.3.6.a, nodurile pot fi împărţite în trei clase:

    1. Cele `i-1` noduri ale **subarborelui stâng**, care au lungimea medie a drumului egală cu $a_{i-1}+1$.

    2. **Rădăcina** care are lungimea drumului `1`.

    3. Cele `n-i` noduri ale **subarborelui drept**, cu lungimea medie a drumului egală cu $a_{n-i}+1$.

- Astfel lungimea medie a drumului de căutare $a_n^{(i)}$ pentru situaţia în care `i` este cheia nodului rădăcină poate fi exprimată ca o sumă de trei termeni [8.3.6.b].

$$a_n^{(i)} = (a_{i-1}+1)\frac{i-1}{n} + 1*\frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n} \qquad \text{[8.3.6.b]}$$

- În consecinţă, a_n poate fi considerat ca şi **media aritmetică** a termenilor $a_n^{(i)}$ pentru toţi `i=1,2,...,n`, respectiv pentru toţi cei `n` arbori care au respectiv cheile `1,2,....,n` drept **rădăcină** [8.3.6.c].

$$a_n = \frac{1}{n}\sum_{i=1}^{n} a_n^{(i)} = \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n}\right] =$$

$$= 1 + \frac{1}{n^2}\sum_{i=1}^{n}\left[(i-1)a_{i-1} + (n-i)a_{n-i}\right] = 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=1}^{n-2} i * a_i \quad \text{[8.3.6.c]}$$

- Ecuaţia [8.3.6.c] este o **relaţie de recurenţă** pentru a_n prezentată în forma $a_n=f_1(a_1, a_2,....,a_{n-1})$.

- Din această formă se poate deduce o **relaţie de recurenţă** de forma $a_n=f_2(a_{n-1})$.

- În acest scop, din formula de mai sus rezultă direct [8.3.6.d] respectiv [8.3.6.e].

$$a_n = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i * a_i = 1 + \frac{2}{n^2}(n-1)a_{n-1} + \frac{2}{n^2}\sum_{i=1}^{n-2} i * a_i \qquad [8.3.6.d]$$

$$a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i * a_i \qquad [8.3.6.e]$$

- Înmulţind relaţia [8.3.6.e] cu $((n-1)/n)^2$ se obţine [8.3.6.f]:

$$\frac{2}{n^2}\sum_{i=1}^{n-2} i * a_i = \frac{(n-1)^2}{n^2}(a_{n-1} - 1) \qquad [8.3.6.f]$$

- Înlocuind pe [8.3.6.f] în [8.3.6.d] rezultă forma dorită a relaţiei de recurenţă [8.3.6.g].

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1} + 2n - 1) \qquad [8.3.6.g]$$

- Pe de altă parte, a_n poate fi exprimat într-o **formă nerecursivă** utilizând termenii **funcţiei armonice** H [8.3.6.h] după cum se prezintă în relaţia [8.3.6.i]

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \qquad [8.3.6.h]$$

$$a_n = 2\frac{n+1}{n}H_n - 3 \qquad [8.3.6.i]$$

- Se poate verifica faptul că relaţia [8.3.6.i] verifică relaţia recursivă [8.3.6.g].

- Dar valoarea aproximativă a lui H_n poate fi determinată în baza formulei lui **Euler** [8.3.6.j].

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + ... \qquad [8.3.6.j]$$

---

unde $\gamma \approx 0.577$ este constanta lui Euler. Înlocuind această valoare în formula [8.3.6.i] rezultă [8.3.6.k].

$$a_n \approx 2\left[\ln(n) + \gamma\right] = 2\ln(n) - c \qquad [8.3.6.k]$$

- Reamintim că această valoare calculată a_n reprezintă **lungimea medie a drumului de căutare** pentru **toţi arborii binari ordonaţi** care pot fi construiţi pornind de la n chei, adică $n!$ arbori.

- Întrucât **lungimea medie a drumului de căutare** într-un **arbore binar perfect echilibrat** cu n chei este [8.3.6.l]:

$$a_n^{/} = \log_2(n) - 1 \qquad [8.3.6.l]$$

- Dacă calculăm **la limită** valoarea raportului dintre cele două lungimi de drumuri, neglijând termenii constanţi care pentru valori mari ale lui n devin neglijabili, obţinem relaţia finală [8.3.6.m].

$$\lim_{n\to\infty} \frac{a_n}{a_n^{/}} = \frac{2\ln(n)}{\log_2(n)} = \frac{2\ln(n)}{\frac{\ln(n)}{\ln 2}} = 2\ln 2 \approx 1.386 \qquad [8.3.6.m]$$

- **Concluzia** este că înlocuind **arborele binar perfect echilibrat** cu un **arbore binar ordonat oarecare**, efortul de căutare creşte în medie cu 39 %.

    - Desigur, creşterea acestui efort poate fi mult mai mare, dacă arborele binar ordonat oarecare este nefavorabil, spre exemplu degenerat într-o listă, dar această situaţie are o probabilitate foarte mică de a se realiza.

- Cele 39 % impun practic limita efortului adiţional de calcul care poate fi cheltuit în mod profitabil pentru reorganizarea structurii după inserarea cheilor.

- În acest sens un rol esenţial îl joacă **raportul** dintre numărul de **accese la noduri** (căutări) şi **numărul de inserţii** realizate în arbore.

    - Cu cât acest raport este mai mare cu atât reorganizarea structurii este mai justificată.

- În general valoarea 39 % este suficient de redusă pentru ca în majoritatea aplicaţiilor să se recurgă la tehnici directe de inserare şi să **nu** se facă uz de reorganizare decât în situaţii deosebite.

8.3.7. Arbori binari parţial ordonaţi

- O structură arbore binar aparte o reprezintă structura **arbore binar parţial ordonat**.

- Caracteristica esenţială a unui astfel de arbore este aceea că cheia oricărui nod **nu** este mai mare (mică) decât cheile fiilor săi.

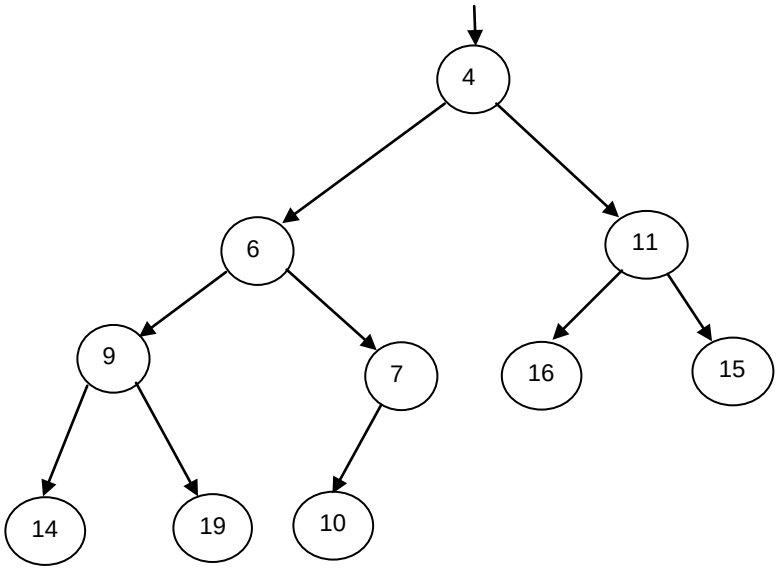


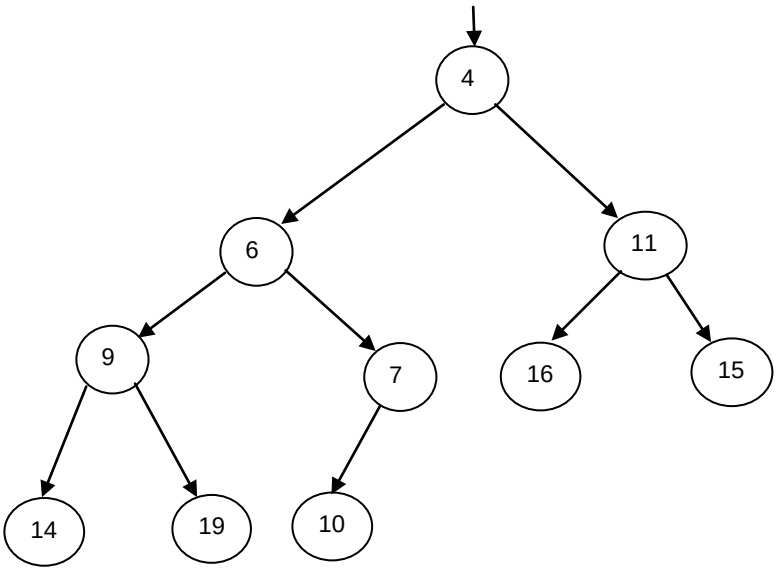
Fig.8.3.7.a. Arbore binar parţial ordonat

- Un exemplu de astfel de arbore apare în figura 8.3.7.a.

- Deoarece un arbore binar parţial ordonat este de fapt un arbore binar, se poate realiza o reprezentare eficientă a sa cu ajutorul unui tablou liniar aplicând tehnica specificată la paragraful 8.2.4.1.

- Această reprezentare este cunoscută şi sub numele de **ansamblu** (heap) şi a fost definită în partea I (sortare prin metoda ansamblelor - Vol.1 &3.2.5.)

- Spre exemplu arborele binar parţial ordonat din figura 8.3.7.a apare reprezentat ca un ansamblu în figura 8.3.7.b.



1	2	3	4	5	6	7	8	9	10
4	6	11	9	7	16	15	14	19	10

Fig.8.3.7.b. Reprezentarea unui arbore binar parţial ordonat ca un ansamblu

- Structura ansamblu permite implementarea eficientă şi foarte elegantă atât a unor metode de sortare (**sortarea prin metoda ansamblelor** - Vol.1 &3.2.5) cât şi a unor structuri de date derivate din liste (**cozi bazate pe prioritate** - Vol.1 &6.5.5.3).

- Se reaminteşte de asemena faptul că aceasta structură a fost extinsă cu un set specific de operatori într-un **tip de date abstract ansamblu** (Vol.1 &6.5.5.5).

8.3.8. Aplicaţii ale arborilor binari ordonaţi

8.3.8.1. Problema concordanţei

- În cadrul acestui paragraf se propune reluarea **problemei concordanţei** prezentată în Vol.1 şi rezolvarea ei cu ajutorul structurilor de date **arbore binar ordonat**.

- Se reaminteşte că **problema concordanţei** constă de fapt în **determinarea frecvenţelor de apariţie** ale cuvintelor unui text.

- **Problema** se formulează astfel:
    - Se consideră un text format dintr-o succesiune de cuvinte.
    - Se parcurge textul şi se delimitează cuvintele.
    - Pentru fiecare cuvânt se verifică dacă **este** sau **nu** la **prima apariţie**.
    - În caz **afirmativ**, cuvântul se înregistrează şi contorul asociat se iniţializează pe valoarea 1.
    - În caz **negativ** se incrementează contorul asociat cuvântului, contor care memorează numărul de apariţii.
    - În final se dispune de **lista (**ordonată) a **tuturor cuvintelor** şi de **numărul de apariţii** ale fiecăruia.
- În acest scop, nodurile reprezentând cuvintele sunt organizate într-o **structură arbore binar ordonat**, pornind de la un arbore vid.
- Procesul de creare a structurii arbore binar ordonat se desfăşoară după cum urmează:
    - Se citeşte un nou cuvânt şi se caută în arbore.
        - Dacă **nu** se găseşte atunci cuvântul se inserează.
        - Dacă cuvântul se găseşte, atunci se incrementează contorul de apariţii al cuvântului respectiv.
    - Procesul continuă până la epuizarea tuturor cuvintelor textului analizat.
- Se presupune că un nod al structurii **arbore binar ordonat** are structura precizată în secvenţa [8.3.8.1.a].

```
-------------------------------------------------------------
/*Problema concordanţei. Implementare bazată pe arbori binari
ordonaţi structuri de date*/

typedef struct  cuvant {
              int cheie;                    /*[8.3.8.1.a]*/
              int contor;
              struct cuvant * stang;
              struct cuvant * drept;
```

```
            } tip_nod;
```

- Fie `radacina` o variabilă pointer care indică rădăcina arborelui binar ordonat.
- Programul care rezolvă problema concordanţei apare în secvenţa [8.3.8.1.b].

```
/*PROGRAM Concordanta – varianta C*/

#include "stdafx.h"
#include <stdlib.h>

/*definire structura nod arbore*/
typedef struct  cuvant {
      int cheie;
      int contor;
      struct cuvant * stang;
      struct cuvant * drept;
} tip_nod;

typedef struct cuvant * ref_tip_nod;

ref_tip_nod radacina; /*arborele cuvintelor*/
int cuv;

void Imprarbore(ref_tip_nod r)
/*afişază arborle cu radacina r parcurgîndu-l în inodrdine*/
{
      if (r != NULL)
      {
            Imprarbore(r->stang);
            printf("cheia %d, are frecventa %d \n", r->cheie,
                  r->contor);
            Imprarbore(r->drept);
      }
} /*Imprarbore*/

void Cauta(int x, ref_tip_nod * p)
/*caută cuvântul x în arborele p; dacă nu îl găseşte îl inserează; dacă
îl găseşte îi incrementează contorul*/
{
      if ((*p) == NULL) /*cuvântul nu exista, deci inserţie*/
      {
            (*p) = (cuvant*)malloc(sizeof(struct cuvant));
            (*p)->cheie = x; (*p)->contor = 1;
            (*p)->stang = NULL; (*p)->drept = NULL;
      }
      else
      if (x<(*p)->cheie)
            Cauta(x, &(*p)->stang);
      else
      if (x>(*p)->cheie)
            Cauta(x, &(*p)->drept);
      else /*cuvânt găsit,incrementare contor*/
            (*p)->contor = (*p)->contor + 1;
} /*Cauta*/

int main()
{
      ref_tip_nod radacina;
      int cuv;
      radacina = NULL;
```

```
        printf("cheie = ");
        scanf_s("%d", &cuv);
        while (cuv != 0)
        {
              Cauta(cuv, &radacina);
              printf("cheie = ");
              scanf_s("%d", &cuv);
        }
        Imprarbore(radacina);
}
```

- Pentru simplificare se presupune ca **textul** analizat constă dintr-o succesiune de **numere întregi** care modelează cuvintele textului, iar cifra `0` este utilizată ca **terminator**.
- Textul se introduce prin furnizarea numerelor de la tastatură.
- Procedura **`Cauta`** realizează următoarele:
  - (1) Caută cheia `cuv` în arborele indicat de pointerul `radacina`.
  - (2) Dacă **nu** o găseşte, inserează cheia în arbore.
  - (3) Dacă găseşte cheia incrementează contorul corespunzător.
- După cum se observă, această procedură este o **combinaţie** a căutării şi creeării arborilor binari ordonaţi.
- Metoda de **parcurgere** a arborelui este cea prezentată la căutarea în arbori binari ordonaţi varianta recursivă (&8.3.3)
- Dacă **nu** se găseşte nici un nod cu cheia `cuv` atunci are loc inserţia similară celei utilizate în cadrul procedurii **`Insereaza`** definită la inserţia în arbori binari ordonaţi varianta 1 (&8.3.4).
- Procedura recursivă **`Imprarbore`** parcurge nodurile arborelui în **inordine** afişându-le unele sub altele, fără a reflecta însă şi structura arborelui, element care diferenţiază această procedură de cea prezentată în secvenţa [8.2.7.1.a.].
- În continuare se prezintă o **a doua variantă**, de data aceasta **nerecursivă** a procedurii **`Cauta`** bazată pe **varianta nerecursivă** a procedurii de inserţie ([8.3.4.c]).
  - Parcurgerea arborelui se face cu ajutorul a doi pointeri `q1` şi `q2` după cum s-a prezentat în paragraful 8.3.4.
  - Codul aferent acestei proceduri apare în [8.3.8.1.c] în forma procedurii **`CautaNerecursiv`**.

```
/*Problema concordanţei. Implementare bazată pe arbori binari
ordonaţi - varianta iterativă pseudocod*/

subprogram CautaNerecursiv(int x, ref_tip_nod radacina)

/*inserează un nod cu cheia x în arborele binar ordonat b -
varianta iterativă*/

    ref_tip_nod q1,q2;
    int d;

    q2=b;
    q1=(*q2).drept;          /*q1=q2->drept;*/
```

```
d=1;
cat_timp((q1!=NULL)&&(d!=0)) /*căutare în arbore binar*/
   q2=q1;
   daca(x<(*q1).cheie)
      q1=(*q1).stang;
      d=-1; /*q1 este fiu stâng*/
      □
     altfel
        daca(x>(*q1).cheie)
           q1=(*q1).drept;                    [8.3.4.c]
           d=1; /*q1 este fiu drept*/
           □
          altfel
            d=0; /*cheia s-a găsit în arbore*/
□ /*terminare căutare*/

daca(d=0) /*cheia s-a găsit în arbore*/
    (*q1).numar=(*q1).numar+1 /*incrementare contor*/
  altfel /*inserţie nod nou pe poziţia lui q1*/
    q1= aloca_memorie(tip_nod); /*creare nod nou*/
    (*q1).cheie=x; (*q1).numar=1;
    (*q1).stang=NULL; (*q1).drept=NULL;
    /*completare înlănţuire părinte*/
    daca d<0          /*daca (x<q2^cheie)  ...*/
       (*q2).stang=q1; /*q1 este fiu stâng*/
     altfel
       (*q2).drept=q1; /*q1 este fiu drept*/
    □
/*CautaNerecursiv*/
```

- Condiţia necesară ca această procedură să lucreze corect este ca arborele să **aibă cel puţin** un nod, motiv pentru care în implementarea structurii arborelui s-a utilizat **tehnica nodului fictiv.**

- De asemenea, pentru funcţionarea corectă a programului **`Concordanta`** cu procedura **`CautaNerecursiv`**, iniţializarea `radacina=NULL` din programul principal indicată cu [*] în secvenţa [8.3.8.1.b], se înlocuieşte cu secvenţa [8.3.8.1.d].

```
---------------------------------------------------------------
          radaciana= (cuvant*)malloc(sizeof(struct cuvant);
          radacina->drept=NULL;                [8.3.8.1.d]
---------------------------------------------------------------
```

- În consecinţă `radacina` indică **un nod fictiv** în cadrul căruia se asignează numai câmpul `drept`.

- Rădăcina efectivă a arborelui apare ca fiu drept al acestui nod. În mod evident trebuie reformulat apelul procedurii de căutare din programul principal.

8.3.8.2. Generator de referinţe încrucişate

- În cadrul acestui paragraf se prezintă un program pentru construirea unui **index** de **referinţe încrucişate** ("cross-reference index") referitor la un text dat.

- Specificaţia programului este următoarea:

- Programul citeşte un **text** din fişierul de intrare.
- Selecţionează şi memorează **toate cuvintele** textului precum şi **numerele rândurilor** în care acestea au fost întâlnite.
- La terminarea parcurgerii textului, programul furnizează o **listă** conţinând **toate cuvintele ordonate alfabetic.**
- Pentru fiecare cuvânt în parte, se furnizează numerele liniilor de text în care el apare (**tabela de referinţe încrucişate**).

- Metoda utilizată este aceea de a construi un **arbore binar ordonat** în care cheia (identificatorul) fiecărui nod este un cuvânt al textului.
- Un astfel de arbore care este ordonat în ordinea alfabetică a cuvintelor se mai numeşte şi **arbore lexicografic**.
- În afara cheii, fiecare nod al arborelui conţine un **pointer** la o **listă liniară** care păstrează numerele rândurilor în care a fost întâlnit respectivul identificator.
- Pentru a accelera procesul de inserţie al unui nod nou în această listă, conform celor precizate la studiul listelor înlănţuite, (Vol.1 &6.3.2.1), este indicat a se păstra o **referinţă** la **sfârşitul listei**, referinţă care apare tot ca şi un câmp distinct al nodului arborelui.
- Se observă astfel că în cadrul acestui exemplu se utilizează structuri de date combinate: arbori şi liste. Avem de fapt de-a face cu **un arbore de liste**.
- Programul complet varianta C apare în secvenţa [8.3.8.2.a] şi el constă din două faze:
  (1) **Faza de parcurgere a textului**, de identificare a cuvintelor şi de creare a **arborelui lexicografic**.
  (2) **Faza de afişare** a tabelei de referinţe încrucişate.
- Pentru urmărirea mai uşoară a programului se fac următoarele precizări:
- Prin **cuvânt** al textului se înţelege orice secvenţă de litere sau cifre care începe cu o literă. Cuvântul se termină la întâlnirea primului caracter diferit de literă sau cifră.
- Funcţia **`cauta`** are rolul de a căuta cuvântul curent în structura **arbore binar ordonat**.
  - Dacă cuvântul **nu** este găsit el se inserează în arbore.
    - Se precizează că la inserţia unui nod nou în arbore, se crează şi primul nod al listei liniare ataşate.
  - Dacă cuvântul este găsit, se adaugă un nou nod listei liniare ataşate conţinând numărul liniei curente.
  - Funcţia returnează un pointer la nodul găsit respectiv nou inserat.
- Funcţia **`listeaza_cuvant`** realizeză afişarea cuvântului asociat unui nod al arborelui lexicografic, urmat de afişarea listei care memorează liniile în care el apare.
- Funcţia **`parcurge_arbore`** realizează **traversarea în inordine a arborelui** şi afişează nodurile întâlnite obţinându-se astfel **tabelul ordonat de referinţe încrucişate.** Ea face uz de funcţia **`listeaza_cuvant`**.
- Programul principal este implementat de funcţia **`parcurge_fisier`**.
  - Această funcţie parcurge textul sursă, delimitează cuvintele şi pentru fiecare cuvânt apelează funcţia **`caută`** care are rolul de a completa, respectiv de a modifica structura arborelui indicat de variabila `radacina`.

- La terminarea parcurgerii textului se afişează **tabela de referinţe încrucişate** (funcţia **parcurge_arbore**) după care se dezafectează spaţiul de memorie alocat prin distrugerea arborelui lexicografic.

```
/* Generator de referinţe încrucişate*/

#include <stdio.h>
#include <malloc.h>
#include <string.h>

#define LMAXCUVANT 64

typedef struct rand
{
     int nr;
     rand *urm;
} rand;

typedef struct cuvant
{
     char cheie[LMAXCUVANT];
     rand *prim, *ultim;
     cuvant *stang, *drept;
} cuvant;

cuvant* cauta(cuvant *pc,char *cheie,int nr_linie)
{
     if(pc==NULL) /*creare nod terminal în arborele binar*/
     {
          rand *pr2;
          pr2=(rand*)malloc(sizeof(rand));
          pr2->nr=nr_linie;
          pr2->urm=NULL;
          cuvant *pc2;
          pc2=(cuvant*)malloc(sizeof(cuvant));
          strcpy(pc2->cheie,cheie);
          pc2->prim=pc2->ultim=pr2;
          pc2->stang=pc2->drept=NULL;

          return pc2;
     }
     else
     {
          if(strcmp(pc->cheie,cheie)>0) /* căutare cuvânt în
                                           subarborele stâng */
          {
               pc->stang=cauta(pc->stang,cheie,nr_linie);
               return pc;
          }
          else
          if(strcmp(pc->cheie,cheie)<0) /* căutare cuvânt în
                                           subarborele drept */
          {
               pc->drept=cauta(pc->drept,cheie,nr_linie);
               return pc;
          }
          else /* cuvântul a fost găsit şi se va insera un
```

```
                    nod în lista înlănţuită */
          {
               rand *pr;
               pr=(rand*)malloc(sizeof(rand));
               pr->nr=nr_linie;
               pr->urm=NULL;
               pc->ultim->urm=pr;
               pc->ultim=pr;
               return pc;
          }
     }
}

void listeaza_cuvant(cuvant *pc) /* listare cuvânt şi linii
                                    apariţie */
{
     rand *pr;
     printf("%s\t",pc->cheie);
     for(pr=pc->prim;pr!=NULL;pr=pr->urm)
          printf("%d ",pr->nr);
     printf("\n");
}

void parcurge_arbore(cuvant *pc) /* parcurgere arbore în
                                    inordine */
{
     if(pc!=NULL)
     {
          parcurge_arbore(pc->stang);
          listeaza_cuvant(pc);
          parcurge_arbore(pc->drept);
     }
}

cuvant* distruge_arbore(cuvant *pc) /* eliberare memorie
                                       arbore */
{
     if(pc!=NULL)
     {
          pc->stang=distruge_arbore(pc->stang);
          pc->drept=distruge_arbore(pc->drept);

          /* se şterge lista simplu înlănţuită din nod */
          rand *pr,*pr2;
          pr=pc->prim;
          while(pr!=NULL)
          {
               pr2=pr;
               pr=pr->urm;
               free(pr2);
          }
          pc->prim=pc->ultim=NULL;

          free(pc);
     }

     return NULL;
}
```

```
void parcurge_fisier(char *nume)
{
     FILE *f=NULL;
     cuvant *radacina=NULL;
     char cuv[LMAXCUVANT],c;
     int i=0;
     int n=1;

     if((f=fopen(nume,"rt"))==NULL)
     {
          printf("Eroare la citire fisier...\n");
          return;
     }

     while(!feof(f))
     {
          fscanf(f,"%c",&c);
          if(((c>='a')&&(c<='z'))||((c>='A')&&(c<='Z')))
                                   /* citire identificatori */
          {

     while((((c>='a')&&(c<='z'))||((c>='A')&&(c<='Z')))&&(i<
               LMAXCUVANT)&&(!feof(f)))
                {
                     cuv[i]=c;
                     i++;
                     fscanf(f,"%c",&c);
                }

                cuv[i]=0;
                radacina=cauta(radacina,cuv,n);
                i=0;
          }

          if((c>='0')&&(c<='9')) /* citire constante
                                   numerice */
          {
               while(((c>='0')&&(c<='9'))&&(!feof(f)))
               {
                    fscanf(f,"%c",&c);
               }
          }

          if(c==10) /* citire rând nou */
          {
               n++; /* incrementare nr. linie */
          }
     }
     fclose(f);

     parcurge_arbore(radacina);
     radacina=distruge_arbore(radacina);
}

void main(int argc, char* argv[])
{
     if(argc>1)
```

```
    {
        parcurge_fisier(argv[1]);
    }
}
-------------------------------------------------------------
```

- Desigur, programul de faţă abordează la nivel schematic partea de analiză a textului şi de evidenţiere a cuvintelor.

- Pornind de la această formă simplificată se poate concepe o abordare mai complexă a construcţiei tabelei de referinţe încrucişate care ţine cont şi de alte aspecte cum ar fi delimitatorii sau caracterele speciale.

8.4. Arbori de regasire (“Trie Trees”)

8.4.1. Definire

- **Arborii de regăsire** sunt structuri de date speciale care pot fi utilizate în repezentarea **mulţimilor de caractere**.

- De asemenea cu ajutorul lor pot fi reprezentate tipuri de date care sunt **şiruri de obiecte** de orice tip sau **şiruri de numere**.

- În literatura de specialitate **arborii de regăsire** sunt cunoscuţi sub denumirea de structuri **trie,** cuvânt derivat din cuvântul **“retrieval”** (regăsire).

- Un **arbore de regăsire** permite implementarea simplă a operatorilor definiţi asupra unei structuri de date **mulţime** ale cărei elemente sunt **şiruri de caractere** (cuvinte).

- Este vorba în principiu despre operatorii:

    - ***Insereaza***.
    - ***Suprimă***.
    - ***Aparţine***.
    - ***Iniţializează***.
    - ***Afişează***.

    - Ultimul operator realizează afişarea tuturor membrilor (cuvintelor) mulţimii.

- Utilizarea arborilor de regăsire este eficientă atunci când există **mai multe cuvinte** care **încep cu aceeaşi secvenţă de caractere**, adică atunci când numărul de **prefixe distincte** al tuturor cuvintelor din mulţime este mult mai redus decât numărul total de cuvinte.

- Într-un **arbore de regăsire** fiecare **drum** de la **rădăcină** spre un **nod terminal** corespunde unui **cuvânt al mulţimii** care este reprezentată de arbore.

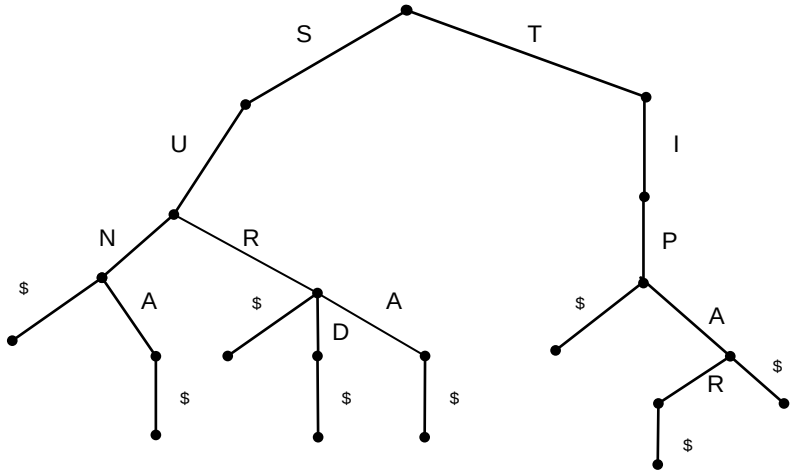
    - Astfel **nodurile** arborelui de regăsire corespund **prefixelor** cuvintelor mulţimii.

    - Pentru delimitarea cuvintelor se foloseşte un **caracter special de sfârşit** (simbolul `$´).

- În figura 8.4.1.a apare un arbore de regăsire reprezentând o mulţime formată din cuvintele SUN, SUNA, SUR, SURD, SURA, TIP, TIPA, TIPAR.

    - Rădăcina corespunde şirului vid, iar cei doi fii ai săi corespund prefixelor S şi T.

    - Parcurgând drumurile arborelui rezultă celelalte cuvinte precizate.



SUN, SUNA, SUR, SURD, SURA, TIP, TIPA, TIPAR

Fig.8.4.1.a. Arbore de regăsire

- În legătură cu **arborii de regăsire** se pot face următoarele observaţii:

  - (1) Fiecare nod al arborelui poate avea cel mult 29 de fii (câte unul pentru fiecare literă a alfabetului, plus caracterul `$´).
  - (2) Cele mai multe dintre noduri vor avea mult mai puţini fii decât 29.
  - (3) Un nod la care se ajunge printr-un ram etichetat cu caracterul `$´, **nu** poate avea nici un fiu şi care atare poate fi eventual omis din structură.

8.4.2. Structura de date "Nod arbore de regăsire"

- Un nod al unui arbore de regăsire poate fi privit ca o **structură asociere** al cărei **domeniu** este mulţimea `{A,B,C,...,Z,$}` şi al cărei **codomeniu** este o **mulţime de valori** aparţinând tipului `ReferinţăNodArboreDeRegăsire`.

- Cu alte cuvinte o **asociere** definită pe mulţimea **caracterelor** cu valori **în mulţimea pointerilor la noduri** (fig.8.4.2.a).

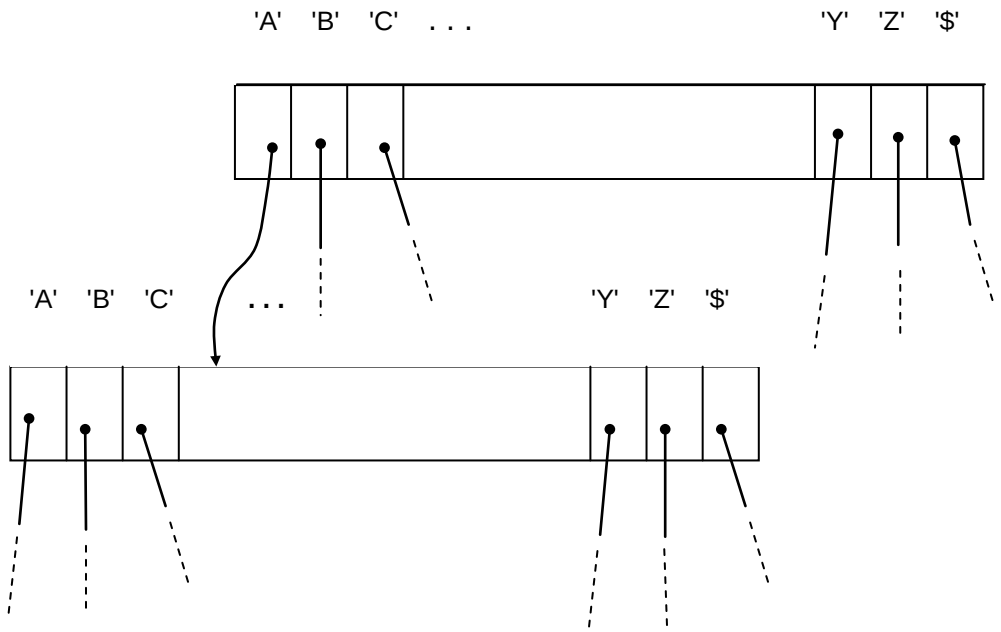


Fig.8.4.2.a. Structura de date `NodArboreDeRegăsire`

- Un **arbore de regăsire** poate fi identificat printr-un pointer la **rădăcina** sa (care este de fapt un nod).

- În consecinţă rezultă că **structurile de date abstracte** "`ArboreDeRegăsire`" şi "`NodArboreDeRegăsire`" pot fi încadrate într-o **aceeaşi** structură de date, deşi **operatorii specifici** care se aplică asupra fiecăreia dintre ele sunt substanţial **diferiţi**.

- Presupunând spre exemplu că:

  - `p` este un pointer la o structură `NodArboreDeRegăsire` care este încadrat în tipul `RefNodArboreDeRegasire`.

  - `Nod` o instanţă a structurii `NodArboreDeRegăsire`

  - `c` este o valoare de tip caracter,

- Asupra structurii `NodArboreDeRegăsire` se definesc următorii operatori:

  1. ***`Initializează`*** *`(NodArboreDeRegasire Nod)`* - operator care face ca Nod să indice asocierea vidă.

  2. ***`Atribuie`*** *`(NodArboreDeRegasire Nod, char c, RefNodArboreDeRegasire p)`*- operator care asociază caracterului *`c`* din nodul *`Nod`* referinţa *`p`*. După cum se observă, *`p`* este o referinţă la un nod al arborelui.

3. *RefNodArboreDeRegasire* ***ValoareNod*** *(NodArboreDeRegasire Nod, char c)* - operator care returnează referinţa asociată caracterului *c* din *Nod*.

4. ***NodNou*** *(NodArboreDeRegasire Nod, char c)*- operator care face ca valoarea lui *Nod* pentru caracterul *c* să fie un pointer la un nod nou iniţializat pe asocierea nulă.

8.4.2.1. Implementare bazată pe tablouri a structurii Nod arbore de regăsire

- O implementare simplă a **nodurilor unui arbore de regăsire** este cea bazată pe un **tablou de pointeri la noduri**, al cărui **set de indici** este format din mulţimea caracterelor `A,B,C,...,Z,$`.

- Se pot defini următoarele **structuri de date** [8.4.2.1.a].

```
-------------------------------------------------------------
/*Structura NodArboreDeRegasire - Implementare bazată pe
tablouri*/

int Dim_Nod=29;
typedef ref_nod_arb_regăsire * nod_arb_regăsire;
                                          /*[8.4.2.1.a]*/
typedef ref_nod_arb_regăsire nod_arb_regăsire[Dim_Nod];

-------------------------------------------------------------
{Structura NodArboreDeRegasire - Implementare bazată pe
tablouri – varianta PASCAL}

TYPE RefNodArboreDeRegăsire=^NodArboreDeRegăsire;
     Caractere=('A', 'B', 'C',..., 'Z' ,'$' );
                                           {[8.4.2.1.a]}
     NodArboreDeRegăsire = ARRAY [Caractere] OF
       RefNodArboreDeRegăsire;
-------------------------------------------------------------
```

- Varianta pseudocod a implementării operatorilor definiţi pe structura `NodArboreDeRegăsire` apare în secvenţa [8.4.2.1.b].

- Pentru a **nu supraîncărca** structura cu noduri frunză care sunt fiii corespunzători caracterului `'$'` se adoptă următoarea **convenţie**:

  - (1) `Nod['$']` are valoarea `null`.

    - În acest caz nodul **nu** are fiu corespunzând caracterului `'$'`.

  - (2) `Nod['$']` are valoarea **pointerului** care indică nodul însuşi.

    - În acest caz se presupune că nodul are un astfel de fiu, care de fapt **nu se crează niciodată.**

- O astfel de situaţie precizează **sfârşitul** unui **cuvânt** pe **nivelul anterior** al structurii arborelui de regăsire, respectiv la părintele nodului respectiv (fig.8.4.2.1.a).

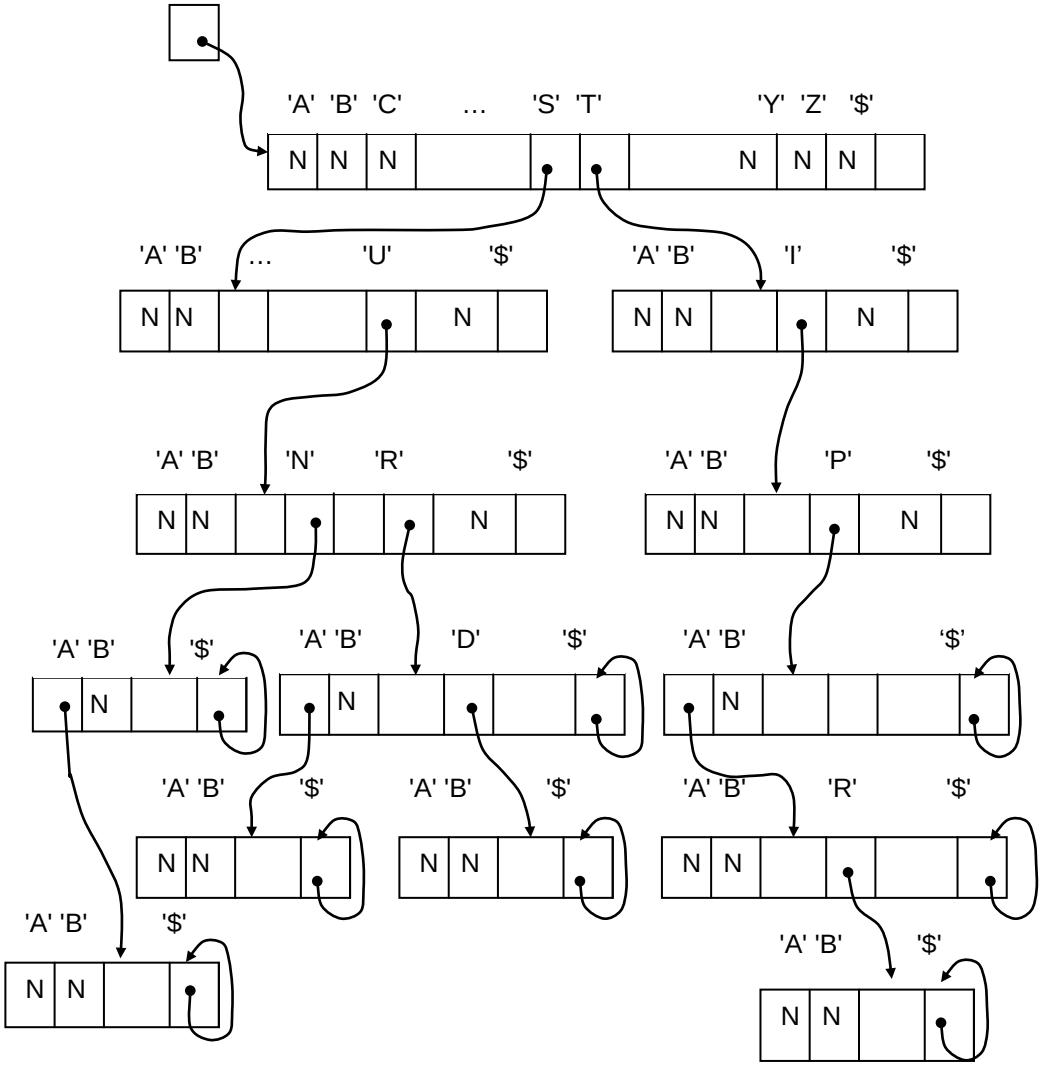


Fig.8.4.2.1.a. Reprezentarea structurii `ArboreDeRegasire`

```
-----------------------------------------------------------------
/*Implementarea bazată pe tablouri a operatorilor
  TDA NodArboreDeRegăsire - varianta pseudocod*/

Subprogram Initializeaza(nod_arb_regăsire Nod)
  CHAR c;
  pentru c='A' pana la '$'
    Nod[c]= NULL;
/*Initializeaza*/

Subprogram Atribuie(nod_arb_regăsire Nod, CHAR c,
  ref_nod_arb_regăsire p)
    Nod[c]= p;                                    [8.4.2.1.b]
/*Atribuie*/

ref_nod_arb_regăsire ValoareNod(nod_arb_regăsire Nod,CHAR c)
    returneaza Nod[c];
/*ValoareNod*/

Subprogram NodNou (nod_arb_regăsire Nod, CHAR c);
```

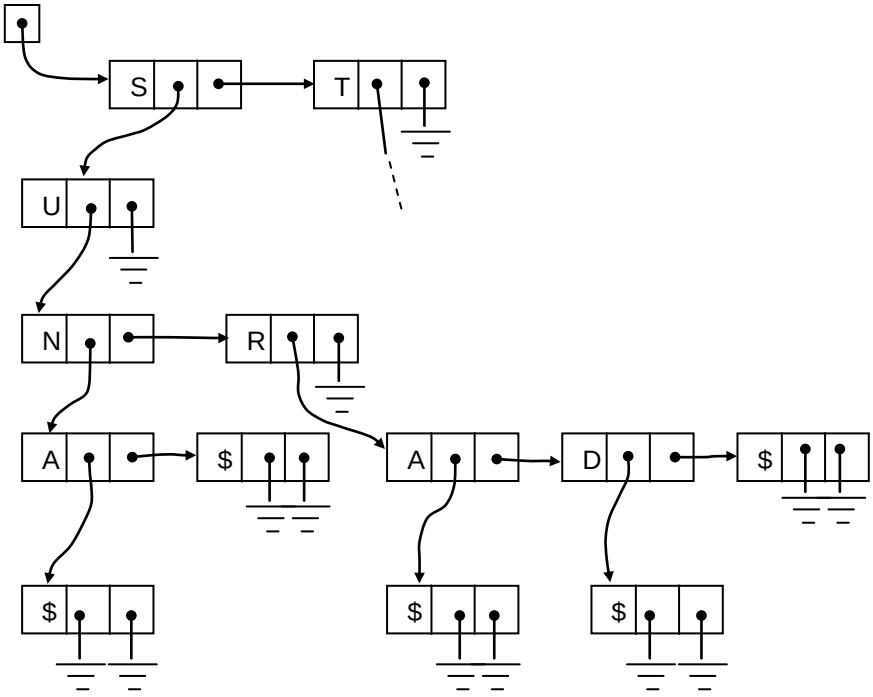
```
    Nod[c]= aloca_memorie(nod_arb_regasire);
    Initializeaza(Nod[c]);
/*NodNou*/
-------------------------------------------------------------
```

8.4.2.2. Implementare bazată pe liste înlănţuite a structurii Nod arbore de regăsire

- Implementarea nodurilor arborilor de regăsire cu ajutorul tablourilor este **ineficientă** deoarece:
  - (1) Se rezervă în **fiecare nod** loc pentru pointeri la **toate literele** alfabetului.
  - (2) Se ajunge astfel ca **volumul de memorie** ocupat de structură să **depăşească** cu mult lungimea totală a cuvintelor mulţimii.
- Sunt posibile însă şi alte reprezentări.
- Pornind de la observaţia că un nod al unui arbore de regăsire este de fapt o **asociere**, în principiu poate fi utilizată **orice reprezentare** a unei astfel de **structuri asociere**.
- În cazul de faţă, deoarece pentru un nod **domeniul** asocierii poate conţine un număr variabil de elemente, apare ca deosebit de potrivită reprezentarea bazată pe **liste înlănţuite**.
- Astfel **asocierea** corespunzătoare unui **nod al unui arbore de regăsire** poate fi reprezentată ca o **listă înlănţuită** de caractere pentru care valoarea asociată este un pointer diferit de NULL [8.4.2.2.a].

```
-------------------------------------------------------------
/*Structura NodArboreDeRegasire - Implementare bazată pe
liste înlănţuite – varianta C*/

typedef Nod_lista * ref_nod_lista;

typedef struct Nod_lista
  {
     char domeniu;                        /*[8.4.2.2.a]*/
     ref_nod_lista valoare; /*indică prima celulă
                                  a listei nodului fiu*/
     ref_nod_lista urmator;  /*indică următoarea
                            celulă a listei nodului curent*/
  } Nod_lista;

typedef ref_nod_lista nod_arb_regăsire;

-------------------------------------------------------------
```

- În figura 8.4.2.2.a apare reprezentată în această manieră o porţiune a arborelui de regăsire din figura 8.4.1.a.



Fig,8.4.2.2.a. Arbore de regăsire implementat cu ajutorul listelor

- În aceste condiţii, operatorii ***Initializeaza***, ***Atribuie***, ***ValoareNod*** şi ***NodNou*** definiţi pentru structura `NodArboreDeRegăsire` se pot implementa cu uşurinţă.

8.4.3. Structura de date Arbore de regăsire

- Pornind de la cele anterior precizate, în continuare se poate defini **structura de date** `ArboreDeRegăsire`:

```
-------------------------------------------------------------
typdef ref_nod_arb_regăsire tip_arb_regăsire; /*[8.4.3.a]*/
-------------------------------------------------------------
```

- Pe această structură de date se pot implementa operatorii specifici:
  Iniţializare, ***Inserează***, ***Aparţine***, ***Suprimă*** şi ***Afişează***.

- Pentru exemplificare se prezintă implementarea operatorului ***Inserează*** utilizând drept suport, operatorii definiţi pe structura `nod_arbore_regasire` implementată la rândul ei cu ajutorul **tablourilor**.

  - Se presupune că în prealabil este definit un `tip_cuvânt` reprezentat printr-un **tablou de caractere**.

  - Acest tablou de caractere conţine un singur cuvânt care se termină cu caracterul `'$'`.

- În acest context, procedura ***Inserează***`(tip_cuvânt x, tip_arb_regăsire cuvinte)` care inserează cuvântul `x` în mulţimea `cuvinte` reprezentată printr-un arbore de regăsire, apare în secvenţa [8.4.3.b].

```
-------------------------------------------------------------
/*Implementarea operatorului Insereaza pentru arbori de
```

```
regăsire implementaţi cu ajutorul tablourilor - varianta
pseudocod*/

typedef char sir[] tip_cuvânt; /*cuvântul de inserat*/

Subprogram Inserează (tip_cuvant x, tip_arb_regasire
cuvinte)
/*inserează cuvântul x în arborele de regăsire cuvinte*/

  int i; /*precizează poziţia caracterului curent în
                  cuvântul x*/
  tip_arb_regasire t; /*utilizat pentru a parcurge
            nodurile arborelui de regăsire corespunzator
            prefixelor lui x*/
  i=0;                                        /*[8.4.3.b]*/
  t=cuvinte; /*rădăcina arbore*/
  cât_timp (x[i]!=`$`)
    daca (ValoareNod (*t, x[i])== NULL) NodNou(*t, x[i]);
      /*dacă nodul curent nu are fiu pentru caracterul
          x[i], atunci se creează unul*/
      t=ValoareNod (*t,x[i] ); /*se trece la fiul lui t
        pentru caracterul x [i] (chiar dacă a fost
                  creat recent)*/
      i=i+1; /*se avansează la caracterul următor în
                       cuvântul x*/
    □ /*cât_timp*/
  /*s-a ajuns la caracterul `$` în cuvantul x*/
  Atribuie(*t,`$`,t) /*se face o buclă pentru `$` pentru a
                        marca un nod terminal*/
/*Insereaza*/
```

- Se presupune că cuvântul de inserat **nu** se găseşte în structură.

- **Procedura** funcţionează după cum urmează:

  - `i` este cursorul de caractere în tabloul `x`.

  - `t` este un pointer utilizat în parcurgerea nodurilor arborelui de regăsire.

  - Se parcurge cuvântul `x` caracter cu caracter în bucla **`cât_timp`** până la întâlnirea caracterului `'$'`.

  - Pentru fiecare caracter `x[i]` diferit de caracterul `'$'` întâlnit, se verifică dacă nodul curent are un fiu pentru acest caracter. Dacă **nu** are, se crează un astfel de fiu.

  - Se trece pe nivelul următor al arborelui la fiul caracterului `x[i]`. Acest lucru se întâmplă chiar dacă nodul a fost creat în iteraţia curentă.

  - Se avansează la caracterul următor în cuvântul `x` şi se reia bucla de prelucrare.

  - Când se ajunge la sfârşitul cuvântului `x,(x[i]='$')`, parcurgerea arborelui de regăsire a avansat până la nivelul următor ultimului caracter introdus.

  - În acest moment se realizează înlănţuirea `Nod['$']=t`, adică se realizează înlănţuirea nodului cu el însuşi pentru a marca **sfârşitul** cuvântului introdus.

- Implementarea operatorilor **`Iniţializare`**, ***`Aparţine`***, ***`Suprimă`*** şi ***`Afişează`*** se recomandă ca şi exerciţiu.

- Într-o manieră similară se poate realiza implementarea aceloraşi operatori utilizând de această dată implementarea bazată pe liste înlănţuite a arborilor de regăsire.

8.4.4. Evaluarea performanţei structurii Arbore de regăsire

- În cele ce urmează ne propunem să realizăm o **analiză comparată** din punctul de vedere al **timpului** şi al **volumului** de memorie necesar pentru **reprezentarea mulţimilor**.

- Se porneşte de la următoarele considerente:
  - Se reprezintă `n` cuvinte.
  - Cuvintele au `p` prefixe diferite.
  - Lungimea totală a tuturor cuvintelor este de `m` caractere.

- Analiza comparată se va realiza utilizând în acest scop structurile **tabelă de dispersie** respectiv **arbore de regăsire**.
  - În cele ce urmează se consideră **pointerii** implementaţi pe `4` octeţi.

- Probabil că cel mai eficient mod de a memora cuvintele, din punct de vedere al spaţiului ocupat, mod care permite implementarea simplă a operatorilor ***`Inserează`*** şi ***`Suprimă`*** este **tabela de dispersie deschisă** bazată pe **înlănţuirea directă** (Vol.1 &7.3.3.1).

- Deoarece cuvintele au lungime variabilă, nodurile listelor asociate tabelei de dispersie **nu** vor conţine chiar cuvintele, ci fiecare nod va conţine două referinţe:
  - Un pointer care înlănţuie într-o **listă înlănţuită** nodurile ale căror **indici primari** sunt indentici.
  - Un idicator care indică începutul cuvântului într-o **zonă de cuvinte**.

- Cuvintele vor fi memorate în **zona de cuvinte**, ca şi un tablou de caractere, sfârşitul fiecărui cuvânt fiind indicat de caracterul `'$'`.

- Spre exemplu cuvintele SUN, SUNA, SURD vor fi memorate astfel:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 ...
S U N $ S U N A $ S  U  R  D  $...
```

- Pointerii la cele trei cuvinte sunt de fapt **cursori** la poziţiile `0`,`4` şi `9` ale tabelului de cuvinte, poziţii care marchează începutul cuvintelor respective.

- **Spaţiul de memorie** necesar pentru implementarea bazată pe structura **tabelă de dispersie deschisă** poate fi determinat pe baza calculului următor:

a) Sunt necesari `8n` octeţi pentru **nodurile listelor asociate tabelei** (avem `n` cuvinte fiecare necesitând câte doi pointeri a 4 octeţi).

b) Sunt necesari `m+n` octeţi pentru cele `n` **cuvinte** memorate în **zona de cuvinte** cu lungimea totală `m` şi caracterele lor de sfârşit;

c) Sunt necesari `4d` octeţi pentru **tabela de dispersie** propriu-zisă, unde `d` este dimensiunea tabelei (un număr prim).

- Însumând aceste valori, în **total** pentru reprezentarea bazată pe structura **tabela de dispersie deschisă** rezultă un necesar de **`9n+m+4d`** octeţi.

- În cazul **arborelui de regăsire** ale cărui noduri sunt implementate ca şi **liste înlănţuite** sunt necesare `p+n` noduri, unde:
  - `p` reprezintă numărul de noduri destinate **prefixelor** (câte unul pentru fiecare prefix).
  - `n` reprezintă numărul de noduri destinate **sfârşiturilor de cuvinte** (câte unul pentru fiecare cuvânt).

- Fiecare **nod** conţine un caracter şi doi pointeri necesitând `9` octeţi de memorie.

- În total reprezentarea bazată pe structura **arbore de regăsire** necesită **`9n+9p`** octeţi.

- Se compară necesarul de `9n+m+4d` octeţi pentru tabela de dispersie cu `9n+9p` octeţi, necesarul pentru arborele de regăsire
  - De fapt trebuie comparate valorile `m+4d` şi `9p`.
  - De regulă în aplicaţii care implementează dicţionare de mari dimensiuni, raportul `m/p` este de obicei mai mic ca `3`, deci `m<3p`.

- În consecinţă, concluzia este că **tabela de dispersie** va utiliza mai puţin spaţiu.

- Din punctul de vedere al **timpului de execuţie**:
  - În cazul **arborelui de regăsire** realizarea operaţiilor ***`Inserează`***, ***`Suprimă`*** şi ***`Aparţine`*** consumă un **timp** proporţional cu lungimea cuvântului implicat.
  - În cazul **tabelului de dispersie**, calculul valorii funcţiei de dispersie consumă un interval de timp de calcul comparabil cu realizarea operaţiei ***`Aparţine`*** într-un arbore de regăsire.
  - Deoarece calculul valorii funcţiei de dispersie **nu** include timpul necesar rezolvării problemelor de **coliziune**, sau realizării efective a inserţiei, a suprimării sau testului de apartenenţă, este de aşteptat ca **arborii de regăsire** să fie **considerabil mai rapizi** decât **tabelele de dispersie**.
- Un alt avantaj al **arborelui de regăsire** este acela că **suportă** realizarea eficientă a operaţiei **`Min`**, obiectiv dificil de realizat în cazul **tabelei de dispersie**.

8.5. Arbori binari echilibraţi. Arbori AVL

8.5.1. Definirea arborilor echilibraţi AVL

- Din **analiza** căutării în **arbori binari ordonaţi** prezentată în &8.3.6. rezultă în mod evident că o procedură de inserare care **restaurează** structura arbore astfel încât ea să fie **tot timpul** *perfect echilibrată* **nu** este viabilă, deoarece activitatea de restructurare este foarte **complexă**.

- Cu toate acestea sunt posibile anumite abordări mai realiste, dacă termenul **"echilibrat"** este definit într-o manieră **mai puţin strictă**.

- Astfel de criterii de echilibrare "**imperfectă**" pot conduce la **tehnici** mai simple de **reorganizare a structurilor arbori binari ordonaţi**, al căror cost deteriorează într-o măsură redusă **performanţa medie de căutare**.

- Una dintre aceste definiţii ale **echilibrării arborilor binari ordonaţi** este cea propusă de **Adelson, Velskii** şi **Landis** în 1962 şi care are următorul enunţ:

    - Un **arbore binar ordonat** este **echilibrat** dacă şi numai dacă pentru oricare nod al arborelui, înălţimile celor doi subarbori diferă cu cel mult 1.

- Arborii care satisfac acest criteriu se numesc "**arbori AVL**" după numele inventatorilor.

- În cele ce urmează, aceşti arbori vor fi denumiţi "**arbori echilibraţi**".

    - Se atrage atenţia asupra faptului că **arborii perfect echilibraţi** sunt de asemenea **arbori AVL**.

- Această definiţie are câteva **avantaje**:

    - (1) Este foarte **simplă**.

    - (2) Conduce la o **procedură** viabilă de re-echilibrare.

    - (3) Asigură o **lungime medie a drumului de căutare** practic identică cu cea a unui **arbore perfect echilibrat.**

- În acest context se vor studia următorii operatori definţi în cadrul structurii **arbore echilibrat**:

    1° **Căutarea** unui nod cu o cheie dată.

    2° **Inserţia** unui nod cu o cheie dată.

    3° **Suprimarea** unui nod cu o cheie dată.

- Toţi aceşti operatori necesită un **efort de calcul** de ordinul $O(log_2\ n)$, unde n este numărul nodurilor structurii, chiar în **cel mai defavorabil caz**.

- Acest lucru este o consecinţă directă a teoremei demonstrată de **Adelson-Velskii** şi **Landis**, care garantează că:

- Un arbore echilibrat **nu** va fi niciodată cu mai mult de **45 %** mai înalt decât omologul său perfect echilibrat, **indiferent** de numărul de noduri pe care-l conţine.

- Dacă se notează cu $h_0(n)$ înălţimea unui **arbore echilibrat cu n noduri**, atunci este valabila relaţia [8.5.1.a].

$$\log(n+1) < h_0(n) < 1.4404 * \log(n+2) - 0.328 \qquad [8.5.1.a]$$

- Optimul este atins de arborii echilibraţi având un număr de noduri $n=2^k-1$.

8.5.2. Arbori Fibonacci

- Se consideră următoarea problemă:

  - Se dă o anumită **înălţime** h şi se cere să se construiască **arborele echilibrat AVL** care conţine **numărul minim de noduri** pentru înălţimea dată.

  - Ca şi în cazul arborelui având pe h minim, valoarea dorită a lui h poate fi atinsă numai pentru **anumite** valori specifice ale lui n.

  - Se notează **arborele de înălţime h** cu număr minim de noduri cu A_h.

  - În acest caz, A_0 este arborele vid iar A_1 este un arbore cu un singur nod.

  - Pentru a construi arborele A_h pentru h>1, se va porni de la rădăcină, căreia i se vor ataşa doi subarbori care le rândul lor vor avea un număr minim de noduri. Aceşti arbori sunt de asemenea de tip A.

  - Evident unul din subarbori trebuie să aibă înălţimea h-1, iar celuilalt îi este permisă o înălţime cu 1 mai mică deci (h-2).

  - În figura 8.5.2.a sunt reprezentaţi arbori de înălţime 2, 3 şi 4.

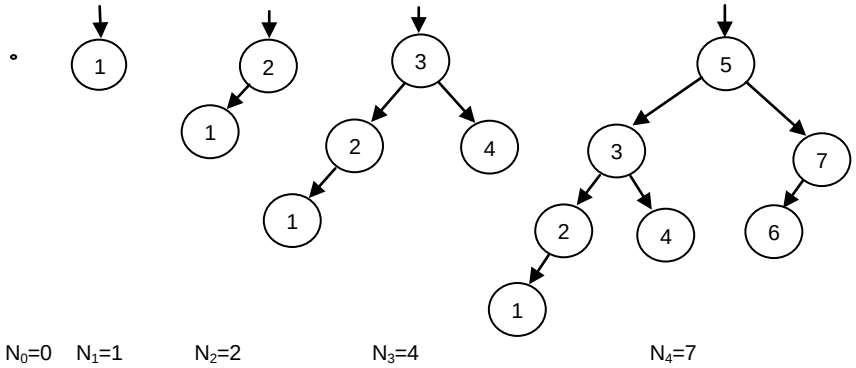


Fig.8.5.2.a. Arbori Fibonacci de înălţime 2, 3 şi 4

- Deoarece principiul lor de alcătuire seamănă foarte mult cu numerele lui Fibonacci, adică fiecare arbore curent se construieşte din cei doi anteriori, ei se numesc **arbori Fibonacci**.

- **Arborii Fibonacci** se definesc **formal** după cum urmează:

    1. Arborele vid A_0 este arborele Fibonacci de înălţime 0.

    2. Arborele cu un singur nod A_1 este arborele Fibonacci de înălţime 1.

    3. Dacă A_{h-1} şi A_{h-2} sunt arbori Fibonacci de înălţime $h-1$ respectiv $h-2$, atunci $A_h=<A_{h-1}, r, A_{h-2}>$ este arborele Fibonacci de înălţime h, având rădacina r.

    4. Nici un alt arbore **nu** este un arbore Fibonacci.

- Numărul de noduri ale lui A_h este definit de următoarea relaţie de recurenţă [8.5.2.a], care justifică de fapt denumirea de **arbori Fibonacci**.

$N_0=0, \quad N_1=1$ [8.5.2.a]

$N_h=N_{h-1}+1+N_{h-2}$

- Numerele N_i sunt numerele de noduri valabile pentru **cazul cel mai defavorabil**, respectiv când se atinge limita superioară a lui h în relaţia [8.5.1.a].

- **Arborii Fibonacci** sunt cazurile cele mai **dezavantajoase** de **arbori AVL**.

8.5.3. Inserţia nodurilor în arbori echilibraţi AVL

- Se dă un **arbore AVL** având rădăcina R şi pe S de înălţime h_S şi pe D de înălţime h_D, pe post de subarbori stâng respectiv drept.

    - Se cere să se insereze un nod nou în acest arbore.

- În cazul **inserţiei** se pot distinge trei cazuri.

    - Se presupune că nodul nou se inserează în **subarborele stâng S**, determinând creşterea cu 1 a înălţimii acestuia.

        - (1) $h_S=h_D$: în urma inserţiei S şi D devin de înălţimi inegale, fără însă a viola criteriul echilibrului.

        - (2) $h_S<h_D$: în urma inserţiei S şi D devin de înălţimi egale, echilibrul fiind îmbunătăţit.

        - (3) $h_S>h_D$: criteriul echilibrului este violat şi arborele trebuie **reechilibrat**.

- Astfel, în arborele echilibrat din figura 8.5.3.a:

    - Nodurile 9 sau 11 pot fi inserate **fără** reechilibrare.

- Inserţia unuia din nodurile `1, 3, 5` sau `7` necesită însă **reechilibrarea** arborelui.

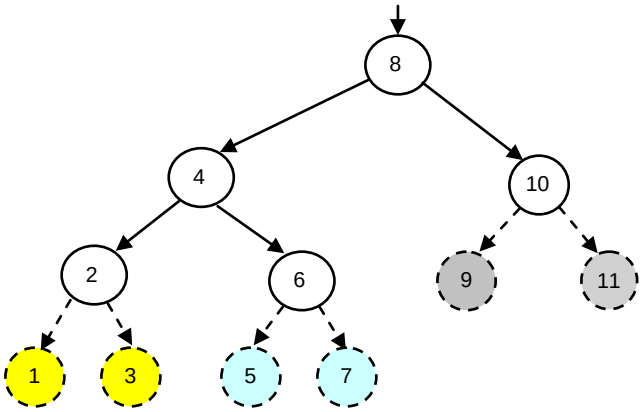


Fig.8.5.3.a. Arbore echilibrat AVL

- O analiză atentă a situaţiilor posibile care rezultă în urma inserţiei evidenţiază faptul că există numai **două configuraţii** care necesită tratamente speciale.

- Celelalte configuraţii pot fi reduse la aceste două situaţii din considerente de **simetrie.**

- **Prima situaţie** se referă la inserţia nodurilor `1` sau `3` în arborele reprezentat cu linie continuă în figura 8.5.3.a.

- Cea de-**a doua sitaţie** se referă la inserţia nodurilor `5` sau `7` în arborele din figura 8.5.3.a.

- Cele două situaţii sunt prezentate în figurile 8.5.3.b şi 8.5.3.c, fiecare în câte trei ipostaze (a), (b) şi (c) care evoluează de la simplu la complicat.

  - Cele două situaţii sunt denumite cazul "**1 Stânga**" respectiv cazul "**2 Stânga**".

  - Ambele cazuri presupun creşterea **subarborelui stâng** S, ca atare reprezintă un **caz stânga**.

    - **Cazul 1 Stînga** presupune creşterea **subarborelui stâng** al **subarborelui stâng** al arborelui în cauză .

    - **Cazul 2 Stânga** presupune creşterea **subarborelui drept** al **subarborelui stâng** al arborelui în cauză.

- Elementele adăugate prin inserţie apar cu linie punctată.

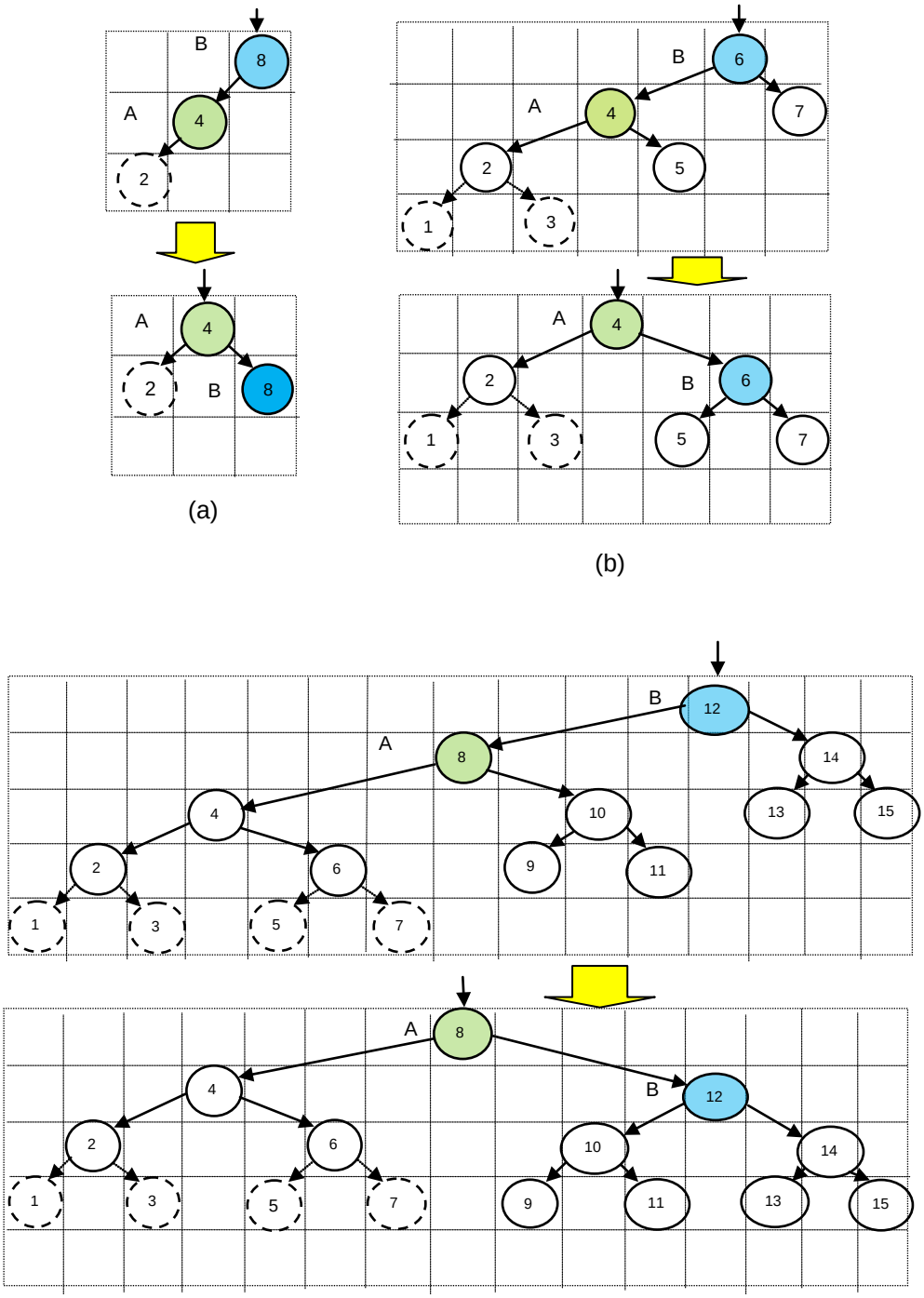
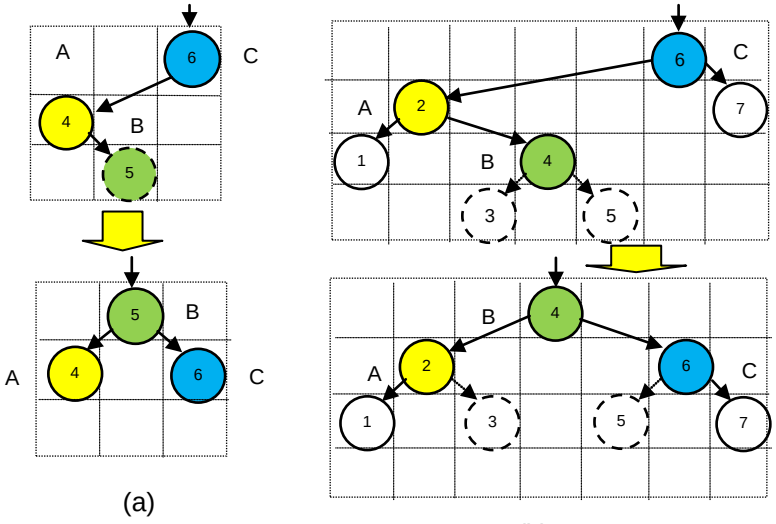


Fig.8.5.3.b. Echilibrarea arborilor AVL. Cazul **1Stânga**



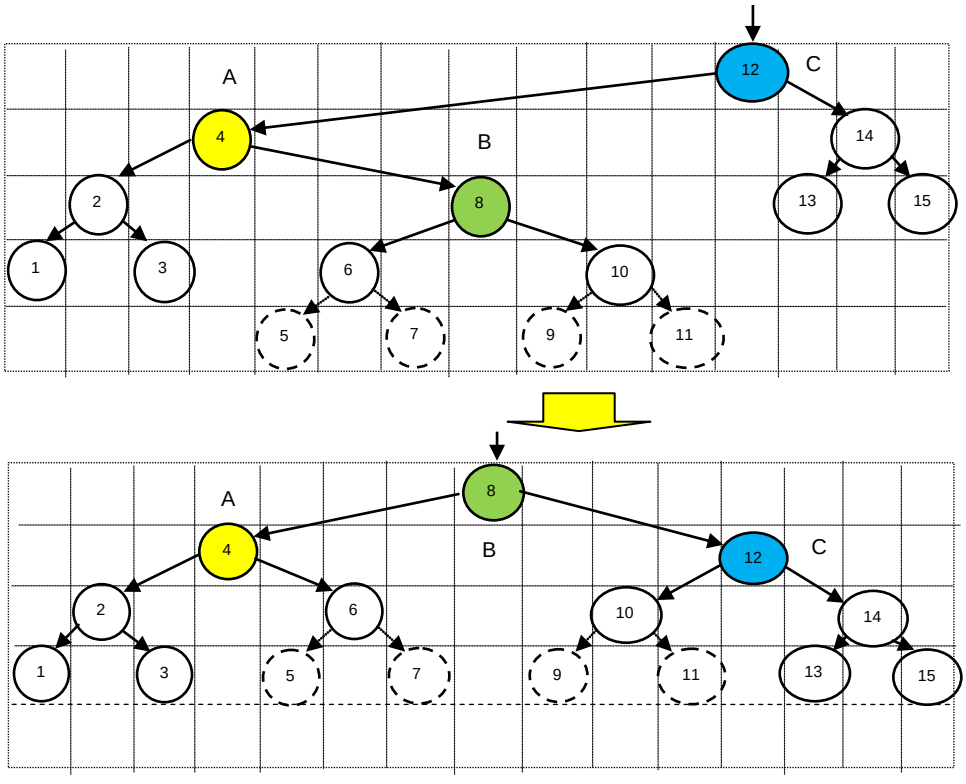


Fig.8.5.3.c. Echilibrarea arborilor AVL. Cazul **2 Stânga**

- Prin **transformări simple**, structurile arbore se reechilibrează.

  - În **cazul 1 Stânga** este vorba despre o rotaţie simplă a două noduri respectiv A şi B.
  - În cazul **2 Stânga** este vorba despre o rotaţie dublă în care sunt implicate trei noduri: A, B şi C.
    - Se subliniază faptul că arborii AVL fiind **arbori ordonaţi**, singurele mişcări permise ale nodurilor sunt cele pe **verticală**.

- **Poziţiile relative** ale proiecţiilor pe orizontală ale nodurilor aparţinând unui arbore AVL, trebuie să rămână **nemodificate**.

- Sinteza acestor cazuri precum şi modul sintetic în care se realizează procesul de echilibrare pentru cazurile pe stânga sunt prezentate în figurile 8.5.3.d şi 8.5.3.e.

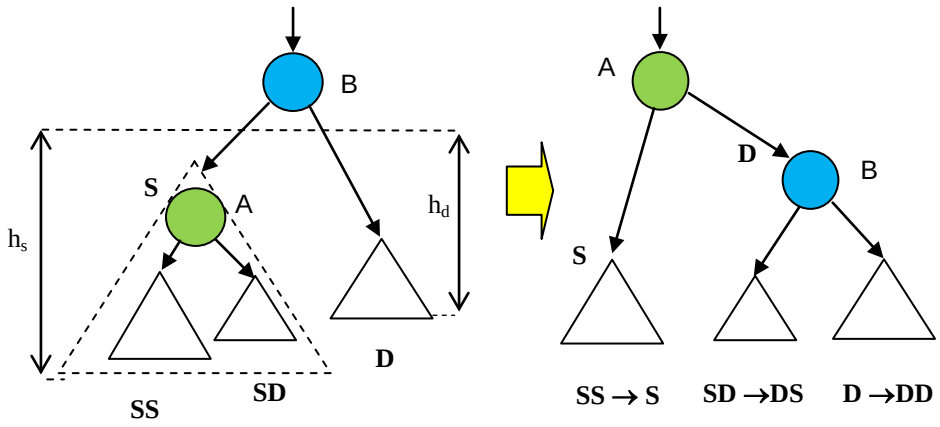


Fig.8.5.3.d. Echilibrarea arborilor AVL**. Cazul 1 Stânga**. Schema generală

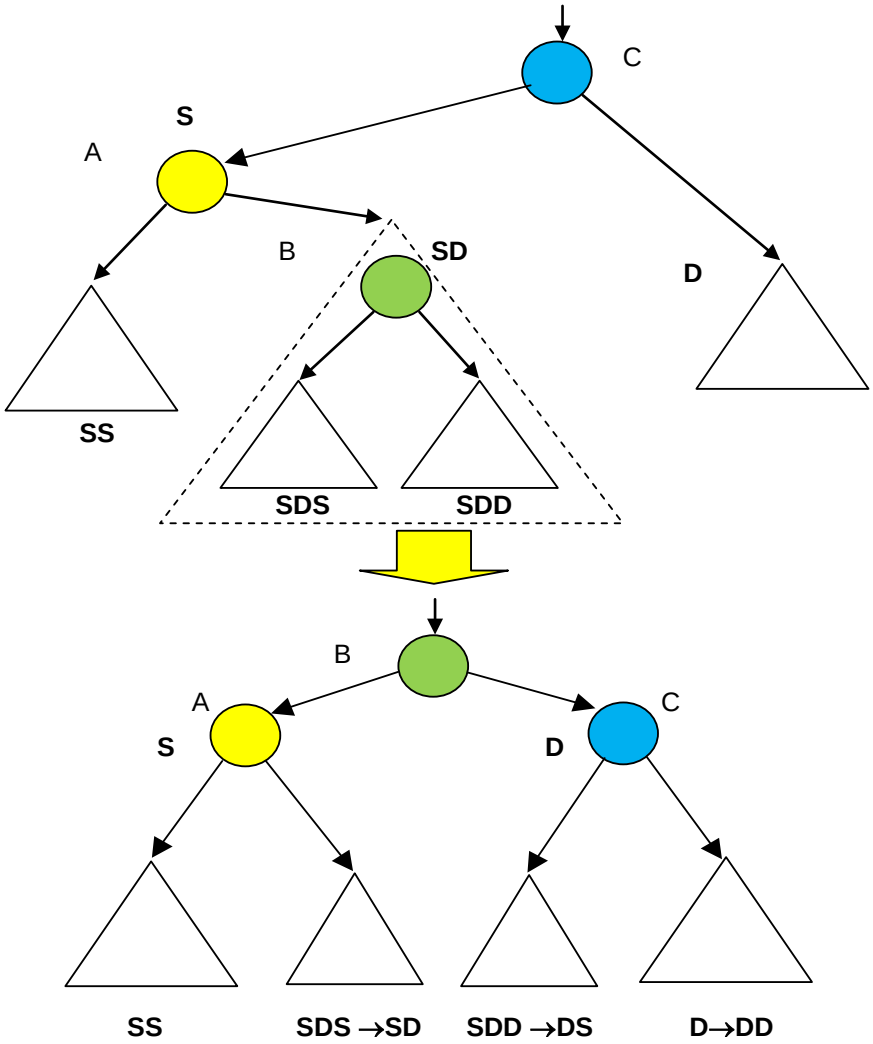


Fig.8.5.3.e. Echilibrarea arborilor AVL. **Cazul 2 Stânga**. Schema generală

- Aceleaşi scheme sintetice de data aceasta pentru cazurile pe **dreapta** apar în figurile 8.5.3.f respectiv 8.5.3.g.

    - Este vorba despre cazurile **1** respectiv **2 Dreapta**.

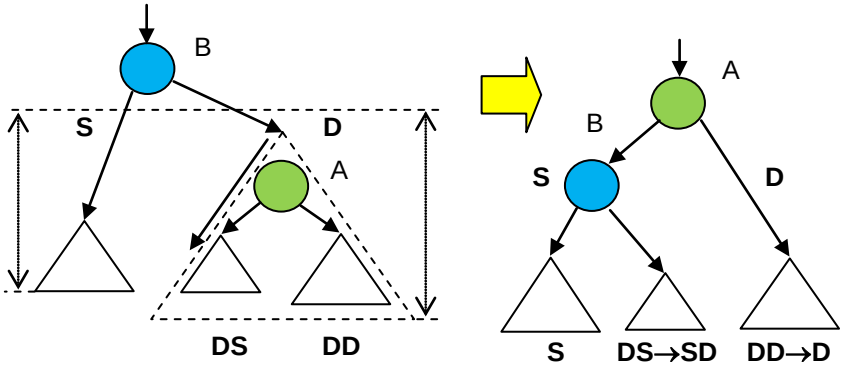


Fig.8.5.3.f. Echilibrarea arborilor AVL. **Cazul 1 Dreapta**. Schema generală

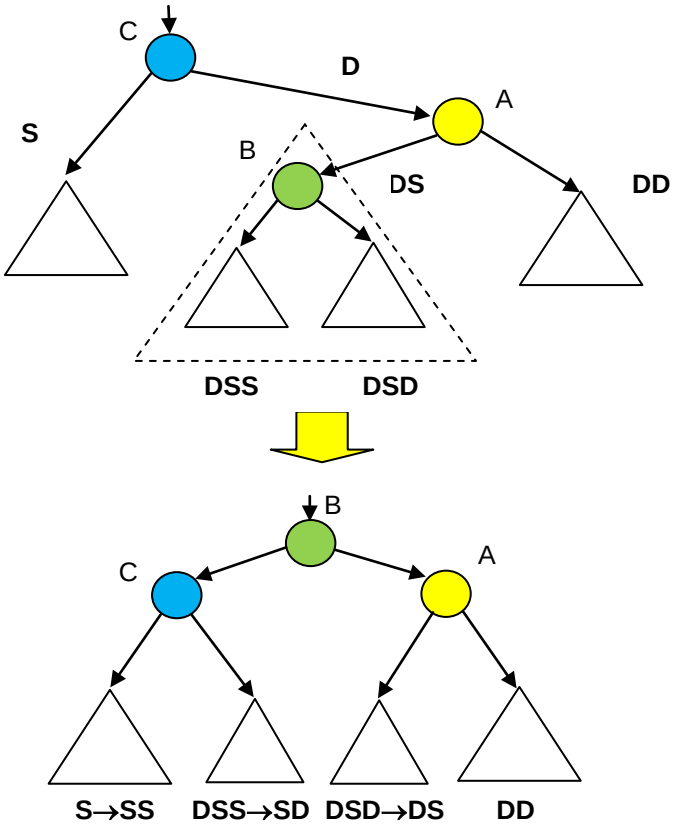


Fig.8.5.3.g. Echilibrarea arborilor AVL. **Cazul 2 Dreapta**. Schema generală

- De această dată a crescut **subarborele drept** al arborelui original şi e necesară reechilibrarea.

- În oglindă cu cazurile pe **stânga** şi aici distingem:

    - **Cazul 1 Dreapta** care presupune creşterea **subarborelui drept** al **subarborelui drept** al arborelui original.

- **Cazul 2 Dreapta** care presupune creşterea **subarborelui stâng** al **subarborelui drept** al arborelui original.

- Şi în acest caz reechilibrarea se rezolvă prin **una** respectiv **două rotaţii** ale nodurilor A şi B, respectiv ale nodurilor A, B şi C.

- **Principial,** procesul de **echilibrare** împreună cu modalitatea efectivă de **restructurare** apar pentru fiecare din cele două cazuri în figurile mai sus precizate.

- Un **algoritm** pentru inserţie şi reechilibrare depinde în **mod critic** de maniera în care este memorată informaţia referitoare la **situaţia echilibrului arborelui.**

- (1) O soluţie posibilă este aceea care exploatează faptul că această informaţie este conţinută în **mod implicit** în structura arborelui.

    - În acest caz factorul de echilibru al unui nod afectat de inserţie, trebuie determinat de fiecare dată prin parcurgerea arborelui, lucru care conduce la o **regie excesivă**.

- (2) O altă soluţie este aceea prin care se atribuie **fiecărui nod** al arborelui un **factor explicit de echilibru.**

    - **Factorul de echilibru** se referă la subarborele a cărui **rădăcină** o constituie nodul în cauză.

    - **Factorul de echilibru** al unui **nod**, va fi interpretat, prin definiţie, ca şi **diferenţa** dintre înălţimea **subarborelui său drept** şi înălţimea **subarborelui său stâng**.

- În acest caz structura unui nod devine [8.5.3.a]:

```
-----------------------------------------------------------------
/*Structura unui nod al unui arbore AVL –varianta C*/

typedef struct Nod_AVL
          {
            int cheie;
            int contor;
            struct nod* stang;                  /*[8.5.3.a]*/
            struct nod* drept;
            int ech; /*ia valorile -1 sau 0 sau +1*/
          } nod;

typedef Nod_AVL* ref_tip_nod_AVL;

-----------------------------------------------------------------
{Structura unui nod al unui arbore AVL – varianta PASCAL}

TYPE TipRef=^TipNod
        TipNod=RECORD
                 cheie:integer;                 [8.5.3.a]
                 contor:integer;
                 stang,drept:TipRef;
                 ech:(-1,0,+1)
               END;
```

- Pornind de la **structura nod** definită în secvenţa [8.5.3.a], **inserţia** unui nod se desfăşoară în trei etape:
  - (1) Se parcurge arborele binar, pentru a verifica dacă nu cumva cheia există deja.
  - (2) Se înserează noul nod şi se iniţializează factorul său de echilibru pe valoarea zero.
  - (3) Se revine pe drumul de căutare şi se verifică factorul de echilibru pentru fiecare nod întâlnit, procedându-se la echilibrare acolo unde este cazul.
- Această metodă necesită unele verificări redundante:
  - Odată echilibrul stabilit, **nu** mai este necesară verificarea factorului de echilibru pentru strămoşii nodului.
- Cu toate acestea se va face totuşi uz de ea, deoarece:
  - (1) Este uşor de înţeles.
  - (2) Se poate implementa printr-o **extindere** a **procedurilor recursive** de **căutare** şi **inserţie** a nodurilor în **arbori binari ordonaţi**, descrise în & 8.3.4.
- Aceste proceduri care includ operaţia de căutare a unui nod, datorită formulării lor **recursive**, asigură în manieră implicită "**revenirea de-a lungul drumului de căutare**".
- **Informaţia** care trebuie transmisă la **revenirea** din fiecare pas este cea referitoare **la modificarea înălţimii subarborelui** în care s-a făcut inserţia.
  - Din acest motiv, în **lista** de **parametri ai procedurii** se introduce parametrul variabil boolean h, a cărui valoare "adevărat" semnifică **creşterea înălţimii subarborelui**.
  - Se presupune că procedura de inserţie revine din **subarborele stâng** la un nod p^ (vezi fig.8.5.3.h), cu indicaţia că **înălţimea subarborelui stâng** a crescut.

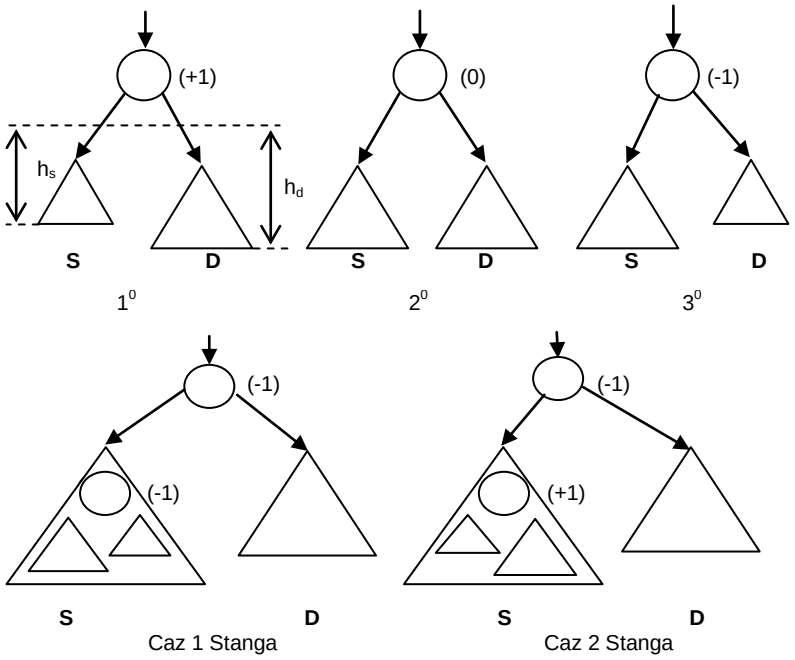


Fig.8.5.3.h. Inserţia în arbori AVL. Cazul Stânga. Schema generală

- Se pot distinge **trei situaţii** referitoare la înălţimea subarborelui **înaintea** respectiv **după** realizarea inserţiei:

    - (1) $h_S<h_D$, deci `p^.ech=+1`; După inserţie factorul de echilibru devine `p^.ech=0,` ca atare inechilibrul anterior referitor la nodul `p` a fost rezolvat.

    - (2) $h_S=h_D$, deci `p^.ech=0`; După inserţie factorul de echilibru devine `p^.ech=-1`, în consecinţă greutatea este acum înclinată spre stânga, dar arborele rămâne echilibrat în sensul AVL.

    - (3) $h_S>h_D$, deci `p^.ech=-1`; ca atare este necesară **reechilibrarea arborelui.**

- În cazul 3^0, **inspectarea** factorului de echilibru al **rădăcinii subarborelui stâng** (`p1^.ech`) conduce la stabilirea cazului **1 Stânga** sau cazul **2 Stânga**.

    - (1) Dacă acest nod are la rândul său înălţimea subarborelui său stâng mai mare ca cea a celui drept, adică factorul de echilibru egal cu `(-1)`, suntem în cazul **1 Stânga.**

    - (2) Dacă factorul de echilibru al acestui nod este egal cu `(+1)` suntem în cazul **2 Stânga** (fig. 8.5.3.h).

    - (3) În această situaţie **nu** poate apare un subarbore stâng a cărui rădăcină are un factor de echilibru nul [Wi76].

- **Operaţia de reechilibrare** constă dintr-o secvenţă de reasignări de pointeri.

    - De fapt pointerii sunt schimbaţi ciclic, rezultând fie o **rotaţie simplă** a două noduri, fie o **rotaţie dublă** a trei noduri implicate.

    - În plus, pe lângă rotirea pointerilor, **factorii de echilibru** respectivi sunt reajustaţi.

- Procedura care realizează acest lucru apare în secvenţa[8.5.3.b]. Principiul de lucru este cel ilustrat în figura 8.5.3.h.

/*Inserţia unui nod într-un arbore echilibrat AVL -varianta pseudocod*/

```
Subprogram InsertEchilibrat(int x, ref_tip_nod_AVL p,
boolean *h)
/*inserează nodul cu cheia x în arborele echilibrat indicat
de p şi returnează h=TRUE dacă arborele îşi modifică
înălţimea. Dacă nodul există deja în arbore îi incrementează
contorul asociat*/

    ref_tip_nod_AVL p1,p2;
    *h=FALSE;
    dacă(p==NULL)  /*cuvântul nu e arbore*/
       |/*se crează şi se inserează un nod nou*/
       |p=aloca_memorie(tip_nod_AVL); /*creare nod nou*/
       |p->cheie=x; p->contor=1;
       |p->stang=NULL; p->drept=NULL; p->ech=0; *h=TRUE;
       |□ /*daca*/
      altfel
        daca(x<p->cheie) /*parcurgere subarbore stâng*/
           |InsertEchilibrat(x,p->stang,&h);
           |daca(*h) /*ramura stângă a crescut în înălţime*/
           |  daca(p->ech==1) /*cazul ech=1*/
           |     |p->ech=0; *h=FALSE;
           |     |revenire;
           |     |□ /*daca*/                    [8.5.3.b]
           |  daca(p->ech==0) /*cazul ech=0*/
           |     |p->ech=-1;
           |     |revenire;
           |     |□ /*daca*/
           |  daca(p->ech==-1)/*cazul ech=-1 reechilibrare*/
           |     |p1=p->stang;
           |     |daca(p1->ech==-1) /*cazul 1 stânga*/
           |     |    |p->stang=p1->drept;
           |     |    |p1->drept=p;
           |     |    |p->ech=0; p=p1;
           |     |    |□ /*daca*/
           |     |  altfel /*cazul 2 stânga*/
           |     |    |p2=p1->drept;
           |     |    |p1->drept=p2->stang;
           |     |    |p2->stang=p1;
           |     |    |p->stang=p2->drept;
           |     |    |p2->drept=p;
           |     |    |daca(p2->ech==-1)
           |     |    |    p->ech=+1;
           |     |    |  altfel
           |     |    |    p->ech=0;
           |     |    |daca(p2->ech==+1)
           |     |    |    p1->ech=-1;
           |     |    |  altfel
           |     |    |    p1->ech=0;
           |     |    |p=p2;
           |     |    |□ /*altfel*/
           |     |p->ech=0; *h=FALSE; revenire;
           |     |□ /*daca*/
           |  □ /*daca*/
           |□ /*daca*/                          [8.5.3.b]
```

```
altfel
   daca(x>p->cheie) /*parcurgere subarbore drept*/
      InsertEchilibrat(x,p->drept,&h);
      daca(*h)/*ramura dreapta a crescut în
                înălţime*/
        daca(p->ech==-1) /*cazul ech=-1*/
          p->ech=0; *h=FALSE
          revenire;
          □ /*daca*/
        daca(p->ech==0) /*cazul ech=-0*/
          p->ech=+1;
          revenire;
          □ /*daca*/
        daca(p->ech==1) /*cazul ech=1*/
          /*reechilibrare dreapta*/
          p1=p->drept;
          daca(p1->ech==+1) THEN
              /*cazul 1 dreapta*/
              p->drept=p1->stang;
              p1->stang=p;
              p->ech=0; p=p1;
              □ /*daca*/
           altfel
              /*cazul 2 dreapta*/
              p2=p1->stang;
              p1->stang=p2->drept;
              p2->drept=p1;
              p->drept=p2->stang;
              p2->stang=p;
              daca(p2->ech==+1)
                  p->ech=-1;
                altfel
                  p->ech=0;
              daca(p2^.ech==-1)
                  p1->ech=+1;
                altfel
                  p1->^.ech=0;
              p=p2;              [8.5.3.b]
              □ /*altfel*/
          p->ech=0; *h=FALSE; revenire;
          □ /*daca*/
        □ /*daca*/
      □ /*daca*/
   altfel
      /*s-a găsit cheia, incrementare contor*/
      p->contor=p->contor+1;
      revenire;
      □ /*altfel*/
/*InsertEchilibrat*/
```

- Procedura **`InsertEchilibrat`** funcţionează după cum urmează:
  - (1) Iniţial se parcurge arborele indicat de referinţa `p` pe stânga respectiv pe dreapta după valoarea cheii `x` care se caută. Parcurgerea se realizează prin apeluri recursive ale procedurii **`InsertEchilibrat`**.
  - (2) Dacă se ajunge la o referinţă `p=NULL` are loc inserţia, cu modificarea lui `h=TRUE` specificând astfel că înălţimea subarborelui a crescut.

- (3) După o astfel de inserţie se revine din apelul recursiv şi se verifică echilibrul nodului curent realizându-se eventual echilibrarea pe **stânga** (dacă se revine din stânga) sau pe **dreapta** (dacă se revine din dreapta).
- (4) Dacă se găseşte o cheie egală cu `x` se incrementează contorul nodului în cauză.
- (5) Cu privire la variabila `h` se fac următoarele precizări:
  - Inserţia îl poziţionează pe `h=TRUE`.
  - Revenirile prin noduri cu factorul de echilibru `0` nu îl modifică pe `h`.
  - Reechilibrarea îl poziţionează pe `h=FALSE`.

- Pentru exemplificare se consideră succesiunea de inserţii într-un arbore AVL precizată în figura 8.5.3.i.

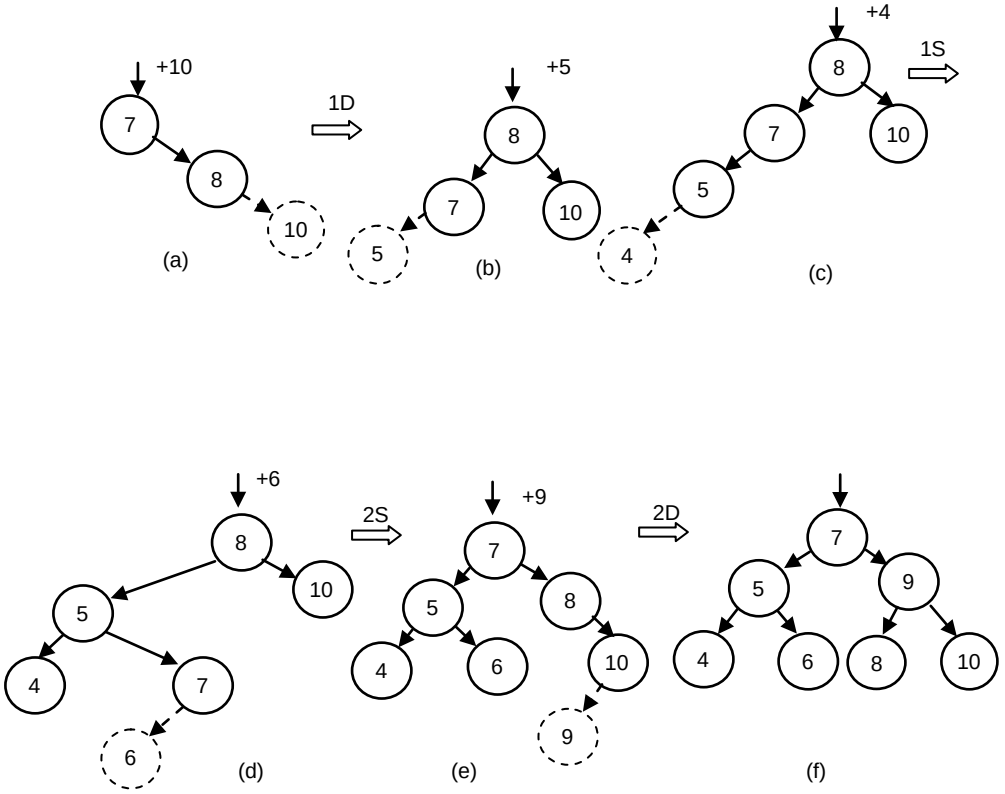


Fig.8.3.5.i. Inserţii succesive într-un arbore echilibrat AVL.

- Se consideră arborele echilibrat AVL (a).
- Inserţia cheii `10` conduce la un arbore dezechilibrat (cazul 1 Dreapta), a cărui echilibrare perfectă se realizează printr-o rotaţie simplă dreapta, fig.8.5.3.i (b).
- Inserţiile nodurilor `5` şi `4` conduc la dezechilibrarea subarborelui cu rădăcina `7`. Echilibrarea sa se realizează printr-o rotaţie simplă (cazul 1 Stânga) (d).
- Inserţia în continuare a cheii `6` produce din nou dezechilibrarea arborelui, a cărui echilibrare se realizează printr-o rotaţie dublă stânga rezultînd arborele (e) (cazul 2 Stânga).

- În sfârşit, inserţia nodului 9 conduce la cazul 2 Dreapta, care necesită în vederea echilibrării arborelui cu rădăcina 8 o rotaţie dublă care conduce la arborele echilibrat AVL (f).

- În legătură cu **performanţele inserţiei într-un arbore echilibrat AVL** se ridică două probleme:
  1. Dacă toate cele n! permutări de n chei apar cu **probabilitate egală**, care este **înălţimea probabilă** a arborelui echilibrat care se construieşte?
  2. Care este **probabilitatea** ca o inserţie să necesite **reechilibrarea** arborelui?
- Analiza matematică a acestui complicat algoritm este încă o problemă parţial rezolvată.
- Teste empirice ale înălţimii arborilor generaţi de algoritmul [8.5.3.b.] conduc la valoarea h=log(n)+c, unde c este o constantă mică (c≈0.25).
- Aceasta înseamnă că în practică, **arborii echilibraţi AVL**, se comportă **la fel de bine** ca şi **arborii perfect echilibraţi**, fiind însă mai uşor de realizat.
- **Testele empirice** sugerează de asemenea că în medie, **reechilibrarea** este necesară aproximativ la fiecare **două inserţii.**
  - Atât rotaţiile simple cât şi cele duble sunt echiprobabile.
- Complexitatea operaţiei de reechilibrare sugerează faptul că arborii echilibraţi trebuie utilizaţi de regulă când operaţiile de căutare a informaţiei sunt mult mai frecvente decât cele de inserare.

8.5.4. Suprimarea nodurilor în arbori echilibraţi AVL

- Şi în cazul arborilor echilibraţi AVL, **suprimarea** este o operaţie mai **complicată** decât inserţia.
- În principiu însă, operaţia de **reechilibrare** rămâne aceeaşi, reducîndu-se la una sau două **rotaţii** la stânga sau la dreapta.
- **Tehnica** care stă la baza suprimării nodurilor în arbori echilibraţi AVL este similară celei utilizate în cazul **arborilor binari ordonaţi** prezentată în &8.3.5.
  - Cazul evident este cel în care, nodul care se suprimă este un **nod terminal** sau are **un singur descendent**.
  - Dacă nodul de suprimat are însă **doi descendenţi**, el va fi înlocuit cu cel mai din dreapta nod al subarborelui său stâng (predecesorul său la parcurgerea în inordine).
- Ca şi în cazul inserţiei, se utilizează **variabila booleeană** h a cărei valoare adevărat semnifică **reducerea înălţimii subarborelui**.
- **Reechilibrarea** se execută **numai** când h este adevărat.
- Variabila h se poziţionează pe adevărat după suprimarea unui nod al structurii, sau dacă reechilibrarea însăşi reduce înălţimea subarborelui.
- Tehnica suprimării nodurilor din arbori echilibraţi AVL este materializată de procedura **SuprimEchilibrat** secvenţa [8.5.4.a]

```
-------------------------------------------------------------
/*Suprimarea unui nod într-un arbore echilibrat AVL -
varianta pseudocod*/
```

```
tip_ref_nod_AVL q; /*variabila globala*/
boolean h; /*variabila globala*/

Subprogram Echilibru1(ref_tip_nod_AVL p, boolean *h)
 /*echilibrează subarborele stâng după o suprimare*/

  ref_tip_nod_AVL p1,p2;
  int e1,e2;

    /*h=adevărat,ramura stânga a devenit mai mică*/
    dacă (p->ech==-1)
       p->ech=0;  /*cazul ech=-1*/
       revenire;
       □ /*dacă*/
    dacă (p->ech==0) /*cazul ech=0*/
       p->ech=+1; *h=FALSE;
       revenire;
       □ /*dacă*/
    dacă (p->ech==+1) /*reechilibrare subarbore stang*/
       p1=p->drept; e1=p1->ech;
       dacă (e1>=0)
          /*cazul 1 dreapta - rotaţie simplă*/
          p->drept=p1->stang; p1->stang=p;
          dacă (e1==0)
             p->ech=+1; p1->ech=-1;
             *h=FALSE;
             □ /*dacă*/
            altfel
             p->ech=0;
             p1->ech=0;
             □ /*altfel*/                          [8.5.4.a]
          p=p1;
          □ /*dacă*/
        altfel
          /*cazul 2 dreapta – rotaţie dublă*/
          p2=p1->stang; e2=p2->ech;
          p1->stang=p2^.drept; p2->drept=p1;
          p->drept=p2->stang;
          p2->stang=p;
          dacă(e2==+1)
              p->ech=-1;
            altfel
              p->ech=0;
          dacă(e2==-1)
              p1->ech=+1;
            altfel
              p1->ech=0;
          p=p2; p2->ech=0;
          □ /*altfel*/
       revenire; /*revenire cazul reeechilibrare stânga*/
       □ /*dacă*/
 /*Echilibru1*/

 Subprogram Echilibru2(tip_ref_nod_AVL p, boolean *h);
 /*echilibrează subarborele drept după o suprimare*/

  tip_ref_nod_AVL p1,p2;
  int e1,e2;
```

```
   /*h=adevarat,ramura dreapta a devenit mai mică*/
   dacă(p->ech==+1)
      p->ech=0;  /*cazul ech=+1*/
      revenire;
      □ /*dacă*/
   dacă (p->ech==0) /*cazul ech=0*/
      p->ech=-1; *h=FALSE
      revenire;
      □ /*dacă*/
   dacă (p->ech==-1) /*reechilibrare subarbore drept*/
      p1=p->stang; e1=p1^.ech;
      dacă(e1<=0)
         /*cazul 1 stânga – rotaţie simplă*/
         p->stang=p1->drept; p1->drept=p;
         dacă(e1==0)
            p->ech=-1; p1->ech=+1;
            *h=FALSE;
            □ /*dacă*/
           altfel
            p->ech=0;
            p1->ech=0;
            □ /*altfel*/
         p=p1
         □ /*dacă*/
        altfel
         /*cazul 2 stânga - rotaţie dublă*/
         p2=p1->drept; e2=p2->ech;
         p1->drept=p2->stang; p2->stang=p1;
         p->stang=p2->drept;
         p2->drept=p;
         dacă(e2==-1)
            p->ech=+1;
           altfel
            p->ech=0;
         dacă(e2==+1)
            p1->ech=-1;
           altfel
            p1->ech=0;
         p=p2; p2->ech=0;
         □ {altfel}
      revenire; /*revenire cazul reeechilibrare dreapta*/
      □ /*dacă*/
  /*Echilibru2*/

Subprogram Suprima(tip_ref_nod_AVL r, boolean *h)
/*suprimă cel mai din dreapta nod al arborelui indicat de r
 şi mută conţinutul său în nodul q*/
    *h=FALSE;
    dacă(r->drept!=NULL)
        Suprima(r->drept,&h);
        dacă(*h)  Echilibru2(r,&h);
        □ /*dacă*/
      altfel                                        [8.5.4.a]
        q->cheie=r->cheie;
        q->contor=r->contor;
        r=r->stang; *h=TRUE;
        □ /*altfel*/
```

```
      revenire;
  /*Suprima*/


Subprogram SuprimaEchilibrat(int x, tip_ref_nod_AVL p,
boolean *h)
/*caută şi suprimă nodul cu cheia x din arborele referit de
p. Dacă arborele îşi modifică înălţimea se returnează
h=TRUE*/

    daca(p==NULL)
       |*scrie('cheia nu e în arbore'); *h=FALSE;
       |□ /*dacă*/
      altfel
        daca(x<p->cheie)
           |SuprimaEchilibrat(x,p->stang,&h);
           |daca(*h)  Echilibru1(p,&h);/*echilibrare stânga*/
           |□ /*dacă*/
          altfel
            daca(x>p->cheie)
               |SuprimaEchilibrat(x,p->drept,&h);
               |daca(*h) Echilibru2(p,&h); /*echilibrare
               |                            dreapta*/
               |□ /*dacă*/
              altfel
               |/*nod gasit, suprimare nod p^*/
               |q=p;
               |daca(q->drept==NULL)          /*[8.5.4.a]*/
               |   |p=q->stang; *h=TRUE;
               |   |□ /*dacă*/
               |  altfel
               |    daca(q->stang==NULL)
               |       |p=q->drept; *h=TRUE;
               |       |□ /*dacă*/
               |      altfel  /*suprimare predecesor*/
               |        |Suprima(q^.stang,&h);
               |        |daca(*h) Echilibru1(p,&h);
               |        |□ /*altfel*/
               |/*elibereaza_memoria(q)*/
               |□ /*altfel*/
    revenire;
  /*SuprimaEchilibrat*/
```

- În cadrul procedurii **SuprimEchilibrat** se definesc trei proceduri:
  - (1) **Echilibru1** care se aplică când **subarborele stâng** s-a redus din înălţime.
  - (2) **Echilibru2** care se aplică când **subarborele drept** s-a redus din înălţime.
  - (3) **Suprima** – are rolul procedurii **Supred** la arbori binari ordonaţi:
    - (3.1) Găseşte şi înlocuieşte nodul de suprimat cu predecesorul său.
    - (3.2) Suprimă predecesorul.

- (3.3) În plus procedura **`Suprima`** realizează eventualele reechilibrări la revenirea recursivă pe drumul parcurs în arbore, apelând când este cazul, procedura **`Echilibru2`** (revine din dreapta întotdeauna).

- Mersul procedurii **`SuprimEchilibrat`** este normal:
  - (1) Se parcurge recursiv arborele AVL pentru căutarea cheii de suprimat, prin apeluri ale procedurii **`SuprimaEchilibrat`** pe stânga sau pe dreapta după cum cheia care se caută e mai mică respectiv mai mare decât cea a nodului curent.
  - (2) Când se găseşte cheia ea se suprimă exact ca şi la arborii binari ordonaţi:
    - Cazul 1 fiu: se rezolvă prin suprimare directă.
    - Cazul 2 fii: se apelează procedura **`Suprima`** descrisă mai sus.
  - (3) Este important de reamintit faptul că după fiecare revenire dintr-un apel recursiv se verifică valoarea lui `h` şi dacă este necesar se apelează procedura corespunzătoare de echilibrare.
- Modul de lucru al procedurii, este prezentat în figura 8.5.4.a.

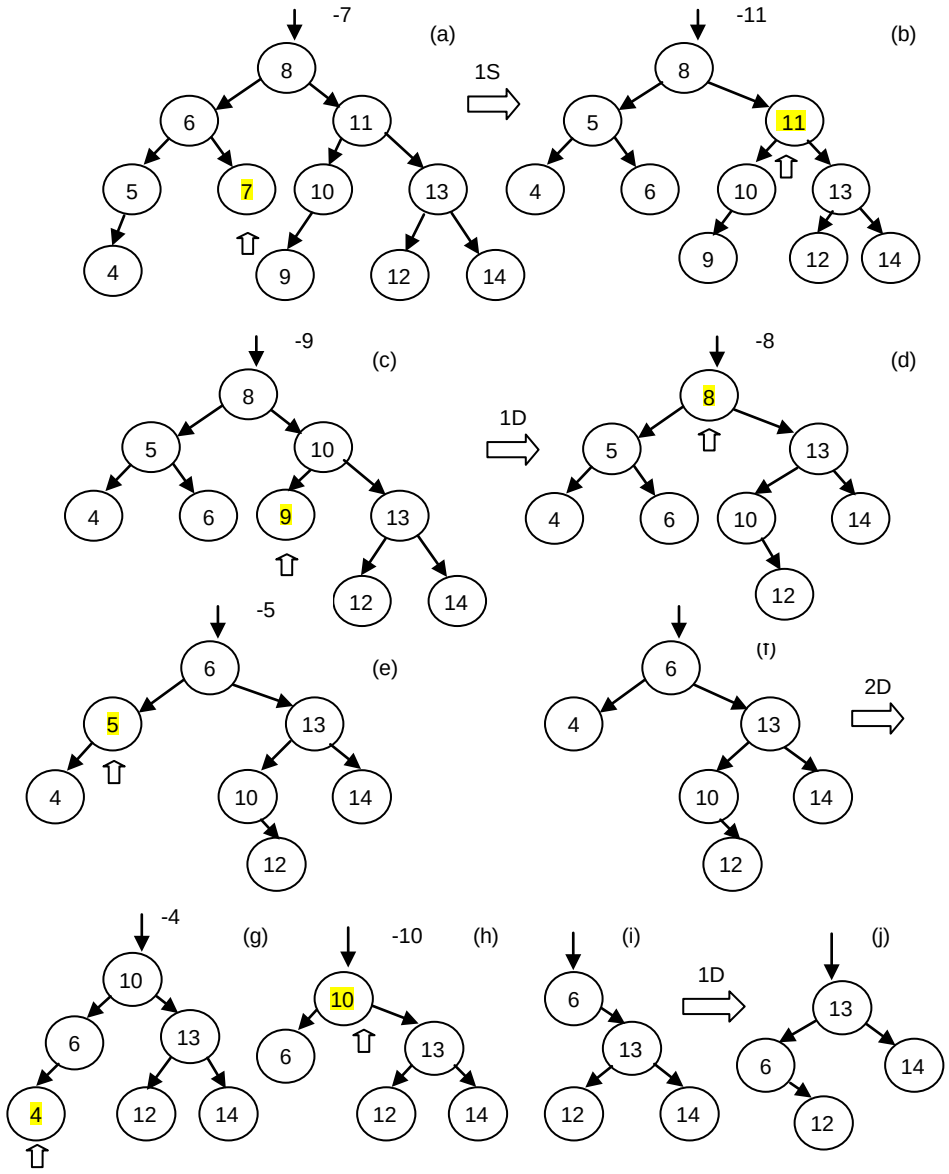


Fig.8.5.4.a. Suprimări succesive în arbori echilibraţi AVL

- Dându-se arborele binar echilibrat (a), se suprimă în mod succesiv nodurile având cheile `7,11,9,8,5,4` şi `10,` rezultând arborii (b)...(j).
    - Suprimarea cheii `7` este simplă însă conduce la subarborele dezechilibrat cu rădăcina `6`. Reechilibrarea acestuia presupune o rotaţie simplă (cazul 1 stânga).
    - Suprimarea nodului `11` nu ridică probleme.
    - Reechilibrarea devine din nou necesară după suprimarea nodului `9`; de data aceasta, subarborele având rădăcina `10`, este reechilibrat printr-o rotaţie simplă dreapta (cazul 1 dreapta).
    - Suprimarea cheii `8` este imediată.
    - Deşi nodul `5` are un singur descendent, suprimarea sa presupune o reechilibrare mai complicată bazată pe o dublă rotaţie (cazul 2 dreapta).
    - Ultimul caz, cel al suprimării nodului cu cheia `10` presupune înainte de reechilibrare, înlocuirea acestuia cu cel mai din dreapta element al arborelui său stâng (nodul cu cheia `6`).
- În cazul **arborilor binari echilibraţi**, suprimarea unui nod se realizează în cel mai defavorabil caz cu performaţa O*(log_2 n)*.
- Diferenţa esenţială dintre **inserţie** şi **suprimare** în cazul arborilor echilibraţi AVL este următoarea:
    - În urma unei **inserţii**, reechilibrarea se realizează **o singură dată** prin una sau două rotaţii (a două sau trei noduri).
    - În cel mai defavorabil caz, **suprimarea** poate necesita o **reechilibrare** a **fiecărui nod** situat pe drumul de căutare.
        - Spre exemplu, în cazul arborelui **Fibonacci**, suprimarea nodului său cel mai din dreapta, necesită numărul maxim de rotaţii, acesta fiind cazul cel mai **defavorabil** de suprimare dintr-un arbore echilibrat.
- În realitate, testele experimentale indică faptul suprinzător că:
    - (1) În cazul **inserţiei** reechilibrarea devine necesară aproximativ la fiecare **a doua** inserţie.
    - (2) În cazul **suprimării** reechilibrarea devine necesară aproximativ la fiecare a **5-a** suprimare.