

ARBORI BINARI DE CĂUTARE

(BINARY SEARCH TREE)

- *Arborii binari de căutare* (ABC) sunt structuri de date des folosite în implementarea containerelor care conțin elemente de tip *TComparable* (sau identificate prin chei de tip *TComparable*) și care suportă următoarele operații:
 1. căutare;
 2. adăugare;
 3. ștergere;
 4. determinare maxim, minim, predecesor, succesor.
- ABC sunt SD care se folosesc pentru implementare:
 - dicționar, dicționar ordonat
 - * **TreeMap** în Java (folosește ABC echilibrat - arbore roșu-negru)
 - * **map** din STL folosește ABC echilibrat ca implementare.
 - C++ 11 - **unordered_map** (tabelă de dispersie)
 - coadă cu priorități;
 - listă (**TreeList** în Java; folosește ABC echilibrat);
 - colecție, mulțime (**TreeSet** în Java; folosește ABC echilibrat - arbore roșu-negru);

Observație: Presupunem în cele ce urmează (fără a reduce generalitatea):

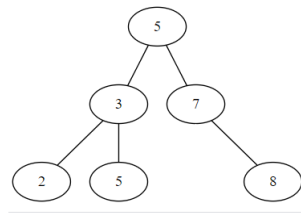
1. *Elementele* sunt identificate printr-o **cheie**.
2. *Cheia* elementului este de tip *TComparable*.
3. Pp. relația “ \leq ” între chei (se poate ușor generaliza la o relație de ordine oarecare).

Definiție 1 *Un ABC este un AB care satisface următoarea **proprietate** (proprietatea ABC):*

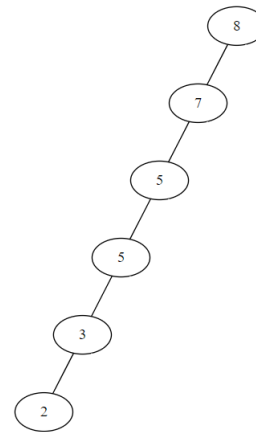
- *dacă x este un nod al ABC, atunci:*
 - $\forall y$ un nod din subarborele stâng al lui x , are loc inegalitatea $cheie(y) \leq cheie(x)$ ($cheie(y) \mathcal{R} cheie(x)$).
 - $\forall y$ un nod din subarborele drept al lui x , are loc inegalitatea $cheie(x) < cheie(y)$ ($\neg(cheie(y) \mathcal{R} cheie(x))$).

Exemplu. Presupunem că în container avem cheile 2 3 5 5 7 8 și relația $\mathcal{R} = \leq$. În Figura 1 sunt indicați 2 ABC care conțin aceste chei.

- operațiile de bază pe ABC consumă timp $O(\text{înălțimea arborelui})$.



(a) ABC echilibrat.



(b) ABC degenerat (lanț).

- dacă ABC este plin \Rightarrow înălțimea este $\theta(\log_2 n)$ - caz (a);
- dacă ABC este degenerat (lanț liniar) \Rightarrow înălțimea este $\theta(n)$ - caz (b).
- forma unui ABC este importantă
 - * operațiile vor avea complexitate timp $O(h)$, h fiind înălțimea arborelui

Proprietate. Traversarea în inordine a unui ABC furnizează cheile în ordine în raport cu relația de ordine (ordine crescătoare dacă $R = \leq$).

- în exemplele (a) și (b) de mai sus, parcurgerea în inordine este 2 3 5 5 7 8

Reprezentarea ABC

Un ABC (ca și un AB) poate fi reprezentat:

1. secvențial;
 2. înlănțuit
 - reprezentarea înlănțuirilor folosind alocare dinamică (pointeri);
 - reprezentarea înlănțuirilor folosind alocare statică (tablouri).
- pentru implementarea operațiilor, pp. în cele ce urmează
 - reprezentare înlănțuită folosind alocare dinamică.
 - $\mathcal{R} = \leq$
 - notăm cu h înălțimea arborelui.
 - notăm cu n numărul de noduri din arbore.

REPREZENTARE

- într-un nod memorăm informația utilă și adresa celor doi descendenți (stâng și drept)

- putem memora (pentru a eficientiza anumite operații) și alte informații
 - adresa părintelui
 - adâncimea, înălțimea nodului

Nod

e: TElement //informația utilă nodului
 st: \uparrow Nod //adresa la care e memorat descendentul stâng
 dr: \uparrow Nod //adresa la care e memorat descendentul drept

Container

rad: \uparrow Nod //adresa rădăcinii ABC

CĂUTARE - pp. chei distincte.

- returnează subarborele a cărui rădăcină conține cheia căutată.

Varianta recursivă.

Modelul matematic recursiv

- arborele binar de căutare îl vom referi sub forma $abc(r, s, d)$
 - r - e informația din rădăcină
 - s - e subarborele stâng
 - d - e subarborele drept

$$cauta_rec(abc(r, s, d), e) = \begin{cases} \emptyset & abc(r, s, d) = \emptyset \\ abc(r, s, d) & r.c = e.c \\ cauta_rec(s, e) & e.c \leq r.c \\ cauta_rec(d, e) & altfel \end{cases}$$

Functia $cauta(abc, e)$

{complexitate timp: $O(h)$ }

pre: abc este un **container** reprezentat folosind un arbore binar de căutare

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui e

$cauta \leftarrow cauta_rec(abc.rad, e)$

SfFunctia

Functia $cauta_rec(p, e)$

{complexitate timp: $O(h)$ }

pre: p este adresa unui nod; $p : \uparrow$ Nod - rădăcina unui subarbore

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui e în subarborele de rădăcină p

{dacă s-a ajuns la subarbore vid sau nodul este cel căutat}

Dacă $p = NIL \vee [p].e.c = e.c$ atunci

```

    cauta_rec ← p
altfel
    Daca  $e.c < [p].e.c$  atunci
        {se caută în subarborele stâng}
        cauta_rec ← cauta_rec([p].st, e)
    altfel
        {se caută în subarborele drept}
        cauta_rec ← cauta_rec([p].dr, e)
    SfDaca
SfDaca
SfFunctia

```

Varianta iterativă.

Functia $cauta(abc, e)$
 {complexitate timp: $O(h)$ }

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui e în arborele abc

```

p ← abc.rad
CatTimp  $p \neq NIL \wedge [p].e.c \neq e.c$  executa
    Daca  $e.c < [p].e.c$  atunci
        {se caută în subarborele stâng}
        p ← [p].st
    altfel
        {se caută în subarborele drept}
        p ← [p].dr
    SfDaca
SfCatTimp
cauta ← p
SfFunctia

```

ADĂUGARE

- returnează arborele rezultat prin inserarea unui element într-un arbore

Modelul matematic recursiv

$$adauga_rec(abc(r, s, d), e) = \begin{cases} abc(e, \emptyset, \emptyset) & abc(r, s, d) = \emptyset \\ abc(r, adauga_rec(s, e), d) & e.c \leq r.c \\ abc(r, s, adauga_rec(d, e)) & altfel \end{cases}$$

Functia $creazaNod(e)$
 {complexitate timp: $\theta(1)$ }

pre: e este de tip $TElement$

post: se returnează pointer spre un nod care conține elementul e
 {se alocă un spațiu de memorare pentru un nod; $p : \uparrow Nod$ }

```

aloca(p)
{se completează componentele nodului}
[p].e ← e

```

```


[p].st  $\leftarrow$  NIL  

[p].dr  $\leftarrow$  NIL  

creeazaNod  $\leftarrow$  p  

SfFunctia


```

```


Subalgoritm adauga(abc, e)  

{complexitate timp:  $O(h)$ }  

pre: abc este un container reprezentat folosind un arbore binar de căutare  

post: abc' este containerul în care a fost adăugat e  

abc.rad  $\leftarrow$  adauga_rec(abc.rad, e)  

SfSubalgoritm


```

```


Functia adauga_rec(p, e)  

{complexitate timp:  $O(h)$ }  

pre: p este adresa unui nod; p : $\uparrow$  Nod - rădăcina unui subarbore  

post: se adaugă e în subarborele de rădăcină p și se returnează rădăcină noului subarbore  

{dacă s-a ajuns la subarbore vid se adaugă}  

Daca p = NIL atunci  

p  $\leftarrow$  creeazaNod(e)  

altfel  

Daca e.c  $\leq$  [p].e.c atunci  

{se adaugă în subarborele stâng}  

[p].st  $\leftarrow$  adauga_rec([p].st, e)  

altfel  

{se adaugă în subarborele drept}  

[p].dr  $\leftarrow$  adauga_rec([p].dr, e)  

SfDaca  

SfDaca  

{se returnează rădăcina subarborelui}  

adauga_rec  $\leftarrow$  p  

SfFunctia


```

MINIM

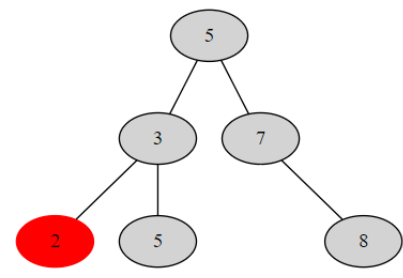


Figura 2: Minim.

```


Functia minim(p)  

{complexitate timp:  $O(h)$ }  

pre: p este adresa unui nod; p : $\uparrow$  Nod; p  $\neq$  NIL  

post: se returnează adresa nodului având cheia minimă din subarborele de rădăcină p  

CatTimp [p].st  $\neq$  NIL executa


```

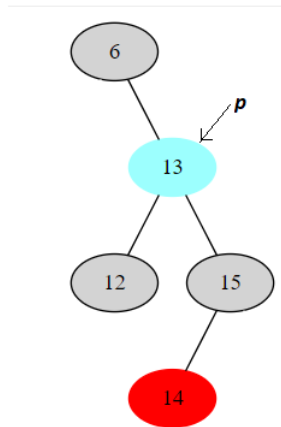
```

     $p \leftarrow [p].st$ 
    SfCatTimp
    minim  $\leftarrow p$ 
    SfFunctia

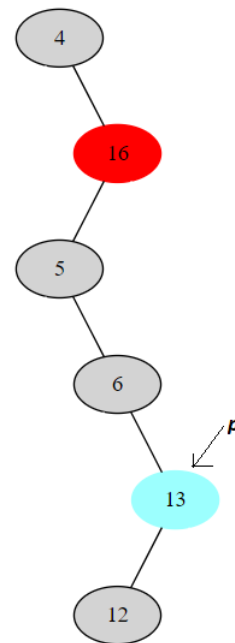
```

SUCCESSOR

- cheia din container imediat mai mare (dacă $\mathcal{R} = \leq$) decât o cheia dintr-un nod p dat
- de exemplu, dacă am avea cheile 2 6 3 4 2 3 1, atunci succesul cheii 4 este 6.



(a) Nodul p are descendent drept. Succesorul e marcat cu **roșu**.



(b) Nodul p nu are descendent drept. Succesorul e marcat cu **roșu**.

Funcția $\text{succesor}(p)$

pre: p este adresa unui nod; $p : \uparrow \text{Nod}$; $p \neq \text{NIL}$

post: se returnează adresa nodului având cheia imediat mai mare decât cheia din p

Dacă $[p].dr \neq \text{NIL}$ atunci

{există subarbore drept al lui p }

$\text{succesor} \leftarrow \text{minim}([p].dr)$

altfel

$\text{prec} \leftarrow \text{parinte}(p)$

CatTimp $\text{prec} \neq \text{NIL} \wedge p = [\text{prec}].dr$ executa

$p \leftarrow \text{prec}$

$\text{prec} \leftarrow \text{parinte}(p)$

SfCatTimp

$\text{succesor} \leftarrow \text{prec}$

SfDaca

SfFunctia

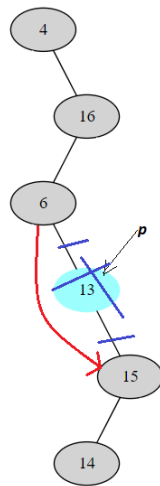
Observație

- dacă un nod memorează adresa părintelui, atunci complexitatea operației este $O(h)$, în caz contrar este $O(h^2)$.

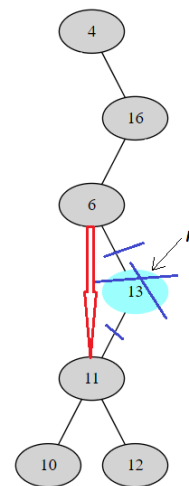
ȘTERGERE

- pp. chei distincte
- se șterge nodul având cheia egală cu cea a unui element dat
- se returnează arborele rezultat în urma ștergerii

Sunt trei cazuri la ștergere, indicate mai jos.



(a) Nodul p nu are descendent stâng.



(b) Nodul p nu are descendent drept.

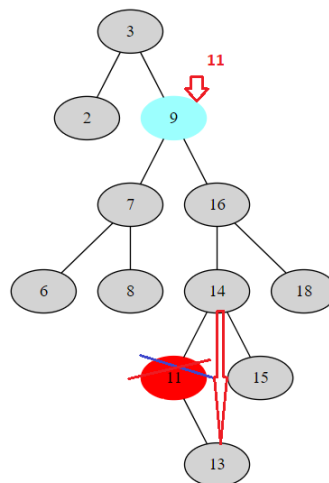


Figura 5: Nodul p are și descendent stâng și drept.

Modelul matematic recursiv

$$sterge_rec(abc(r, s, d), e) = \begin{cases} \emptyset & abc(r, s, d) = \emptyset \\ abc(r, sterge_rec(s, e), d) & e.c < r.c \\ abc(r, s, sterge_rec(d, e)) & e.c > r.c \\ d & s = \emptyset \\ s & d = \emptyset \\ abc(minim(d), s, sterge_rec(d, minim(d))) & altfel \end{cases}$$

Subalgoritm **sterge**(*abc*, *e*)

{complexitate timp: $O(h)$ }

pre: *abc* este un **container** reprezentat folosind un arbore binar de căutare

post: *abc'* este containerul din care a fost şters *e*

abc.rad \leftarrow **sterge_rec**(*abc.rad*, *e*)

SfSubalgoritm

Funcția **sterge_rec**(*p*, *e*)

{complexitate timp: $O(h)$ }

pre: *p* este adresa unui nod; *p* : \uparrow *Nod* - rădăcina unui subarbore

post: se şterge nodul având cheia egală cu cheia lui *e* din subarboarele de rădăcină *p* şi se returnează rădăcină noului subarbore

{dacă s-a ajuns la subarbore vid}

Dacă *p*=NIL atunci

sterge_rec \leftarrow NIL

altfel

Dacă *e.c* < [*p*].*e.c* atunci

{se şterge din subarboarele stâng}

[*p*].*st* \leftarrow **sterge_rec**([*p*].*st*, *e*)

sterge_rec \leftarrow *p*

altfel

Dacă *e.c* > [*p*].*e.c* atunci

{se şterge din subarboarele drept}

[*p*].*dr* \leftarrow **sterge_rec**([*p*].*dr*, *e*)

sterge_rec \leftarrow *p*

altfel

{am ajuns la nodul care trebuie şters}

Dacă [*p*].*st* \neq NIL \wedge [*p*].*dr* \neq NIL atunci

{nodul are şi subarbore stâng şi subarbore drept}

temp \leftarrow **minim**([*p*].*dr*)

{se mută cheia minimă în *p*}

[*p*].*e* \leftarrow [*temp*].*e*

{se şterge nodul cu cheia minimă din subarboarele drept}

[*p*].*dr* \leftarrow **sterge_rec**([*p*].*dr*, [*p*].*e*)

sterge_rec \leftarrow *p*

altfel

temp \leftarrow *p*

Dacă [*p*].*st* = NIL atunci

{nu există subarbore stâng}

repl \leftarrow [*p*].*dr*

altfel


```

        {nu există subarbore drept, [p].dr=NIL}
        repl ← [p].st
    SfDaca
    {dealocă spațiul de memorare pentru nodul care trebuie șters}
    dealoca(temp)
    sterge_rec ← repl
SfDaca
SfDaca
SfDaca
SfDaca
SfDaca
SfFunctia

```

În directorul asociat Cursului 12 (pe pagina cursului) găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** cu elemente de tip comparabil, reprezentarea este sub forma unui ABC reprezentat înlănțuit, cu alocare dinamică a nodurilor). Iteratorul (în inordine) nu este implementat.

Probleme

1. Scrieți o operație nerecursivă care determină părintele unui nod p dintr-un ABC.
2. Scrieți varianta iterativă pentru operația de adăugare într-un ABC.
3. Presupunând că dorim ca fiecare nod din arbore să memoreze următoarele: informația utilă, referință către subarborele stâng, referință către subarborele drept, referință către părinte. Folosind reprezentarea înlănțuită cu alocare dinamică a nodurilor, scrieți operația de adăugare în ABC (varianta iterativă, varianta recursivă).
4. Presupunând ca elementele sunt de forma (cheie, valoare) și relația de ordine între chei este " \leq ", scrieți operația **MAXIM** care determină elementul din ABC având cea mai mare cheie.
5. Presupunând ca elementele sunt de forma (cheie, valoare), relația de ordine între chei este " \leq ", și arborele este reprezentat înlănțuit cu alocare dinamică a nodurilor, scrieți operația **PREDECESOR** care pentru un nod p dintr-un ABC determină elementul având cea mai mare cheie mai mică decât cheia lui p .
6. Implementați operațiile pe ABC generalizând relația " \leq " de la proprietatea unui ABC la o relație de ordine oarecare R .