

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 $cheie(x) < cheie(y)$ ($\neg(cheie(y) \mathcal{R} cheie(x))$).
- operațiile de bază pe ABC consumă timp $O(\text{înălțimea arborelui})$.
 - dacă ABC este plin \Rightarrow înălțimea este $\theta(\log_2 n)$;

- dacă ABC este degenerat (lanț liniar) \Rightarrow înălțimea este $\theta(n)$.

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

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.
 - notăm cu h înălțimea arborelui.
 - notăm cu n numărul de noduri din arbore.

CĂUTARE - pp. chei distincte.

Varianta recursivă.

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 subarboarele 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` \leftarrow *p*

altfel

Dacă *e.c* < [*p*].*e.c* atunci

{se caută în subarboarele stâng}

`cauta_rec` \leftarrow `cauta_rec([p].st, e)`

altfel

{se caută în subarboarele drept}

`cauta_rec` \leftarrow `cauta_rec([p].dr, e)`

SfDaca

SfDaca

SfFunctia

Varianta iterativă.

```
Functia caută(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 ≠ NIL ∧ [p].e.c ≠ 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
    caută ← p
SfFunctia
```

ADĂUGARE

```
Functia creeazaNod(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 : ↑ Nod}
    aloca(p)
    {se completează componentele nodului}
    [p].e ← e
    [p].st ← NIL
    [p].dr ← NIL
    creeazaNod ← 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 ← 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 subarboarele 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 subarboarele stâng}
 $[p].st \leftarrow adauga_rec([p].st, e)$

altfel
 {se adaugă în subarboarele drept}
 $[p].dr \leftarrow adauga_rec([p].dr, e)$

SfDaca
 SfDaca
 {se returnează rădăcina subarborelui}
 $adauga_rec \leftarrow p$

SfFunctia

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 subarboarele de rădăcină p

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

SfCatTimp
 $minim \leftarrow p$

SfFunctia

Functia `succesor(p)`

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

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

Daca $[p].dr \neq NIL$ atunci
 {există subarbore drept al lui p }
 $succesor \leftarrow minim([p].dr)$

altfel
 $prec \leftarrow parinte(p)$
 CatTimp $prec \neq NIL \wedge p = [prec].dr$ executa
 $p \leftarrow prec$
 $prec \leftarrow parinte(p)$

SfCatTimp
 $succesor \leftarrow prec$

SfDaca
 SfFunctia

ȘTERGERE

Pp. chei distincte.

```

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 șters e
           abc.rad  $\leftarrow$  sterge_rec(abc.rad, e)
SfSubalgoritm

Functia sterge_rec(p, e)
    {complexitate timp:  $O(h)$ }
pre:    p este adresa unui nod; p : $\uparrow$  Nod - rădăcina unui subarbore
post:    se șterge nodul având cheia egală cu cheia lui e din subarboarele de rădăcină p și se
           returnează rădăcină noului subarbore
           {dacă s-a ajuns la subarbore vid}
Daca p=NIL atunci
    sterge_rec  $\leftarrow$  NIL
altfel
Daca e.c < [p].e.c atunci
    {se șterge din subarboarele stâng}
    [p].st  $\leftarrow$  sterge_rec([p].st, e)
    sterge_rec  $\leftarrow$  p
altfel
Daca e.c > [p].e.c atunci
    {se șterge din subarboarele drept}
    [p].dr  $\leftarrow$  sterge_rec([p].dr, e)
    sterge_rec  $\leftarrow$  p
altfel
    {am ajuns la nodul care trebuie șters}
Daca [p].st  $\neq$  NIL  $\wedge$  [p].dr  $\neq$  NIL atunci
    {nodul are și subarbore stâng și subarbore drept}
    temp  $\leftarrow$  minim([p].dr)
    {se mută cheia minimă în p}
    [p].e  $\leftarrow$  [temp].e
    {se șterge nodul cu cheia minimă din subarboarele drept}
    [p].dr  $\leftarrow$  sterge_rec([p].dr, [p].e)
    sterge_rec  $\leftarrow$  p
altfel
    temp  $\leftarrow$  p
Daca [p].st = NIL atunci
    {nu există subarbore stâng}
    repl  $\leftarrow$  [p].dr
altfel
    {nu există subarbore drept, [p].dr=NIL}
    repl  $\leftarrow$  [p].st
SfDaca
    {dealocă spațiul de memorare pentru nodul care trebuie șters}
    dealoca(temp)
    sterge_rec  $\leftarrow$  repl
SfDaca
SfDaca
SfDaca
SfDaca

```

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 .