ARBORI BINARI DE CĂUTARE
(BINARY SEARCH TREE)

- *Arborii binari de căutare* (ABC) sunt structuri de date des folosite în implementarea containerelor care conţin elemente de tip $TComparabil$ (sau identificate prin chei de tip $TComparabil$) şi care suportă următoarele operaţii:

  1. căutare;
  2. adăugare;
  3. ştergere;
  4. determinare maxim, minim, predecesor, succesor.

- ABC sunt SD care se folosesc pentru implementare:

  – dicţionar, dictionar ordonat
    * **TreeMap** în Java (foloseşte ABC echilibrat - arbore roşu-negru)
    * **map** din STL foloseşte ABC echilibrat ca implementare.
      · C++ 11 - **unordered_map** (tabelă de dispersie)

  – coadă cu priorităţi;

  – listă (**TreeList** în Java; foloseşte ABC echilibrat);

  – colecţie, mulţime (**TreeSet** în Java; foloseşte ABC echilibrat - arbore roşu-negru);

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

1. *Elementele* sunt identificate printr-o **cheie**.
2. *Cheia* elementului este de tip $TComparabil$.
3. Pp. relaţia "$\leq$" între chei (se poate uşor generaliza la o relaţie de ordine oarecare).

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

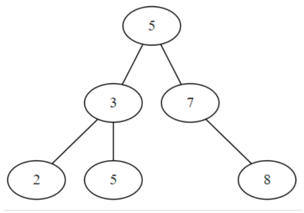
- *dacă x este un nod al ABC, atunci:*

  – *$\forall$ y un nod din subarborele stâng al lui x, are loc inegalitatea $cheie(y) \leq cheie(x)$ ($cheie(y)\ \mathcal{R}\ cheie(x)$).*

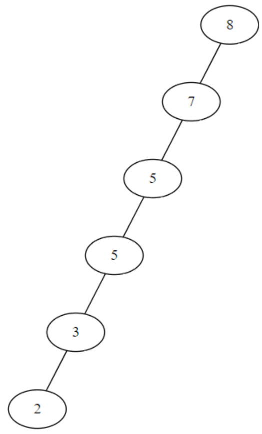
  – *$\forall$ y un nod din subarborele drept al lui x, are loc inegalitatea $cheie(x) < cheie(y)$ ($\neg(cheie(y)\ \mathcal{R}\ cheie(x))$).*

Exemplu. Presupunem că în container avem cheile 2 3 5 5 7 8 şi relaţia $\mathcal{R} = \leq$. În Figura 1 sunt indicaţi 2 ABC care conţin aceste chei.

- operaţiile de bază pe ABC consumă timp O(**înălţimea** arborelui).



(a) ABC echilibrat.



(b) ABC degenerat (lanţ).

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

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

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

Reprezentarea ABC

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

1. secvenţial;
2. înlănţuit
   - reprezentarea înlănţuirilor folosind alocare dinamică (pointeri);
   - reprezentarea înlănţuirilor folosind alocare statică (tablouri).

- pentru implementarea operaţiilor, pp. în cele ce urmează
  - reprezentare înlănţuită folosind alocare dinamică.
  - $\mathcal{R} =\leq$

- notăm cu h înălţimea arborelui.

- notăm cu n numărul de noduri din arbore.

REPREZENTARE

- într-un nod memorăm informaţia utilă şi adresa celor doi descendenţi (stâng şi drept)

- putem memora (pentru a eficientiza anumite operaţii) şi alte informaţii
  - adresa părintelui
  - adâncimea, înalţimea nodului

<u>Nod</u>
    e: TElement *//informaţia utilă nodului*
    st: ↑ Nod *//adresa la care e memorat descendentul stâng*
    dr: ↑ Nod *//adresa la care e memorat descendentul drept*

<u>Container</u>
    rad: ↑ Nod *//adresa rădăcinii* ABC

CĂUTARE - pp. chei distincte.

- returnează subarborele a cărui rădăcină conţine cheia căutată.

<u>Varianta recursivă</u>.

Modelul matematic recursiv

- arborele binar de căutare îl vom referi sub forma $abc(r,s,d)$
  - r - e informaţia 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 conţine 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 conţine 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}
  `Daca` $p = NIL \vee [p].e.c = e.c$ `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 conţine 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
    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 creeazaNod(e)
    {complexitate timp: θ(1)}
```

pre: e este de tip $TElement$

post: se returnează pointer spre un nod care conţine elementul e

```
    {se alocă un spaţiu de memorare pentru un nod; p :↑ 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 şi 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

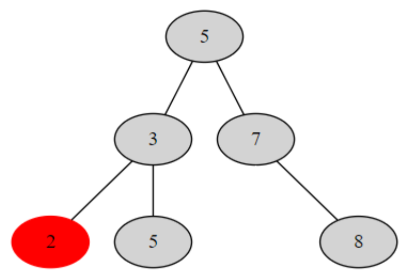


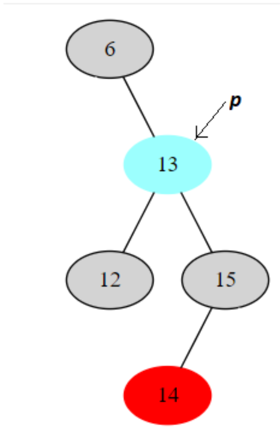
Figura 2: Minim.

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

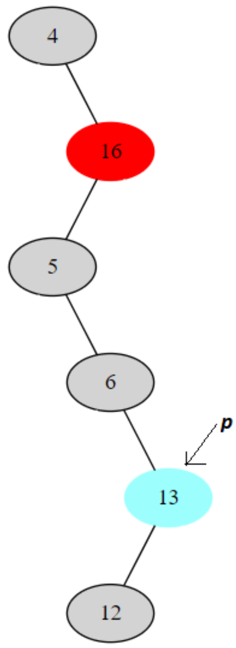
$p \leftarrow [p].st$
SfCatTimp
minim $\leftarrow p$
SfFunctia

SUCCESOR

- 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 succesorul cheii 4 este 6.



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



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

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

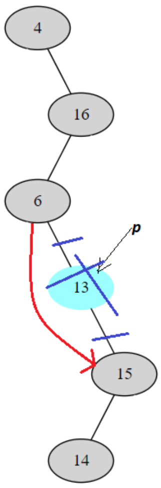
Observaţie

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

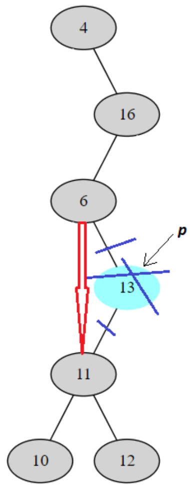
ŞTERGERE

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

Sunt trei cazuri la ştergere, indicate mai jos.



(a) Nodul p nu are descendent stâng.



(b) Nodul p nu are descendent drept.

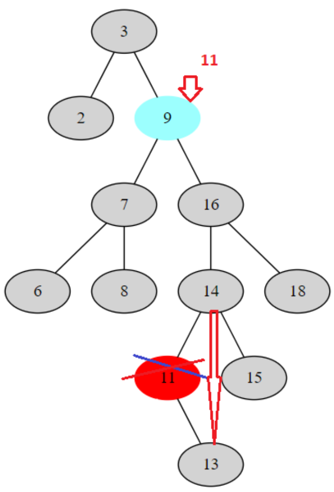


Figura 5: Nodul p are şi descendent stâng şi drept.

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 ← sterge_rec(abc.rad, e)
SfSubalgoritm

Functia sterge_rec(p, e)
   {complexitate timp: O(h)}
pre:   p este adresa unui nod; p :↑ Nod - rădăcina unui subarbore
post:  se sterge nodul având cheia egală cu cheia lui e din subarborele de rădăcină p şi se
   returnează rădăcină noului subarbore
   {dacă s-a ajuns la subarbore vid}
   Daca p=NIL atunci
      sterge_rec ←NIL
   altfel
      Daca e.c < [p].e.c atunci
         {se şterge din subarborele stâng}
         [p].st ←sterge_rec([p].st, e)
         sterge_rec ← p
      altfel
         Daca e.c > [p].e.c atunci
            {se şterge din subarborele drept}
            [p].dr ← sterge_rec([p].dr, e)
            sterge_rec ← p
         altfel
            {am ajuns la nodul care trebuie şters}
            Daca [p].st ≠ NIL ∧ [p].dr ≠ NIL atunci
               {nodul are şi subarbore stâng şi subarbore drept}
               temp ← minim([p].dr)
               {se mută cheia minimă în p}
               [p].e ← [temp].e
               {se şterge nodul cu cheia minimă din subarborele drept}
               [p].dr ← sterge_rec([p].dr, [p].e)
               sterge_rec ← p
            altfel
               temp ← p
               Daca [p].st = NIL atunci
                  {nu există subarbore stâng}
                  repl ← [p].dr
               altfel
```

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

În directorul asociat Cursului 12 găsiţi implementarea parţială, în limbajul C++, a containerului **Colecţie** cu elemente de tip comparabil, reprezentarea este sub forma unui ABC reprezentat înlănţuit, cu alocare dinamică a nodurilor). Iteratorul (în inordine) nu este implementat.

Probleme

1. Scrieţi o operaţie nerecursivă care determină părintele unui nod p dintr-un ABC.
2. Scrieţi varianta iterativă pentru operaţia de adăugare într-un ABC.
3. Presupunând că dorim ca fiecare nod din arbore să memoreze următoarele: informaţia utilă, referinţă către subarborele stâng, referinţă către subarborele drept, referinţă către părinte. Folosind reprezentarea înlănţuită cu alocare dinamică a nodurilor, scrieţi operaţia de adăugare în ABC (varianta iterativă, varianta recursivă).
4. Presupunând ca elementele sunt de forma (cheie, valoare) şi relaţia de ordine între chei este "$\leq$", scrieţi operaţia **MAXIM** care determină elementul din ABC având cea mai mare cheie.
5. Presupunând ca elementele sunt de forma (cheie, valoare), relaţia de ordine între chei este "$\leq$", şi arborele este reprezentat înlănţuit cu alocare dinamică a nodurilor, scrieţi operaţia **PREDECESOR** care pentru un nod p dintr-un ABC determină elementul având cea mai mare cheie mai mică decât cheia lui p .
6. Implementaţi operaţiile pe ABC generalizând relaţia "$\leq$" de la proprietatea unui ABC la o relaţie de ordine oarecare R.