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 ca implementare.

  – 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 0.1 *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 $cheie(x) < cheie(y)$ ($\neg(cheie(y)\ \mathcal{R}\ cheie(x))$).*

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

  – dacă ABC este plin $\Rightarrow$ înălţimea este $\theta(log_2 n)$;

  – dacă ABC este degenerat (lanţ liniar) $\Rightarrow$ înălţimea este $\theta(n)$.

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

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.
- notăm cu h înălţimea 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 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` $\leftarrow p$
  `altfel`
    `Daca` $e.c < [p].e.c$ `atunci`
      {se caută în subarborele stâng}
      `cauta_rec` $\leftarrow$ `cauta_rec`($[p].st, e$)
    `altfel`
      {se caută în subarborele drept}
      `cauta_rec` $\leftarrow$ `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

```
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 ← NIL
   [p].dr ← NIL
   creeazaNod ← p
SfFunctia
```

```
Subalgoritm adauga(abc, e)
   {complexitate timp: O(h)}
```
pre: abc este un arbore binar de căutare

post: abc' este arbore binar de căutare, e a fost adăugat în abc'

```
   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 subarborele de rădăcină p şi se returnează rădăcină noului subarbore

```
   {dacă s-a ajuns la subarbore vid se adaugă}
   Daca p = NIL atunci
      p ←creeazaNod(e)
   altfel
      Daca e.c ≤ [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

`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`

`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`

ŞTERGERE

PP. chei distincte.

`Subalgoritm sterge`(abc, e)
{complexitate timp: $O(h)$}
pre: abc este un arbore binar de căutare
post: abc' este arbore binar de căutare, e a fost sters din abc'
$abc.rad \leftarrow$ `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 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
```

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.