Arbori binari de căutare

2015

Arbori binari de căutare

- căutare
- inserare
- ștergere

Arbori binari de căutare

- căutare
- inserare
- ștergere

- analog al listei ordonate (performanța operației de căutare)

Arbori binari de căutare - definiții

Structură:

- Arbore binar

Arbori binari de căutare - definiții

Structură:

- Arbore binar
- cu chei de un tip total ordonat

Arbori binari de căutare - definiții

Structură:

- Arbore binar
- cu chei de un tip total ordonat
- pentru orice nod u al său avem relațiile:

(1) `info[u] > info[v]`, $\forall v \in$ `left[u]`

(2) `info[u] < info[w]`, $\forall w \in$ `right[u]`

a.b.c. - definiție recursivă

Un arbore binar T este:

1. fie un arbore vid ($T = \emptyset$),

a.b.c. - definiție recursivă

Un arbore binar T este:

1. fie un arbore vid ($T = \emptyset$),
2. fie este nevid
   și atunci conține un nod numit *rădăcină*, cu `info` de un tip totul ordonat
   împreună cu doi subarbori binari de căutare disjuncți (numiți *subarborele stâng*, `left`, respectiv *subarborele drept*, `right`, astfel încât:
   (1') `info[root(T)] > info[root(left[u])]`
   (2') `info[root(T)] < info[root(right[u])]`

a.b.c. - Chei multiple

1) Contorizare apariții
2) Reprezentare efectivă a cheilor multiple:

a.b.c. - Chei multiple

1) Contorizare apariții
2) Reprezentare efectivă a cheilor multiple:

Numim arbore binar de căutare *nestrict la stânga* un arbore binar T cu proprietatea că în fiecare nod u al să avem relațiile:

(3) $\texttt{info[u]} \geq \texttt{info[v]}, \forall v \in \texttt{left[u]}$

(4) $\texttt{info[u]} < \texttt{info[w]}, \forall w \in \texttt{right[u]}$

a.b.c. - Chei multiple

1) Contorizare apariții
2) Reprezentare efectivă a cheilor multiple:

Numim arbore binar de căutare *nestrict la stânga* un arbore binar T cu proprietatea că în fiecare nod u al să avem relațiile:

(3) info[u] $\geq$ info[v], $\forall v \in$ left[u]

(4) info[u] < info[w], $\forall w \in$ right[u]

Analog, un arbore binar de căutare *nestrict la dreapta*, cu relațiile:

(5) info[u] > info[v], $\forall v \in$ left[u]

(6) info[u] $\leq$ info[w], $\forall w \in$ right[u]

Legătura cu *sortarea*

Parcurgerea în inordine (SRD) a unui a.b.c. produce o listă ordonată crescător a cheilor.

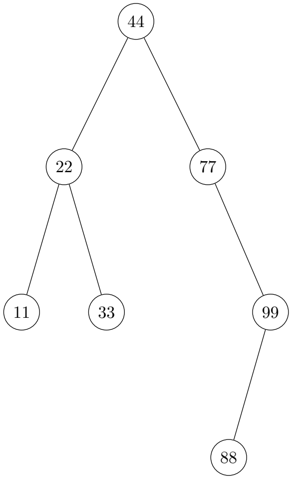
Legătura cu *sortarea*

Parcurgerea în inordine (SRD) a unui a.b.c. produce o listă ordonată crescător a cheilor.

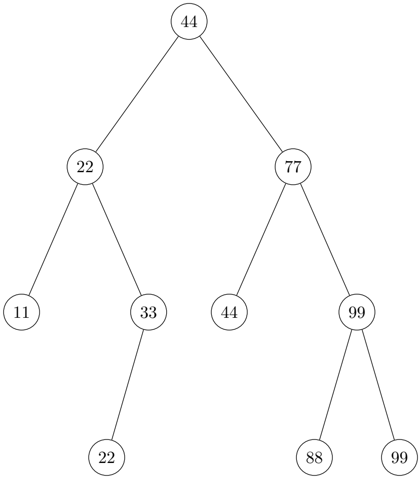
De aceea, structura de arbore binar de căutare este potrivită pentru seturi de date pe care, pe lângă inserări și ștergeri, se cere destul de frecvent ordonarea totală a cheilor.

Arbori binari de căutare - exemple

(a) Arbore binar de căutare strict



(b) Arbore binar de căutare nestrict la dreapta. Cheile 22, 44 şi 99 sunt chei multiple



Căutarea în a.b.c. - iterativ

arbore Search (arbore *Root, int Val)*

```
{ // caută în Root valoarea Val, iterativ si returneaza ptr curent Loc
   arbore *Loc;
   bool found;
   Loc = Root;
   found = false;
   while ((Loc != NULL) && (found == false))
      if (Loc->info == Val)
         {
         found = true;
         return Loc                                   // căutare cu succes
         }
      else
         if (Loc->info > Val)
            Loc = Loc->left;
         else
            Loc = Loc->right;
   return Loc;                  // Loc = NULL codifică o căutare fără succes
}
```

Căutarea în a.b.c. - recursiv

arbore SearchRec (arbore* Root, int Val)*

```
{
// caută în Root valoarea Val, recursiv
// NULL codifică o căutare fără succes
  if (Root == NULL)
    return NULL;                    // căutare fără succes
  else
    if (Root->info == Val)
      return Root;                  // căutare cu succes
    if (Root->info > Val)
      return SearchRec(Root->left, Val);
    else if (Root->info < Val)
      return SearchRec(Root->right, Val);
}
```

Căutare cu inserare în a.b.c. - rec.

arbore SearchInsRec (int Val, arbore* Root)*

```
{ // NULL codifică o căutare fără succes
// pentru cheile multiple se incrementează un câmp contor
  if (Root == NULL)                    // Val nu a fost găsit și va fi inserat
     // inserează nod nou cu Val
     // incrementează contor
  else
  {
    if (Root->info > Val)
      return SearchInsRec(Val, Root->left);
    else if (Root->info < Val)
      return SearchInsRec(Val, Root->right);
    }
}
```

Creare a.b.c. prin inserări repetate

```
Root = NULL;                    // inițializarea arborelui cu arborele vid
while (mai există data x de inserat)
{
  citeste x
  SearchInsRec(x, Root);
}
```

Ștergere nod din a.b.c.

Avem pointerul `p` pe nodul de șters și pointerul `tp` pe tatăl său.

Ștergere nod din a.b.c.

Avem pointerul `p` pe nodul de șters și pointerul `tp` pe tatăl său.
Caz 1) Nodul de șters are cel mult 1 fiu nevid

  `tp` devine tatăl unicului fiu nevid

Mai exact, `tp->left` sau `tp->right` (în funcție de unde se afla `p`) preia unicul fiu nevid al lui `p`.

Ștergere nod din a.b.c.

Avem pointerul `p` pe nodul de șters și pointerul `tp` pe tatăl său.
Caz 1) Nodul de șters are cel mult 1 fiu nevid

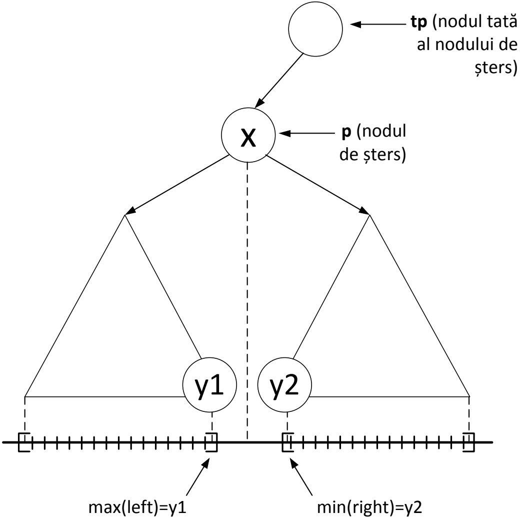
`tp` devine tatăl unicului fiu nevid

Mai exact, `tp->left` sau `tp->right` (în funcție de unde se afla `p`) preia unicul fiu nevid al lui `p`.
Caz 2) Nodul de șters are ambii fii nevizi

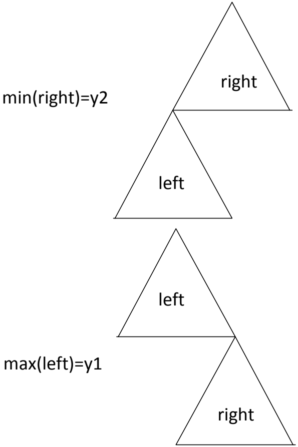
eliminăm doar valoarea `p->info`

căutăm o altă valoare în arbore, care
- să poată *urca* în acest nod (*separator*)
- să fie într-un nod ușor de șters (caz 1)

Explicații ștergere nod din a.b.c.



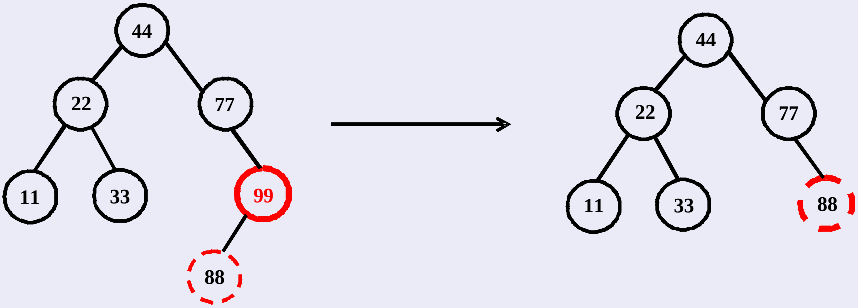
Candidații ideali pentru ca ștergerea să afecteze arborele cât mai puțin:



Ștergere nod din a.b.c. - Exemplu

Caz 1: Nodul de șters are cel mult 1 fiu nevid

Nodul de șters este 99

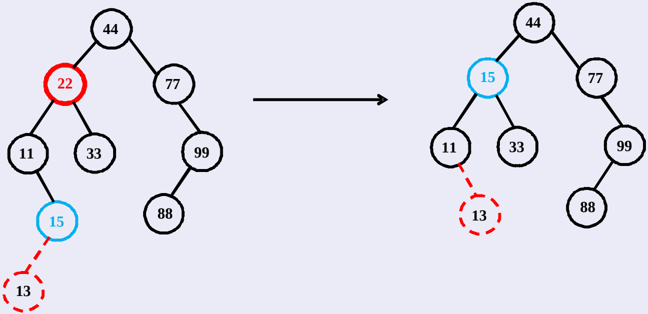


Subarborele stâng (nevid) al nodului cu cheia 99 devine subarborele drept pentru nodul cu cheia 77.

Ștergere nod din a.b.c. - Exemplu

Caz 2: Nodul de șters are ambii fii nevizi

Nodul de șters este 22



- Se determină predecesorul în inordine (15)
- Valoarea cheii se copiază în locația nodului de șters
- Se șterge nodul cu cheia 15

Complexitatea operațiilor la a.b.c.

Operațiile de inserare și ștergere de noduri într-un arbore binar de căutare depind în mod esențial de operația de căutare. *Căutarea* revine la parcurgerea, eventual incompletă, a unei ramuri, de la rădăcină până la un nod interior în cazul căutării cu succes, sau până la primul fiu vid întâlnit în cazul căutării fără succes (și al inserării).

Complexitatea operațiilor la a.b.c.

Operațiile de inserare și ștergere de noduri într-un arbore binar de căutare depind în mod esențial de operația de căutare. *Căutarea* revine la parcurgerea, eventual incompletă, a unei ramuri, de la rădăcină până la un nod interior în cazul căutării cu succes, sau până la primul fiu vid întâlnit în cazul căutării fără succes (și al inserării).

Performanța căutării depinde de lungimea ramurilor pe care se cauta; media ei va fi dată de *lungimea medie a ramurilor*, iar dimensiunea maximă de lungimea celor mai lungi ramuri, adică de *adâncimea arborelui*. **Forma arborelui**, deci si adâncimea depind, cu algoritmii dați, **de ordinea introducerii cheilor** și putem avea cazul cel mai nefavorabil, în care adâncimea arborelui este n, numărul nodurilor din arbore, adică performanța căutării rezultă $O(n)$.

Complexitatea operațiilor la a.b.c.

Operațiile de inserare și ștergere de noduri într-un arbore binar de căutare depind în mod esențial de operația de căutare. *Căutarea* revine la parcurgerea, eventual incompletă, a unei ramuri, de la rădăcină până la un nod interior în cazul căutării cu succes, sau până la primul fiu vid întâlnit în cazul căutării fără succes (și al inserării).

Performanța căutării depinde de lungimea ramurilor pe care se cauta; media ei va fi dată de *lungimea medie a ramurilor*, iar dimensiunea maximă de lungimea celor mai lungi ramuri, adică de *adâncimea arborelui*. **Forma arborelui**, deci si adâncimea depind, cu algoritmii dați, **de ordinea introducerii cheilor** și putem avea cazul cel mai nefavorabil, în care adâncimea arborelui este n, numărul nodurilor din arbore, adică performanța căutării rezultă $O(n)$.

În viitor, vom face operația de completare canonică a unui arbore binar la unul strict, în care fiecare fiu vid se înlocuiește cu un nod special, frunza. Tot acolo se estimează lungimea medie a drumului de la rădăcină până la o frunză și adâncimea unui asemenea arbore. Anticipând puțin, avem o limită inferioară pentru adâncime de ordinul lui $\log_2 n$, ceea ce înseamnă că performanța operației de căutare nu poate coborî sub ea. Ne punem problema dacă putem *atinge această valoare optimă*.

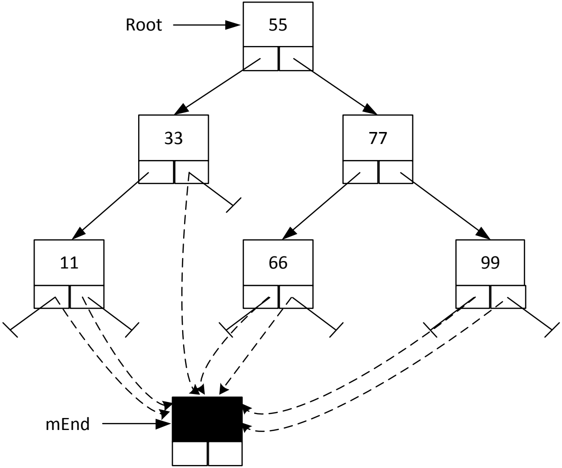
Detalii implementare C

Căutarea în a.b.c.

arbore Search (arbore *Root, int Val)*

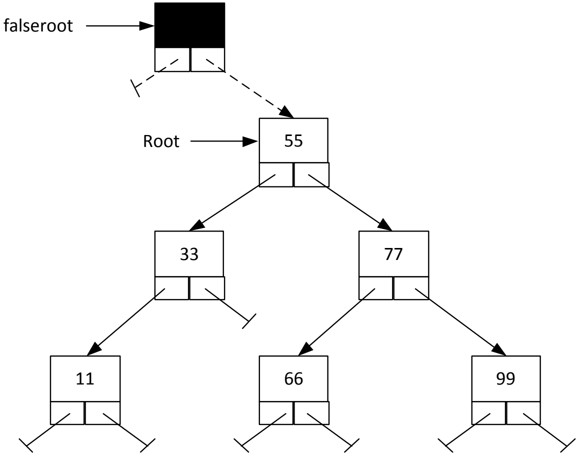
```
{                           // caută în Root valoarea Val, iterativ si returneaza ptr curent Loc
   arbore *Loc;
   bool found;
   Loc = Root;
   found = false;
   while ((Loc != NULL) && (found == false))
      if (Loc->info == Val)
         {
         found = true;
         return Loc                                                   // căutare cu succes
         }
      else
         if (Loc->info > Val)
            Loc = Loc->left;
         else
            Loc = Loc->right;
   return Loc;                              // Loc = NULL codifică o căutare fără succes
}
```

Căutarea în a.b.c. cu nod marcaj



Arbore binar de căutare completat cu nod marcaj la sfârșit

Căutarea în a.b.c. cu nod marcaj



Arbore binar de căutare completat cu nod marcaj la început

SearchIns în a.b.c. - iterativ

*void SearchInsIterativ (int x, arbore *Root, arbore *p)*

```
{   arbore *p1;
   int d; // inițializarea pointerilor pentru parcurgere
   p1 = NULL;
   p = root;
   d = 1;
   while ((p != NULL) && (d != 0))
      if (x < p->info)
         { p1 = p;
         p = p->left;
         d = -1; }
      else
            if (x > p->info)
            {  p1 = p;
            p = p->right;
            d = 1; }
               else                                             // x == p->info
         d = 0;
```

```
  if (p != NULL)                                   // d = 0 și am găsit x în arborele root,
    p->contor = p->contor + 1;                       // deci trebuie incrementat contorul
  else                                                // p == NULL și facem inserarea
    { p->info = x;
    p->contor = 1;
    p->left = NULL;
    p->right = NULL;
// legarea noului nod la tată
      if (p1 == NULL)
        root = p1;                                    // cazul inserării într-un arbore vid
      else
        { if (d < 0)
          p1->left = p;
        else
          p1->right = p;
        }
      }
  }
```

Ștergere nod din a.b.c.

SearchDel

In: `int x`, `pnod root`
// `pnod p1, p2` sunt pointeri curenți
// `pnod falseroot` este un nod fals înainte de rădăcină

Ștergere nod din a.b.c.

SearchDel

In: `int x`, `pnod root`
// `pnod p1, p2` sunt pointeri curenți
// `pnod falseroot` este un nod fals înainte de rădăcină

Delete1

In: `pnod p`
// șterge nod cu cel mult un fiu nevid

Ștergere nod din a.b.c.

SearchDel

In: `int x`, `pnod root`
// `pnod p1, p2` sunt pointeri curenți
// `pnod falseroot` este un nod fals înainte de rădăcină

Delete1

In: `pnod p`
// șterge nod cu cel mult un fiu nevid

Delete2

In: `pnod p`
// șterge nod cu doi fii nevizi

Ștergere nod din a.b.c.

SearchDel

In: `int x`, `pnod root`
// `pnod p1, p2` sunt pointeri curenți
// `pnod falseroot` este un nod fals înainte de rădăcină

Delete1

In: `pnod p`
// șterge nod cu cel mult un fiu nevid

Delete2

In: `pnod p`
// șterge nod cu doi fii nevizi

*void Delete1(arbore *p)*

```
{  // șterge nodul p cu cel mult un succesor
   if (p->left == NULL)
      p = p->right;
   else
      p = p->left;
}
```

*void Delete2 (arbore *p)* // șterge nodul p cu doi succesori

```
{  // caută predecesorul în inordine al lui p->info mergând un pas la stânga,
//  apoi la dreapta, cât se poate
// parcurgerea se face cu r și q = tatăl lui r
   arbore *q, *r;                          // d1 = -1 ⇔ r = q->left
   int d1;                                 // d1 = 1 ⇔ r = q->right
// (a)
   q = p;
   r = p->left;
   d1 = -1;
   while (r->right != NULL)
      { q = r;
      r = r->right;
      d1 = 1; }
// (b)
   p->info = r->info;                      // se copiază în p valorile din r
   p->contor = r->contor;
// (c) se leagă de tată, q, subarborele stâng al lui r
   if (d1 < 0)
      q->left = r->left;
   else
      q->right = r->left;
}
```

*void Search Del (int x, arbore *Root)*

```
{ arbore *p1, *p2, *falseroot; int found;
   falseroot = new nod; falseroot->right = root;          // adăgăm nod marcaj
   p1 = root; p2 = falseroot;
   d = 1; found = false;
   while ((p1 != NULL) && (found == false))
      { p2 = p1;
      if (x < p->info)
         { p2 = p1; p1 = p->left; d = -1; }
      else
         { if (x > p->info)
            { p1 = p->left; d = 1; }
         else
            found = true; }
      }
   if (found == false)
       // Nu am găsit
   else                              // found == true și trebuie să șterg nodul p1
      if ((p1->left == NULL) || (p1->right == NULL))      // ștergere caz 1
            Delete1(p1);
      else  Delete2(p1)                                   // ștergere caz 2
       // legarea noului nod p1 de tatăl său p2
      if (d > 0) p2->right = p1;
      else p2->left = p1;
}
```