laborator 7

>> Structuri arborescente II

Conţinut

- Arbori binari de căutare echilibraţi AVL
- Ansamble (heaps)
- Cozi cu priorităţi (priority queues)

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap 15*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Tema 7*
- **F. Pfenning**, *Lecture Notes on AVL Trees, 15-122: Principles of Imperative Computation*, Lecture 18, 22 martie 2011
- **D.E. Knuth.** *Tratat de programare a calculatoarelor - Sortare şi căutare*, Editura Tehnică, 1973 (şi ediţiile ulterioare)

Arborii binari de căutare sunt eficienţi (optimi) doar atunci când sunt *echilibraţi* (balanced). Ideal este ca înălţimea arborelui să fie $O(\log n)$, unde n este numărul de noduri din arbore.

Soluţia va fi reechilibrarea (rebalancing) arborelui în timpul operaţiei de inserare astfel încât să se păstreze şi proprietatea arborelui binar de căutare.

Există mai mulţi algoritmi pentru menţinerea **arborilor binari de căutare echilibraţi**: arbori AVL, arbori roşu-negru (red/black), arbori splay sau arbori binari de căutare construiţi aleator (randomized). Ei diferă prin invarianţii pe care îi asigură (în afara celui de ordine) şi prin momentul şi modul în care se face reechilibrarea.

Arborii AVL (introduşi în 1962) poartă numele inventatorilor săi: G.M. Adelson-Velskii şi E.M. Landis.

Arbori AVL

Arborii AVL asigură **i**nvariantul de înălţime:

Fie un nod x din arbore. Înălţimea subarborelui său stâng şi cea a subarborelui său drept diferă prin cel mult 1 $\Leftrightarrow \left|h(x->left)-h(x->right)\right| \leq 1$.

Există trei cazuri în care se poate afla orice nod `x` din arbore, în funcţie de relaţia înălţimilor celor doi subarbori ai săi.

Caz 1: Subarborele drept este mai înalt:

```
h(x->right) - h(x->left) = 1
```

Caz 2: Subarborii sunt egali:

```
h(x->right) - h(x->left) = 0
```

Caz 3: Subarborele stâng este mai înalt:

```
h(x->right) - h(x->left) = -1
```

Valorea `h(x->right) - h(x->left)` $\in \{-1,0,1\}$ va fi reţinută într-un câmp suplimentar în fiecare nod, câmp ce va fi notat `bal` şi care se numeşte *factor de echilibrare (balance factor)*.

Deci, vom memora nodurile unui arbore AVL folosind următorul tip de nod:

```
struct nod {
    int info;
    nod *left;
    nod *right;
    int bal;
};
```

Puteam alege să reţinem în fiecare nod înălţimea subarborelui cu rădăcina în acel nod (`h`) *şi să calculăm factorul de echilibru atunci când este necesar.*

Căutarea

Căutarea într-un arbore AVL este aceeaşi ca la abori binari de căutare, deoarece invariantul de înălţime intervine doar la operaţia de inserare.

Inserarea atât în cazul arborilor binari de căutare (inclusiv cei echilibraţi AVL) se comportă ca şi operaţia de căutare. Este posibil să le combinăm într-o procedură, ce poate fi recursivă sau iterativă, care efectuează căutarea unui nod, iar în cazul în care nodul nu se află în arbore îl inserează.

Inserarea

În mod evident, inserarea poate conduce la creşterea înălţimii unui subarbore astfel încât proprietatea arborilor AVL să nu mai fie respectată (deci factorul de echilibru al nodului să devină 2 sau -2.

Observăm că singurele noduri afectate (cărora este posibil să le fi fost modificată înălţimea) de inserare se află pe drumul între rădăcină şi locul unde s-a produs inserarea. Atunci, ideea de bază a reechilibrării este ca, după inserare, să se parcurgă nodurile (de jos în sus) de la nodul nou inserat la rădăcină şi să se actualizeze înălţimea (respectiv factorul de echilibru). În plus, la fiecare pas, pentru a „reechilibra” arborele folosim *rotaţii*. Acestea sunt aplicate nodurilor cu un factor de echilibrare nepermis (2 sau -2). După ce am întâlnit un astfel de nod (într-unul din cele patru cazuri de mai jos) nu va mai fi nevoie să continuăm parcurgerea. Deci parcurgerea de jos în sus se termină fie în acest caz, fie atunci când ajungem la rădăcină.

Putem întâlni un nod x ce necesită reechilibrare în patru situaţii diferite, pentru fiecare dintre ele se aplică câte un algoritm de rotaţie.

Situaţiile ce corespund rotaţiilor simple se mai numesc şi cazuri externe.

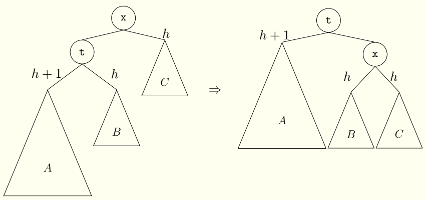
Rotaţii simple

Caz 1 Inserarea s-a făcut în subarborele **stâng** al fiului **stâng** al lui x. Rotaţia se numeşte **rotaţie dreapta-dreapta (DD)** sau, simplu **rotaţie dreapta (D)**.

În mod normal, procedura s-ar numi mai potrivit ROTAŢIE–SPRE–DREAPTA.

►► ROTAŢIE–DREAPTA(x)

```
tempref  ← x->left
x->left  ← tempref->right
tempref->right  ← x
x->bal  ←  x->bal + (1 - min(tempref->bal, 0))
tempref->bal  ←  tempref->bal + (1 + max(x->bal, 0))
x  ← tempref
```



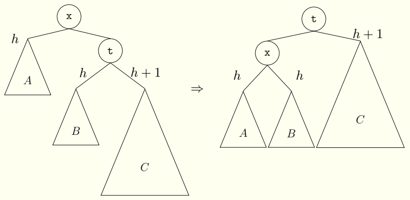
Figură 1 Rotaţie dreapta în nodul x (înainte şi după rotaţie))

Caz 2 Inserarea s-a făcut în subarborele **drept** al fiului **drept** al lui x.Rotaţia se numeşte **rotaţie stânga-stânga (SS)** sau, simplu **rotaţie stânga (S)**.

În mod normal, procedura s-ar numi mai potrivit ROTAŢIE–SPRE–STÂNGA.

►► ROTAŢIE–STÂNGA(x)

```
tempref  ← x->right
x->right  ← tempref->left
tempref->left  ← x
x->bal  ←  x->bal - (1 + max(tempref->bal, 0))
tempref->bal  ←  tempref->bal - (1 - min(x->bal, 0))
x  ← tempref
```



Figură 2 Rotaţie stânga în nodul x (înainte şi după rotaţie))

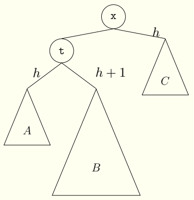
Rotaţii duble

Caz 3 Inserarea s-a făcut în subarborele **drept** al fiului **stâng** al lui x. Rotaţia se numeşte **rotaţie stânga-dreapta (SD)**.

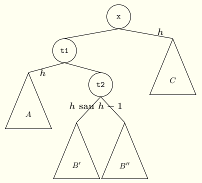
Situaţiile ce corespund rotaţiilor duble se mai numesc şi cazuri interne.

▶▶ Rotaţie–Stânga–Dreapta(x)

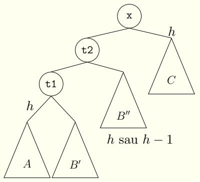
1. Rotaţie–Stânga (x->left)
2. Rotaţie–Dreapta (x)



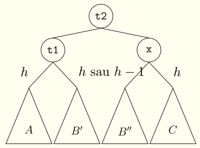
(**a**) Subarborele x iniţial



(**b**) Detaliere



(**c**) După rotaţia stânga în t1 (x->left)



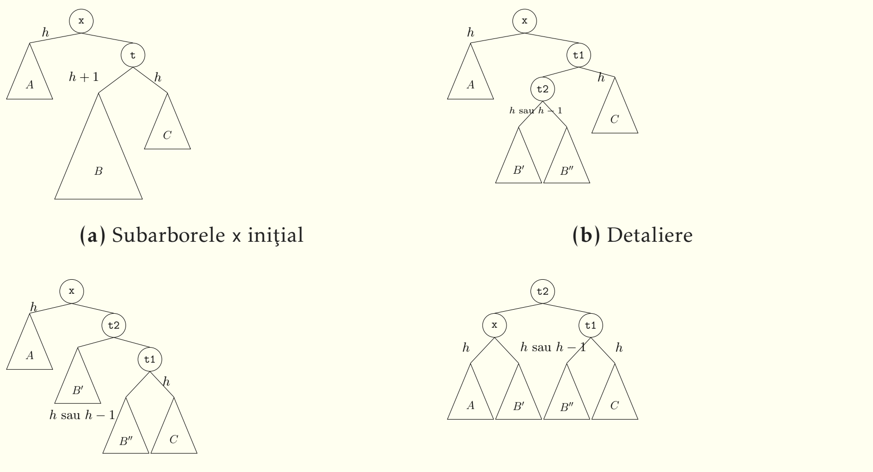
(**d**) După rotaţia dreapta în x

Figură 3 Rotaţie stânga-dreapta în nodul x

Caz 4 Inserarea s-a făcut în subarborele **stâng** al fiului **drept** al lui x. Rotaţia se numeşte **rotaţie dreapta-stânga (DS)**.

▶▶ Rotaţie–Dreapta–Stânga(x)

1. Rotaţie–Dreapta (x->right)
2. Rotaţie–Stânga (x)



(**a**) Subarborele x iniţial

(**b**) Detaliere

(**c**) După rotaţia dreapta în t1 (x->right)

(**d**) După rotaţia stânga în x

Figură 4 Rotaţie dreapta-stânga în nodul x

▶▶ Creează–Nod(*val*)

```
alocare spaţiu pentru nodul nou x
x->info = val
x->left = NIL
x->right = NIL
x->bal = 0
return x
```

▶▶ Echilibrează(*p*, *val*)

```
if p->bal = -2 then
  if p->left->bal = 1 then
    Rotaţie–Stânga–Dreapta(p)
  else
    Rotaţie–Dreapta(p)
  endif
else if p->bal = 2 then
  if p->right->bal = -1 then
    Rotaţie–Dreapta–Stânga(p)
  else
    Rotaţie–Stânga(p)
  endif
endif
```

▶▶ Caută–şi–Inserează–Recursiv(p, val)

```
if p = NIL then
  p = Creează–Nod(val)
  return true
endif
if p->info = val then
  return false
endif
if p->info > val then
  if Caută–şi–Inserează–Recursiv(p->left, val) = true then
    p->bal ← p->bal - 1
  else
    return false
  endif
else
  if Caută–şi–Inserează–Recursiv(p->right, val) = true then
    p->bal ← p->bal + 1
  else
    return false
  endif
endif
if p->bal = 0 then
  return false
else if (p->bal = 1) or (p->bal =-1) then
  return true
else
  Balansează(p)
  return false
endif
```

Atenţie la implementare, parametrul de intrare p (care este un pointer la nodul curent din parcurgere) trebuie transmis prin referinţă, ca şi parametrii procedurilor de rotaţie.

Ştergerea

Spre deosebire de operaţia de inserare, în cazul ştergerii unui nod, o singură rotaţie pentru reechilibrare nu va fi suficientă. În schimb, va trebui să parcurgem toate nodurile aflate între poziţia nodului şters şi rădăcină, efectuând, dacă este necesar, rotaţii pentru reechilibrarea arborelui.

Ştergerea în arborii echilibraţi AVL urmează acelaşi raţionament ca şi la arborii binari de căutare simpli.

i. Dacă z este o frunză (nu are niciun fiu), atunci eliminăm nodul z.

ii. Dacă z are un singur fiu, atunci înlocuim conţinutul (câmpul `info` al) nodului z cu cel al fiului. Acest lucru este posibil, deoarece o consecinţă a faptului că arborele este echilibrat este că acest unic fiu este un nod terminal. Prin copierea conţinutului practic îl înlocuim pe z cu fiul său. Apoi îl ştergem pe fiu, ştergere care corespunde cazului i..

iii. Dacă z are ambii fii nevizi, atunci găsim nodul y, succesorul lui z şi înlocuim conţinutul lui z cu cel al lui y. Astfel dorim să îl înlocuim pe z cu succesorul său. După copiere, ştergem nodul y, ştergere care corespunde unuia dintre cele două cazuri anterioare (i. sau ii.), pentru că nodul y, fiind succesor, are cel mult un fiu nevid.

De fapt, toate cele trei cazuri s-au redus la ştergerea unei frunze. După efectuare ştergerii, trebuie să „urcăm” în arbore, de la frunza respectivă spre rădăcină, traversând fiecare strămoş al frunzei. La fiecare nod din această traversare trebuie:

a) să actualizăm câmpul `bal` (factorul de echilibrare al nodului) şi

b) să utilizăm rotaţii pentru reechilibrare, dacă este necesar.

▶▶ CAUTĂ–ŞI–ŞTERGE–RECURSIV(p, val)

```
if p = NIL then
  return false
endif
if p->info > val then
  if CAUTĂ–ŞI–ŞTERGE–RECURSIV(p->left, val) = true then
    p->bal ← p->bal + 1
  else
    return false
  endif
else if p->info < val then
  if CAUTĂ–ŞI–ŞTERGE–RECURSIV(p->right, val) = true then
    p->bal ← p->bal - 1
  else
    return false
  endif
else
  if ( p->left = NIL ) or ( p->right = NIL ) then
    if p->left ≠ NIL then
      p->info ← p->left->info
      p->left ← NIL
      p->bal ← p->bal + 1
      return true
    else if p->right ≠ NIL then
      p->info ← p->right->info
      p->right ← NIL
      p->bal ← p->bal - 1
      return true
    else
      p ← NIL
    endif
    return true
  else
    y ← MINIM(p->right)
    p->info = y->info
    if CAUTĂ–ŞI–ŞTERGE–RECURSIV(p->right, y->info) = true then
      p->bal ← p->bal - 1
    else
      return false
    endif
  endif
endif
if (p->bal = 2) or (p->bal = -2) then
  Balansează(p)
endif
if p->bal = 0 then
  return true
else
  return false
endif
```

Ansamble (heaps)

Un ansamblu (binar) (**heap**) este un vector A prin care se reprezintă ca un arbore binar aproape complet (ultimul nivel poate să nu fie complet, el este plin de la stânga la dreapta). Fiecare element din vector reprezintă un nod din arbore (în vector se vor reţine informaţiile ataşate nodurilor). Dacă un vector este un ansamblu, el are două atribute:

1. *lungimea* (numărul elementelor din vector), pe care o vom nota n
2. *dimensiunea* (numărul elementelor din ansamblu memorate în vector), pe care o notăm sizeA

Rezultă că: în vector putem reţine elemente în A care nu aparţin ansamblului. În algoritmii ce urmează acordaţi atenţie suplimentară gestiunii celor două atribute.

Rădăcina arborelui este A[1].

Determinarea părintelui, fiului stâng şi fiului drept ale unui nod i se poate face prin formulele din procedurile următoare.

PĂRINTE(i)

1. **return** $\left\lfloor \frac{i}{2} \right\rfloor$

STÂNGA(i)

1. **return** 2i

DREAPTA(i)

1. **return** 2i + 1

Pentru orice nod i, exceptând rădăcina, este adevărată **proprietatea de ansamblu**:

- fie A[PĂRINTE(i)] ≥ A[i] (max-ordonare),
- fie A[PĂRINTE(i)] ≤ A[i] (min-ordonare).

În primul caz ansamblul este un **max-ansamblu (max-heap)**: valoarea dintr-un nod este mai mică sau egală decât cea a părintelui său; rădăcina reţine maximul. În al doilea caz ansamblul este un **min-ansamblu (min-heap)**: valoarea dintr-un nod este mai mare sau egală decât cea a părintelui său; rădăcina reţine minimul.

Procedurile referitoare la ansamble date în continuare lucrează pe o structură de max-ansamblu.

Inserarea

Procedura ASAMBLEAZĂ(A, i) primeşte ca date de intrare un vector A şi un indice i şi asigură respectarea proprietăţii de ansamblu.

Procedura ASAMBLEAZĂ(A, i) se întâlneşte în literatură şi cu denumirea de Reconstituie-ansamblu, *respectiv* Heapify.

Procedura presupune că subarborii de rădăcină STÂNGA(i), respectiv DREAPTA(i), sunt ansamble. Pentru a respecta proprietatea de ansamblu trebuie ca elementul A[i] să fie mai mare decât descendenţii săi, adică toate nodurile din subarborii de rădăcină STÂNGA(i) şi, respectiv DREAPTA(i). Prin urmare, pe poziţia i trebuie să fie adus elementul maxim, iar elementul A[i] iniţial să fie aşezat corespunzător în vector.

Procedura funcţionează recursiv, aducând elementul maxim (dintre A[i], A[STÂNGA(i)] şi A[DREAPTA(i)]) pe poziţia A[i]. Dacă maximul este chiar A[i] atunci algoritmul se termină. Însă, atunci când interschimbăm maximul (unul dintre variantele rămase A[STÂNGA(i)] şi A[DREAPTA(i)]) cu elementul A[i], acesta ajunge pe poziţia A[maxim] care poate să nu reprezinte poziţia corectă. Deci, trebuie să reapelăm procedura pentru această porţiune a ansamblului.

Datorită ipotezei că subarborii de rădăcină STÂNGA(i), respectiv DREAPTA(i), sunt ansamble, elementul maxim, dacă nu este A[i], nu poate fi decât fie nodul A[STÂNGA(i)], fie nodul A[DREAPTA(i)].

▶▶ Asamblează(A, i)

```
s ← Stânga(i)
d ← Dreapta(i)
max ← i
if s ≤ sizeA and A[s] > A[i] then
  max ← s
endif
if d ≤ sizeA and A[d] > A[max] then
  max ← d
endif
if max ≠ i then
  Interschimbă(A[i], A[max])
  Asamblează(A, max)
endif
```

Atenţie la faptul că transmiterea doar a vectorului A şi a numărului de elemente n ca parametrii unei funcţii ce implementează proceduri ce au ca date de intrare un ansamblu nu este suficientă. Trebuie transmis şi atributul `sizeA`. Putem folosi o structură în C++ care să conţină vectorul de elemente, dimensiunea n şi dimensiunea `sizeA` pentru a defini tipul variabilelor ce desemnează un ansamblu.

Crearea structurii de ansamblu pe un vector A[1..n] se face de jos în sus (plecând de la frunze spre rădăcină). Toate elementele din subvectorul $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 1..n\right]$ sunt frunze. De aceea, pot fi considerate ansamble cu un singur element. Prin urmare, este necesar să apelăm procedura de asamblare doar pentru restul nodurilor. Aşa cum s-a precizat, traversarea acestor noduri se face de jos în sus (deci de la $\left\lfloor \frac{n}{2} \right\rfloor$ spre 1). Astfel, subarborii unui nod vor forma ansamble înainte ca procedura Asamblează să fie apelată pentru acel nod (aminitiţi-vă că aceasta era condiţia procedurii de asamblare). La fiecare iteraţie a ciclului **for** elementul A[i] este dus la locul său în ansamblul creat până în acel moment.

▶▶ Construieşte–Ansamblu(A)

```
sizeA ← n
for i ← ⌊n/2⌋ to 1 do
  Asamblează(A, i)
endfor
```

Pentru transformarea unui vector într-un ansamblu se poate utiliza algoritmul Construieşte–Ansamblu(A, n).

▶▶ Creează–Ansamblu(A', n')

```
Alocă MAX_SIZE spaţiu pentru vectorul A
n ← min(MAX_SIZE, n')
Copiază n elemente din A' în A
sizeA ← 0
```

Procedura care realizează inserarea unui element nou într-un vector pe care se menţine o structură de ansamblu este dată în continuare.

►► Inserează–În–Ansamblu(A, val)

```
if sizeA = MAX_SIZE
  Overflow
else
  sizeA ← sizeA + 1
  if n < sizeA then
    n ← n+1
  endif
  i ← sizeA
  while (i > 1) and (A[Părinte(i)] < val) do
    A[i] ← A[Părinte(i)]
    i ← Părinte(i)
  endwhile
  A[i] ← val
endif
```

Putem utiliza această procedură pentru construcţia ansamblului prin inserţie.

►► Construieşte–Ansamblu–2(A)

```
sizeA ← 1
for i ← 2 to n do
  Inserează–În–Ansamblu(A, A[i])
endfor
```

Ştergerea

Ştergerea specifică ansamblului extrage rădăcina ansamblului (maximul), motiv pentru care se numeşte *decapitare a ansamblului*. Varianta prezentată utilizează procedura de asamblare.

►► Decapitează–Ansamblu(A)

```
if sizeA < 1 then
  Underflow
else
  max ← A[1]
  Interschimbă(A[1], A[sizeA])
  sizeA ← sizeA-1
  Asamblează(A, 1)
  return max
endif
```

Reprezentarea cozilor cu priorităţi (priority queues)

Ansamblele se pretează pentru implementarea cozilor cu priorităţi, deoarece ne permit efectuarea inserărilor şi ştergerilor în timp $O(\log n)$. Această variantă de reprezentare este de preferat când numărul de elemente este mare.

În funcţie de tipul ansamblului (max-ansamblu sau min-ansamblu) folosit pentru reprezentarea cozii cu priorităţi, aceasta poate avea două forme: max-coadă cu priorităţi (max-priority queue) sau min-coadă cu priorităţi (min-priority queue). În continuare, vom utiliza varianta ce foloseşte max-ansamble.

Într-un laborator anterior am prezentat cozile cu priorităţi implementate folosind structuri liniare. Tipul de coadă cu priorităţi de atunci aplica regula „cel mai mic introdus este primul extras”. Acest tip corespunde min-cozii cu priorităţi, iar o max-coadă cu priorităţi aplică regula „cel mai mare introdus este primul extras”.

Coada cu priorităţi este utilizată pentru menţinerea unei mulţimi E de elemente, ce au asociate o valoare numită *cheie* (prioritate, pondere).

Obţinerea şi extragerea maximului

Operaţia de accesare a elementului cu cheia maximă din coadă se poate face în timp $\Theta(1)$.

►► OBŢINE–MAXIM(A)

```
return A[1]
```

- Complexitate: $O(\log n)$

Procedura următoare realizează extragerea (cu ştergere) a elementului cu cheie maximă.

►► EXTRAGE–MAXIM(A)

```
if sizeA < 1 then
  UNDERFLOW
else
  max ← A[1]
  A[1] ← A[sizeA]
  sizeA ← sizeA - 1
  ASAMBLEAZĂ(A, 1)
  return max
endif
```

Incrementarea unei chei

- Complexitate: $O(\log n)$

Procedura Incrementează–Cheie(A, cheie, i) asociază celui de-al i-lea element din coadă cheia transmisă ca parametru, dacă aceasta este mai mare decât cea curentă. Indicele i este poziţia elementului în vector a cărui cheie dorim să o modificăm. Deoarece actualizarea cheii poate duce la încălcarea regulii unei max-cozi de priorităţi, trebuie să ne asigurăm că elementul se află la poziţia corectă în ansamblu. Cum cheia nu poate fi decât mai mare decât cea anterioară, rezultă că, dacă elementul $A[i]$ nu este plasat deja corect, poziţia sa finală nu poate fi decât mai sus în ansamblu (spre rădăcină). De aceea, comparăm cheia sa cu cea a părintelui cât timp părintele are o cheie mai mică şi efectuăm mereu o interschimbare.

►► Incrementează–Cheie(A, i, $cheie$)

```
if cheie < A[i] then
  Cheie nouă necorespunzătoare
else
  A[i] ← cheie
  while (i > 1) and (A[Părinte(i)] < A[i]) do
    Interschimbă(A[i], A[Părinte(i)])
    i ← Părinte(i)
  endwhile
endif
```

Inserarea

- Complexitate: $O(\log n)$

Pentru inserare folosim operaţia de mai jos, care are elementul de inserat (cu prioritatea cheie) ca parametru de intrare. Întâi se măreşte ansamblul prin adăugarea unui element nou cu prioritate minimă, apoi se apelează algoritmul de incrementare a cheii pentru acest element cu cheia transmisă, urmând ca aici să se modifice cheia la valoarea dorită şi să se „aducă” elementul în poziţia corespunzătoare.

►► Inserează–Cheie(A, $cheie$)

```
sizeA ← sizeA + 1
A[sizeA] ← −∞
Incrementează–Cheie(A, sizeA, cheie)
```

1. **(10p)** Să se implementeze un arbore binar de căutare echilibrat cu următoarele operaţii (cu echilibrare după fiecare operaţie, acolo unde este necesar):
   - **a.** `add(root, x)` – inserează cheia `x` în arborele de rădăcină `root`;
   - **b.** `search(root, x)` – returnează 1 dacă elementul `x` se află în arborele de rădăcină `root` şi 0 în caz contrar;
   - **c.** `findMax(root)` – returnează elementul maxim din arborele de rădăcină `root`, fără a-l şterge din arbore;
   - **d.** `delete(root, x)` – şterge nodul cu cheia `x` din arborele de rădăcină `root` (păstrând proprietatea de arbore binar de căutare şi, eventual, echilibrarea);
   - **e.** `print(root)` – afişează cheile din arborele de rădăcină `root`, în ordine crescătoare.

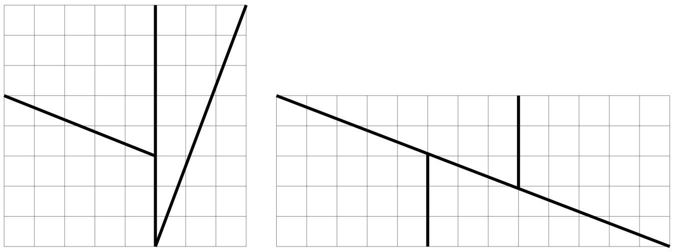
2. **(3p)** Să se scrie algoritmul pentru sortarea unui şir de numere folosind metoda heap sort. Structura de heap va fi implementată că un arbore binar într-una din cele două forme care urmează:

   (a) max - Heap – arbore binar în care fiecare nod are cheia mai mare decât oricare dintre fiii săi

   (b) min - Heap – arbore binar în care fiecare nod are cheia mai mică decât oricare dintre fiii săi

   Scrieţi funcţii pentru crearea ansamblului şi pentru decapitarea lui.

3. **(5ps)** Fiind dată o tablă de şah de 8×8 pătrate, putem să o tăiem în două trapeze şi două triunghiuri, ca în imaginea din stânga. O reasamblăm apoi după cum este indicat în figura din dreapta. Aria tablei din stânga este $8 \times 8 = 64$, pe când aria tablei din dreapta este $13 \times 5 = 65$. Explicaţi paradoxul.



   Notând cu $F(n)$ al n-lea număr Fibonacci, cum putem generaliza paradoxul? Găsiţi o relaţie între $F(n-1)$; $F(n)$ şi $F(n+1)$ pentru a explica paradoxul.

4. **(3p)** Să se implementeze o coadă cu priorităţi folosindu-se un heap. Elementele cozii vor avea doua câmpuri: `prioritate` şi `cheie`. Se vor implementa funcţii pentru următoarele operaţii:
   - **a.** `insert(q, x)` – inserează nodul `x` în coada q;
   - **b.** `findMax(q)` – returnează elementul de prioritate maximă din coada q, fără a-l şterge;
   - **c.** `extractMax(q)` – returnează elementul de prioritate maximă din coada q, eliminându-l din coadă.

■ TERMEN DE PREDARE: **Săptămâna 11 (10–14 decembrie 2012) inclusiv.**

■ DETALII: **Studenţii pot obţine un maxim de 21 puncte. Problemele 1 şi 4 sunt obligatorii. Problema 2 este suplimentară. Problema 3 este facultativă, iar termenul de predare pentru ea este săptămâna 10 (3–7 decembrie). Un singur student poate rezolva problema facultativă.**