---------- BINARY SEARCH TREE ----------

1.
Je dán BVS s n uzly. Máme za úkol spočítat hodnotu součtu všech klíčů v tomto stromě. Když to uděláme efektivně, bude složitost této operace

a) $\Theta(n \cdot \log(n))$
b) $O(\log(n))$
c) $\Theta(n^2)$
d) $O(1)$
e) $\Theta(n)$

1.
An effective algorithm computes the sum of values of all keys in the BST. The complexity of this task is

f) $\Theta(n \cdot \log(n))$
g) $O(\log(n))$
h) $\Theta(n^2)$
i) $O(1)$
j) $\Theta(n)$

1.
The complexity of the INSERT operation in a balanced BST is (*n* denotes the number of nodes in BST)

a) $\Omega(n)$
b) $O(1)$
c) $\Theta(n)$
d) $O(\log n)$

1.
The complexity of the INSERT operation in a non-balanced BST is (*n* denotes the number of nodes in BST)

a) $\Omega(1)$
b) $O(\log n)$
c) $O(1)$
d) $\Theta(n)$

1.
Kolik jednoduchých rotací se maximálně provede při jedné operaci Insert v BVS s ohraničeným vyvážením s *n* uzly?

a) jedna
b) dvě
c) log(*n*)
d) *n*

Provádějí-li se rotace vůbec, pak leda jednoduché (L, R) nebo dvojité (LR, RL). Každá dvojitá rotace je však složením dvou jednoduchých rotací. Při ohraničeném vyvážení se bere potaz pouze počet uzlů v jednotlivých podstromech. Došlo-li k rozvážení v uzlu X, pak rotace rozvážení opět napraví a počet uzlů v podstromu s kořenem X se nezmění. Nezmění se proro ani koeficient vyvážení ve vešech uzlech ležících „nad“ X, tj. na cestě z X ke kořeni celého stromu, tudíž se žádne další rotace po operaci Insert neprovedou a platí možnost b).

1.
The depth of a binary tree is 2 (root depth is 0 always). The number of leaves in the tree is:

a) minimum 0 a maximum 2
b) minimum 1 a maximum 3
c) minimum 1 a maximum 4
d) minimum 2 a maximum 4
e) unlimited

1.
Složitost operace Insert ve vyváženém BVS s *n* uzly je vždy

- o O(log(*n*))
- o O(log(hloubka BVS))
- o Θ(*n*)
- o O(1)
- ✓ Θ(log(*n*))
- o Θ(log(hloubka BVS))

2.
Složitost operace Delete v BVS s *n* uzly je vždy

- o O(log(*n*))
- o O(log(hloubka BVS))
- o Θ(*n*)
- ✓ O(*n*)
- o Θ(log(*n*))
- o Θ(log(hloubka BVS))

2a.
Podobné otázky jako v předchozí úloze pro případ operací Find, Insert, Delete ve vyvážených nebo nevyvážených stromech jen opakují fakta z přednášky, resp. okamžitý jednoduchý náhled na celou věc.

3.
Binární vyhledávací strom:

a) musí splňovat podmínku haldy *(na tohle příjdeme později)*
b) udržuje v každém uzlu referenci na uzel s nejbližším větším klíčem
c) udržuje v každém uzlu referenci na uzel s nejbližším větším i s nejbližším menším klíčem
d) po každém vložení prvku do BVS musí proběhnout vyvážení stromu
e) po průchodu v pořadí inorder vydá seřazenou posloupnost klíčů

Řešení Podmínku haldy musí splňovat pouze halda, která ani není BVS. Reference na jiné uzly než jen bezprostřední potomky nejsou v BVS povinností a vyvažování BVS také není povinné. Zbývá tak jen možnost e), o níž se hovoří i při výkladu pořadí inorder jako takového.

4.
Čísla ze zadané posloupnosti postupně vkládejte do prázdného binárního vyhledávacího stromu (BVS), který nevyvažujte. Jak bude vypadat takto vytvořený BVS?
Poté postupně odstraňte první tři prvky. Jak bude vypadat výsledný BVS?

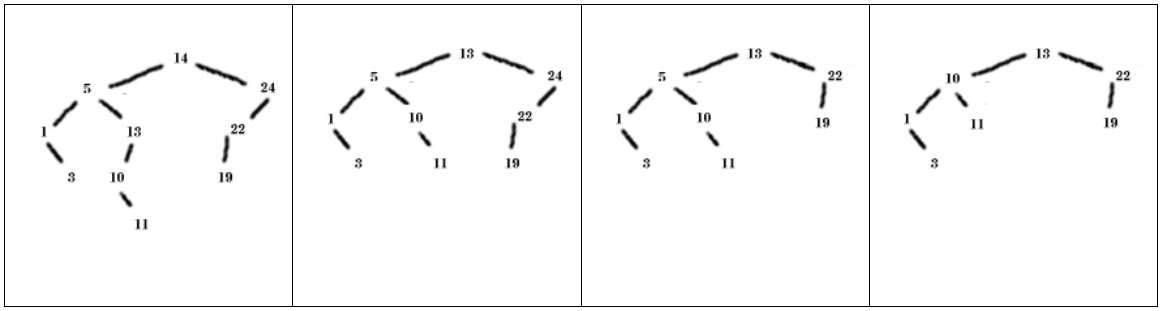
Řešení Nejprve opakování pravidel:
Při vytváření BVS je potřeba dodržet jednoduché pravidlo: v levém podstromu každého vnitřního uzlu BVS (kořeni podstromu) jsou klíče s menší hodnotou, v pravém podstromu zase klíče s větší hodnotou, než má kořen podstromu.
Nový prvek vkládáme vždy odshora. Začneme porovnáním s kořenem stromu a podle výše zmíněného pravidla postupujeme vlevo, když je klíč vkládaného prvku menší, nebo vpravo, když je větší, než klíč kořene. To opakujeme tak dlouho, dokud je ve zvoleném směru nějaký uzel. V okamžiku, když už nemůžeme dál pokračovat, vložíme nový uzel tam, kam by vedl další postup. Pamatujte si, že nově vkládaný prvek je vždy listem.

Odstranění prvku z BVS není už tak jednoduché. Zatímco při vkládání vznikne vždy list, odebírání se týká všech uzlů BVS. V případě, že je rušený prvek uvnitř stromu, musíme jeho podstromy znovu „zapojit“ do BVS tak, aby výsledkem byl zase BVS.

1. Odstranění prvku na pozici listu je jednoduché. Prostě jej vypustíme.
2. Pokud má odstraňovaný prvek jeden podstrom, stačí podstrom napojit na rodiče rušeného prvku.
3. Nejtěžší případ nastane při rušení prvku, který má oba podstromy. Místo manipulace s podstromy nahradíme rušený prvek vhodným kandidátem vybraným z jednoho z podstromů. Vzhledem k tomu, že i po zrušení prvku musí být strom stále BVS, je potřeba zajistit, aby nově vložený prvek byl větší, než zbylé prvky levého podstromu a menší, než zbylé prvky pravého podstromu. Tuto podmínku splňují 2 kandidáti: největší prvek z levého podstromu, nebo nejmenší prvek z pravého podstromu. Je jedno, který z nich si vybereme.
   1. Najdeme největší prvek v levém podstromu. U toho je zaručeno, že určitě sám nemá pravý podstrom (jinak by nebyl největší) a proto nebude problém jej pomocí postupu z bodů 1, nebo 2 odstranit.
   2. Hodnotu z tohoto prvku vložíme do rušeného uzlu.

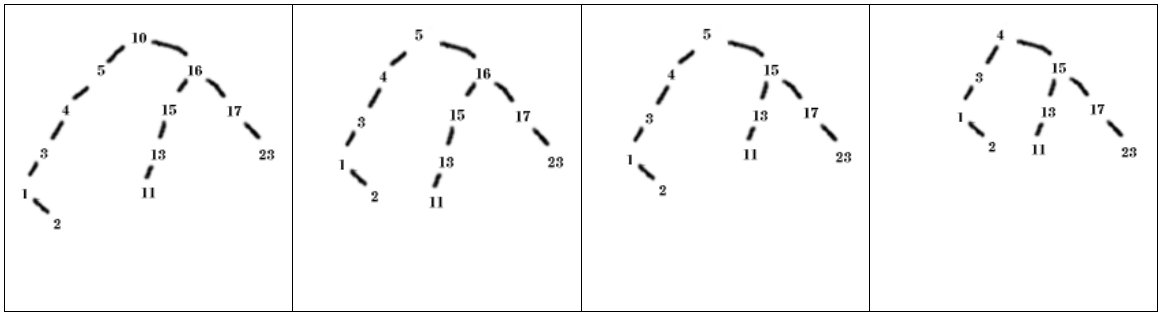
Posloupnost a)

14 24 5 13 1 3 22 10 19 11



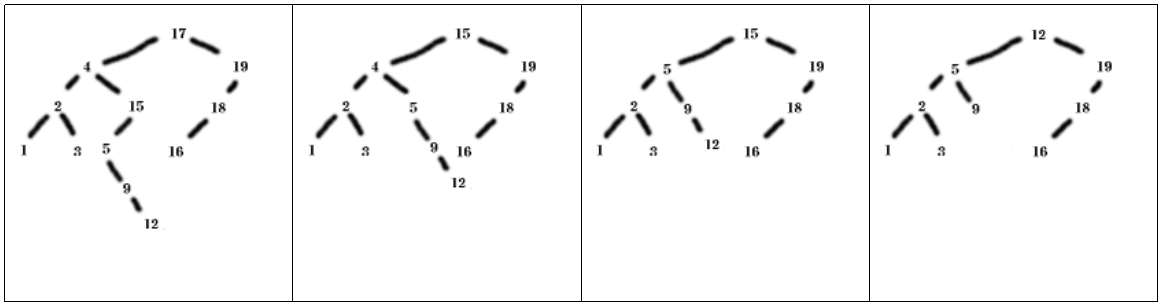
Posloupnost b)

10 16 5 17 4 15 3 1 23 13 2 11



Posloupnost c)

17 4 15 2 5 9 1 12 3 19 16 18



1.

Mějme klíče 1, 2, 3, ..., *n*. Číslo *n* je liché. Nejprve vložíme do BVS všechny sudé klíče v rostoucím pořadí a pak všechny liché klíče také v rostoucím pořadí. Jaká bude hloubka výsledného stromu? Změnil by se nějak tvar stromu, kdybychom lichá čísla vkládali v náhodném pořadí?

Řešení Všechny sudé klíče vytvoří jedinou dlouhou větev doprava z kořene. Uzly s lichými klíči se budou jednotlivě navěšovat vlevo od uzlů se sudými klíčí o jedna většími, vznikne pravidelný maximálně nevyvážený strom. na pořadí vkládaných lichých klíčů nezáleží.

12.
Uzel binárního binárního vyhledávacího stromu obsahuje tři složky: Klíč a ukazatele na pravého a levého potomka. Navrhněte rekurzívní funkci (vracející `bool`), která porovná, zda má dvojice stromů stejnou strukturu. Dva BVS považujeme za strukturně stejné, pokud se dají nakreslit tak, že po položení na sebe pozorovateli splývají, bez ohledu na to, jaká obsahují data.

Řešení Z uvedeného popisu by měl být zřejmý následující rekurzvní vztah: Dva binární stromy A a B (ani nemusí být vyhledávací) jsou strukturně stejné, pokud

a) jsou buď oba prázdné
b) levý podstrom kořene A je strukturně stejný jako levý podstrom kořene B a zároveň také pravý podstrom kořene A je strukturně stejný jako pravý podstrom kořene B.

Toto zjištění již snadno přepíšeme do programovacího jazyka (zde jen pseudokódu):

```
boolean stejnaStruktura(node root1, node root2) {
  if ((root1 == null) && (root2 == null)) return true;
  if ((root1 == null) || (root2 == null)) return false;
  return (stejnaStruktura(root1.left, root2.left) &&
          stejnaStruktura(root1.right, root2.right));
}
```

13.
Upravte sami (velmi snadno) předchozí funkci tak, aby zjistila, zda jsou dva BVS shodné, t.j. zda se shodují strukturou i svými daty.

Řešení Samostatně.

14.
Napište rekurzivní verze operací: TreeMinimum, která vrátí referenci na uzel s nejmenší hodnotou v BVS.

Řešení Stručná nápověda:

TreeMinimum(tree): Konec rekurze – tree->left == null. Pokud není, zavolej TreeMinimum(tree->left). Napřed ještě překontrolovat tree == null.

16.
Je dána struktura popisující uzel binárního vyhledávacího stromu takto

```
struct node
{
      valType val;
      node * left, right;
      int count;
}
```

Nebo v Javě takto

```
class Node {
   valType val;    // na tom, co je valType, v tomto případě nesejde
   Node left;
   Node right;
   int count;
}
```

Navrhněte nerekurzívní proceduru, která do složky count v každém uzlu zapíše počet vnitřních uzlů v podstromu, jehož kořenem je tento uzel (včetně tohoto uzlu, pokud sám není listem).

Řešení Musíme volit průchod v pořadí postorder, protože počet vnitřních uzlů v daném stromu zjistíme tak, že sečteme počet vnitřních uzlů v levém podstromu a pravém podstromu kořene a nakonec přičteme jedničku za samotný kořen.
Nepotřebujeme tedy nic jiného než procházet stromem a před definitivním opuštěním uzlu sečíst počet vnitřních uzlů v jeho levém a pravém podstromu, pokud existují. Pokud neexistuje ani jeden, pak je uzel listem a registrujeme v něm nulu.
Na zásobník budeme s každým uzlem, který ještě budeme zpracovávat, ukládat také počet návštěv, které jsme již v něm vykonali. Po třetí návštěve již uzel ze zásobníku definitivně vyjmeme – v implementaci to znamená, že již jej zpět do zásobníku nevložíme.

```
void pocetVnitrUzlu(node root) {
if root == null return
stack.init();
stack.push(root, 0);
while (stack.empty() == false) {
   (aktUzel, navstev) = stack.pop();
   // pokud je uzel listem:
   if (aktUzel.left == null) && (aktUzel.right == null))
       aktUzel.count = 0;
   // pokud uzel neni listem:
   else {
      if(navstev == 0) {
         stack.push(aktUzel, 1);  // uz jsme v akt uzlu byli jednou
         if (aktUzel.left != null) stack.push(aktUzel.left, 0);
      }
      if(navstev == 1) {
          stack.push(aktUzel, 2); // uz jsme v akt uzlu byli dvakrat
          if (aktUzel.right != null) stack.push(aktUzel.right, 0);
      }
      if(navstev == 2) {     // ted jsme v akt uzlu potreti
         aktUzel.count = 1;  // akt uzel je vnitrnim uzlem stromu
         if (aktUzel.left != null) aktUzel.count += aktUzel.left.count;
         if (aktUzel.right != null) aktUzel.count += aktUzel.right.count;
      }
   }
} // end of while
}
```

18.
Napište funkci, jejímž vstupem bude ukazatel (=reference) na uzel X v BVS a výstupem ukazatel (=reference) na uzel s nejbližší vyšší hodnotou ve stromu.

Řešení Celkem může nastat v uzlu X několik případů.
A) X má pravého potomka.
Pak je hledaným uzlem nejlevější uzel v pravém podstromu X.
B) X nemá pravého potomka ani žádného rodiče.
Pak X je kořenem bez pravého pdstromu a hledaný uzel neexistuje.
C) X nemá pravého potomka a je levým potomkem svého rodiče Y.
Pak buď Y nemá pravý podstrom je tedy sám hledaným uzlem nebo Y pravý podstrom má a hledaným uzlem je tedy nejlevější uzel v tomto pravém podstromu.
D) X nemá pravého potomka a je pravým potomkem svého rodiče Y.
Pak budeme postupovat od X přes Y směrem doleva nahoru a pokud nalezneme uzel Y2, který je levým potomkem svého rodiče, aplikujeme na uzel Y2 postup popsaný v bodě C) pro uzel X. Pokud takový uzel nenajdeme, pak ani neexistuje a uzel X je nejpravějším uzlem celého stromu.

```
uzel nejvicLvPpodStromu(uzel x) {
   x = x.right;
   while (x.left != null) x = x.left;
   return x;
```

```
}

uzel neblizsiVetsi(uzel x) {
  // varianta A)
  if (x.right != null) return nejvicLvPpodStromu(x);

  // varianta B)
  if (x.parent == null) return null;

  // varianta C)
  if (x.parent.left == x) {
    if (x.parent.right == null) return x.parent;
    else return nejvicLvPpodStromu(x.parent);
  }

  // varianta D)
  x = x.parent;
  while (x.parent != null) {
      if (x.parent.left ==  x)  {
         if (x.parent.right == null) return x.parent;
         else return nejvicLvPpodStromu(x.parent);
      }
     x = x.parent;
  }
  return null;
}
```

Při výpisu hodnot uzlů BVS v pořadí inorder získáme uspořádanou posloupnost hodnot. Napište nerekurzivní funkci, která v každém uzlu daného BVS zamění levý a pravý podstrom. (Po této úpravě bude strom „zrcadlovým obrazem“ původního a výpisem v pořadí inorder bychom získali opačně uspořádanou posloupnost.)

Úpravu stromu můžeme provést v pořadí postorder i v pořadí preorder. Pořadí preorder se ale snáz implementuje, pokud je nerekurzivní, protože na násobník není nutno ukládat počet návštěv konkrétního uzlu. Zachyceno pseudokódem:

```
stack.init();
stack.push(strom.root);
while (stack.empty() == false) {
  aktUzel = stack.pop;
  if (aktUzel !=  null) {
     stack.push(aktUzel.right);
     stack.push(aktUzel.left);

     // cele zpracovani:
     pomocnyUk = aktUzel.left;
     aktUzel.left = aktUzel.right;
     aktUzel.right = pomocnyUk;
  }
// hotovo
```

19.
Navrhněte algoritmus, který spojí dva BVS *A* a *B*. Spojení proběhne tak, že všechny uzly z *B* budou přesunuty do *A*, přičemž se nebudou vytvářet žádné nové uzly ani se nebudou žádné uzly mazat. Přesun proběhne jen manipulací s ukazateli. Předpokládejte, že v každém uzlu v *A* i v *B* je k dispozici ukazatel na rodičovský uzel.

Řešení Základem je procházení stromem *B* v pořadí Postorder. Tím zajistíme, že se uzly v *B* budou zpracovávat v pořadí „zdola“, resp. v okamžiku zpracovávání uzlu, to jest v okamžiku jeho přesunu do *A*, budou již všechni jeho bývalí potomci již také přesunuti do *A*. Samotný přesun uzlu *x* z *A* do *B* pak znamená pouze nalezení místa pro *x* ve stromu *A* stejně jako v operaci Insert a následné přepsání nejvýše tří referencí: z *x* na nového rodiče v *A*, z tohoto nového rodiče zpět na *x* a zrušení reference z původního rodiče *x* v B (pokud *x* není kořenem *B*) na uzel *x*.

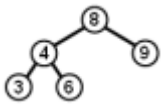
1.
Napište funkci, která obrátí pořadí prvků v binárním vyhledávacím stromu. Obrácené pořadí znamená, že po výpisu v pořadí inorder (který neimplementujte!), budou prvky srovnány od největšího k nejmenšímu.

1.
Datový typ popisující uzel binárního vyhledávacího stromu obsahuje klíč, ukazatel na levého potomka, ukazatel na pravého potomka a celočíselnou složku count. Navrhněte nerekurzívní proceduru, která do složky count v každém uzlu daného stromu zapíše počet listů v podstromu, jehož kořenem je tento uzel (včetně tohoto uzlu, je-li sám listem).

---------- AVL TREE ----------

1.
Do AVL stromu na obrázku vložíme klíč s hodnotou 5. Přitom musíme provést

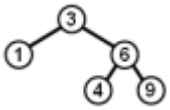
a) jednu R rotaci
b) jednu L rotaci
c) jednu LR rotaci
d) jednu RL rotaci
e) žádnou rotaci



2.

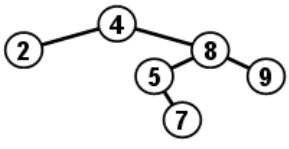
Do AVL stromu na obrázku vložíme klíč s hodnotou 7. Přitom musíme provést

a) jednu R rotaci
b) jednu L rotaci
c) jednu LR rotaci
d) jednu RL rotaci
e) žádnou rotaci



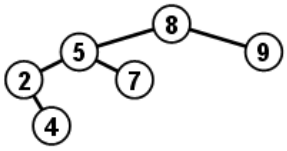
3.

Daný AVL strom není vyvážený. Po provedení správné rotace se změní tvar stromu a potom bude pravý podstrom kořene obsahovat uzly s klíčí:



a) 4 5 7 8 9
b) 5 7 8 9
c) 7 8 9
d) 8 9
e) 9

4.
Daný AVL strom není vyvážený. Po provedení správné rotace se změní tvar stromu a potom bude levý podstrom kořene obsahovat uzly s klíči:



a) žádný uzel
b) 2
c) 2 4
d) 2 4 5
e) 2 4 5 6

5.
V AVL stromu se algoritmus pro vyvažování musí rozhodnout, kterou z rotací pro vyvážení stromu použije po vložení uzlu E. Má k dispozici pouze hodnoty rozvážení ve všech uzlech, tj. vnitřních uzlech i listech. Musí se rozhodnout správně.

a) Použije levou rotaci v uzlu B, protože je nalevo od F a hodnota rozvážení je -1
b) Použije levou rotaci v uzlu B následovanou pravou rotací v uzlu F, protože rozvážení je v uzlu F rovno hodnotě 2 a rozvážení v B je záporné
c) Použije pravou rotaci v kořeni F, protože rozvážení je v tomto uzlu rovno hodnotě 2
d) Vůbec rotovat nemusí, protože strom vyhovuje kritériím AVL stromu

6.
V AVL stromu se algoritmus pro vyvažování musí rozhodnout, kterou z rotací pro vyvážení stromu použije po vložení uzlu C. Má k dispozici pouze hodnoty rozvážení ve všech uzlech, tj. vnitřních uzlech i listech. Musí se rozhodnout správně.

a) Použije pravou rotaci v kořeni F, protože rozvážení je v tomto uzlu rovno hodnotě 2
b) Použije levou rotaci v uzlu B, protože je nalevo od F a hodnota rozvážení je -1
c) Vůbec rotovat nemusí, protože strom vyhovuje kritériím AVL stromu
d) Použije LR rotaci v uzlu F, protože rozvážení je v tomto uzlu rovno hodnotě 2 a rozvážení v B je záporné.

7.
Jednoduchá levá rotace v uzlu *u* má operační složitost
a) závislou na výšce levého podstromu uzlu *u*
b) konstantní
c) mezi $O(1)$ a $\Omega(n)$
d) závislou na hloubce uzlu *u*

Zde je nutno vědět, jak přibližně rotace vypadá. Mění jen několik ukazatelů na uzly stromu. Varianty a) a d) neplatí, varianta c) je nesmyslná sama o sobě, průnik množin $O(1)$ a $\Omega(n)$ je prázdný. Platí samozřejmě b).

8.
Jednoduchá pravá rotace v uzlu *u* má operační složitost
a) $\Theta(1)$
b) závislou na hloubce uzlu *u*
c) mezi $O(1)$ a $\Omega(\log n)$
d) $\Omega(n)$

Zde je nutno vědět, jak přibližně rotace vypadá. Mění jen několik ukazatelů na uzly stromu. Intuitivně by mělo pak být jasné, že má složitost konstantní. Varianty b) a d) v takovém případě neplatí.

Varianta c) tvrdí, že dotyčná složitost je mezi O(1) a Ω (log *n*), což znamená, že sice roste pomaleji než log(*n*), ale zároveň rychleji než konstanta (která „neroste vůbec“). Platí ovšem a).

9.
Rotace ve vyváženém binárním vyhledávacím stromě

a) se provádí vždy, když je výška jednoho z podstromů váhově vyváženého stromu alespoň o dvě vyšší než ve druhém podstromu
b) se provádí při každém vkládání nového uzlu
c) je invariantní vůči procházení stromem v pořadí *inorder*
d) se provádí pouze při vkládání levého syna v pravém podstromu a pravého syna v levém podstromu

10.
Left rotation in node *u* (in the sub-tree with the root node *u*)
a) moves the right sun u_R of the root node *u* to the root. Node *u* becomes the left son of the node u_R and the left sub-tree of u_R becomes the right sub-tree of the node *u*.
b) moves the left sun u_L of the root node *u* to the root. Node *u* becomes the right son of the node u_L and the left sub-tree of u_L becomes the right sub-tree of the node *u*.
c) moves the right sun u_R of the root node *u* to the root. Node *u* becomes the left son of the node u_R and the right sub-tree of u_R becomes the left sub-tree of the node *u*.
d) moves the left sun u_L of the root node *u* to the root. Node *u* becomes the right son of the node u_L and the right sub-tree of u_L becomes the left sub-tree of the node *u*.

11.
Right rotation in node *u* (in the sub-tree with the root node *u*)
a) moves the right sun u_R of the root node *u* to the root. Node *u* becomes the left son of the node u_R and the left sub-tree of u_R becomes the right sub-tree of the node *u*.
b) moves the left sun u_L of the root node *u* to the root. Node *u* becomes the right son of the node u_L and the left sub-tree of u_L becomes the right sub-tree of the node *u*.
c) moves the right sun u_R of the root node *u* to the root. Node *u* becomes the left son of the node u_R and the right sub-tree of u_R becomes the left sub-tree of the node *u*.
d) moves the left sun u_L of the root node *u* to the root. Node *u* becomes the right son of the node u_L and the right sub-tree of u_L becomes the left sub-tree of the node *u*.

12.
The keys 40 20 10 30 were inserted one after another into an originally empty AVL tree. The process resulted in

a) one R rotation
b) one L rotation
c) one RL rotation
d) one LR rotation
e) no rotation at all

13.
The keys 10 30 20 40 were inserted one after another into an originally empty AVL tree. The process resulted in

f) one R rotation
g) one L rotation
h) one RL rotation
i) one LR rotation
j) no rotation at all

14.
Simple right rotation in node *u*

a) decreases the depth of the right son of *u*
b) swaps the contents of node *u* with the contents of its left son
c) decreases the depth of the left son of *u*
d) moves the inner node left from the right son of *u* to the root

15.
Simple left rotation in node *u*

a) decreases the depth of the left son of *u*
b) swaps the contents of node *u* with the contents of its left son
c) moves the inner node left from the right son of *u* to the root
d) decreases the depth of the right son of *u*

16.
Jednoduchá levá rotace v uzlu *u* má operační složitost
a) závislou na výšce levého podstromu uzlu *u*
b) mezi O(1) a $\Theta(n)$
c) závislou na hloubce uzlu *u*
d) konstantní

Řešení Provedení kterékoli rotace nezávisí na poloze uzlu ve stromu a vyžaduje jen změnu ukazatelů na předem známý počet uzlů – nejvýše 3 v jednoduché rotaci (malujte si obrázek) případně o jeden více uvažujeme-li i ukazatele na rodiče uzlu, v němž rotace probíhá. To ovšem znamená, že se jedná o konstantní složitost – varianta d).

17a.
LR rotace v uzlu u lze rozložit na
a) levou rotaci v pravém synovi uzlu u následovanou pravou rotací v uzlu u
b) pravou rotaci v pravém synovi uzlu u následovanou levou rotací v uzlu u
c) levou rotaci v levém synovi uzlu u následovanou pravou rotací v uzlu u
d) pravou rotaci v levém synovi uzlu u následovanou levou rotací v uzlu u

Řešení Namalujte si tři obrázky. Nejprve (dostatečně velký) binarní strom a pojmenujte(!) v něm uzly. Na druhém obrázku bude tento strom po LR rotaci v kořeni. Na třetím obrázku nejprve původní strom překreslete po L rotaci v levém následníku kořene a tento naposled jmenovaný strom pak ještě po R rotaci v kořeni. Výsledek třetího obrázku je shodný se druhým obrázkem, takže třetí možnost je správně.

17b.
RL rotace v uzlu u lze rozložit na
a) levou rotaci v pravém synovi uzlu u následovanou pravou rotací v uzlu u
b) pravou rotaci v pravém synovi uzlu u následovanou levou rotací v uzlu u
c) levou rotaci v levém synovi uzlu u následovanou pravou rotací v uzlu u
d) pravou rotaci v levém synovi uzlu u následovanou levou rotací v uzlu u

Řešení Namalujte si tři obrázky. Nejprve (dostatečně velký) binarní strom a pojmenujte(!) v něm uzly. Na druhém obrázku bude tento strom po RL rotaci v kořeni. Na třetím obrázku nejprve původní strom překreslete po R rotaci v pravém následníku kořene a tento naposled jmenovaný strom pak ještě po L rotaci v kořeni. Výsledek třetího obrázku je shodný se druhým obrázkem, takže druhá možnost je správně.

18.
Kolik jednoduchých rotací se maximálně provede při jedné operaci Insert v AVL stromu s *n* uzly?
e) jedna
f) dvě
g) log(*n*)
h) *n*

Řešení Provádějí-li se rotace vůbec, pak leda jednoduché (L, R) nebo dvojité (LR, RL). Každá dvojitá rotace je však složením dvou jednoduchých rotací. Pokud došlo k rozvážení v uzlu X, platí toto: Měl-li podstrom s kořenem v X před operací insert hloubku h, pak po operaci insert stoupla hloubka na h+1 (protože X je najednou rozvážený), ale rotace opět tuto hloubku sníží na původní h (malujte si obrázky!). Protože tedy operace insert a následná rotace v X nezměnila hloubku podstromu s kořenem v X, nemohlo ani dojít k rozvážení uzlů kteří leží „nad" X, tj. na cestě z X ke kořeni celého stromu, tudíž se žádne další rotace po operaci Insert neprovedou a platí možnost b).

19.
LR rotace v uzlu *u* má operační složitost
a) závislou na hloubce uzlu *u*
b) Θ (log *n*)
c) konstantní
d) lineární

Řešení Provedení kterékoli rotace nezávisí na poloze uzlu ve stromu a vyžaduje jen změnu ukazatelů na předem známý počet uzlů – nejvýše 6 ve dvojité rotaci (malujte si obrázek) případně o jeden více uvažujeme-li i ukazatele na rodiče uzlu, v němž rotace probíhá. To ovšem znamená, že se jedná o konstantní složitost – varianta c).

20a.
Levá rotace v uzlu *u*
a) v podstromu s kořenem *u* přemístí pravého syna u_p uzlu *u* do kořene. Přitom se uzel *u* stane levým synem uzlu u_p a levý podstrom uzlu u_p se stane pravým podstromem uzlu *u*
b) v podstromu s kořenem *u* přemístí levého syna u_l uzlu *u* do kořene. Přitom se uzel *u* stane pravým synem uzlu u_l a levý podstrom uzlu u_l se stane pravým podstromem uzlu *u*
c) v podstromu s kořenem *u* přemístí pravého syna u_p uzlu *u* do kořene. Přitom se uzel *u* stane levým synem uzlu u_p a pravý podstrom uzlu u_p se stane levým podstromem uzlu *u*
d) v podstromu s kořenem *u* přemístí levého syna u_l uzlu *u* do kořene. Přitom se uzel *u* stane pravým synem uzlu u_l a pravý podstrom uzlu u_l se stane levým podstromem uzlu *u*

Řešení Zde můžeme jen doporučit nakreslení obrázku, případně nahlédnutí do vhodné publikace, možnost a) se ukáže být jedinou správnou.

20b.
Pravá rotace v uzlu *u*
a) v podstromu s kořenem *u* přemístí pravého syna u_p uzlu *u* do kořene. Přitom se uzel *u* stane levým synem uzlu u_p a levý podstrom uzlu u_p se stane pravým podstromem uzlu *u*
b) v podstromu s kořenem *u* přemístí levého syna u_l uzlu *u* do kořene. Přitom se uzel *u* stane pravým synem uzlu u_l a levý podstrom uzlu u_l se stane pravým podstromem uzlu *u*
c) v podstromu s kořenem *u* přemístí pravého syna u_p uzlu *u* do kořene. Přitom se uzel *u* stane levým synem uzlu u_p a pravý podstrom uzlu u_p se stane levým podstromem uzlu *u*
d) v podstromu s kořenem *u* přemístí levého syna u_l uzlu *u* do kořene. Přitom se uzel *u* stane pravým synem uzlu u_l a pravý podstrom uzlu u_l se stane levým podstromem uzlu *u*

Řešení Zde můžeme – stejně jako ve vedlejším oddělení – jen doporučit nakreslení obrázku, případně nahlédnutí do vhodné publikace, možnost d) se ukáže být jedinou správnou.

21.
Vyvážení BVS některou z operací rotace
a) je nutno provést po každé operaci Insert
b) je nutno provést po každé operaci Delete
c) je nutno provést po každé operaci Insert i Delete
d) snižuje hloubku stromu
e) není pro udržování BVS nezbytné

Řešení Binární vyhledávací stromy se vyvažovat mohou a také nemusí. Možnosti a) b) c) tedy opadají a zřejmě platí možnost e). Možnost d) neplatí, protože rotace nemusí snížit hloubku stromu, je-li provedena dostatečně vysoko ve stromu v některé z kratších větví (jednoduché cvičení – najděte na to příklad, budete potřebovat pět pater stromu).

22.
Napište proceduru *strom leváRotace(strom s).* Napřed si namalujte obrázek a rozmyslete si, kterých ukazatelů se změna dotkne.

Řešení

```
Tree leftRotation( Tree root ) {
   if( root == null ) return root;
   Tree p1 = root.right;              // (init)
   if (p1 == null) return root;
   root.right = p1.left;
   p1.left    = root;
   return p1;
}
```

Dtto pro LR rotaci. Domyslete, kde chybí testy typu if(root==null) return null;

Řešení

```
Tree leftRightRotation( Tree root ) {
   Tree p1 = root.left;
   Tree p2 = p1.right;

   root.left = p2.right;
   p2.right = root;
   p1.right = p2.left;
   p2.left = p1;
   return p2;
}
```