1.
Operace INSERT ve vyváženém BVS s *n* uzly má operační složitost

- o $\Omega(n)$
- ✓ $O(\log n)$
- o $O(1)$
- o $\Theta(n)$

Operace insert při svém průchodu stromem se v každém uzlu zdrží nanejvýš konstantní dobu, hloubka vyváženého stromu je úměrná $\log_2(n)$, celkem tedy je zřejmě správně druhá možnost.

2.
Operace DELETE ve vyváženém BVS s *n* uzly má operační složitost

- o $\Omega(n)$
- ✓ $O(\log n)$
- o $\Omega(1)$
- o $\Theta(n)$

Operace Delete při svém průchodu stromem se v každém uzlu zdrží nanejvýš konstantní dobu, hloubka vyváženého stromu je úměrná $\log_2(n)$, celkem tedy je zřejmě správně druhá možnost. Třetí možnost také nelze, ale také neříká vůbec nic, každá operace kdekoli má složitost $\Omega(1)$, snad tu mělo být napsáno alespoň $\Omega(\text{sqrt}(-1))$?.

3.
Operace DELETE v nevyváženém BVS má operační složitost (*n* je počet uzlů ve stromě)

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

Maximální možná hloubka nevyváženého BVS je s n uzly je n. Každá operace, která při svém provádění postupuje od kořene k listům (listu) a v každém uzlu stráví čas nejvýše konstantní, má asymptotickou složitost O(n). To je v nevyváženém BVS i případ operace Delete. Platí varianta a).

4.
Operace INSERT v nevyváženém BVS má operační složitost (n je počet uzlů ve stromě)

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

Maximální možná hloubka nevyváženého BVS je s n uzly je n. Každá operace, která při svém provádění postupuje od kořene k listům (listu) a v každém uzlu stráví čas nejvýše konstantní, má asymptotickou složitost O(n). To je v nevyváženém BVS i případ operace Inser. Platí varianta d).

5.
Klíče daného binárního vyhledávacího stromu vypíšeme v pořadí preorder.
Vznikne posloupnost 4 1 8 5 9. Celkový počet uzlů v pravém podstromu kořene tohoto BVS je:

a) 0
b) 1
c) 2
d) 3
e) 4

Ve výpisu v pořadí preorder je vždy na prvním místě kořen stromu. Kořen stromu tedy v našem případě obsahuje klíč s hodnotou 4. Náš strom je ale také vyhledávací, tudíž všechny hodnoty větší než 4 leží v pravém podstromu kořene. Tyto hodnoty jsou 8, 5, 9 a jsou celkem tři. Platí tedy varianta d).

6.
Klíče daného binárního vyhledávacího stromu vypíšeme v pořadí postorder.
Vznikne posloupnost 4 3 7 9 6. Celkový počet uzlů v levém podstromu kořene tohoto BVS je:

a) 0
b) 1
c) 2
d) 3
e) 4

Ve výpisu v pořadí postorder je vždy na posledním místě kořen stromu. Kořen stromu tedy v našem případě obsahuje klíč s hodnotou 6. Náš strom je ale také vyhledávací, tudíž všechny hodnoty menší než 6 leží v levém podstromu kořene. Tyto hodnoty jsou 4 a 3 a jsou celkem dvě. Platí tedy varianta c).

7.
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
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.

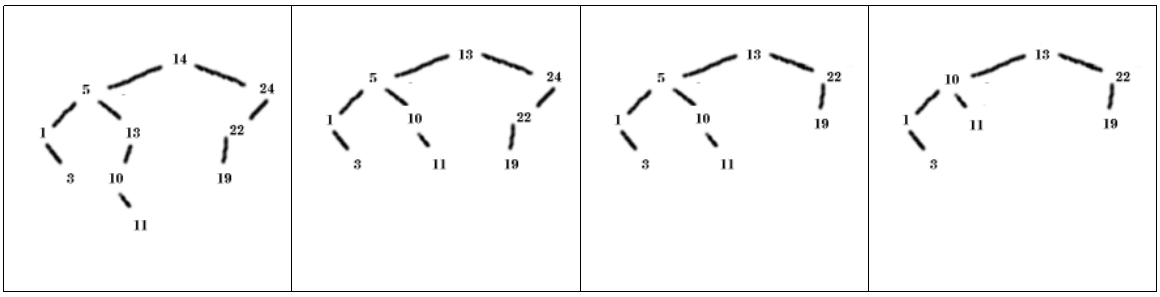
1. Odstranění prvku na pozici listu je jednoduché. Prostě jej vypustíme.
2. Pokud má odstraňovaný prvek jeden podstrom, stačí podstrom napojit na rodiče rušeného prvku.
3. Nejtěžší případ nastane při rušení prvku, který má oba podstromy. Místo manipulace s podstromy nahradíme rušený prvek vhodným kandidátem 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 (tj. předchůdce mazaného prvku). 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. Úplně stejně to funguje, najdeme-li již zmíněného následníka prvku, tj. nejmenší prvek v pravém podstromu.
2. Hodnotu z tohoto prvku vložíme do rušeného uzlu.

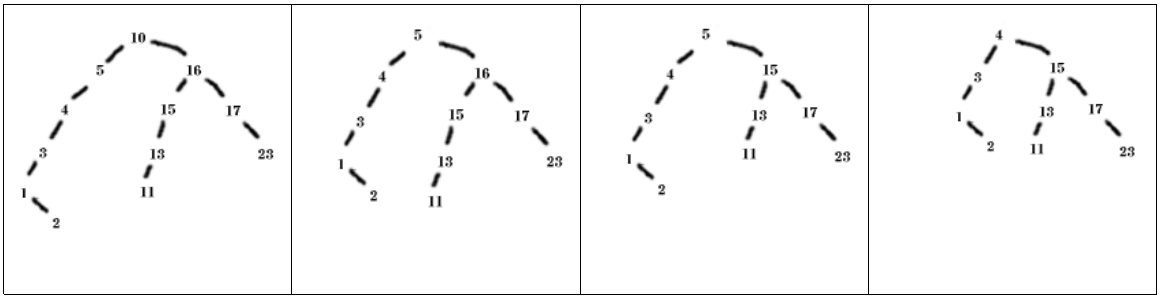
Posloupnost a)

14 24 5 13 1 3 22 10 19 11



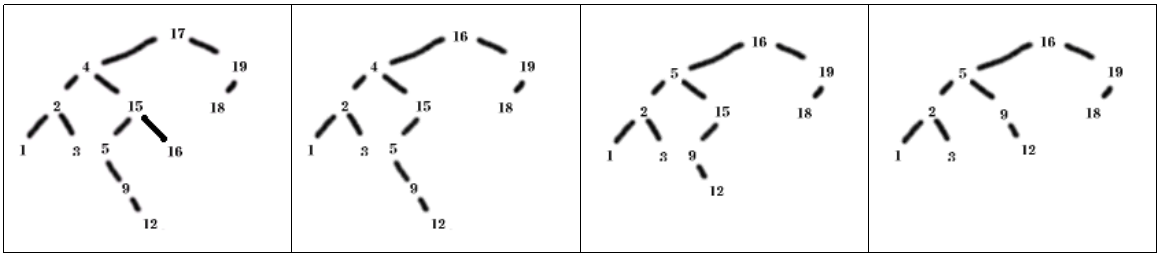
Posloupnost b)

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



8.

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

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 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 sám hledaným uzlem nebo pravý podstrom má a pak je hledaným uzlem nejlevější uzel v jeho 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;
}
```

9.
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
```

10.

Je dána struktura popisující uzel binárního vyhledávacího stromu takto

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

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

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
}
```

11.

Je dána struktura popisující uzel binárního vyhledávacího stromu takto

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

navrhněte nerekurzívní proceduru, která do proměnné count v každém uzlu zapíše počet listů v podstromu, jehož kořenem je tento uzel (včetně tohoto uzlu, je-li sám listem).

Musíme volit průchod v pořadí postorder, protože počet listů v daném stromu zjistíme tak, že sečteme počet listů v levém podstromu a pravém podstromu kořene.
Nepotřebujeme tedy nic jiného než procházet stromem a před definitivním opuštěním uzlu sečíst počet listů v jeho levém a pravém podstromu, pokud existují. Pokud neexistuje ani jeden, pak je uzel listem a registrujeme v něm jedničku.
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 pocetListu(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 = 1;
   // 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 = 0;
         if (aktUzel.left != null) aktUzel.count +=
aktUzel.left.count;
         if (aktUzel.right != null) aktUzel.count +=
aktUzel.right.count;
      }
   }
} // end of while
}
```

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 stromy považujeme za strukturně stejné, pokud se dají nakreslit tak, že po položení na sebe pozorovateli splývají.

Z uvedeného popisu by měl být zřejmý následující rekurzvní vztah: Dva binární stromy (ani nemusí být vyhledávací) A a B 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.
Uzel binárního vyhledávacího stromu obsahuje čtyři složky: Klíč, celočíselnou hodnotu **`count`** a ukazatele na pravého a levého potomka.
Navrhněte nerekurzívní proceduru, která naplní zadané pole hist[0,maxInt] počtem výskytů hodnot v proměnné **`count`** (histogram hodnot uložených v **`count`**).

Není v zásadě zapotřebí nic jiného, než projít stromem a v každém uzlu provést operaci: hist[aktUzel.count]++;
Průchod stromem zvolíme v pořadí preorder, protože to se nejsnáze implementuje nerekurzivně. Na zásobník stačí po zpracování aktuálního uzlu ukládat pouze pravého a levého potomka (pokud existují).

Celý pseudokód by pak mohl vypadat asi takto:

```
void histogram(node root, int [ ] hist) {
NaplnPoleNulami(hist);
```

```
if (root == null) return;
stack.init;
stack.push(root);
while (stack.empty() == false) {
  aktUzel = stack.pop();
  hist[aktUzel.count]++;
  if (aktUzel.right != null) stack.push(aktUzel.right);
  if (aktUzel.left != null) stack.push(aktUzel.left);
  }
}
```

14.

Navrhněte algoritmus, který spojí dva BVS: A a B. Spojení proběhne tak, že všechny uzly z B budou přesunuty do A na patřičné místo, 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.

2.

Budeme procházet stromem B a ukazatel na každý uzel X, který najdeme, předáme funkci moveToA(node RootA, node X).

Volání funkce moveToA(rootA, X) udělá následující:

-- najde místo ve stromu A pro uzel X (stejně jako to dělá std. operace Insert),
tj. najde v A uzel R, který bude rodičem X v A.

-- přesune uzel X ze stromu B do stromu A pod uzel R pouze manipulací s ukazateli.

Přesunutím uzlu X z A do B ovšem ztratí strom B veškerou referenci na tento uzel a tím také na jeho případné potomky v B. Aby tedy nedošlo ke ztrátě informace, musíme uzel X vybírat vždy tak, aby sám neměl žádné potomky.

To lze naštěstí zajistit snadno, protože procházení stromu v pořadí **postorder** má právě tu vlastnost, že zpracovává strom počínaje listy. Zpracování listů v naší úloze ale znamená odstranění listů. Máme tedy pořadím postorder zaručeno (malujte si obrázek!), že kdykoli při procházení stromu B v tomto pořadí začneme zpracovávat uzel, nebude mít již tento uzel žádné potomky.

Celý algoritmus pak vypadá například takto:

```
void presunVse(node rootA, node nodeB) { // nodeB je aktuální uzel v B
    // presunVse má vlastnosti procházení postorder:
    if (nodeB == NULL) return;
      presunVse(rootA, nodeB.left);
      presunVse(rootA, nodeB.right);
      moveToA(rootA, nodeB);
}

node moveToA(node rootA, node X); {
/*
1.     najdi místo ve stromu A pro klíč uzel s hodnotou klíče rootB.key
       stejně jako v operaci Insert. Rodičem nového uzlu ať je uzel R.
2.     proveď přesun uzlu X z B do A:
  */
   // přidej X do A
   if (R.key < X.key)
      R.right = X;
```

```
  else
      R.left = X.

    // vyjmi X z B
    if (X.rodic.left == X)
          X.rodic.left = NULL;
    else
          X.rodic.right = NULL;

    // a nastav správně rodiče X
    X.parent = R;
}
```

Uvedené řešení nepočítá s možností, že by strom A byl na začátku prázdný, vhodné doplnění algoritmu ponecháváme zájemcům jako jednoduché cvičení.

15.

Implementujte operaci Delete v binárním vyhledávacím stromu. Předpokládejte, že každý uzel obsahuje klíč a ukazatele na levého a pravého potomka a ukazatel na rodičovský uzel.

Operace Delete je popsána i s kódem jak v přednáškách tak i v leckteré literatuře, doplnění kódu o korektní nastavení ukazatelů na rodičovské uzly ponecháváme zájemcům jako snadné cvičení.

16.

Napište kód pro operaci, která dostane na vstupu ukazatel na uzel v binárním vyhledávacím stromu a provede v něm LR rotaci. Předpokládejte, že každý uzel obsahuje klíč a ukazatele na levého a pravého potomka a ukazatel na rodičovský uzel.

LR rotace je popsána i s kódem jak v přednáškách tak i v leckteré literatuře, doplnění kódu o korektní nastavení ukazatelů na rodičovské uzly ponecháváme zájemcům jako snadné cvičení.