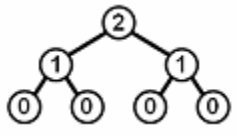
1.

```
void ff(int x) {
  if (x > 0) ff(x-1) ;
  abc(x);
  if (x > 0) ff(x-1) ;
}
```

Daná funkce ff je volána s parametrem 2: **`ff(2);`**. Funkce **`abc(x)`** je tedy celkem volána

a) 1 krát
b) 3 krát
c) 5 krát
d) 7 krát
e) 8 krát

Jedno volání funkce reprezentujeme uzlem ve stromu rekurzivního volání, hodnotou uzlu bude hodnota parametru funkce v daném volání.



Protože při každém volání funkce ff se zavolá funkce abc() právě jednou, je počet volání funkce abc() právě 7. Platí varianta d).

2.

```
void ff(int x) {
  if (x >= 0) ff(x-2) ;
  abc(x);
  if (x >= 0) ff(x-2) ;
}
```

Daná funkce ff je volána s parametrem 2: **`ff(2);`**. Funkce **`abc(x)`** je tedy celkem volána

a) 1 krát
b) 3 krát
c) 5 krát
d) 7 krát
e) 8 krát

Jedno volání funkce reprezentujeme uzlem ve stromu rekurzivního volání, hodnotou uzlu bude hodnota parametru funkce v daném volání.

Protože při každém volání funkce ff se zavolá funkce abc() právě jednou, je počet volání funkce abc() právě 7. Platí varianta d).

3.
Funkce
```
int ff(int x, int y) {
 if (x > 0) return ff(x-1,y)+y;
 return 0;
}
```

a) sčítá dvě libovolná přirozená čísla
b) násobí dvě libovolná přirozená čísla
c) násobí dvě přirozená čísla, pokud je první nezáporné
d) vrací nulu za všech okolností
e) vrací nulu nebo y podle toho, zda x je kladné nebo ne

(Zde došlo k překlepu, místo slova „přirozená“ má být všude slovo „celá“, niceméně respondenti takto zamaskovanou správnou odpověď přesto odhalili.)

Při každém návratu z funkce (kromě prvního s nulou) je vrácena hodnota předchozího volání zvětšená o hodnotu y, která se během jednotlivých volání nemění. Celkem je tedy vrácen vícenásobný součet hodnoty y, takže se jedná o násobení. Jediná podmínka ve funkci zároveň také určuje, že násobení proběhne pouze při nezáporném prvním parametru, takže správná je varianta c).

4.
Funkce
```
int ff(int x, int y) {
 if (x > 0) return ff(x-1, y)-1;
 return y;
}
```

a) pro kladná x vrací 0, jinak vrací y
b) odečte x od y, pokud x je nezáporné
c) odečte y od x, pokud x je nezáporné
d) vrací –y pro kladné x, jinak vrací y
e) spočte zbytek po celočíselném dělení y%x

První návrat z volání funkce vrátí hodnotu y, která se během všech volání nemění. Každý další návrat odečte od výsledku jedničku, počet vnořených volání je roven hodnotě x. Tudíž výsledkem bude hodnota y–x. Celé to nastane ovšem pouze tehdy, když počáteční hodnota x bude nezáporná. Tomuto popisu vyhovuje právě varianta b).

5.
Funkce
```
int ff(int x, int y) {
 if (x < y) return ff(x+1,y);
 return x;
}
```

a) buď hned vrátí první parametr nebo jen „do nekonečna“ volá sama sebe
b) vrátí maximální hodnotu z obou parametrů
c) vrátí součet svých parametrů

d) vrátí x+1
e) neprovede ani jednu z předchozích možností

Když je x větším (nebo stejným) z obou parametrů, jeho hodnota je vrácena ihned. V opačném případě se v rekurzivním volání jeho hodnota zvětšuje tak dlouho, dokud nedosáhne hodnoty y tedy původně většího z obou parametrů. Při návratu z rekurze již k žádným změnám nedochází, opět je tedy vrácena hodnota většího z obou parametrů. Platí možnost b).

6.

Funkce

```
int ff(int x, int y) {
 if (y>0) return ff(x, y-1)+1;
 return x;
}
```

a) sečte x a y, je-li y nezáporné
b) pro kladná y vrátí y, jinak vrátí x
c) spočte rozdíl x–y, je-li y nezáporné
d) spočte rozdíl y–x, je-li y nezáporné
e) vrátí hodnotu svého většího parametru

Pro y záporné nebo nulové vrátí fukce hodnotu x, Pro kladné y volá sama sebe. Počet volání je roven hodnotě y (neboť tento parametr se při každém volání o jedničku zmenší) a při návratu z rekurze se návratová hodnota z většuje pkaždé o 1. Počet volání je y, nejvnitřnější volání vrátí hodnotu x, takže návrat z rekurze přičte k x ještě hodnotu y. Správná odpověď je a).

7.

Při volání rekurzivní funkce f(n) vznikne binární pravidelný ideálně vyvážený strom rekurzivního volání s hloubkou $\log_2(n)$. Asymptotická složitost funkce f(n) je tedy

a) $\Theta(\log_2(n))$
b) $\Theta(n \cdot \log_2(n))$
c) $O(n)$
d) $O(\log_2(n))$
e) $\Omega(n)$

Celkem je známo, že hloubka vyváženého stromu s n uzly je zhruba $\log_2(n)$. To je i případ stromu z našeho zadání. Funkce f při své rekurzivní činnosti navštíví každý uzel tohoto stromu, takže vykoná alespoň *konst* $\cdot n$ operací. Možná, že má práce v každém uzlu (= při každém svém volání) ještě více, takže celková doba její činnosti je buď úměrná n, nebo je asymptoticky ještě větší. Právě to je vyjádřeno pátou možností e). Třetí možnost nepřipouští, že by práce v každém jednom uzlu mohlo být mnoho, zbylé možnosti jsou vůbec nesmysl, vata, vycpávka.

8.
Při volání rekurzivní funkce f(n) vznikne binární pravidelný ideálně vyvážený strom rekurzivního volání s hloubkou n. Asymptotická složitost funkce f(n) je tedy

a) $\Theta(n)$
b) $O(n)$
c) $\Theta(\log_2(n))$
d) $\Omega(2^n)$
e) $O(n!)$

Když má dotyčný strom např. m uzlů, jeho hloubka je úměrná $\log_2(m)$.
Tudíž, když má hloubku n, jeho počet uzlů je úměrný 2^n.
Symbolicky:
$n \approx \log_2(m) \Rightarrow 2^n \approx m$.
Nevíme ovšem, co při každém rekurzivním volání (= v uzlu stromu rekurze) funkce dělá, třeba tam řeší něco složitého, takže celkem její složitost může být i větší než $\Theta(2^n)$.
Jiná možnost než d) prostě není, všechny ostatní jsou jsou pouhé „křoví“ bez jakéhokoli nároku (a úmyslu!) na podobnost s realitou.

9.
Vypočtěte, kolik celkem času zabere jedno zavolání funkce `rekur(4);` za předpokladu, že provedení přikazu `xyz();` trvá vždy jednu milisekundu a že dobu trvání všech ostatních akcí zanedbáme.

```
void rekur(int x) {
  if (x < 1) return;
  rekur(x-1);
  xyz();
  rekur(x-1);
}
```

Strom rekurzivního volání funkce `rekur` je binární vyvážený strom. při volání `rekur(4)` bude mít tento strom hloubku 5, uzly posledního (=nejhlubšího) „patra“ však budou odpovídat pouze provedení řádku `if (x < 1) return;` a příkaz `xyz()` se tu neprovede.
V každém uzlu stromu rekurzivního volání do hloubky 4 bude tedy jednou proveden příkaz `xyz()`. Počet těchto uzlů je 1 + 2 + 4 + 8 = 15. Stejněkrát bude proveden i příkaz `xyz()`.

10.
Určete, jakou hodnotu vypíše program po vykonání příkazu `print(rekur(4));`, když rekurzivní funkce `rekur()` je definována takto:

```
int rekur(int x) {
  if (x < 1) return 2;
  return (rekur(x-1)+rekur(x-1));
}
```

Nedokážete-li výsledek přímo zapsat jako přirozené číslo, stačí jednoduchý výraz pro jeho výpočet.

Rekurzivní volání `rekur(x-1)+rekur(x-1)` můžeme zapsat jako `2*rekur(x-1)` a potom máme:

```
rekur(4) = 2*rekur(3) = 2*2*rekur(2) = 2*2*2*rekur(1) =
= 2*2*2*2*rekur(0) = 2*2*2*2*2 = 32.
```

11.
Obecný binární strom

a) má vždy více listů než vnitřních uzlů
b) má vždy více listů než vnitřních uzlů, jen pokud je pravidelný
c) má vždy méně listů než vnitřních uzlů
d) může mít více kořenů
e) může mít mnoho listů a žádné vnitřní uzly

Má-li strom alespoň dva uzly, má i kořen, který je v takovém případě vnitřním uzlem. Když má strom jen jeden uzel, je to kořen, a i kdybychom jej deklarovali jako list, byl by to list jediný, což lze jen těžko označit jako „mnoho listů“, takže možnst e) nepřichází v úvahu.
Možnost d) je zřejmý nesmysl. Obecný binární strom může mít jen list jediný a mnoho vnitřních uzlů (= jedna „holá větev“), odpadá možnost a). Vyvážený strom se třemi uzly vyvrací možnost c), takže zbývá jediná korektní možnost b). V pravidelném binárním stromu (s alespoň 3 uzly) platí, že počet listů je o 1 větší než počet vnitřních uzlů.

12.
Binární strom má n uzlů. Šířka jednoho „patra“ (tj. počet uzlů se stejnou hloubkou) je tedy

a) nejvýše $\log(n)$
b) nejvýše $n/2$
c) alespoň $\log(n)$
d) alespoň $n/2$
e) nejvýše n

Binární strom nemusí být nutně pravidelný, může to být jen jediná „dlouhá větev“, takže nejmenší možná šíře je 1, možnost třetí a čtvrtá odpadají. „Co nejširší patro“ odpovídá stromu „co nejvíce do šířky roztaženému“, na což se zdá být ideálně vyvážený pravidelný strom alespoň kandidátem. V něm je nejspodnější „patro“ listů. Při n uzlech ve stromu je v onom „patře“ $(n+1)/2$ uzlů, možnost první a druhá odpadají také.

13.
Daný binární strom má tři listy. Tudíž

a) má nejvýše dva vnitřní uzly
b) počet vnitřních uzlů není omezen

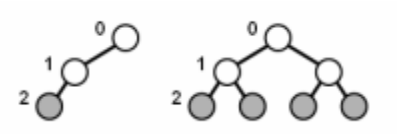
c) všechny listy mají stejnou hloubku
d) všechny listy nemohou mít stejnou hloubku
e) strom je pravidelný

Binární strom nemusí být nutně pravidelný, může mít dlouhatánské „lineární“ nevětvící se větve (nevětvící se větve, hmm...) s jediným uzlem na konci. Stačí tady tři takové větve (jedan doprava z kořene a dvě z jeho levého potomka) a jejich délka může být zcela libovolná. Takový strom není pravidelný (ne paté možnosti), jeho tři listy mohou a nemusí ležet stejně hluboko (ne třetí a čtvrté možnosti) a zbytek již byl řečen.

14.
Binární strom má hloubku 2 (hloubka kořene je 0). Počet listů je

a) minimálně 0 a maximálně 2
b) minimálně 1 a maximálně 3
c) minimálně 1 a maximálně 4
d) minimálně 2 a maximálně 4

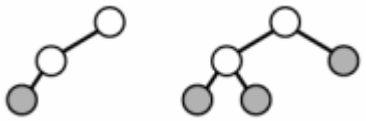
Minimální a maximální počet listů je vyznačen na obrázku s naznačenými hloubkami, platí varianta c).



15.
Binární strom má 2 vnitřní uzly. Má tedy

e) minimálně 0 a maximálně 2 listy
f) minimálně 1 a maximálně 3 listy
g) minimálně 1 a maximálně 4 listy
h) minimálně 2 a maximálně 4 listy

Minimální a maximální počet listů je vyznačen na obrázku, platí varianta b).



16.
Algoritmus A provádí průchod v pořadí inorder binárním vyváženým stromem s n uzly a v každém uzlu provádí navíc další (nám neznámou) akci, jejíž složitost je $\Theta(n^2)$. Celková asymptotická složitost algoritmu A je tedy

a) $\Theta(n)$
b) $\Theta(n^2)$
c) $\Theta(n^3)$
d) $\Theta(n^2+\log_2(n))$
e) $\Theta(n^2 \cdot \log_2(n))$

Při průchodu stromem v pořadí pre/in/postorder má režie na příchod a odchod do/z uzlu složitost úměrnou konstantě. V každém uzlu – jichž je n – se navíc podle zadání provedou operace, jejichž počet je úměrný n^2. Celkem je tedy na zpracování úkolu zapotřebí čas úměrný výrazu $n \cdot (\text{konstanta} + n^2) = n \cdot \text{konstanta} + n^3 \in \Theta(n^3)$. Platí varianta c).

17.
Algoritmus A provede jeden průchod binárním stromem s hloubkou *n*. Při zpracování celého *k*-tého „patra“ (=všech uzlů s hloubkou *k*) provede *k*+*n* operací. Operační (=asymptotická) složitost algoritmu A je tedy

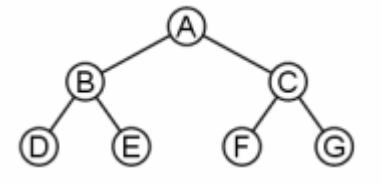
a) $\Theta(k+n)$
b) $\Theta((k+n) \cdot n)$
c) $\Theta(k^2+n)$
d) $\Theta(n^2)$
e) $\Theta(n^3)$

Celkem je provedený počet operací při zpracování všech uzlů roven
$(1+n) + (2+n) + (3+n) + ... + (n+n) = n \cdot ((1+n) + (n+n))/2 = n \cdot (1+3n)/2 = 3n^2/2 + n/2 \in \Theta(n^2)$

Považíme-li navíc ještě režii na přesun od jednoho „patra“ stromu k následujícímu patru, která může mít složitost nanejvýš $\Theta(n)$, připošteme k výsledku ještě člen $\Theta(konst \cdot n \cdot n)$, který ovšem na složitosti $\Theta(n^2)$ nic nemění. Platí varianta d).

18.
Obsah uzlů daného stromu vypíšeme v pořadí postorder. Vznikne posloupnost

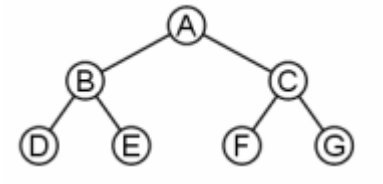
a) G F E D C B A
b) F C G D B E A
c) G C F A E B D
d) D E B F G C A



Přímo z definice pořadí postorder plyne, že je nutno volit odpověď d).

19.
Obsah uzlů daného stromu vypíšeme v pořadí preorder. Vznikne posloupnost

a) A B C D E F G H
b) D B E A F C G
c) A B D E C F G
d) D E F G B C A



Přímo z definice pořadí postorder plyne, že je nutno volit odpověď c).

20.
Výpis prvků binárního stromu v pořadí postorder provede následující:

a) vypíše prvky v opačném pořadí, než v jakém byly do stromu vloženy
b) pro každý podstrom vypíše nejprve kořen, pak obsah jeho levého a pak pravého podstromu
c) pro každý podstrom vypíše nejprve obsah levého podstromu kořene, pak obsah pravého podstromu a pak kořen
d) pro každý podstrom vypíše nejprve obsah pravého podstromu kořene, pak obsah levého podstromu a pak kořen
e) vypíše prvky stromu v uspořádání zprava doleva

Definice praví, že se jedná o variantu c).

21.
Výpis prvků binárního stromu v pořadí preorder provede následující:

a) vypíše prvky stromu ve stejném pořadí, v jakém byly do stromu vloženy
b) vypíše prvky stromu v uspořádání zleva doprava
c) pro každý podstrom vypíše nejprve kořen, pak obsah jeho levého a pak pravého podstromu
d) pro každý podstrom vypíše nejprve obsah levého podstromu kořene, pak obsah pravého podstromu a pak kořen
e) vypíše prvky stromu seřazené vzestupně podle velikosti

Definice praví, že se jedná o variantu c).

22.
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.

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 B do A 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 ponechávám zájemcům jako jednoduché cvičení.

23.

Projděte binárním stromem a vypište obsah jeho uzlů v pořadí preorder bez použití rekurze, zato s využitím vlastního zásobníku.

Pořadí preorder znamená, že nejprve zpracujeme aktuální uzel a potom jeho levý a pravý podstrom. Budeme tedy v cyklu zpracovávat vždy aktuální vrchol, referenci na nějž vždy odebereme z vrcholu zásobníku (-- odkud jinud také!) a hned poté si do zásobníku uložíme informaci o tom, co je ještě třeba zpracovat, tedy ukazatele na levého a pravého potomka aktuálního uzlu.

Celý základní cyklus tedy bude vypadat takto:

```
while (stack.empty == false) {
    currNode = stack.pop();
    print(currNode.key);
    if (currNode.left != null)  stack.push(currNode.right);
    if (currNode.right != null)  stack.push(currNode.left);
}
```

Předtím, než tento cyklus spustíme, musíme ovšem inicializovat zásobník a a vložit do něj kořen stromu:

```
If (kořenStromu == null) return;
stack.init();
stack.push(kořen.Stromu);
základní cyklus uvedený výše;
```

To je celé, vstupem algoritmu je ukazatel na kořen stromu, výstupem hledaná posloupnost, pro zásobník je nutno pro jistotu alokovat tolik místa, kolik může nejvíc mít strom uzlů.

24.

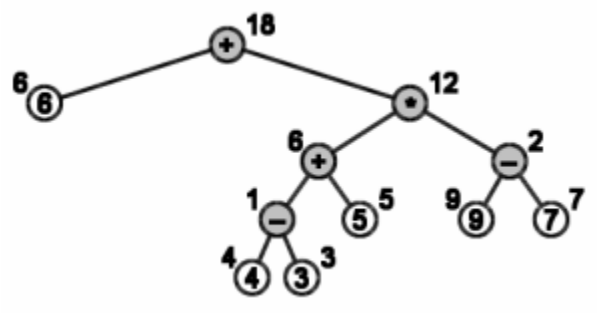
Aritmetický výraz obsahující celá čísla, závorky a operace +,–,*,/ (celočíselné dělení) může být reprezentován jako pravidelný binární strom. Popište, jak takový strom obecně vypadá, navrhněte implementaci uzlu a napište funkci, jejímž vstupem bude ukazatel na kořen stromu a výstupem hodnota odpovídajícího aritmetického výrazu.
(Body = 1 popis + 1 uzel + 3 funkce)

Vnitřní uzly stromu budou představovat operace, listy pak jednotlivá čísla. Například výraz
6+(4-3+5)*(9-7)
lze reprezentovat následujícím stromem:

(Obrázek je bitmapa, nejprve olíznutá z obrazovky s powerpointem, v němž byl obrázek vytvořen, pak upravena v paintShopu a posleze vložená do wordu jako rastrový obrázek. Produkty firmy mikro**t si totiž neumí mezi sebou navzájem předávat svou vektorovou grafiku, hm…)

Hodnotou každého listu je číslo v listu uložené, hodnotou vnitřního uzlu je <hodnota levého podstromu> <operace v uzlu zapsaná> <hodnota pravého podstromu>.

Hodnota kořene pak představuje hodnotu celého výrazu.
Na dalším obrázku jsou hodnoty u jednotlivých uzlů připsány:



V implementaci uzlu tedy pořebujeme
-- **typ** - složka, která udává, zda jde o vnitřní uzel nebo list
-- **op** - složka udávající operaci (např. znak)
-- **val** - složka pro číslo, je-li uzel listem, jinak hodnota uzlu
-- **left** a **right** – potomci uzlu

(též je možno první složku vypustit a indikovat např. prázdným znakem ve druhé složce, že jde o list, apod…)

Celá vyhodnocovací funkce (předpokládající, že strom je neprázdný) pak může vypadat takto:

```
int hodnota(node n) {
  if (n.typ == list) return n.val;
  switch (n.typ) {
    case '+': return (hodnota(n.left) + hodnota(n.right)); break;
    case '-': return (hodnota(n.left) - hodnota(n.right)); break;
    case '*': return (hodnota(n.left) * hodnota(n.right)); break;
    case '/': return (hodnota(n.left) / hodnota(n.right)); break;
  }
}
```

(mimochodem, tím že kompilátor zajistí, že se ve výrazu hodnota(n.left) + hodnota(n.right) a dalších nejprve spočtou hodnoty funkcí a pak teprve se provede sčítání (odčítání,.. atd.), zajistí vlastně také průchod oním stromem v pořadí postorder...)

25.
Deklarujte uzel binárního stromu, který bude obsahovat celočíselné složky **výška** a **hloubka**. Ve složce **hloubka** bude uložena hloubka daného uzlu ve stromu, ve složce **výška** jeho výška. Výška uzlu X je definována jako vzdálenost od jeho nejvzdálenějšího potomka (= počet hran mezi uzlem X a jeho nejvzdálenějším potomkem).
Napište funkci, která každému uzlu ve stromu přiřadí korektně hodnotu jeho hloubky a výšky.

Při počítaní hloubky stačí každému potomku uzlu X přiřadit o 1 větší hloubku než má uzel X. Sama rekurzivní procedura mluví za dlouhé výklady:

```
void setHloubka(node x, int depth) {
   if (x == null) return;
   x.hloubka = depth;
   setHloubka(x.left, depth+1);
   setHloubka(x.right,depth+1);
}
```

Přiřazení hloubky každému uzlu ve stromu pak provedeme příkazem
`setHloubka(ourTree.root, 0);`

Všimněte si, že hloubky se uzlům přiřazují – celkem logicky – v pořadí preorder.

Při přiřazování výšky uzlu X musíme naopak znát výšku jeho potomků, takže se nabízí zpracování v pořadí postorder:

```
void setVyska(node x) {
   int vyskaL = -1;  // nejprve pripad, ze uzel nema L potomky.
   int vyskaR = -1;  // dtto

   if (x.left != null) {
       setVyska(x.left); vyskaL = x.left.vyska;
   }
   if (x.right != null) {
       setVyska(x.right); vyskaR = x.right.vyska;
   }
   x.vyska = max(vyskaL, vyskaR) + 1;
}
```

Přiřazení výšky každému uzlu ve stromu pak provedeme příkazem
`if (ourTree.root != null) setVyska(ourTree.root);`

Celý proces s výškou lze zachytit ještě úsporněji:

```
int setVyska(node x) {
   if (x == null) return -1;
   x.vyska = 1+ max(setVyska(x.left), setVyska(x.right));
   return x.vyska;
}
```

Přiřazení výšky každému uzlu ve stromu pak provedeme např. příkazem
`zbytecnaProm = setVyska(ourTee.root);`

26.

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.

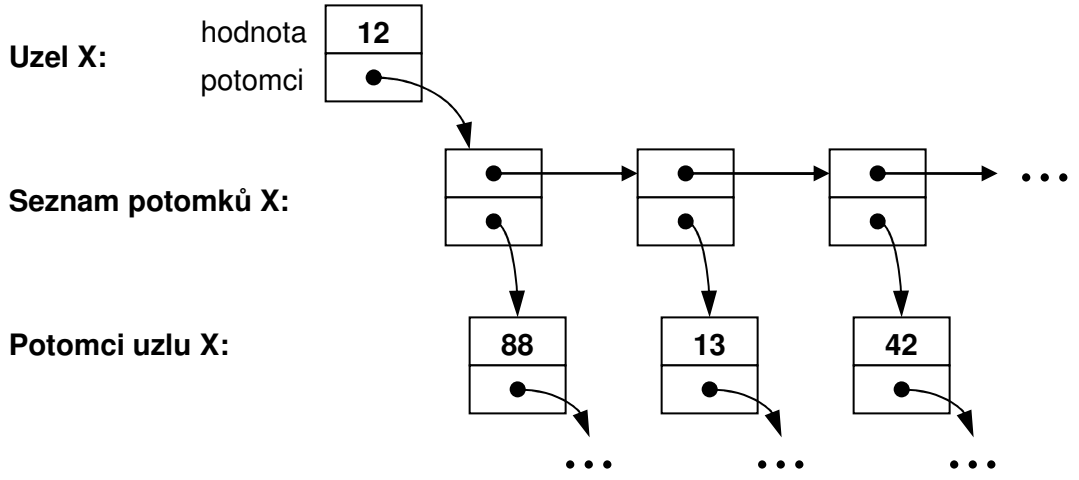
Z uvedeného popisu by měl být zřejmý následující rekurzvní vztah: Dva binární stromy (ani nemusí být vyhledávací) A a B jsou strukturně stejné, pokud

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));
}
```

27.
Napište funkci, která vytvoří kopii obecného kořenového stromu. Ten je reprezentován takto:
Uzel X stromu obsahuje dvě složky: **hodnota** a **potomci**. Složka **potomci** je ukazatel na zřetězený seznam obsahující jednotlivé ukazatele na potomky uzlu X. Nemá-li uzel X žádné potomky, složka **potomci** ukazuje na null. Každý prvek seznamu potomků obsahuje jen ukazatel na potomka X a ukazatel na další prvek v seznamu potomků. Strom je obecný, potomci libovolného uzlu nejsou nijak uspořádáni podle svých hodnot.



Jen tak cvičně si napřed řekněme, jak by probíhalo kopírování binárního stromu. Nejprve bychom vytvořili kopii kořene, potom kopii levého a pravého podstromu kořene a pak bychom tyto kopie podstromů připojili ke kopii kořene. Kopie podstromů bychom ovšem vyrobili stejnou rekurzivní procedurou.

Zachyceno pseudokódem:

```
node kopieBinStromu (node root) {
  if (root == null) return null;
  new root2 = kopieUzlu(root);
```

```
  root2.left = kopieBinStromu(root.left);
  root2.right = kopieBinStromu(root.right);
  return root2;
}
```

V případě obecného kořenového stromu je situace zcela obdobná. Jediný rozdíl je v tom, že nemáme v uzlu pevnou strukturu pro levého a pravého následníka ale jen ukazatel na seznam obsahující ukazatele na potomky.
Ať jeho struktura uzlu např. takováto

```
{ int hodnota;
  seznamPotomku potomci; }
```

A struktura prvku seznamu ukazatelů na potomky:

```
{ seznamPotomku next;
  node potomek}
```

Zachyceno pseudokódem:

```
node kopieBinStromu (node root) {
  if (root == null) return null;
  new rootKopie = kopieUzlu(root);  // kopírujme pouze uzel,
                                    // nikoli seznam ukazatelů na
potomky

  // připravme si seznam zkopírovaných potomků,
  // který nakonec připojíme k root2
  seznamPotomku potomciRootKopie = null;
  seznamPotomku potomciRoot = root.potomci
   while (potomciRoot != null) {

      node kopiePodstromu = kopieBinStromu(potomciRoot.potomek);
// rekurze
      seznamPotomku kopieUkazatele = new(seznamPotomku);
      kopieUkazatele.potomek = kopiePodstromu;
      zaradNaKonecSeznamu(PotomciRootKopie, kopieUkazatele);

      potomciRoot = potomciRoot.next;
   }
   // teď seznam ukazatelů na zkopírované podstromy připojíme k
rootKopie
   rootKopie.potomci = potomciRootKopie;

   return root2;
}  // je hotovo
```

28.
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);
  }
}
```

29.
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
}
```

30.

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
}
```

31.

Napište rekurzívně a nerekurzívně funkci, která projde n-ární strom (každý uzel může mít až n podstromů. Odkazy na podstromy jsou uloženy v každém uzlu v poli délky n). Datová struktura uzlu (1 bod). Rekurzívní verze (2b), nerekurzívní (3b.)

uzel obsahuje dvě složky

```
{
/*napr.*/ int value;
          potomci [n];
}

// rekurzivne ve variante "preorder":
void projdi(node root) {
  if (root == null) return;
  nejakaAkce(root); // tady by se provedla patricna akce
  for (i = 0; i < n; i++)
     projdi(potomci[i]);


// nerekurzivne
```

```
// pouzijeme opet postup "preorder" protoze se pri nem nemusime
na zasobnik
// ukladat zadne dodatecne informace o navstevach uzlu

void projdiNonRec(node root) {
if (root== null) return;
stack.init();
stack.push(root);
while (stack.empty() == false) {
    aktUzel = stack.pop();  // aktualni uzel definitivne ze
zasbniku mizi
    nejakaAkce(aktUzel);    // tady by se provedla patricna akce
           // na zasobnik se ulozi potomci
    for (i = 0; i < n; i++)
       stack.push(aktUzel.potomci[i]);
    }
}
```