Níže uvedené úlohy představují přehled otázek, které se vyskytly v tomto nebo v minulých semestrech ve cvičení nebo v minulých semestrech u zkoušky. Mezi otázkami semestrovými a zkouškovými není žádný rozdíl, předpokládáme, že připravený posluchač dokáže zdárně zodpovědět většinu z nich.

Tento dokument je k dispozici ve variantě převážně s řešením a bez řešení.

Je to pracovní dokument a nebyl soustavně redigován, tým ALG neručí za překlepy a jazykové prohřešky, většina odpovědí a řešení je ale pravděpodobně správně :-).

---------- STACK ----------

1.
Se zásobníkem Z provedeme operace **init(), push(a), pop(), push(b)**. Pak Z obsahuje prvky (počínaje vrcholem zásobníku):

a) a b
b) b a
c) a
d) b
e) žádný prvek

2.
The following operations were performed on a stack S: **init(), push(a), pop(), push(b)**.
The resulting contents of S is then (list starts at the top of S):

a) a b
b) b a
c) a
d) b
e) no element

3.
On the given stack S operations **init(), push(z), push(w), pop()** have been performed. Now S contains the elements (list starts at the stack top):

f) w
g) z
h) w z
i) z w
j) 0 z w

4.
Zásobníkový automat je (zjednodušeně vzato) zařízení, které čte znak po znaku vstupní řetězec a po každém přečteném znaku provede nějakou operaci se zásobníkem. Než automat začne číst řetězec, je zásobník prázdný. Kromě toho jsou specifikována pravidla, jaké operace se zásobníkem se mají provést po přečtení určitého znaku vstupního řetězce.
Předpokládejme, že ve vstupním řetězci mohou být pouze znaky A ... Z a že na zásobník lze ukládat celá čísla. Předpokládejme dále, že zásobník má neomezenou kapacitu a že při pokusu o operaci POP na prázdném zásobníku je ohlášena chyba (samotný prázdný zásobník chybu neznamená).
Pravidla pro práci automatu lze zaznamenat tabulkou

čtený znak	operace
A	PUSH (1)
B	POP()
jiný	žádná

Určete, jaký bude obsah zásobníku po přečtení řetězce
a) AAB
b) ABAB

c) BAA
d) ABAABAAAB

Řešení
a) 1
b) prázdný zásobník
c) chyba
d) 1 1 1

5.
Uvažujme zásobníkový automat **A2** s odlišnou tabulkou operací

čtený znak	operace
A	PUSH (0)
B	PUSH(POP() + 1)
C	POP()
jiný	žádná

Určete, jaký bude obsah zásobníku po přečtení řetězce
a) AABB
b) AABBC
c) AABCB
d) ABABBABBB

Řešení
a) 0 2
b) 0
c) 1
d) 1 2 3 (vrchol zásobníku je vpravo)

6.
Automat **A2** z předchozí úlohy přečetl určitý řetězec *r* a má nyní prázdný zásobník. Co lze říci o počtu znaků A a C v řetězci *r*?

Řešení Přečtení znaku B nemění počet hodnot v zásobníku. Když automat začíná i končí s prázdným zásobníkem, musí být počet znaků A a C v řetězci totožný. (A kromě v žádném počátečním úseku pole nesmí být počet znaků C stejný jako počet znaků A, aby auromat neukončil svopu práci dříve, ale to již vlastně nepatří do odpovědi, neboť jsem byli tázáni pouze na celkový počet znaků A a C.)

7.
Navrhněte takový vstupní řetězec pro automat A2 z předchozí úlohy, aby po jeho přečtení zásobník obsahoval právě hodnoty 4 3 2 1 (vrchol zásobníku je vpravo). Kolik řešení má tato úloha?

Řešení Lze použít řetězec ABBBBABBBBABBAB. Řešení je více, např.každý řetězec ve tvaru ACAC...ACACABBBBABBBBABBAB naplní zásobník stejnými hodnotami.

---------- QUEUE ----------

1.
Nad frontou F provedeme operace **init(), ins_last(a), ins_last(b), del_front()**. Potom F obsahuje prvky (počínaje čelem fronty):

a) a b
b) b a
c) a
d) b
e) žádný prvek

2.

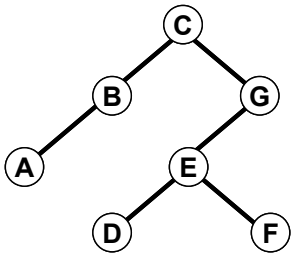
On the given queue Q operations **init(), insLast(x), insLast(y), delFront()** have been performed. Now Q contains the elements (list starts at the queue front):

a) x
b) y
c) x y
d) y x
e) 0 x y

3.
Implementujte cyklickou frontu v poli pole[délka]. Dokud není pole zcela naplněno, musí být fronta stále funkční, tzn. musí být možné do ní stále přidávat i z ní ubírat.
Narazí-li konec nebo začátek fronty na konec nebo začátek pole, neposunujte všechny prvky v poli, zvolte výhodnější „cyklickou“ strategii, která zachová konstantní složitost operace Vlož a Vyjmi.

---------- BREADTH-FIRST SEARCH ----------

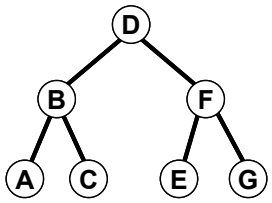
1.
Strom na obrázku procházíme do šířky. V určitém okamžku jsou ve frontě následující uzly (s tím, že čelo fronty je vlevo):



a) BGA
b) GA
c) AG
d) AEG
e) GEA
f) AE

Řešení V případech a) d) e) obsahuje fronta s některým uzlem zároveň i jeho potomka. To v průchodu do šířky není možné. V případě c) se ve frontě nachází uzel uzel hlubšího patra před uzlem mělčího patra (= blíže ke kořeni), což rovněž není možné. Zbývají korektní případy b) a f).

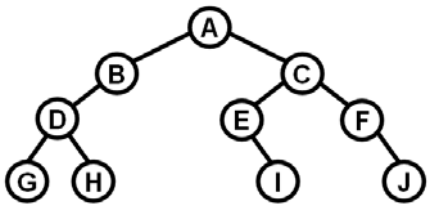
2.
Strom na obrázku procházíme do šířky. V určitém okamžku jsou ve frontě následující uzly (s tím, že čelo fronty je vlevo



a) D
b) DF
c) FB
d) BGE
e) BAC
f) ABCD

Řešení Případ a) popisuje standardní stav na začátku průchodu, je to tedy správná varianta odpovědi. Případy c) a d) popisují spraávně obsah fronty, pokud stromem procházíme zprav doleva. Algoritmus procházení do šířky nijak nespecifikuje, kterým směrem se prochází (co je to směr v interní reprezentaci v paměti počítače??), tudíž tyto varianty jsou korektní. Zbývající varianty umisťují do fronty některý uzel zároveň s jeho potomkem, což není možné.

3.
Zopakujme stručně princip procházení do šířky.
Krok 0. Vlož kořen do prázdné fronty
Dokud je fronta neprázdná, dělej
Krok 1. Vyjmi první prvek z fronty, a zpracuj ho.
Krok 2. Vlož do fronty všechny potomky právě vyjmutého listu.

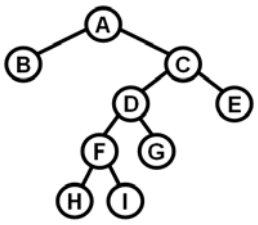


Projděte do šířky daný strom a před každým provedením kroku 1. zaznamenejte obsah fronty.

Řešení
A
BC
CD
DEF
EFGH
FGHI
GHIJ
HIJ
IJ
J

4.
Úlohou je rekonstruovat tvar pravidelného stromu, pokud známe průběžný obsah fronty. Dejme tomu, že před každým provedením kroku 1. v uvedeném postupu zaregistrujeme aktuální obsah fronty.
Získáme posloupnost (předpokládáme, že čelo fronty je vlevo):
A
BC
C
DE
EFG
FG
GHI
HI
I

Řešení Zřejmě A je kořenem a jeho potomci jsou B a C. Po zpracování B (krok 1.) žádní potomci ve frontě v kroku 2. nepřibydou, tudíž B nemá žádné potomky. Naopak, po zpracování C přibydou ve frontě uzly D a E, které jsou proto potomky C. Podobnou úvahou ověříme, že potomky D jsou F a G, zatímco po odstranění E nic ve frontě nepřibyde, tedy E je listem. Zbývá, opět analogicky, ověřit , že potomky F jsou listy H a I a že G je rovněž listem. Celkem tedy vzhled stromu je patrný na obrázku.



5.
Formulujte obecný algoritmus, jak z dané posloupnosti obsahů fronty (jako v předchozí úloze) rekonstruovat pravidelný strom.

Řešení Označme posloupnost symbolem *P*, její i-tý prvek (řetězec) symbolem *P*[i] ($i \geq 0$).
Protože strom budujeme po patrech, musíme si pořídit vlastní frontu na uzly a s její pomocí budeme „stínově“ kopírovat původní průchod stromem do šířky, z něhož zbyla jen posloupnost řetězců
Při vkládání uzlu do fronty zároveň naplníme jeho datovou část. Po vyjmutí uzlu *x* z fronty poznáme podle délky aktuálního a předchozího výpisu v posloupnosti *P*, zda *x* má či nemá portomky. Prodloužení výpisu znamená, že má, že do ní potomci přibyli, zkrácení (o vyjmutý první prvek) znamená, že uzel *x* potomky nemá.
Strom musíme předpokládat úplný, protože z fronty nepoznáme, zda jediný potomek uzlu je pravý či levý.

```
Node root = new Node(P[0].charAt(0) // take P[i][j] as node value
Node currNode;
Queue q = new Queue();
q.enqueue(root);

for (int i = 1; i < P.length(); i++);
   currNode = q.dequeue();
   if (P[i].length() == P[i-1].length()+1) {
         currNode.left = new Node(P[i].charAt(P[i].length()-2));
         currNode.left = new Node(P[i].charAt(P[i].length()-1));
         q.enqueue(currNode.left);
         q.enqueue(currNode.left);
         }
```

```
        else {  // P[i].length() == P[i-1].length()+1,
                // currNode is a leaf
         currNode.left = null;
         currNode.left = null;
        }
 } // end of for loop
```

6.
Přeformulujte předchozí algoritmus, aby na základě posloupnosti obsahů fronty rekonstruoval původní strom, kterým v tomto případě je obecný BVS.

Řešení Samostatně

7.
Vypište hodnoty uzlů daného binárního stromu tak, že budete postupovat „po patrech“, tj. nejprve vypíšete hodnoty všech uzlů s hloubkou 0, pak hodnoty všech uzlů s hloubkou 1, pak hodnoty všech uzlů s hloubkou 2, atd. Nápověda: použijte datový typ fronta.

Úloha tato známa jest v kruzích zasvěcenců coby „procházení do šířky“. Její myšlenka je jednoduchá: Jakmile navštívíme nějaký uzel, zařadíme do fronty nejprve jeho levého a pak pravého potomka. Navštěvovat budeme postupně ty uzly, které budeme odebírat z čela fronty.
Začneme tím, že do fronty vložíme kořen. Po jeho zpracování (a odebrání z fronty!) zbydou ve frontě jeho dva potomci, tedy celé „první patro“ v pořadí odleva doprava. S postupným procházením prvků ve frontě se budou na konec fronty zařazovat uzly z následujícího „patra“ a právě proto, že se budou zařazovat na konec fronty, nedostane se algoritmus ke zpracování dalšího „patra“ dříve, než zpracuje všechny uzly v aktuálním patře. Pseudokód celého algoritmu následuje.

```
if (strom.root == null) return;
fronta.init();
fronta.insert(strom.root);
while (fronta.empty == false) {
  aktUzel = fronta.pop();
  print(aktUzel.hodnota);
  if (aktUzel.left != null) fronta.push(aktUzel.left);
  if (aktUzel.right != null) fronta.push(aktUzel.right);
}
```

---------- GENERAL TREE ----------

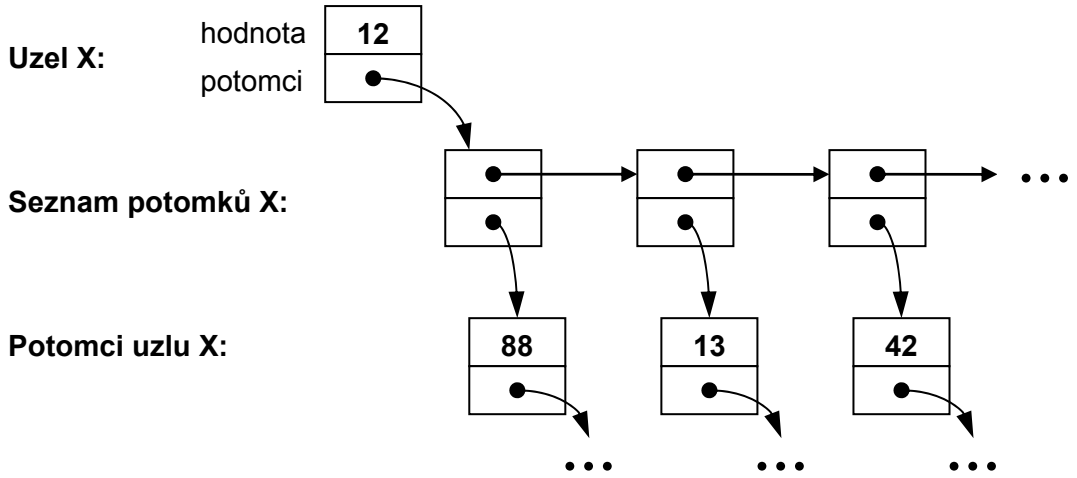
1.
Daný obecný strom má *n* uzlů. Kolik má hran? Zdůvodněte.

Řešení Zvolme ve stromu libovolný uzel jako kořen a představme si strom nakreslený v typické podobě s kořenem nahoře a následníky pod d ním. Z každého uzlu kromě kořene vede jedna hrana do jeho předchůdce. Uzlů je *n*, hran je tedy *n*−1.

2.

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).

3.
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.



Řešení 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
```

4.
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.)

Řešení 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]);
    }
}
```

---------- BIN TREE ----------

1.
Daný pravidelný (kořenový) binární strom má n uzlů. Kolik má listů? Zdůvodněte.

Řešení Představme si jeden takový strom s alespoň 3 uzly. Když nyní jej nyní budeme chtít zvětšit, musíme jeden z dosavadních listů změnit na vnitřní uzel tak, že k němu připojíme 2 nové listy. (Musíme připojit 2 listy a ne jeden nebo více než dva, protože strom je binární a pravidelný.) Při vzniku tohoto nového stromu se z jednoho listu stal vnitřní uzel a dva jiné listy přibyly. To znamená, že počet vnitřních uzlů i počet listů se zvětšil přesně o 1. Nyní si vezměme nejmenší možný pravidelný binární strom, ten obsahuje jen tři uzly -- jeden kořen a dva listy. Zřejmě každý větší pravidelný binární strom může vzniknout pomocí operace popsané výše, přidáním dvojice listů, která zvýší počet vnitřních uzlů i počet

listů o 1. Nejmenší strom má 1 vnitřní uzel (v tomto případě kořen) a 2 listy, má tedy o jeden list více než vnitřních uzlů. Z našeho rozboru plyne, že tento vztah se zachová i po přidání libovolného počtu dalších dvojic listů. Uzavíráme, že v binárním stromu je vždy počet listů o 1 větší než počet vnitřních uzlů. Pokud je celkový počet uzlů n, (o čemž jsme právě ukázali, že je liché číslo), je počet listů roven $(n+1)/2$ a počet vnitřních uzlů $(n-1)/2$.

2.
Jsou dány dva kusy papíru. Některý z nich rozstříhneme na dva kusy, některý ze všech kusů rozstříhneme opět na dva kusy, atd. Kolik kusů celkem vznikne, když bylo rozstříženo celkem k kusů papíru (nezávisle na jejich velikosti)?

Řešení Každým rozstřihnutím jednoho kusu na dva přibyde jeden kus do celkového počtu. Po $k = 0$ střihnutích jsme měli 2 kusy, po k střihnutích budeme proto mít $k+2$ kusy.

3.
Turnaje ve stolním tenisu, který se hraje vylučovacím způsobem, se zúčastnilo celkem 19 hráčů. Šest vylosovaných hráčů muselo sehrát předkolo, z nějž do turnaje postoupili tři hráči. Nakreslete strom reprezentující možný průběh turnaje a určete počet jeho vnitřních uzlů tj. počet utkání, která byla během turnaje sehrána.
Kolik utkání včetně předkola by bylo nutno sehrát v turnaji, jehož se by se účastnilo147 hráčů?

Řešení Celý turnaj i s předkolem znázorníme jako binární pravidelný kořenový strom. Utkání představují vnitřní uzly stromu, hráči jsou listy. Díky úloze 2 víme, že počet listů v takovém stromě je o 1 větší než počet vnitřních uzlů, takže celkový počet zápasů je o 1 menší než počet hráčů. Lze také triviálně uvažovat tak, že si řekneme, že po každém odehraném zápasu turnaj ztrácí jednoho prohravšího hráče. Protože nakonec zbývá jediný vítěz, musí být opět počet zápasů o 1 menší než počet hráčů.

4.
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) $\Omega(n)$

Řešení 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.

5.
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) $O(n!)$

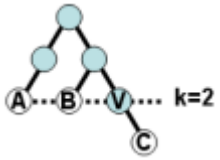
Řešení Když má dotyčný strom např. m uzlů, jeho hloubka je úměrná $\log_2(m)$. Tudíž, když má hloubku n, jeho počet uzlů je úměrný 2^n. Symbolicky: $n \approx \log_2(m) \Rightarrow 2^n \approx m$.
Nevíme ovšem, co při každém rekurzivním volání (= v uzlu stromu rekurze) funkce dělá, třeba tam řeší něco 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.

6.
V obecném binárním stromu s n uzly, který není vyhledávací, hledáme klíč s maximální hodnotou. Když to uděláme efektivně, bude složitost této operace

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

7.
Je dána konstanta k taková, že hloubka každého listu v daném binárním stromu je větší nebo rovna k. Pro hloubku každého vnitřního uzlu platí

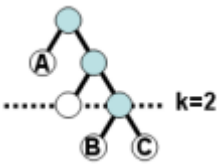
a) je menší než k
b) je menší nebo rovna k–2
c) je větší než k/2
d) je menší nebo rovna $\log_2(k)$
e) nic z předchozího



Pozorujme obrázek. Variantu a) i b) vyvrací vnitřní uzel V. Variantu c) vyvrací kořen stromu. Variantu d) vyvrací opět vnitřní uzel V. Zbývá jediná správná varianta e).

8.
Je dána konstanta k taková, že hloubka každého vnitřního uzlu v daném binárním stromu je menší nebo rovna k. Pro hloubku každého listu platí

a) je rovna k+1
b) je také menší nebo rovna k
c) je menší nebo rovna k+1
d) je větší než k–1
e) je menší nebo rovna $\log_2(k)$



Pozorujme obrázek. Domněnku a) vyvrací list A, B i C. Domněnku b) vyvrací listy B a C. Domněnku d) vyvrací list A, domněnku e) vyvracejí opět listy B a C. Zbývá jediná správná varianta c).

9.
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

Řešení 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.

10.
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

Řešení 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.

11.

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

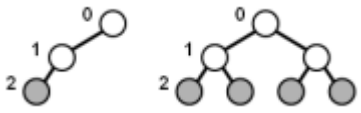
Řešení Binární strom nemusí být nutně pravidelný, může mít dlouhatánské „lineární“ nevětvící se větve (nevětvící se větve, hmm...) s jediným uzlem na konci. Stačí tady tři takové větve (jedna doprava z kořene a dvě z jeho levého potomka) a jejich délka může být zcela libovolná. Takový strom není pravidelný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

12.

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

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

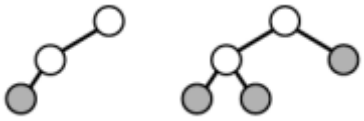


13.

Binární strom má 2 vnitřní uzly. Má tedy

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

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



14.

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) $\Theta(n^2 \cdot \log_2(n))$

Řešení 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).

15.

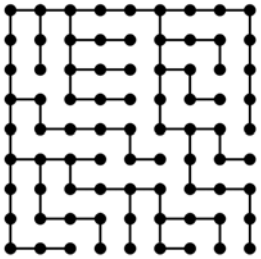
Algoritmus A provede jeden průchod binárním stromem s hloubkou n. Při zpracování celého k-tého „patra“ (=všech uzlů s hloubkou k) provede $k+n$ operací. Operační (=asymptotická) složitost algoritmu A je tedy

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

Řešení 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).

16.
Navrhněte strukturu binárního stromu s n prvky tak, aby hloubka stromu nebyla vyjádřena výrazem ani $\Theta(\log(n))$ ani $\Theta(n)$, ale výrazem $\Theta(\sqrt{n})$.

Řešení Jedna z přímočarých možností je znázorněna na obrázku. Uzly budou mřížové body čtvercové sítě, kořen bude např. levý horní roh čtverce a každý uzel kromě kořene pak bude mít coby rodiče ve stromě svého bezprostředního souseda směrem vlevo nebo nahoru ve čtverci. Hloubka tohoto stromu je $2m-1$, pokud uvažujeme čtverec s $m \times m$ mřížovými body. Označme celkový počet uzlů symbolem $n = m \times m$. Potom hloubka je vyjádřena předpisem $2\sqrt{n} - 1$, který náleží do třídy funkcí $\Theta(\sqrt{n})$.



17.
Představme si, že máme navštívit všechny uzly v ideálně vyváženém stromu se 7 uzly. Můžeme chodit podle každé hrany tam i zpět, pohyb podél jedné hrany mezi dvěma uzly trvá právě jednu sekundu.

a) Jak dlouho se budeme ve stromu pohybovat, máme-li (celkem přirozeně) začít i skončit v kořeni?
b) Změní se nějak doba průchodu stromem v případě že strom bude obsahovat stejný počet uzlů ale bude maximálně nevyvážený?
c) Jaká bude doba průchodu obecně v případě, že strom bude obsahovat n uzlů?

Řešení Když si trasu pochodu zlehka načrtnete tužkou, zjistíte nejspíše, že musíte každou hranou projít právě dvakrát -- tam a zpět. Je to celkem logické, do každého uzlu (nebo podstromu) se lze z kořene dostat jen jedinou cestou a tudíž je nutno se také stejnou cestou vrátit. Celková doba procházky tedy bude v sekundách rovna dvojnásobku počtu hran ve stromu. Zjistit počet hran ve stromu není žádná potíž, do každého kromě kořene uzlu vstupuje právě jedna hrana. Ve stromu je tedy 6 hran, celkem bude procházka trvat 6*2 = 12 sec. Zřejmým zobecněním našeho nálezu dostáváme pravidlo:
Doba procházky stromem je přímo úměrná dvojnásobku počtu uzlů zmenšenému o 2.

18.
Uvažujme opět ideálně vyvážený strom s n uzly. Naší úlohou bude opět navštívit všechny uzly, tentokrát ale bude strategie odlišná. Smíme se pohybovat pouze vpřed, z libovolného uzlu ale můžeme kdykoli skočit zpátky do kořene. Cesta podél jedné hrany trvá jednu sekundu, skok do kořenu je okamžitý, jeho doba trvání je zanedbatelná.

Řešení Inu, musíme do každého listu, těch je (n+1)/2. Každá cesta bude mít délku přesně $\log_2(n+1)$ – tento nález zdůvodněte a vysvětlete, proč je $\log_2(n+1)$ v našem případě celé číslo. Celkem tedy putovním strávíme (n+1)/2 * log(2,n+1) sekund. Totéž (že musíme do každého listu) platí pro maximálně nevyvážený strom. Tam jsou ovšem délky cest rovny postupně od nejdelší (ty jsou dvě) až po nejkratší (každá kratší délka se vyskytne jen jednou) :
(n-1)/2, (n-1)/2, (n-1)/2-1, (n-1)/2-2, (n-1)/2-3, ... , 3, 2, 1.
Součet této posloupnosti je ((n-1)/2+1)((n-1)/2+2)/2 -1 = (n+1)(n+3)/8 -1.

19.

Máme k dispozici 1024 sekundy tj. něco málo přes 1/4 hodiny. Jaká je maximální možný počet uzlů ve vyváženém stromu z předchozí úlohy, abychom jej stihli za tuto dobu projít? Jaká je maximální velikost maximálně nevyváženého stromu pro stejnou úlohu?

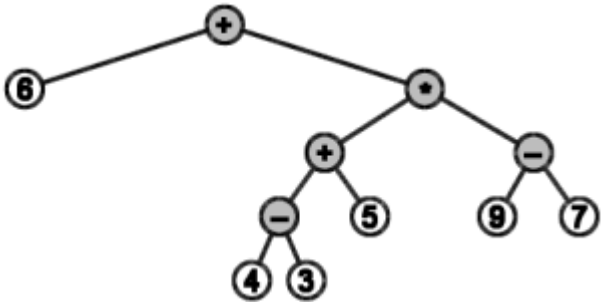
Řešení Zřejmě v případě vyváženého stromu hledáme takové maximální *k*, pro které platí
(2^*k*)/2 * log(2,2^*k*) <= 1024
2^(*k*-1) * *k* <= 1024
k = 7 => 64*7 = 448
k = 8 => 128*8 = 1024
tedy *n* = 255.
Pro druhý případ máme:
(*n*+1)(*n*+3)/8 -1 <= 1024
(*n*+1)(*n*+3)/8 <= 1025
(*n*+1)(*n*+3) <= 8200
To si spanilá čtenářka snadno vyřeší:
(*n*+2)(*n*+2)−1 <= 8200
(*n*+2)(*n*+2) <= 8201
n <= sqrt(8201)-2
n = 88;

20.
Vyzkoušejme ještě jednu variantu předchozí úlohy: Co když traverzování jedné hrany trvá 10^(-8) sec a my máme na projití stromu 1/10 sec?

Řešení Jsme schopni vykonat 10^7 kroků.
Vyvážený strom:
2^19-1 = *n* = 524 287 (půlmilionový)
Rozvážený strom:
Stejná úvaha jako v předchozí úloze vede na:
(*n*+2)(*n*+2) <= 10 000 001
n <= sqrt(10 000 001)-2
n <= 3160.27781828226
n = 3160.

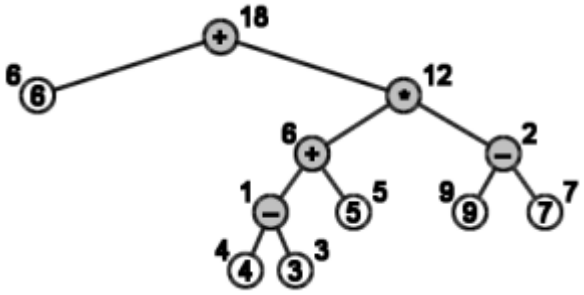
21.
Aritmetický výraz obsahující celá čísla, závorky a operace +,–,*,/ (celočíselné dělení) může být reprezentován jako pravidelný binární strom. Popište, jak takový strom obecně vypadá, navrhněte implementaci uzlu a napište funkci, jejímž vstupem bude ukazatel na kořen stromu a výstupem hodnota odpovídajícího aritmetického výrazu.

Řešení 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**ft 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...)

22.
Napište funkci, která z binárního stromu odstraní všechny listy. Předpokládejte, že každý uzel obsahuje ukazatel na svého rodiče.

23.
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.

Řešení 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);
```

---------- **POST IN PREORDER** ----------

1.
Všechny klíče uložené v uzlech binárního stromu vypíšeme v pořadí postorder. Strom má k listů. Ve vypsané posloupnosti se objeví

a) všechny klíče listů na prvních k pozicích
b) všechny klíče listů na posledních k pozicích
c) polovina klíčů listů na prvních k/2 pozicích a polovina na posledních k/2 pozicích
d) polovina klíčů vnitřních uzlů na prvních k/2 pozicích a polovina na posledních k/2 pozicích
e) klíče uzlů v žádné z předchozích konfigurací

2.
Klíče daného binárního vyhledávacího stromu vypíšeme v pořadí postorder. Vznikne posloupnost 2 9 4 5 1. Celkový počet uzlů v pravém podstromu kořene tohoto BVS je:

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

3.
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).

4.
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).

5.
The keys of the given BST are written out using the postorder scheme. The printed sequence is 4 3 7 9 6. The total number of nodes in the left subtree of the root is:

f) 0
g) 1
h) 2
i) 3
j) 4

6.
Klíče daného binárního vyhledávacího stromu vypíšeme v pořadí preorder. Vznikne posloupnost 2 9 4 5 1. Celkový počet uzlů v pravém podstromu kořene tohoto BVS je:

f) 0
g) 1
h) 2
i) 3
j) 4

7.
The keys of a binary search tree are printed out in using the postorder traversal scheme. The printed sequence is 2 9 4 5 1. Total number of the nodes in the right subtree of the tree root is

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

8.

Všechny klíče uložené v uzlech binárního stromu vypíšeme v pořadí preorder. Strom má k listů. Ve vypsané posloupnosti budou

a) všechny klíče listů na prvních k pozicích
b) všechny klíče listů na posledních k pozicích
c) polovina klíčů listů na prvních k/2 pozicích a polovina na posledních k/2 pozicích
d) polovina klíčů vnitřních uzlů na prvních k/2 pozicích a polovina na posledních k/2 pozicích
e) klíče uzlů v žádné z předchozích konfigurací

9.
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

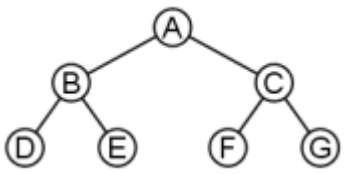
Řešení Definice praví, že se jedná o variantu c).

10.
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

Řešení Definice praví, že se jedná o variantu c).

11.
Obsah uzlů daného stromu vypíšeme v pořadí postorder. Vznikne posloupnost

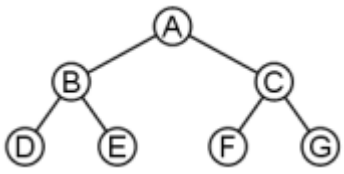
a) G F E D C B A
b) F C G D B E A
c) G C F A E B D
d) D E B F G C A



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

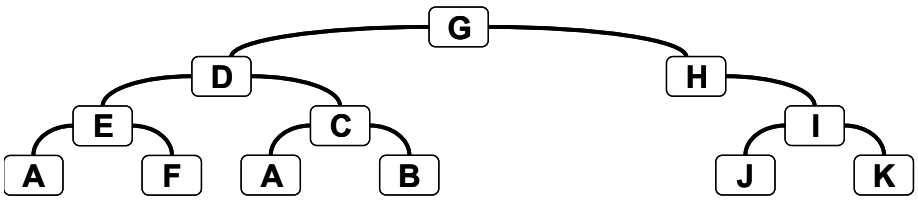
12.
Obsah uzlů daného stromu vypíšeme v pořadí preorder. Vznikne posloupnost

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



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

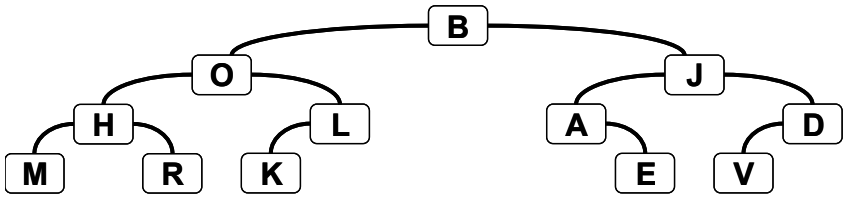
13.
Vypište obsah jednotlivých uzlů daného stromu při průchodu v pořadí
a) preorder
b) postorder



Řešení a) Preorder nejprve zpracuje (=vypíše) kořen aktuálního stromu a pak zpracuje levý a pak pravý podstrom tohoto kořene. Výpis pro daný strom tedy bude:
G D E A F C A B H I J K
b) Postorder nejprve zpracuje levý, pak pravý podstrom kořene aktuálního stromu a nakonec zpracuje (=vypíše) kořen. Výpis pro daný strom tedy bude:
A F E A B C D J K I H G

14.
Vypište obsah jednotlivých uzlů daného stromu při průchodu v pořadí

a) preorder
b) postorder



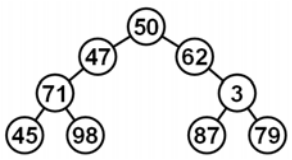
Řešení a) Preorder nejprve zpracuje (=vypíše) kořen aktuálního stromu a pak zpracuje levý a pak pravý podstrom tohoto kořene. Výpis pro daný strom tedy bude:
B O H M R L K J A E D V
b) Postorder nejprve zpracuje levý, pak pravý podstrom kořene aktuálního stromu a nakonec zpracuje (=vypíše) kořen. Výpis pro daný strom tedy bude:
M R H K L O E A V D J B

15.
Jaký musí mít binární strom tvar, aby
a) průchody In- Pre- a Postorder zpracovaly uzly ve stejném pořadí?
b) průchody In- Preorder zpracovaly uzly ve stejném pořadí?
c) průchody In- a Postorder zpracovaly uzly ve stejném pořadí?

Řešení a) Průchod Preorder zpracuje kořen jako první v pořadí a průchod Postorder jej zpracuje jako poslední v pořadí. Mají–li pořadí být identická, zbývá jen možnost, že strom obsahuje nejvýše jeden uzel – kořen.
b) V pořadí Inorder jsou všechny uzly levého podstromu uzlu *x* vždy zpracovány před uzlem *x*, zatímco v pořadí preorder, jsou zpracovány až po uzlu *x*. Aby tedy byla pořadí Preorder a Inorder identická, žádný uzel stromu nesmí obsahovat levý podstrom. Strom je tak tvořen jedinou holou větví táhnoucí se od kořene doprava.
c) Odpověď je analogická případu b), strom představuje jedinou větev bez větvení táhnoucí se o od kořene doleva.

16.
Při pruchodu daným stromem pořadí Inorder a pak Preorder získáme následující posloupnosti hodnot uložených v jeho jednotlivých (celkem devíti) uzlech:
Inorder: 45 71 98 47 50 62 87 3 79
Preorder: 50 47 71 45 98 62 3 87 79
a) Rekonstruujte tvar stromu.
b) Navrhněte a formulujte algoritmus, který z uvedených dvou posloupností pro libovolný strom rekonstruuje jeho podobu.

Řešení a) Strom je patrný na obrázku.



b)Tělo rekurzivní funkce bude řešit následující dvě úlohy. Nejprve najde kořen. To je snadné, je to první hodnota v poli Preorder. Dále je nutno určit, které části polí Inorder a Preorder odpovídají levému a pravému podstromu nalezeného kořene a tyto úseky předat k rekurzivnímu zpracování. Návratovou hodnotou funkce bude ukazatel (reference) na nalezený kořen v daném podstromu.

Vezmeme první hodnotu *x* v poli Preorder, to je kořen, najdeme *x* v poli Inorder. Všechny hodnoty v poli Inroder před výskytem *x* odpovídají levému podstromu *x*, všechny hodnoty za výskytem *x* v poli Inorder odpovídají pravému podstromu *x*. V poli Preorder leží všechny hodnoty levého podstromu *x* bezprostředně za *x,* jejich počet je ovšem totožný s počtem hodnot vlevo od prvku x v poli Inorder, tj počtu prvku v levém podstromu *x*. Ostatní hodnoty v poli Preorder představují pravý podstrom. Po tomto rozboru již celkem snadno sestavíme celou rekurzivní funkci. Jejími parametry budou obě pole a indexy začátku a konce úseku v každém poli odpovídajícího aktuálně zpracovávanému podstromu.

```
Node treeFromInPre (int[] inArr, int inFirst, int inLast,
                    int[] preArr, int preFirst, int preLast){

  if (preFirst > preLast) return null;   // empty array, no node

  // find root
  int inRoot = inFirst;
  while (inArr[inRoot] != preArr[preFirst]) inRoot++;

  // create and return current root node with its both L and R children
  Node left = treeFromInPre(inArr, inFirst, inRoot-1,
                            preArr, preFirst+1, preFirst+ inRoot-inFirst );
  Node right = treeFromInPre(inArr, inRoot+1, inLast,
                             preArr, preFirst+inRoot-inFirst+1, preLast);

  return new Node(inArr[inRoot], left, right);
}
```

Funkce předpokládá jednoduchou deklaraci uzlu typu **Node** s konstruktorem (využitým v posledním řádku funkce), jehož parametry jsou klíč a reference na kořen levého a pravého podstromu. Strom vybudujeme voláním

```
Node root =
     treeFromInPre(inArr, 0, inArr.lenght-1, preArr, 0, preArr.lenght-1);
```

kde **inArr** resp. **preArr** jsou pole stejné délky s vypsanými klíči v pořadí Inorder resp. Preorder.

17.

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

Řešení 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.

18.
Řešte předchozí úlohu pro pořadí inorder a postorder.
Řešení Samostatně.