DatovÉ struktury

Typy rozlišujeme *datovÉ* a *funkcionÁlní*.

DatovÉ typy lze zavÉst pomocí funkcionÁlních, případně funkcionÁlní typy lze simulovat pomocí nekonečných datových typů, ale kvůli efektivnosti implementace se rozlišují a uvažují se zvlÁšť.

Hodnoty funkcionÁlních typů jsou *funkce* a *procedury*. JednotlivÉ funkční hodnoty (či vedlejší efekty procedur) nejsou znÁmy předem, ale dospěje se k nim po výpočtu. Ten je spuštěn tzv. *volÁním* čili *aplikací na argumenty*.

Hodnoty datových typů jsou obvykle znÁmy a uloženy v paměti, ale jejich složky je třeba najít a zpřístupnit (například najít prvek pole podle indexu apod).

Datové typy mohou být *skalární* nebo *složené*.

Skalární datové typy v daném programovacím jazyce obvykle zahrnují číselné typy (celá nebo desetinná čísla z určité konečné množiny), znakové typy, typ pravdivostních hodnot apod. Data skalárního typu zabírají vždy konstantní a malé množství paměti (typicky do 10 B). Zpřístupnění hodnoty skalárního typu trvá konstantní dobu.

Složené datové typy jsou například záznamy (n-tice s pojmenovanými složkami), uniony, posloupnosti, množiny apod.
Data složených typů se nazývají *datové struktury*.

- Datové struktury pevné velikosti (n-tice, statické pole, ...) — tzv. *statické datové struktury*
- Datové struktury proměnné velikosti — tzv. *dynamické datové struktury*

StatickØ datovØ struktury mají konstantní velikost a časovÆ složitost zpřístupnění libovolnØho prvku je konstantní.

DynamickØ datovØ struktury mají velikost danou nějakou proměnnou n, jejíž hodnota se může měnit. ČasovÆ složitost zpřístupnění libovolnØho prvku dynamickØ datovØ struktury je neklesající funkcí zÆvislou na n; často lineÆrní, u efektivních datových struktur lepší, typicky logaritmickÆ.

DynamickØ datovØ struktury

Seznam

Seznam nad bÆzovým typem B je lineÆrní datovÆ struktura typu S s nÆsledujícími operacemi:

$\texttt{nil} : S$

$\texttt{cons} : B \times S \to S$

$\texttt{head} : S \dashrightarrow B$

$\texttt{tail} : S \dashrightarrow S$

$\texttt{null} : S \to \texttt{Bool}$

Pro tyto operace a každou hodnotu x typu B a každý seznam s typu S musí platit

Axiomy seznamu

```
null(nil)  =  True
null(cons(x,s))  =  False
head(cons(x,s))  =  x
tail(cons(x,s))  =  s
```

ZÆsobník

ZÆsobník nad bÆzovým typem B je lineÆrní datovÆ struktura typu S s nÆsledujícími operacemi:

$$\texttt{empty} : S$$

$$\texttt{push} : B \times S \to S$$

$$\texttt{top} : S \dashrightarrow B$$

$$\texttt{pop} : S \dashrightarrow S$$

$$\texttt{isempty} : S \to \texttt{Bool}$$

Pro tyto operace a každou hodnotu x typu $\mathbb{B}$ a každý zÆsobníks typu $\mathbb{S}$ musí platit tzv.

Axiomy zÆsobníku

$$\texttt{isempty(empty)} = \texttt{True}$$

$$\texttt{isempty(push(}x\texttt{,}s\texttt{))} = \texttt{False}$$

$$\texttt{top(push(}x\texttt{,}s\texttt{))} = x$$

$$\texttt{pop(push(}x\texttt{,}s\texttt{))} = s$$

Jak je vidět, zÆsobník a seznam je tatÆž datovÆ struktura.

Liší se pouze v použití:

O *zÆsobníku* obvykle mluvíme, když na něm používÆme jen zÆkladní operace a pracujeme jen s jedním zÆsobníkem nebo pevně daným malým počtem zÆsobníků. ZÆsobníky bývají dÆny předem: během výpočtu se mění jejich obsah, ale nemění se počet zÆsobníků, tj. neruší se a nevznikají nové zÆsobníky. Proto se v imperativních jazycích zÆkladní operace často implementují jako procedury; navíc jejich parametr zÆsobník se vynechÆvÆ, pracuje-li se jen s jedním zÆsobníkem.

U *seznamů* často definujeme složitější operace (zpřístupnění či změnu n-tØho prvku, rozdělení seznamu na dva, spojení dvou seznamů,...) a během výpočtu seznamy vytvÆříme a rušíme.

Například změnu třetího prvku (alespoň tříprvkovØho) seznamu s na číslo 5 lze realizovat složenou operací $\texttt{cons}(a,\texttt{cons}(b,\texttt{cons}(5,t)))$, kde $a = \texttt{head}(s)$, $b = \texttt{head}(\texttt{tail}(s))$, $t = \texttt{tail}(\texttt{tail}(\texttt{tail}(s)))$.

Fronta

Fronta nad bÆzovým typem B je lineÆrní datovÆ struktura typu Q s nÆsledujícími operacemi:

$$\texttt{empty} : Q$$

$$\texttt{head} : Q \dashrightarrow B$$

$$\texttt{enqueue} : B \times Q \to Q$$

$$\texttt{dequeue} : Q \dashrightarrow Q$$

$$\texttt{isempty} : Q \to \texttt{Bool}$$

Pro tyto operace a každØ hodnoty x, y typu B a každou frontu q typu Q musí platit

Axiomy fronty

$$\texttt{isempty(empty)} = \texttt{True}$$

$$\texttt{isempty(enqueue(}x\texttt{,}q\texttt{))} = \texttt{False}$$

$$\texttt{head(enqueue(}x\texttt{,empty))} = x$$

$$\texttt{head(enqueue(}x\texttt{,enqueue(}y\texttt{,}q\texttt{)))} = \texttt{head(enqueue(}y\texttt{,}q\texttt{))}$$

$$\texttt{dequeue(enqueue(}x\texttt{,empty))} = \texttt{empty}$$

$$\texttt{dequeue(enqueue(}x\texttt{,enqueue(}y\texttt{,}q\texttt{)))} = \texttt{enqueue(}x\texttt{,dequeue(enqueue(}y\texttt{,}q\texttt{)))}$$

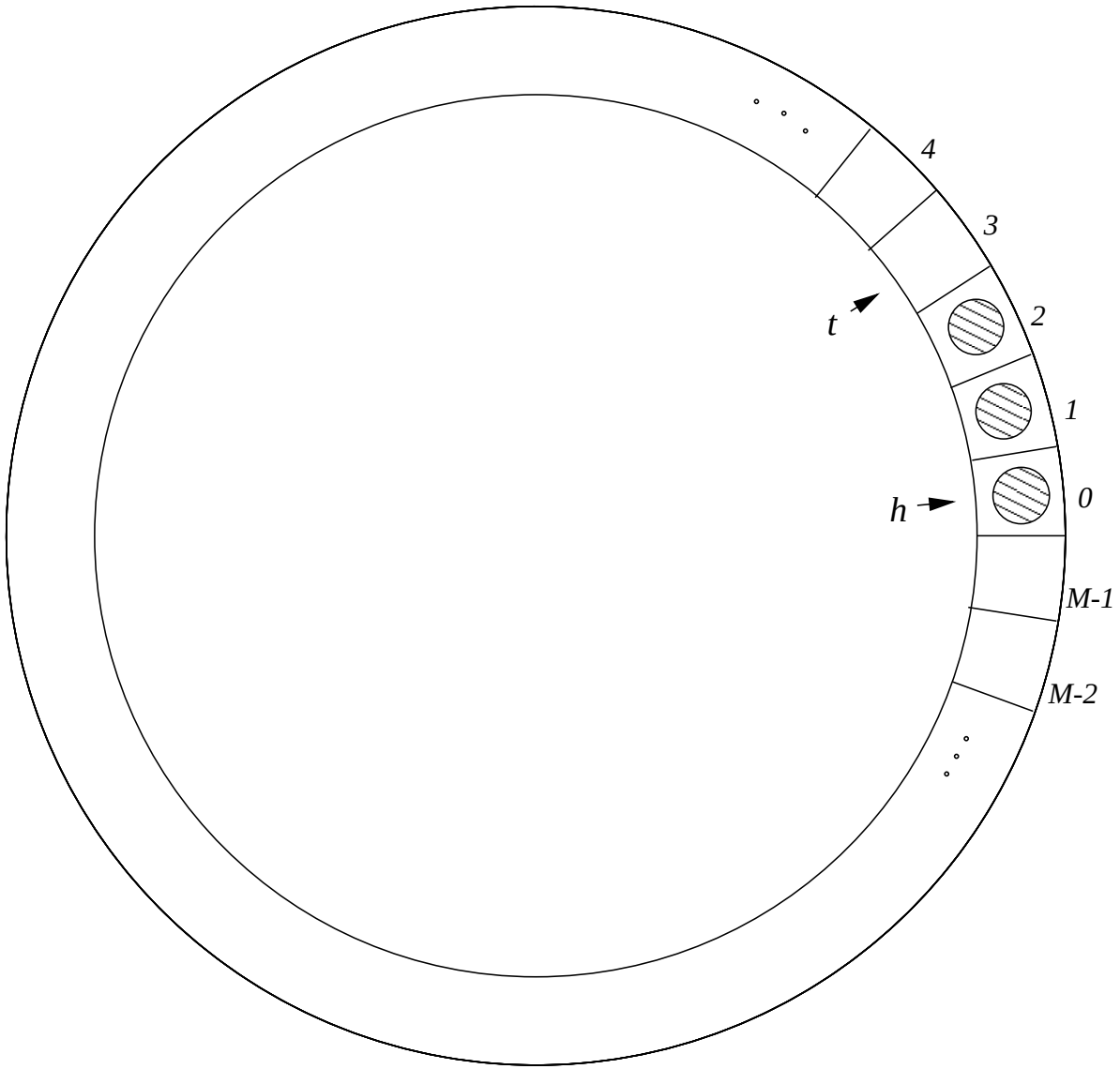
Implementace datových struktur

Jestliže typ datovÉ struktury není součÁstí jazyka, je nutnÉ vyjÁdřit datovou strukturu a operace nad ní pomocí jiných struktur a operací. TakovÉ kolekci definic říkÁme *implementace datovÉ struktury*.

Například zÁsobník (omezenÉ dÉlky) lze implementovat pomocí (statickÉho) pole,
frontu lze implementovat pomocí dvou zÁsobníků,
frontu omezenÉ dÉlky pomocí cyklickÉho pole apod.

PoznÆmka: Jazyky s malou podporou dynamických datových struktur (Pascal, C,...) často poskytují alespoň dynamickou datovou strukturu „nízkØ œrovnì“: *paměť* $\mathtt{M}$ spolu s typem *ukazatel* (adresa) $\mathtt{P}$, nulÆrní operací $\mathtt{nil}:\mathtt{P}$ prÆzdnØho ukazatele a unÆrní operací zpřístupnění (dereferencovÆní) čÆsti paměti $\mathtt{deref}:\mathtt{P}\dashrightarrow\mathtt{M}$.

Pak například seznam se běžně implementuje pomocí zÆznamů v $\mathtt{M}$, jejichž složky jsou ukazatele na další zÆznamy, apod.


4
3
t
2
1
h
0
M-1
M-2

Příklad: Implementace ohraničenØ fronty pomocí cyklickØho pole

```
{ procedury pracují pouze s jedinou globální frontou }

const M   = 256;  { délka pole }
      MAX = M-1;  { kapacita fronty }
type R = 0..MAX;
     Elem = Integer;
var Q : record
          a : array [R] of Elem;
          h, t : Integer
        end;
```

```
procedure mkemptyq;
  begin
    Q.h := 0;  Q.t := 0
  end;

function isempty : Boolean;
  begin
    isempty := Q.h = Q.t
  end;

function isfull : Boolean;
  begin
    isfull := Q.h = (Q.t + 1) mod M
  end;
```

```
function headq : Elem;
  begin
    if isemptyq then err("headq prázdné fronty")
                else headq := Q.a[Q.h]
  end;

procedure enqueue (x:Elem);
  begin
    if isfull
       then err("enqueue do plné fronty")
       else begin Q.a[Q.t] := x;
                  Q.t := (Q.t + 1) mod M
            end
  end;
```

```
procedure dequeue;
  begin
    if isemptyq then err("dequeue prázdné fronty")
                else Q.h := (Q.h + 1) mod M
  end;
```

Příklad: Implementace fronty pomocí dvou seznamů

```
data  Queue  =  Q [Int] [Int]

emptyq ::  Queue
emptyq  =  Q [] []

isemptyq  ::  Queue → Bool
isemptyq (Q [] [])  =  True
isemptyq _          =  False

enqueue :: Int → Queue → Queue
enqueue x (Q h t)  =  Q h (x:t)
```

```
headq :: Queue → Int
headq (Q (x:_) _)  =  x
headq q            =  head h
                      where  Q h _ = revq q

dequeue :: Queue → Queue
dequeue (Q (_:h) t)  =  Q h t
dequeue q            =  Q u []
                        where  Q (_:u) [] = revq q

revq (Q [] t)  =  Q (reverse t) []
```

Cvičení: Funkce `headq` a `dequeue` z předešlé strany zůstávají nedefinovány při aplikaci na prázdnou frontu. Doplňte jejich definice tak, aby jejich aplikace na prázdnou frontu způsobila chybové hlášení. Využijte haskellovské funkce `error :: String -> a`.

Dá se lehce spočítat, že časové složitosti operací `isemptyq` a `enqueue` z předchozího příkladu jsou konstantní, zatímco obě operace `headq` a `dequeue` jsou lineární vzhledem k velikosti fronty. V nepříznivém případě je totiž nutné volat pomocnou operaci `revq`, která je sama lineární.

Přesto i na tyto „dražší“ operace lze v kontextu programu, v němž jsou použity, pohlížet v jistém smyslu jako na operace v průměrném (očekávaném) případě konstantní.

AmortizovanÆ časovÆ složitost

Def: Nechť f je operace na danØ datovØ struktuře D velikosti (nejvýše) n. Uvažujme všechny výpočty $C_1, \ldots, C_m$ podle algoritmů pracujících s datovou strukturou D. Z každØho takovØho výpočtu vybereme všechna volÆní operace f (tj. všechny aplikace f na D), čímž dostaneme m posloupností volÆní operace f. Průměrnou dØlku výpočtu operace f v i-tØm výpočtu označíme τ_i^{av}. Potom *amortizovanÆ složitost* operace f na datovØ struktuře D je funkce $T^{\text{amort}} : \mathbb{N} \to \mathbb{N}$ definovanÆ pro každou velikost dat n takto

$$T^{\text{amort}}(n) = \max\{\tau_i^{\text{av}} | 1 \leq i \leq m\}$$

Příklad: Amortizovaná složitosti operací `headq` a `dequeue` z příkladu implementace fronty dvěma seznamy jsou konstantní.

Každé „drahé“ volání operace `dequeue` nebo `headq` se totiž „rozpustí“ v následujících aspoň n „levných“ voláních těchto operací.

Poznámka: Je-li časová složitost (tj. složitost v nejhorším případě) nějaké operace v $O(f)$, pak také její amortizovaná složitost je v $O(f)$.

Poznámka: Jestliže má nějaká operace amortizovanou složitost $\Theta(g)$, může to být méně příznivé, než když má složitost $\Theta(g)$, protože drahé volání se *může* vyskytnout. Na druhé straně je to příznivější než *průměrná* složitost $\Theta(g)$, protože amortizovaná složitost dává horní ohraničení pro průměrnou složitost v rámci jediného výpočtu, a to pro *každý* výpočet.

Stromy

Obecně *strom* je souvislý graf bez kružnic. Nejčastěji pracujeme s tzv. *kořenovými stromy*, tj. stromy, v nichž je jeden vyznačený uzel, *kořen*, a hrany jsou implicitně orientovÆny směrem od kořene k listům.

Je-li u uzel stromu a do uzlů $u_1, \ldots, u_k$ vedou z uzlu u hrany, pak uzly $u_1, \ldots, u_k$ nazývÆme *(bezprostředními) nÆsledníky* uzlu u.

Na bezprostředních nÆslednících každØho uzlu často bývÆ zavedeno œplnØ uspoøÁdÆní. Pak se nÆsledníci uzlu znÆzorňují zleva doprava; u binÆrních stromů se tedy rozlišuje *levý* a *pravý* nÆsledník.

Uzly bez nÆsledníků se nazývají *listy*, ostatní uzly stromu jsou *vnitřní*.

Stromy pevnØ arity (s omezeným větvením)

Def: Je dÆno přirozenØ číslo n a tzv. bÆzovÆ množina (typ$B$). Definujeme n*-ÆrnÍ strom* nad B takto:

- PrÆzdný strom$\emptyset$ je n-ÆrnÍ strom.
- Jsou-li $T_1, \ldots, T_n$ n-ÆrnÍ stromy a$b \in B$, pak ($n+1$)-tice $(b, T_1, \ldots, T_n)$ je n-ÆrnÍ strom.

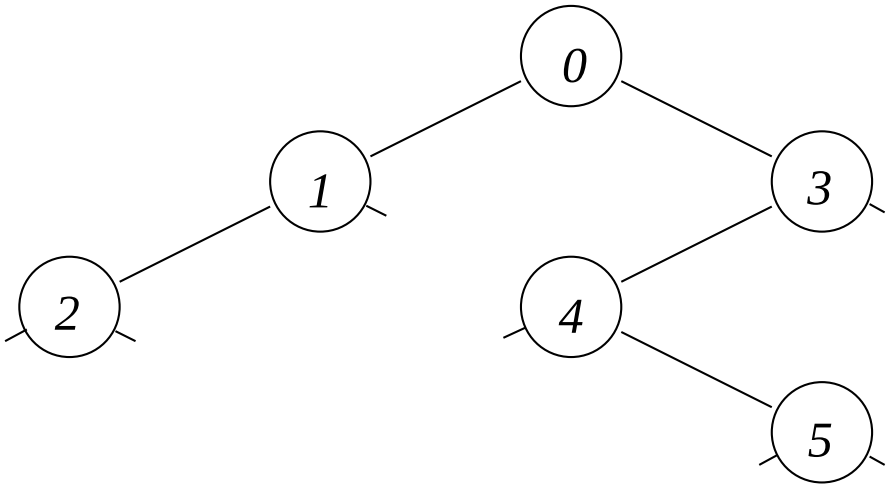
Listy n-ÆrnÍho stromu jsou uzly tvaru$(x, \emptyset, \ldots, \emptyset)$

Def: Cestu z kořene stromu do uzlu, jehož aspoň jeden bezprostřednÍ nÆsledník je prÆzdný strom, se nazývÆ*větev* stromu.

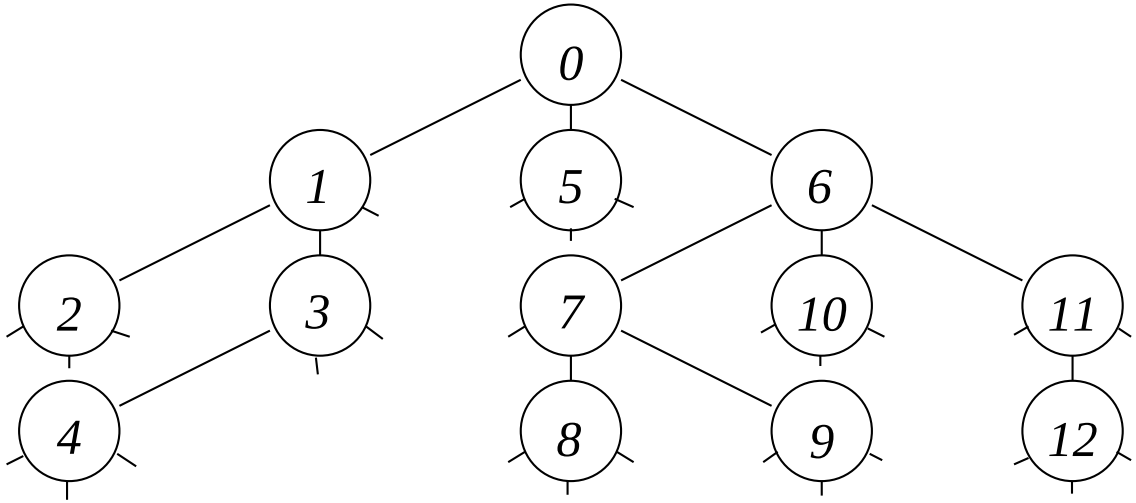
Opovídající datový typ zapsaný v Haskellu např. pro $n = 3$ je

```
data Tree3 b  =  Empty  |  Node b (Tree3 b) (Tree3 b) (Tree3 b)
```

BinÆrní strom (s pevným počtem nÆsledníků)



TernÆrní strom (s pevným počadím nÆsledníků)



BinÆrní stromy

Def: *BinÆrní strom* je strom arity 2.

ZÆkladní operace nad binÆrním stromem

$$\texttt{empty} : \texttt{T}$$

$$\texttt{node} : \texttt{B} \times \texttt{T} \times \texttt{T} \rightarrow \texttt{T}$$

$$\texttt{rootval} : \texttt{T} \dashrightarrow \texttt{B}$$

$$\texttt{left} : \texttt{T} \dashrightarrow \texttt{T}$$

$$\texttt{right} : \texttt{T} \dashrightarrow \texttt{T}$$

$$\texttt{isempty} : \texttt{T} \rightarrow \texttt{Bool}$$

PoznÆmka: ZÆkladní operace mají konstantní složitost, slouží k popisu datovØ struktury a k její implementaci a k implementaci dalších operací, jako například vyhledÆní / pidÆní / odebrÆní uzlu, vyvÆžení stromu a podobn.

Pro zÆkladní operace, každou hodnotu x typu `B` a každØ binÆrní stromy l, r typu `T` musí platit

Axiomy binÆrního stromu

$$\texttt{isempty(empty)} = \texttt{True}$$
$$\texttt{isempty(node(}x\texttt{,}l\texttt{,}r\texttt{))} = \texttt{False}$$
$$\texttt{rootval(node(}x\texttt{,}l\texttt{,}r\texttt{))} = x$$
$$\texttt{left(node(}x\texttt{,}l\texttt{,}r\texttt{))} = l$$
$$\texttt{right(node(}x\texttt{,}l\texttt{,}r\texttt{))} = r$$

Implementace binÆrních stromů v Haskellu

```
data Tree b  =  Empty  |  Node b (Tree b) (Tree b)

isempty :: Tree b → Bool
isempty Empty          =  True
isempty (Node _ _ _)   =  False

rootval :: Tree b → b
rootval (Node x _ _)   =  x

left :: Tree b → Tree b
left (Node _ l _)   =  l

right :: Tree b → Tree b
right (Node _ _ r)   =  r
```

Stromy s neomezeným větvením

Def: Je dÆna bÆzovÆ množina (typ) B. Nechť $b \in B$. Pak pro každØ přirozenØ číslo $k \geq 0$ a každou k-prvkovou posloupnost neprÆzdných stromů nad B je dvojice $(b, [T_1, \ldots, T_k])$ neprÆzdným *stromem s neomezeným větvením* nad B.

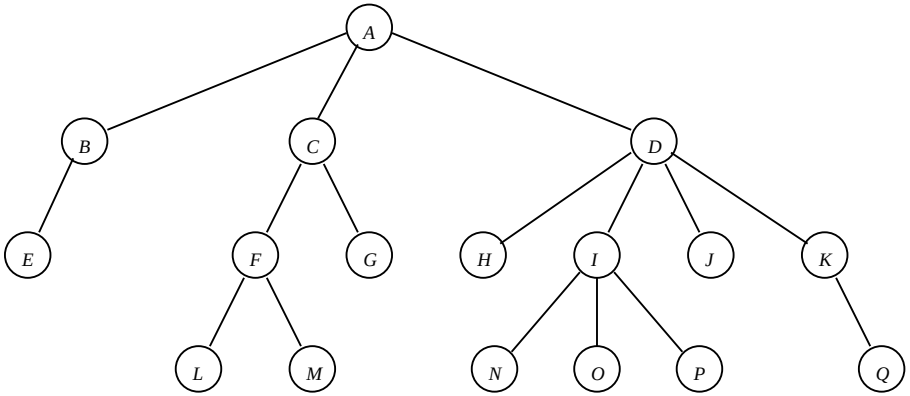
PoznÆmka: Druhou složkou uspořÆdanØ dvojice $(b, [T_1, \ldots, T_k])$ je k-prvkovÆ posloupnost stromů, tj. seznam dØlky k. Je-li $k = 0$, je seznam prÆzdný a dvojice reprezentuje jednouzlový strom (list) s ohodnocením b.

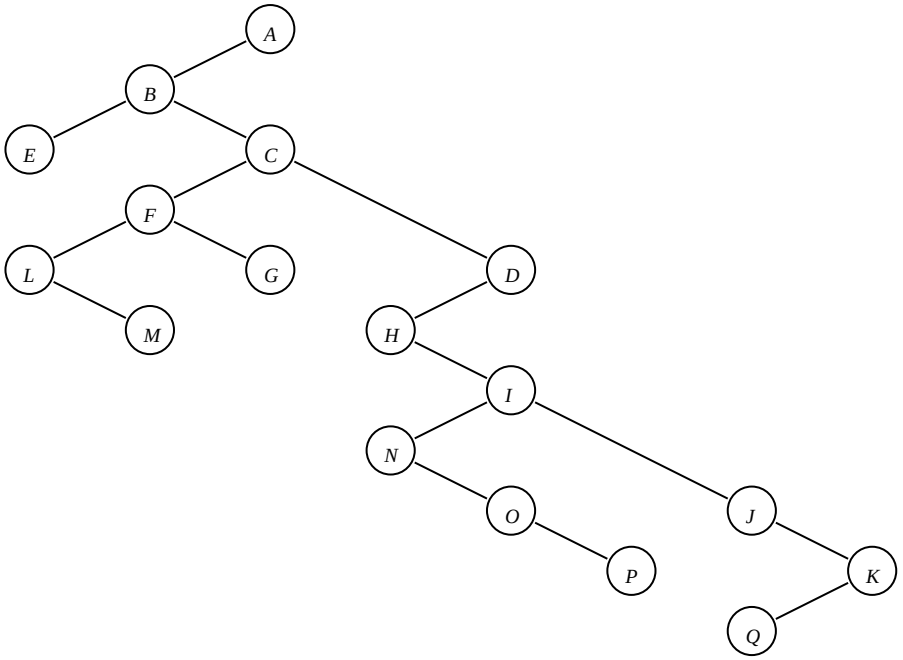
Stromy s neomezeným větvením se od stromů pevnØ arity liší tím, že číslo k není předem pevně danØ, ale může být v jednom stromu pro různØ uzly různØ.

PoznÆmka: Stromy s neomezeným větvením lze reprezentovat binÆrními stromy.

Stromy s neomezeným větvením a binÆrní stromy

```
data NTree a   =  NNode a [ NTree a ]
data BTree a   =  BEmpty  |  BNode a (BTree a) (BTree a)

fb                    ::  [ NTree a ] → BTree a
fb []                  =  BEmpty
fb (NNode v fr : frb)  =  BNode v (fb fr) (fb frb)

nb :: NTree a → BTree a
nb t  =  fb [t]

bf                ::  BTree a → [ NTree a ]
bf BEmpty          =  []
bf (BNode v ec ys)  =  NNode v (bf ec) : bf ys

bn ::  BTree a → NTree a
bn t  =  head (bf t)
```


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q


A
B
E
C
F
L
G
D
M
H
I
N
O
J
P
K
Q

VyhledÆvÆní

Def: Nechť $(K, \leq)$ je œplně uspořÆdanÆ množina tzv. *klíčů*, V je libovolnÆ množina tzv. doplňujících œdajů. Nechť $U = K \times V$ je množina dvojic (k, v), v níž se každý klíč k vyskytuje nejvýše jednou (tj. každý zÆznam z množiny U je určen jednoznačně svým klíčem).

ProblØmu nalØzt k danØmu klíči k zÆznam $(k, v) \in U$ se říkÆ *vyhledÆvací problØm*

PoznÆmka: Například zÆznamy mohou být osobní data a klíči jsou rodnÆ čísla, anebo zÆznamy jsou œdaje o knihÆch v knihovnØ a klíče jsou knihovní signatury apod.

PoznÆmka: V praxi mÆ vÄtšinou smysl pouze případ, kdy množina V je netriviÆlní, aby bylo co hledat.

V ukÆzkových algoritmech se však doplňující œdaje často neuvažují, tj. vyhledÆvají se jen klíče. PříslušnØ rozšíření těchto algoritmů je totiž přímočarØ.

Algoritmus binÆrního vyhledÆvÆní

```
type Elem = Integer;
     Pole = array [1..999] of Elem;

function bSearch (k: Elem; var D: Pole; n: Integer): Integer;
  { Posl. D je rostoucí. Když D[i]=k, tak bSearch(k)=i, jinak bSearch(k)=-1 }

  function bs (l, r: Integer) : Integer;
    var m : Integer;
    begin
      if l > r then bs := -1  {nenalezeno}
               else begin m := (l+r) div 2;
                          if        k < D[m] then bs := bs(l,m-1)
                            else if k > D[m] then bs := bs(m+1,r)
                            else   {k = D[m]}     bs := m
                    end
    end {bs};

  begin  bSearch := bs (1,n)  end
```

Věta: Algoritmus binÆrního vyhledÆvÆní mÆ logaritmickou časovou složitost, tj. jeho složitost je v $\Theta(\log n)$, kde n je dØlka prohledÆvanØho pole.

PoznÆmka: Extrasekvenční paměťovÆ složitost algoritmu je konstantní, protože rekursivní volÆní v něm jsou *prostÆ*

Cvičení: Implementujte algoritmus binÆrního vyhledÆvÆní bez použití rekurse.

Cvičení: Dokažte totÆlní korektnost algoritmu binÆrního vyhledÆvÆní.

HašovÆní

O množině K všech možných klíčů obecně předpoklÆdÆme jen to, že na ní existuje œplnØ uspořÆdÆní. Nikdy však tato množina může mít další speciÆlní vlastnosti, jichž lze využít pro efektivní vyhledÆvÆní. Například je-li $K = \{1, \ldots, m\}$ a číslo m je malØ, můžeme data z množiny V uložit do jednorozměrnØho pole a klíče využít jako indexy. VyhledÆvÆní v takovØto datovØ struktuře je efektivní – stejně rychlØ jako indexovÆní pole. Navíc, pokud se počet n skutečně uložených prvků bude blížit počtu m všech možných klíčů, bude efektivní i využití paměti.

Pokud je $|K| = m$, ale klíče nejsou čísla, lze to obejít pomocí bijektivní funkce $h : K \to \{1, \ldots, m\}$ a pole indexovat pomocí fukčních hodnot $h(k)$, kde $k \in K$. Je však důležitØ, aby výpočet hodnot funkce h byl rychlý, v ideÆlním případě aby měl konstantní časovou složitost. Funkci h nazývÆme *hašovací funkcí* a pole indexovanØ jejími hodnotami nazývÆme *hašovací tabulkou*.

Hašovací tabulky

Hašovací tabulka je datovÆ struktura, pomocí níž lze prakticky efektivně realizovat „slovníkovØ“ operace vyhledÆní, přidÆní a zrušení položky.

„Prakticky efektivně“ znamenÆ, že operace mají příznivou *průměrnou* časovou složitost (tedy ne nutně časovou složitost v nejhorším případě).

Hašovací tabulka je jednorozměrnØ pole H indexovanØ čísly $1, \ldots, n$. Převod klíčů na čísla realizuje *hašovací funkce* $h : K \to \{1, \ldots, n\}$. Výpočet hodnot hašovací funkce musí být efektivní, nejlØpe složitosti $\Theta(1)$.

Častým případem však je, že počet n skutečně uložených prvků je podstatně menší než počet m všech možných klíčů. I v tomto případě lze postupovat podobně a data uklÆdat do n-prvkovØho pole, až na to, že funkce $h : K \to \{1, \ldots, n\}$ nebude injektivní. Bude tedy existovat index i a klíče $k_1, \ldots, k_r$, tak, že $h(k_1) = \cdots = h(k_r) = i$. To nemusí vadit, pokud je splněna nÆsledující podmínka. Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, kterÆ budou skutečně uložena (tedy $|K'| \leq n$). Pak požadujeme, aby z každØ množiny klíčů, kterØ hašovací funkce zobrazí na stejný index, byl v množině K' nejvýše jeden klíč. MÆ-li pro pevně danou množinu K' klíčů skutečně uložených dat hašovací funkce tuto vlastnost, nazývÆme ji *dokonalou hašovací funkcí pro klíče z* K'. Výhodou tabulek s dokonalými hašovacími funkcemi je optimÆlní složitost vyhledÆní, vložení i zrušení prvku; je stejnÆ jako pro pole.

Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, kterÆ budou skuteènì uložena. Klíče z množiny K' nazveme *použitØ klíče*.

Je-li zœžení $h|_{K'} : K' \to \{1, \ldots, n\}$ injektivní, říkÆme, že hašovací funkce je *dokonalÆ* vzhledem k množině použitých klíčů K'.

Věta: MÆ-li výpoèet hašovací funkce konstantní složitost a hašovÆní je dokonalØ pro množinu použitých klíčů K', pak operace vyhledÆní, vložení, resp. zrušení prvku v hašovací tabulce mají stejnou časovou složitost, jako vyhledÆní, vložení, resp. zrušení prvku v poli.

Nevýhodou dokonalých hašovacích funkcí je, že zÆvisí na množině K'. Tato množina musí být znÆma předem, abychom mohli dokonalou hašovací funkci sestrojit. V praxi však uklÆdanÆ datařpředem neznÆme a tato data se mění. Proto v praxi používanØ hašovací funkce většinou nebývají dokonalØ a musí se počítat s takzvanými *kolizemi*.

Kolize je případ, kdy mÆ být více dat uloženo na jednØ pozici hašovací tabulky, tj. chceme uložit data s různými klíči $k_1, \ldots, k_r$ a $h(k_1) = \cdots = h(k_r)$.

Kolize

Tzv. kolize vznikají při nedokonalém hašování: hašovací funkce zobrazuje více použitých klíčů na stejnou pozici pole.

Nejjednodušší řešení kolizí je ukládat do každé pozice pole *seznam* prvků. V něm se pak hledá sekvenčně.

Složitost přidání prvku zůstává stejná jako složitost indexování, tedy složitost vyhledání i složitost zrušení prvku je $\Theta(n)$. To je sice velmi špatný výsledek, ale v praxi nevadí, protože *průměrná* časová složitost dopadne pro vhodně sestrojenou hašovací funkci mnohem lépe.

Věta: Nechť hašovací funkce zobrazuje použité klíče na indexy $1, \ldots, n$ rovnoměrně, tj. pravděpodobnost, že $h(k) = i$, je stejná pro všechny indexy i, $1 \leq i \leq n$.

Pak průměrná časová složitost vyhledání, přidání a zrušení prvku pak je $O\left(\left|K'\right| / n\right)$, za předpokladu, že složitosti výpočtu hašovací funkce i indexování pole jsou konstantní.

Pravděpodobnostní rozložení výskytů klíčů v množině K' nemusí být rovnoměrnØ, ale dobře navrženÆ hašovací funkce transformuje toto rozložení na rovnoměrnØ v množině indexů $\{1, \ldots, n\}$. To znamenÆ, že pro nÆhodně zvolený klíč $k \in K'$ a index i, $1 \leq i \leq n$, je pravděpodobnost, že hodnota $h(k) = i$, stejnÆ nezÆvisle na k a i a rovnÆ $\frac{1}{n}$.

Nyní pro jednoduchost předpoklÆdejme, že $K = \{1, \ldots, m\}$ a $m \geq n$, kde n je velikost hašovací tabulky. Nechť p je prvočíslo, $p \geq m$, a nechť a, b jsou celÆ čísla, $0 < a < p$, $0 \leq b < p$. Hašovací funkci $h_{a,b}$ zavedeme takto:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod n$$

Pak při volbě parametrů a, b nezÆvisle na K (což je v praxi splněno) mají hodnoty $h(k)$ rovnoměrnØ pravděpodobnostní rozložení na množině indexů $\{0, \ldots, n-1\}$.

Cvičení: Mějme tabulku indexovanou $0, \ldots, 8$, hašovací funkci h, $h(k) = k \bmod 9$.
Vložte do tabulky postupně klíče $14, 5, 28, 19, 15, 20, 33, 12, 17, 10, 18, 24$.

BinÆrní vyhledÆvací stromy

Def: *BinÆrní vyhledÆvací strom* je binÆrní strom nad œepÉuspoøÆdanou množinou (tzv. *klíčů*) $(K, \leq)$ takový, že pro každý jeho podstrom t platí:
hodnoty uzlů v podstromu $\texttt{left}(t)$ jsou menší než $\texttt{rootval}(t)$ a hodnoty uzlů v podstromu $\texttt{right}(t)$ jsou větší než $\texttt{rootval}(t)$.

Operace nad binÆrními vyhledÆvacími stromy

Datový typ

```
data  STree a  =  Empty  |  Node a (STree a) (STree a)
```

ZjišťovÆní příslušnosti

```
member                :: a → STree a → Bool
member _ Empty         = False
member k (Node v l r) =    k == v
                        || k < v && member k l
                        || k > v && member k r
```

VyhledÆvÆní

```
search ::  a → STree a → STree a
search _ Empty           = Empty
search k t@(Node v l r)
       | k == v          = t
       | k < v           = search k l
       | otherwise       = search k r
```

VklÆdÆní uzlu

```
insert                    :: a → STree a → STree a
insert k Empty             = Node k Empty Empty
insert k t@(Node v l r)
       | k < v             = Node v (insert k l) r
       | k > v             = Node v l (insert k r)
       | otherwise         = t
```

Rušení uzlu

```
delete :: a → STree a → STree a
delete _ Empty          = Empty
delete k (Node v l r)
       | k < v          = Node v (delete k l) r
       | k > v          = Node v l (delete k r)
       | otherwise      = join l r

join :: STree a → STree a → STree a
join l     Empty  =  l
join Empty r      =  r
join l     r      =  Node u (delete u l) r
                     where u = rightmostkey l

rightmostkey (Node v _ Empty) = v
rightmostkey (Node _ _ r)     = rightmostkey r
```

Cvičení: Vybudujte vyhledÆvací strom z posloupnosti klíčů
$9, 12, 10, 7, 12, 1, 8, 5, 11, 4, 0, 14, 13, 3, 6, 2$.

Cvičení: Zrušte v něm uzel s klíčem 5.

Cvičení: Definujte v Haskellu funkci `leftmostkey`.

Cvičení: Dokažte, že binÆrní strom vzniklý vložením resp. zrušením uzlu z vyhledÆvacího stromu pomocí funkcí `insert` resp. `delete` bude opět vyhledÆvací.

Cvičení: Modifikujte funkce `member`, `search`, `insert`, `delete` tak, aby pracovaly s realističtějším vyhledÆvacím stromem, v němž jsou kromě klíčů uložena i vlastní data:

```
data STree a b = Empty | Node (a,b) (STree a b) (STree a b)

member :: a → STree a b → Bool

search :: a → STree a b → Maybe b

insert :: (a,b) → STree a b → STree a b

delete :: a → STree a b → STree a b
```

Složitost vyhledÆvÆní ve vyhledÆvacích stromech

Označíme-li $T : \mathbb{N} \to \mathbb{N}$ složitost funkce `search`, pak

$$\begin{aligned} T(0) &= c \\ T(h) &= c' + \max\{T(h'), T(h'')\} \end{aligned}$$

kde h je hloubka vyhledÆvacího stromu,h' je hloubka jeho levØho podstromu,h'' je hloubka jeho pravØho podstromu ac, c' jsou konstanty.

Zřejmě $\max\{T(h'), T(h'')\} = h - 1$ a řešením uvedenØ rekursivní soustavy rovnic je lineÆrní funkce.

Složitost vyhledÁvÁní ve vyhledÁvacích stromech

Složitosti operací vyhledÁní, pidÁní a zrušení položky ve vyhledÁvacím stromě jsou přímo úmČrnÉ hloubce stromu. Ta je v nejhorším případě lineÁrně zÁvislÁ na velikosti stromu (počtu jeho uzlů).

Složitost vyhledÁvÁní, vklÁdÁní a rušení v obecnÉm vyhledÁvacím stromě je tedy lineÁrní.

AVL stromy

NazvanØ podle G. M. Adeľsona-VeľskØho a E. M. Landise.

Def: VyhledÆvací binÆrní strom je *AVL*, když hloubka levØho a pravØho podstromu libovolnØho uzlu se liší nejvýše o jednu.

Věta: Hloubka AVL stromu v zÆvislosti na počtu jeho uzlů je vždy v $\Theta(\log)$.

AVL stromy tedy mají logaritmickou hloubku — použijeme-li je jako vyhledÆvací stromy, pak mÆ operace vyhledÆní položky logaritmickou složitost.

Def: *Fibonacciho strom* řÆdu k definujeme takto:

$$\begin{aligned} FT_0 &= \texttt{empty} \\ FT_1 &= \texttt{node}(\texttt{empty}, \texttt{empty}) \\ \text{pro } k \in \mathbb{N} \text{ je } FT_{k+2} &= \texttt{node}(FT_k, FT_{k+1}) \end{aligned}$$

Lemma: Pro $k \geq 1$ je Fibonacciho strom FT_k *minimÆlní* (vzhledem k počtu uzlů) AVL strom hloubky $k - 1$.

Důkaz: Indukcí přes k se snadno ukÆže, že hloubka neprÆzdnØho stromu FT_k je $k - 1$.

Odtud a z definice Fibonacciho stromů vyplývÆ, že Fibonacciho stromy jsou AVL.

Minimalita je důsledkem předchozích dvou bodů: odebrÆním nějakØho uzlu z jinØ než nejpravější větve by někde vznikly sousední podstromy s hloubkami lišícími se aspoň o dvě; odebrÆním uzlu z nejpravější větve by se snížila hloubka celØho stromu.

Věta: Hloubka AVL stromu v zÆvislosti na počtu jeho uzlů je vždy v $\Theta(\log)$.

Důkaz: Označme $N(k)$ velikost stromu FT_k. Tedy $N : \mathbb{N} \to \mathbb{N}$ a

$$\begin{aligned} N(0) &= 0 \\ N(1) &= 1 \\ \text{pro } k \in \mathbb{N} \text{ je } N(k+2) &= 1 + N(k) + N(k+1) \end{aligned}$$

Protože funkce N majorizuje Fibonacciho funkci, je $N \in \Omega(\varphi^k)$, kde $\varphi = \dfrac{1+\sqrt{5}}{2}$ a za k bereme řÆd stromu. Ale víme, žeřÆd stromu a hloubka je (skoro) totØž, takže dostÆvÆme, že počet uzlů Fibonacciho stromu hloubky h roste aspoň tak rychle jako φ^h.

Protože Fibonacciho strom je minimÆlní AVL strom, mÆme, žepočet uzlů *každØho* AVL stromu hloubky h roste aspoň tak rychle jako φ^h. ObrÆceně, každý AVL strom velikosti n mÆ hloubku $O(\log_\varphi n)$.

Víme, že hloubka každØho binÆrního stromu velikosti n je $\Omega(\log_2 n)$.

Dohromady mÆme hloubku v $O(\log_\varphi n) \cap \Omega(\log_2 n) = \Theta(\log)$.

Operace na AVL stromech

DatovÆ struktura:

```
data AVL a = Empty | Node Int a (AVL a) (AVL a)
```

Každý uzel nese informaci o hloubce (jakožto parametr konstruktorovØ funkce `Node`).

VyhledÆvÆní v AVL stromu se neliší od vyhledÆvÆní v obecnØm vyhledÆvacím stromu.

Po přidÆní nebo odebrÆní položky ovšem může nastat situace, že vyhledÆvací strom přestane splňovat podmínku AVL. Pak je nutnØ pozměnit strukturu stromu.

PřidÆvÆní položky (novØho uzlu) do AVL stromu

StejnØ jako u bžnØho vyhledÆvacího stromu, ale s kontrolou vyvÆženosti.

PřidÆvaný uzel oznaíme x.

Pokud se poruší vyvÆženost stromu, nalezne se nejmenší podstrom F, který je nevyvÆžený. Oznaíme-li h jeho hloubku před přidÆním uzlu x, bude po přidÆní jeho hloubka $h + 1$. Kořen stromu F označíme f.

Bez œejmy na obecnosti lze předpoklÆdat, že uzel x je přidÆvÆn do *levØho* podstromu stromu F. Nechť B je levý podstrom stromu F, G je pravý podstrom stromu F.

Strom B je neprÆzdný (jinak by přidÆvÆní uzlu x nemohlo porušit vyvÆženost stromu F). Označíme A resp. D jeho levý resp. pravý podstrom, b bude kořen stromu B.

Rozlišíme dva případy:

1. Uzel x je přidÆn do stromu A. Pak strom A mÆ hloubku $h-1$, stromy D, G mají hloubku $h-2$.

   Vytvoříme strom $T = \texttt{Node}\ h\ b\ A\ (\texttt{Node}\ (h{-}1)\ f\ D\ G)$.

   Pak T je AVL strom hloubky h se stejnými uzly jako v F.

2. Uzel x je přidÆn do stromu D. Pak stromy A, G mají hloubku $h-2$, strom D mÆ hloubku $h-1$. Označíme d resp. C resp. E kořen resp. levý podstrom resp. pravý podstrom stromu D.

   Vytvoříme strom

$$T = \texttt{Node}\ h\ d\ \ (\texttt{Node}\ (h{-}1)\ b\ A\ C)\ \ (\texttt{Node}\ (h{-}1)\ f\ E\ G)$$

Dva dílčí podpřípady jsou:

(a) Uzel x byl přidÆn do stromu C. Pak C mÆ hloubku $h-2$, E mÆ hloubku $h-3$.

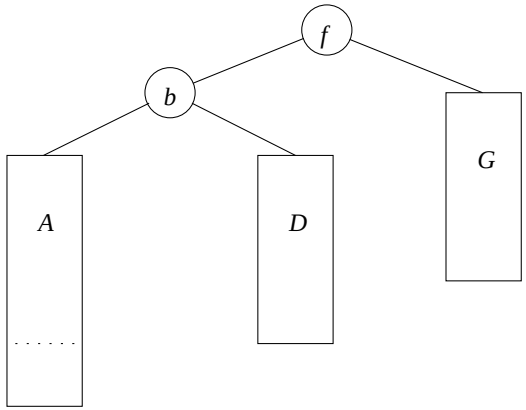
(b) Uzel x byl přidÆn do stromu E. Pak C mÆ hloubku $h-3$, E mÆ hloubku $h-2$.

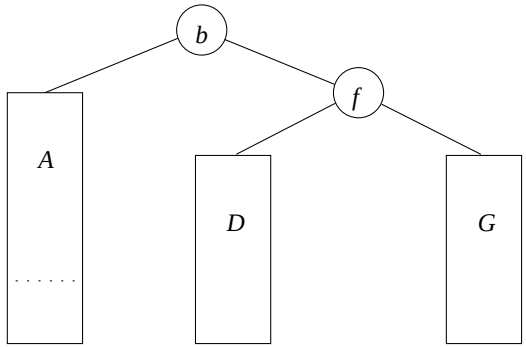
V obou případech však strom T mÆ hloubku h, je AVL a mÆ stejnØ uzly jako strom F.

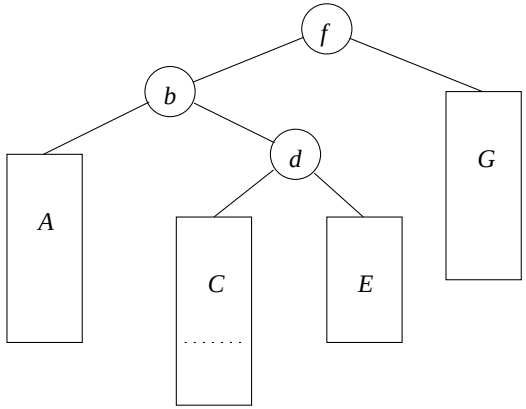
V původním AVL stromu nahradíme podstrom F stromem T. Jelikož T mÆ stejnou hloubku jako měl původní podstrom bez uzlu x, zůstane celý strom vyvÆžený (AVL).

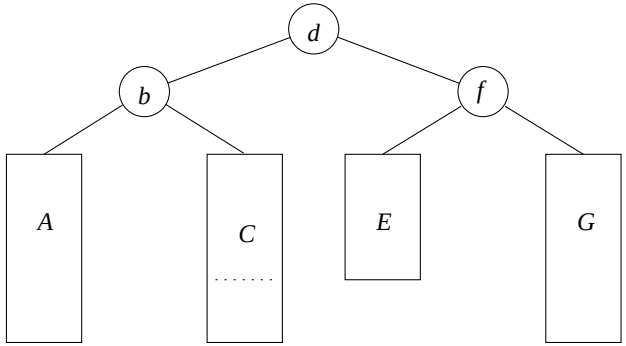
PřepočítÆme hloubky na cestě od kořene stromu T k uzlu x.

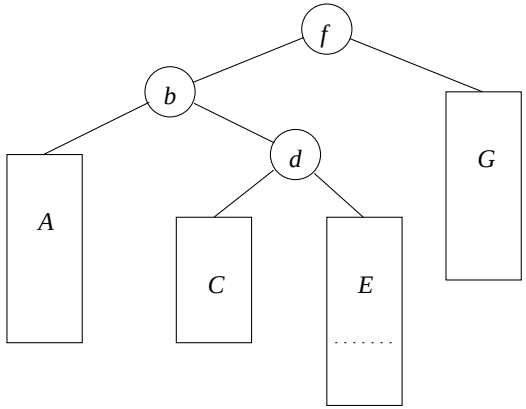
Byl-li uzel x přidÆvÆn do *pravØho* podstromu stromu F, je postup analogický (stranově převrÆcený).

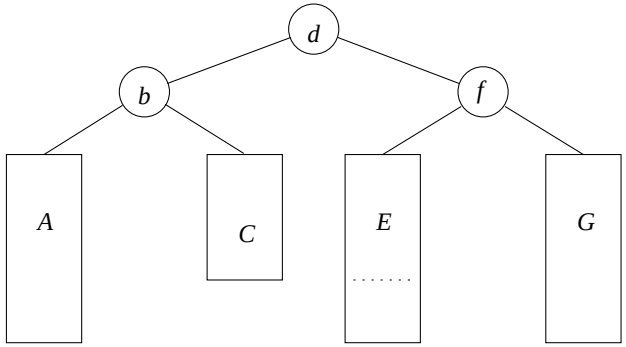

f
b
A
D
G


b
f
A
D
G


f
b
d
G
A
C
E


d
b
f
A
C
E
G


f
b
d
G
A
C
E


d
b
f
A
C
E
G

Složitost operace přidÆní položky do AVL

Hloubka AVL stromu je logaritmickÆ vzhledem k počtu jeho uzlů.

PřidÆní uzlu x: $\Theta(\log n)$.

Nalezení nejmenšího nevyvÆženØho podstromu:
Protože hloubky podstromů jsou spočteny a uloženy v datovØ struktuře, stačí po cestě k uzlu x testovat, zda rozdíl spočtených hloubek levØho a pravØho podstromu každØho uzlu je v množině $\{-1, 0, 1\}$. Nejnižší uzel, který tuto podmínku nesplňuje, je kořen podstromu F. Jeho nalezení trvÆ $\Theta(\log n)$.

Hloubky podstromů se přepočítÆvají jen po cestě od kořene k x, tedy složitost tØto œpravy je $\Theta(\log n)$.

CelkovÆ složitost přidÆní uzlu je tedy $\Theta(\log n)$.

Rušení položky (uzlu) z AVL stromu

Rušení vnitřního uzlu převedeme na rušení listu (podobně jako u vyhledÆvacího stromu).

Odebíraný list označíme x.

Pokud se poruší vyvÆženost stromu, nalezne se nejmenší podstrom B, který je nevyvÆžený. Označíme h jeho hloubku (před i po zrušení uzlu x je stejnÆ).

Bez œejmy na obecnosti lze předpoklÆdat, že uzel x byl odebrÆn z *levØho* podstromu stromu B. Nechť A je levý podstrom stromu B, F je pravý podstrom stromu B.

Strom F je neprÆzdný (jinak by rušení uzlu x nemohlo porušit vyvÆženost stromu B). Označíme D resp. G jeho levý resp. pravý podstrom, f bude kořen stromu F.

Rozlišíme dva případy:

1. Hloubka stromu G je větší nebo rovna hloubce stromu D.

   Vytvoříme strom $T = \texttt{Node}\ h'\ f\ (\texttt{Node}\ (h'-1)\ b\ A\ D)\ G$.

   Pak T je AVL strom hloubky h' se stejnými uzly jako v B, $h' = h$ nebo $h' = h - 1$.

2. Hloubka stromu G je menší než hloubka stromu D. Označíme d resp. C resp. E kořen resp. levý podstrom resp. pravý podstrom stromu D.

   Vytvoříme strom

   $$T = \texttt{Node}\ h'\ d\ \ (\texttt{Node}\ (h'-1)\ b\ A\ C)\ \ (\texttt{Node}\ (h'-1)\ f\ E\ G)$$

   Pak strom T má hloubku $h' = h - 1$, je AVL a má stejné uzly jako měl strom B.

V původním AVL stromu nahradíme podstrom B stromem T.

Přepočítáme hloubky na cestě od kořene stromu T k uzlu x.

V případě rušení uzlu se může stÆt, že hloubka podstromu T bude menší než hloubka původního podstromu, který byl na jeho místě.

Proto je nutnØ proces vyvažovÆní opakovat: nalezne se nejmenší nadstrom T_2 stromu $T = T_1$, který není vyvÆžený, vyvÆžíme ho, nalezneme další nevyvÆžený nadstrom T_3 . . . atd., až je vyvÆžený celý strom, T_k.

Hloubky však stačí přepočítÆvat vždy od kořene stromu T ke kořenu nejbližšího vyvažovanØho nadstromu.

Složitost operace rušení položky z AVL

Zrušení listu: $\Theta(\log n)$.

Nechť hloubka nejmenšího nevyvÆženØho podstromu T_1 je h_1. Nalezení tohoto podstromu trvÆ $\Theta(h_1)$, jeho vyvÆžení (rotace uzlů) trvÆ konstantní dobu, přepočítÆní hloubek trvÆ $\Theta(h_1)$.

Nechť pro $1 < i \leq k$ je h_i dØlka cesty z kořene podstromu T_{i-1} do kořene stromu T_i. Pak každØ nalezení dalšího nevyvÆženØho podstromu T_i trvÆ $\Theta(h_i)$, jeho vyvÆžení trvÆ $\Theta(1)$, přepočítÆní hloubek trvÆ $\Theta(h_i)$.

To znamenÆ, že celkovÆ složitost rušení uzlu je $\Theta\left(\sum_{i=1}^{k} h_i\right) = \Theta(\log n)$.

Cvičení: Na vstupu jsou čísla $8, 3, 5, 0, 2, 4, 1, 6, 9, 7$ a při jejich načítÆní se postupně vytvÆří AVL strom s uzly ohodnocenými těmito čísly. Nakreslete tento AVL strom v každØm kroku vytvÆření (tj. po přidÆní každØho uzlu).

Cvičení: Do AVL stromu, který je zpočÆtku prÆzdný, se postupně přidÆvají položky $4, 1, 2, 7, 5, 6$, pak se odebere položka 7, přidÆ položka 3, odeberou se postupně $5, 6, 1$. Určete stav AVL stromu v každØm kroku.

Cvičení: AVL strom mÆ uzly ohodnocenØ čísly 1 až 20 a mÆ strukturu Fibonacciho stromu šestØhořÆdu. Popište a nakreslete proces rušení listu obsahujícího klíč 2.

Cvičení: Fibonacciho strom čtvrtØhořÆdu na sedmi uzlech je ohodnocen jako AVL strom čísly $1, 3, 5, 7, 9, 11, 13$. Do tohoto stromu přidÆme jako další položku nÆhodně zvolenØ sudØ číslo k, $0 \leq k \leq 14$. JakÆ je pravděpodobnost, že budeme muset rotovat uzly, abychom zachovali AVL vyvÆženost?

ČernobílØ stromy

ČernobílØ stromy jsou binÆrní vyhledÆvací stromy, jejichuzly nesou kromě klíče další atribut — *barvu* — černou nebo bílou.

Def: *Černobílý strom* je binÆrní vyhledÆvací strom, jehož každý uzel je obarven černou nebo bílou barvou. Musí splňovat tyto podmínky:

1. Kořen stromu je černý.
2. Je-li vnitřní uzel bílý, jeho nÆsledníci (pokud existují) jsoučerní.
3. Všechny větve obsahují stejný počet černých uzlů.

PoznÆmka: Vedle nÆzvučernobílØ *stromy* se lze setkat i s doslovnými překlady anglickØho nÆzvu*red-black tree* (drzewo czerwono-czarne, rot-schwarzer Baum, ruĝnigra arbo, vörös-fekete fa,. . .).

Def: *ČernÆ hloubka* černobílØho stromu t je počet černých uzlů na libovolnØ větvi. Značíme ji $\mathrm{bh}(t)$.

Lemma: Hloubka černobílØho stromu t na n uzlech je nejvýše $2\log_2(n+1)$.

Důkaz: Indukcí podle hloubky stromu se ukÆže, že každý černobílý strom t' mÆ aspoň $2^{\mathrm{bh}(t')} - 1$ uzlů. Odtud vyplývÆ $\mathrm{bh}(t') \leq \log_2(n+1)$. Ale podle definice černobílØho stromu jeho hloubka nepřevyšuje dvojnÆsobek jeho černØ hloubky. Odtud plyne tvrzení.

Důsledek: VyhledÆvÆní (operace `member` a `search`) v černobílØm stromě mají složitost $\Theta(\log n)$.

ČernobílØ stromy v Haskellu

```
data  Barva  =  Ce  |  Bi
data  CBS a  =  E  |  N Barva a (CBS a) (CBS a)
```

VyhledÆvÆní v černobílØm stromu je stejnØ jako v nevyvÆženØm vyhledÆvacím stromu.

ZjišťovÆní příslušnosti

```
member ::  a → CBS a → Bool
member _ E            = False
member k (N _ v l r) =    k == v
                       || k < v && member k l
                       || k > v && member k r
```

VyhledÆvÆní

```
search :: a → CBS a → CBS a
search _ E                  = E
search k t@(N _ v l r)
       | k == v             = t
       | k < v              = search k l
       | otherwise          = search k r
```

PřidÆvÆní uzlu do černobílØho stromu

PřidÆvaný uzel bude bílý. Tím se nezmění černÆ hloubka podstromů, ale mohou se dostat pod sebe dva bílØ uzly.

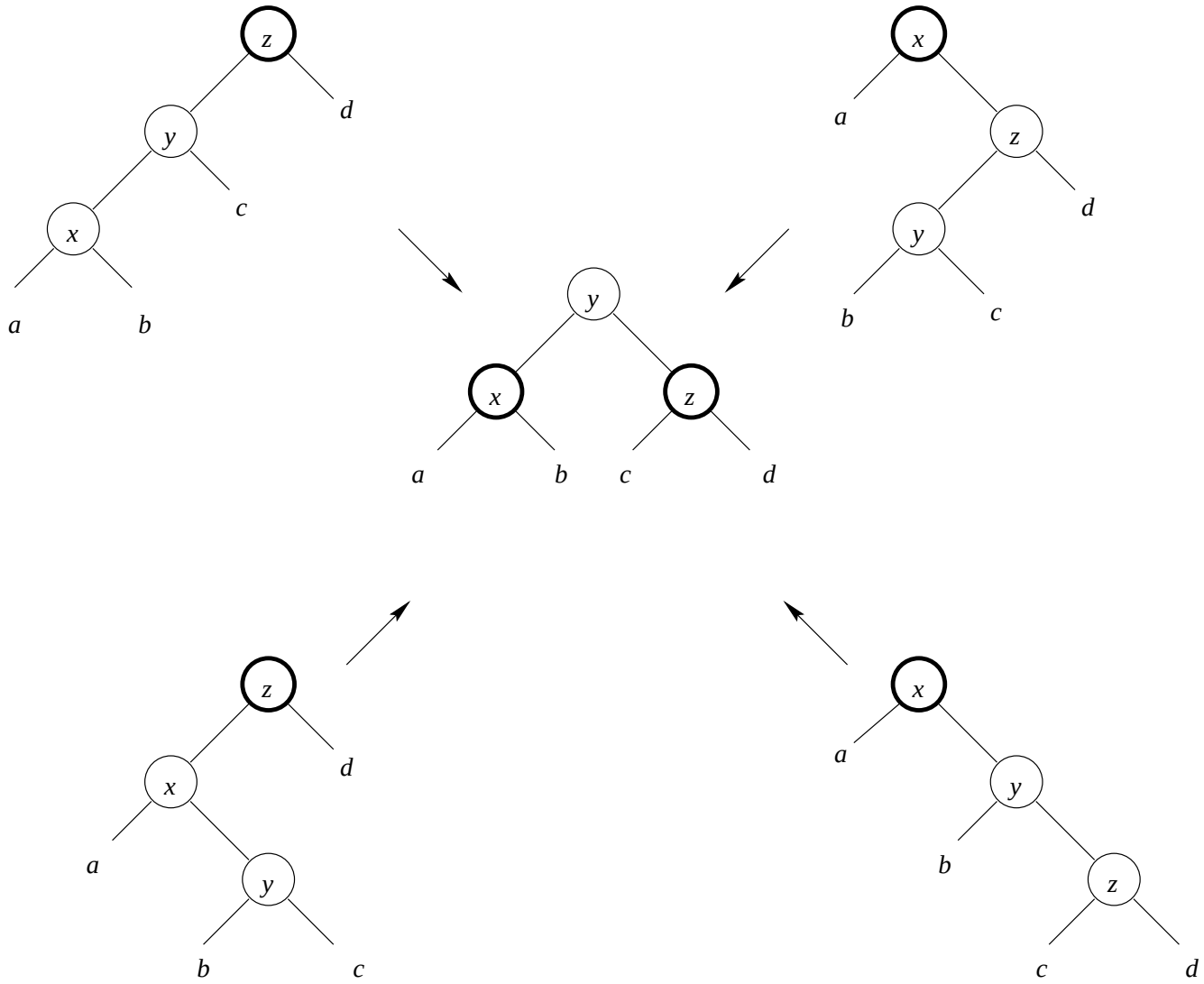
Se dvěma bílými uzly nad sebou a s černým uzlem nad nimi provedeme takovou rotaci, abychom snížili hloubku stromu, ale přebarvíme je tak, aby černÆ hloubka stromu zůstala zachovÆna: kořen bude bílý a jeho dva nÆslednící černí.

Tím se opět mohly dostat pod sebe dva bílØ uzly. Proto celý postup opakujeme tak dlouho, dokud jsou někde pod sebou dva bílØ uzly.

Zůstane-li bílý kořen, přebarvíme ho na černo. Tím se černÆ hloubka celØho stromu zvýší o jedničku.

PřidÆvÆní uzlu

```
insert ::  a → CBS a → CBS a
insert k s = N Ce y tl tr
             where N _ y tl tr       = ins s
                   ins E             = N Bi k E E
                   ins t@(N b y l r)
                       | k < y       = bal (N b y (ins l) r)
                       | k > y       = bal (N b y l (ins r))
                       | otherwise   = t

bal :: CBS a → CBS a
bal (N Ce z (N Bi y (N Bi x a b) c) d)  =  rt
bal (N Ce z (N Bi x a (N Bi y b c)) d)  =  rt
bal (N Ce x a (N Bi z (N Bi y b c) d))  =  rt
bal (N Ce x a (N Bi y b (N Bi z c d)))  =  rt
bal t                                  =  t
    where  rt  =  N Bi y (N Ce x a b) (N Ce z c d)
```


z
y
x
a
b
c
d
x
a
z
y
b
c
d
y
x
z
a
b
c
d
z
x
y
a
b
c
d
x
a
y
b
z
c
d

Složitost operace přidÆní uzlu

Věta: Operace `insert` přidÆní položky dočernobílØho stromu velikosti n mÆ časovou složitost $\Theta(\log n)$.

Důkaz: VyplývÆ z logaritmickØ hloubky černobílØho stromu a z toho, že operace vyvÆženíbal mÆ konstantní složitost.

Cvičení: Proč se při přidÆvÆní položky do černobílØho stromu obarvuje kořen celØho stromu na černo?

Cvičení: Na vstupu je posloupnost $6, 5, 4, 2, 3, 1, 0$, z jejíchž prvků se postupně vytvÆří černobílý strom. JakØ jsou tyto stromy v každØm kroku?

Cvičení: Nechť černožlutobílý strom je binÆrní strom splňující podmínky:

- každý uzel je obarven jednou barvou — černou, žlutou, anebo bílou
- počet černých uzlů na každØ větvi je stejný
- bezprostřední předchůdce žlutØho uzlu nesmí být žlutý
- bezprostřední předchůdce bílØho uzlu nesmí být bílý ani žlutý
- kořen stromu je vždy černý

JakÆ je minimÆlní a jakÆ maximÆlní hloubka černožlutobílØho stromu na n uzlech? Dokažte.

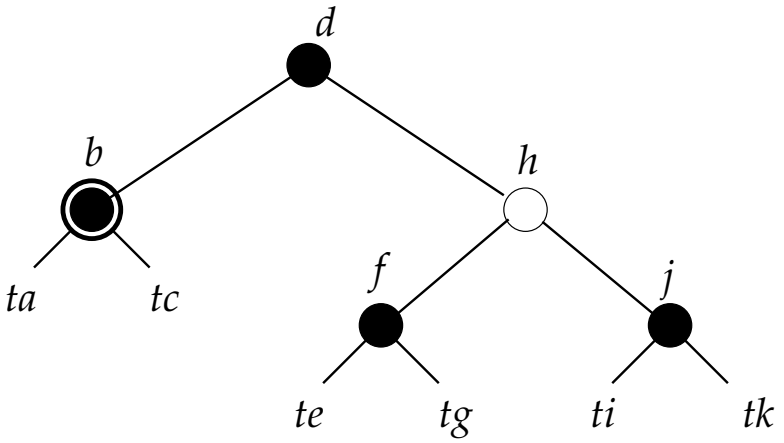
Rušení uzlu z černobílého stromu

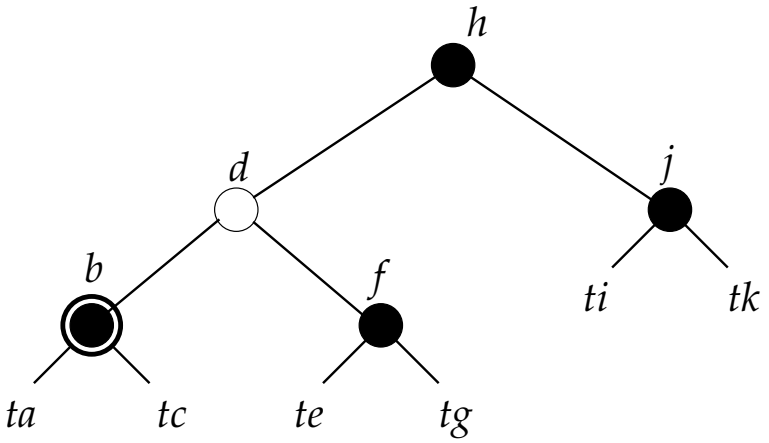
Rušení vnitřního uzlu převedeme známým způsobem na rušení listu. Je-li rušený list bílý, je zrušení triviální — strom i bez listu zůstane černobílý. Je-li rušený list černý, je nutno ho nejdříve „odbarvit“. Odbarvíme-li černý list, stane se tento list bílým a můžeme ho snadno zrušit.

Operace odbarvení spočívá v přesunutí přebytečné černé barvy blíže ke kořenu tak, aby černé délky všech větví zůstaly stejné. Přitom může dojít k „přibarvení“ černého uzlu – přesuneme-li černou barvu na uzel, který byl sám černý, stane se tento uzel „dvojnásobně černý“ a je nutno ho dále odbarvovat (přesouvat čerň stále blíže ke kořenu), aby byl každý uzel „nejvýše jednou černý“.

Stane-li se dvojnásobně černý kořen celého stromu, přebytečnou černou barvu z něho smažeme a necháme ho „jednoučerný“. Tím se sníží černá hloubka celého stromu.

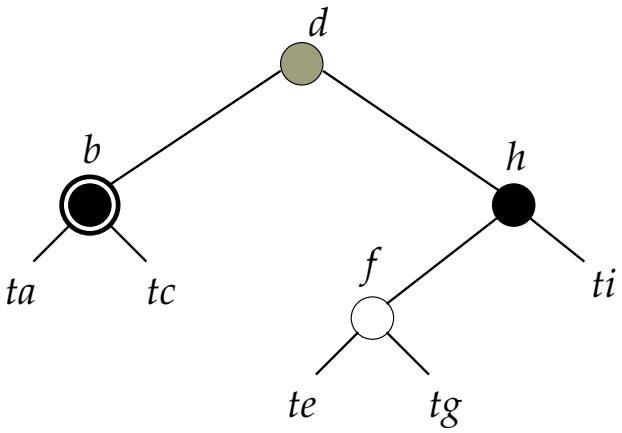
1. Bratr odbarvovanØho uzlu je bílý — převede se na případ 2

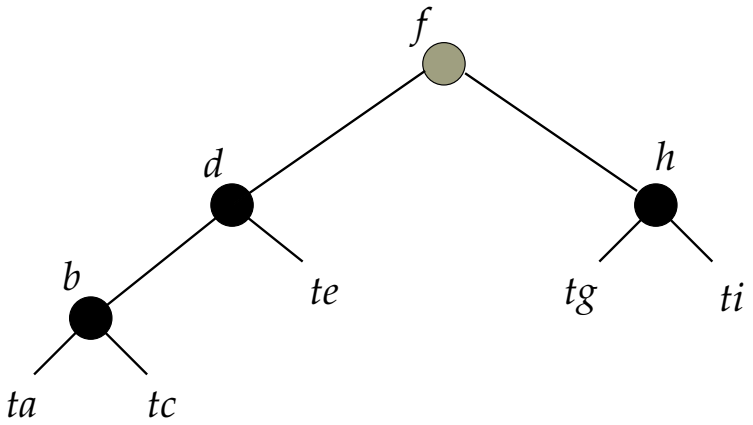




2. Bratr odbarvovanØho uzlu ječerný

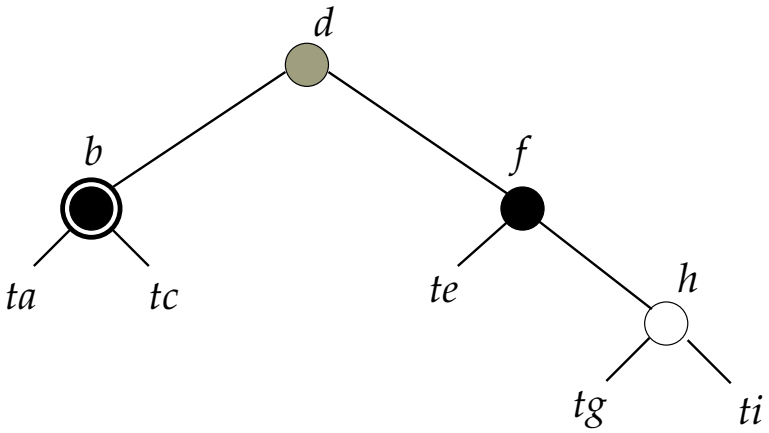
   2a bližší synovec je bílý

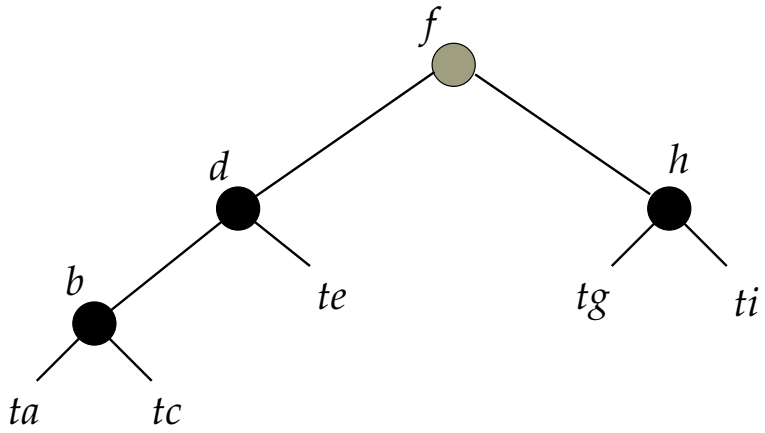




2. Bratr odbarvovanØho uzlu je černý

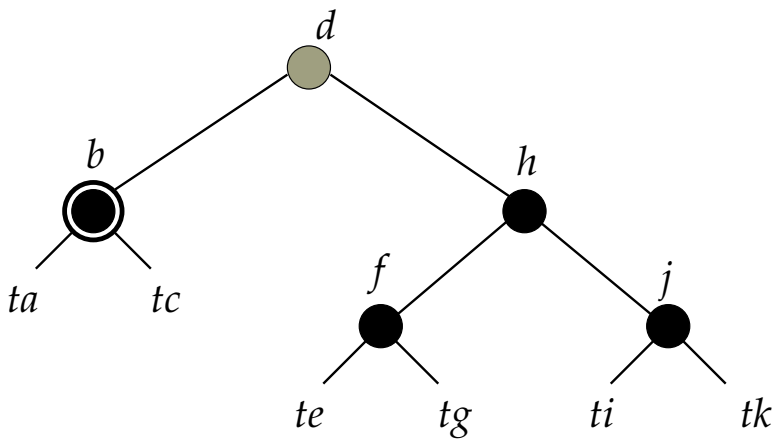
   2b vzdÆlenější synovec je bílý

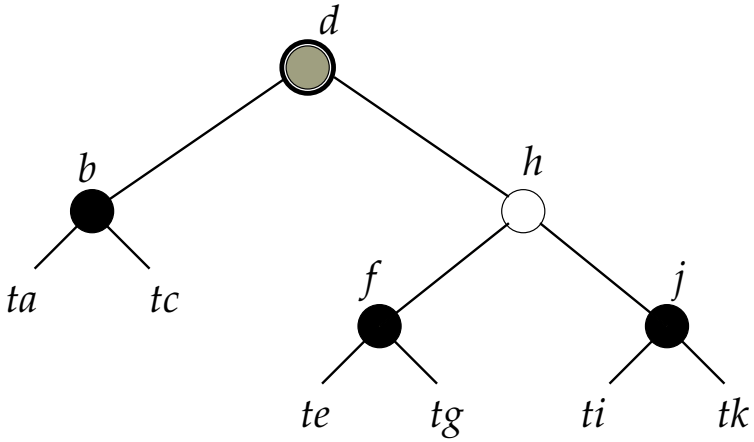




2. Bratr odbarvovanØho uzlu ječerný

   2c žÆdný synovec není bílý





Rušení uzlu

```
delete :: a → CBS a → CBS a
delete k t = cbs (del t)
             where del  E           = E
                   del (N b v l r)
                       | k < v     = lbal b v (del l) r
                       | k > v     = rbal b v l (del r)
                       | otherwise  = join b l r

join :: Barva → CBS a → CBS a → CCBS a
-- definice jako u binárního vyhledávacího stromu
```

```
lbal :: Barva → CCBS a → CBS a → CCBS a
lbal Ce d (CeCe tb) (N Bi h tf tj)              =  -- (1)
        N Ce h (lbal Bi d (CeCe tb) tf) tj
lbal co d (CeCe tb) (N Ce h (N Bi f te tg) ti) =  -- (2a)
        N co f (N Ce d tb te) (N Ce h tg ti)
lbal co d (CeCe tb) (N Ce f te (N Bi h tg ti)) =  -- (2b)
        N co f (N Ce d tb te) (N Ce h tg ti)
lbal co d (CeCe tb) (N Ce h tf tj)              =  -- (2c)
        cern (N co d tb (N Bi h tf tj))
lbal co d tb tf                                 =  -- (3)
        N co d tb tf

rbal :: Barva → CBS a → CCBS a → CCBS a
-- analogicky jako lbal
```

```
data  CCBS a  =  CeCe (CBS a)  |  Cbs (CBS a)

cern :: CBS a → CCBS a
cern E = Cbs E
cern (N Bi v l r) = Cbs (N Ce v l r)
cern (N Ce v l r) = CeCe (N Ce v l r)

cbs :: CCBS a → CBS a
cbs (CeCe t) = t
cbs (Cbs t)  = t
```

Složitost rušení uzlu z černobílého stromu

Element rní přesun černé barvy trvá konstantní dobu, ale v nejhorším případě je nutné odbarvovat až ke kořenu, tj. $\Theta(\log n)$.

Nalezení rušeného uzlu a nahrada rušení vnitřního uzlu za rušení listu trvá také logaritmickou dobu, takže celá operace rušení zabere čas $\Theta(\log n)$.

Cvičení: Implementujte v Haskellu vyvažovací operaci `rbal`.

Cvičení: Napište definici funkce `join`.

Další typy vyhledávacích stromů s logaritmickou složitostí operací

B-stromy Stromy s proměnným počtem následníků, ale omezeným zdola číslem k a shora číslem $2k$ pro pevně zvolenou konstantu k, tzv. řád B-stromu (viz PV062). Složitost operací přidání/zrušení položky je $\Theta\left(\log_k n\right)$.

2-3-4-stromy Speciální případ B-stromů. Každý vnitřní uzel má buď dva, tři, anebo čtyři následníky. Mají opět logaritmickou hloubku.