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, statická pole,...) — tzv. *statické datové struktury*
- Datové struktury proměnné velikosti — tzv. *dynamické datové struktury*

Statické datové struktury mají konstantní velikost a časová složitost zpřístupnění libovolného prvku je konstantní.

Dynamické datové struktury mají velikost danou nějakou proměnnou n, jejíž hodnota se může měnit. Časová složitost zpřístupnění libovolného prvku dynamické datové struktury je neklesající funkcí závislou na n; často lineární, u efektivních datových struktur lepší, typicky logaritmická.

Dynamické datové struktury

Seznam

Seznam nad bázovým typem $\mathbb{B}$ je lineární datová struktura typu $\mathbb{S}$ s následujícími operacemi:

$\texttt{nil} : \mathbb{S}$

$\texttt{cons} : \mathbb{B} \times \mathbb{S} \rightarrow \mathbb{S}$

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

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

$\texttt{null} : \mathbb{S} \rightarrow \texttt{Bool}$

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

Axiomy seznamu

$$\texttt{null(nil)} = \texttt{True}$$

$$\texttt{null(cons(}x\texttt{,}s\texttt{))} = \texttt{False}$$

$$\texttt{head(cons(}x\texttt{,}s\texttt{))} = x$$

$$\texttt{tail(cons(}x\texttt{,}s\texttt{))} = s$$

Zásobník

Zásobník nad bázovým typem $\mathbb{B}$ je lineární datová struktura typu $\mathbb{S}$ s následujícími operacemi:

$\texttt{empty} : \mathbb{S}$

$\texttt{push} : \mathbb{B} \times \mathbb{S} \rightarrow \mathbb{S}$

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

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

$\texttt{isempty} : \mathbb{S} \rightarrow \texttt{Bool}$

Pro tyto operace a každou hodnotu x typu B a každý zásobník s typu S musí platit tzv.

Axiomy zásobníku

isempty(empty) = True

isempty(push(x,s)) = False

top(push(x,s)) = x

pop(push(x,s)) = s

Jak je vidět, zásobník a seznam je tatáž datová struktura.

Liší se pouze v použití:

O *zásobníku* obvykle mluvíme, když na něm používáme jen základní operace a pracujeme jen s jedním zásobníkem nebo pevně daným malým počtem zásobníků. Zásobníky bývají dány předem: během výpočtu se mění jejich obsah, ale nemění se počet zásobníků, tj. neruší se a nevznikají nové zásobníky. Proto se v imperativních jazycích základní operace často implementují jako procedury; navíc jejich parametr zásobník se vynechává, pracuje-li se jen s jedním zásobníkem.

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

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

Fronta

Fronta nad bázovým typem $\mathtt{B}$ je lineární datová struktura typu $\mathtt{Q}$ s následujícími operacemi:

$\mathtt{empty} : \mathtt{Q}$

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

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

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

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

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

Axiomy fronty

`isempty(empty) = True`

`isempty(enqueue(`x`,`q`)) = False`

`head(enqueue(`x`,empty)) = `x

`head(enqueue(`x`,enqueue(`y`,`q`))) = head(enqueue(`y`,`q`))`

`dequeue(enqueue(`x`,empty)) = empty`

`dequeue(enqueue(`x`,enqueue(`y`,`q`))) =`

`enqueue(`x`,dequeue(enqueue(`y`,`q`)))`

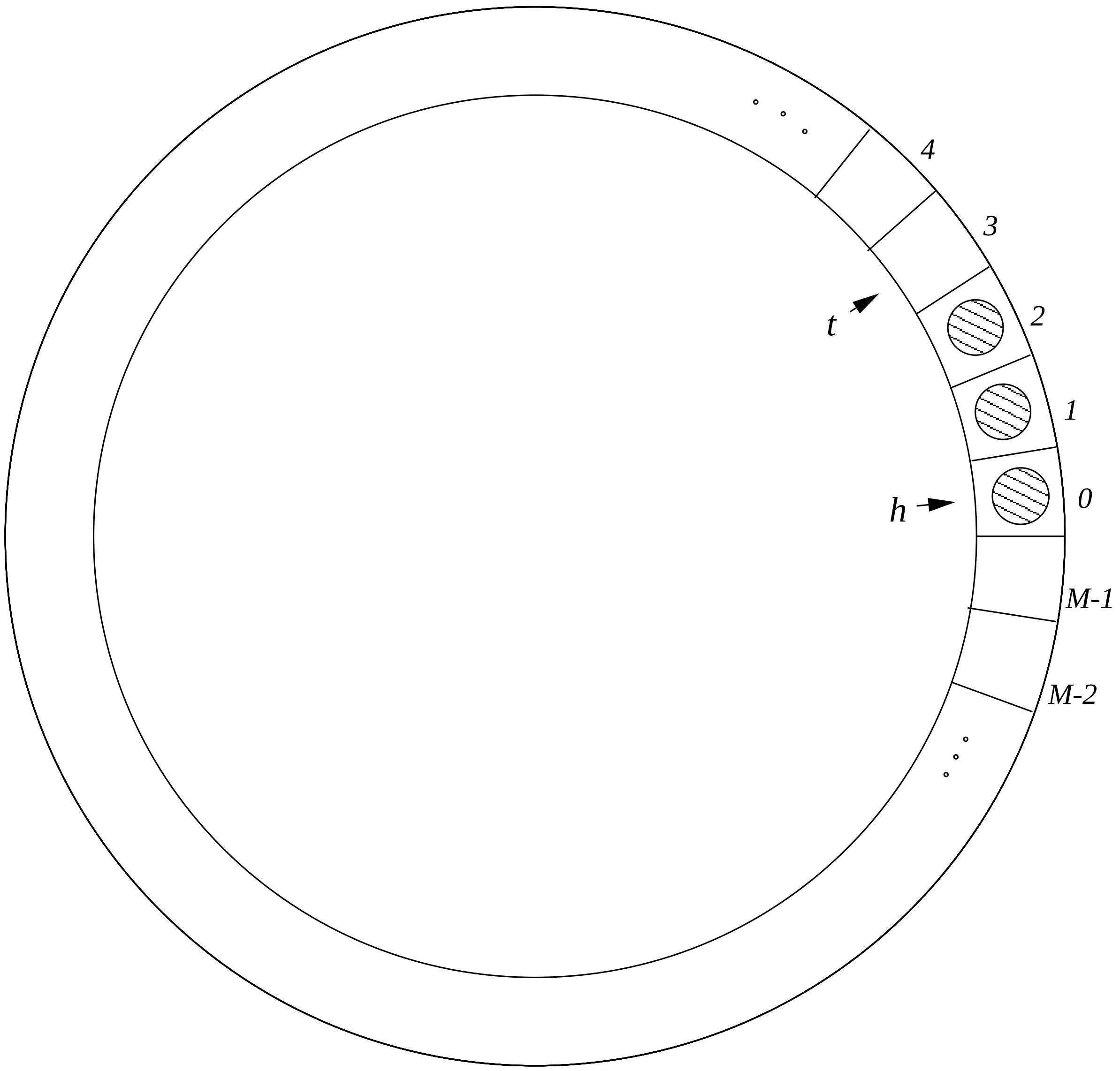
Implementace datových struktur

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.

Poznámka: Jazyky s malou podporou dynamických datových struktur (Pascal, C,...) často poskytují alespoň dynamickou datovou strukturu „nízké úrovně“: *paměť* $\mathtt{M}$ spolu
s typem *ukazatel* (adresa) $\mathtt{P}$,
nulární operací $\mathtt{nil:P}$ prázdného ukazatele
a unární operací zpřístupnění (dereferencování) části paměti $\mathtt{deref:P} \dashrightarrow \mathtt{M}$.

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


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

Příklad: Implementace ohraničené fronty pomocí cyklického 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.

Amortizovaná časová složitost

Def: Nechť f je operace na dané datové struktuře D velikosti (nejvýše) n. Uvažujme všechny výpočty $C_1, \ldots, C_m$ podle algoritmů pracujících s datovou strukturou D. Z každého takového výpočtu vybereme všechna volání operace f (tj. všechny aplikace f na D), čímž dostaneme m posloupností volání operace f. Průměrnou délku výpočtu operace f v i-tém výpočtu označíme τ_i^{av}. Potom *amortizovaná složitost* operace f na datové struktuře D je funkce $T^{\text{amort}} : \mathbb{N} \to \mathbb{N}$ definovaná pro každou velikost dat n takto

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

Příklad: Amortizované 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 orientovány 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 $u_1, \ldots, u_k$ nazýváme *(bezprostředními) následníky* uzlu u.

Na bezprostředních následnících každého uzlu často bývá zavedeno úplné uspořádání. Pak se následníci uzlu znázorňují zleva doprava; u binárních stromů se tedy rozlišuje *levý* a *pravý* následník.

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

Stromy pevné arity (s omezeným větvením)

Def: Je dáno přirozené číslo n a tzv. bázová množina (typ) B. Definujeme *n-ární strom* nad B takto:

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

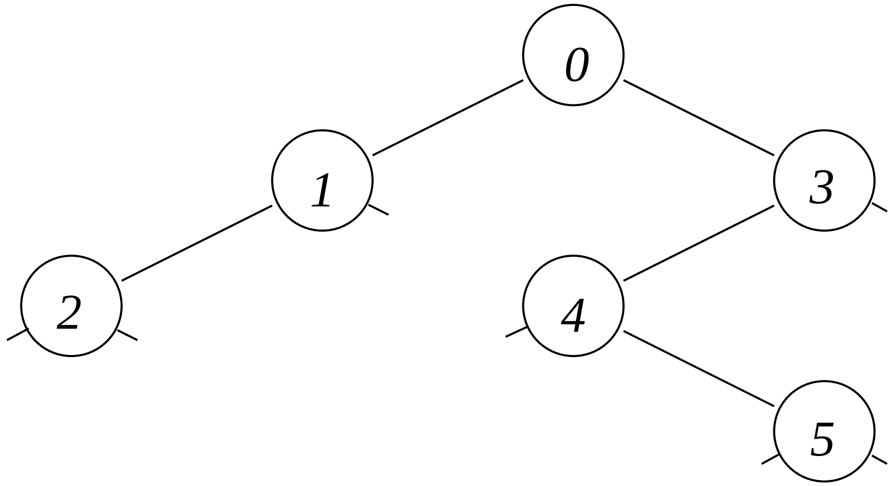
Listy n-árního stromu jsou uzly tvaru $(x, \emptyset, \ldots, \emptyset)$

Def: Cestu z kořene stromu do uzlu, jehož aspoň jeden bezprostřední následník je prázdný strom, se nazývá *větev* stromu.

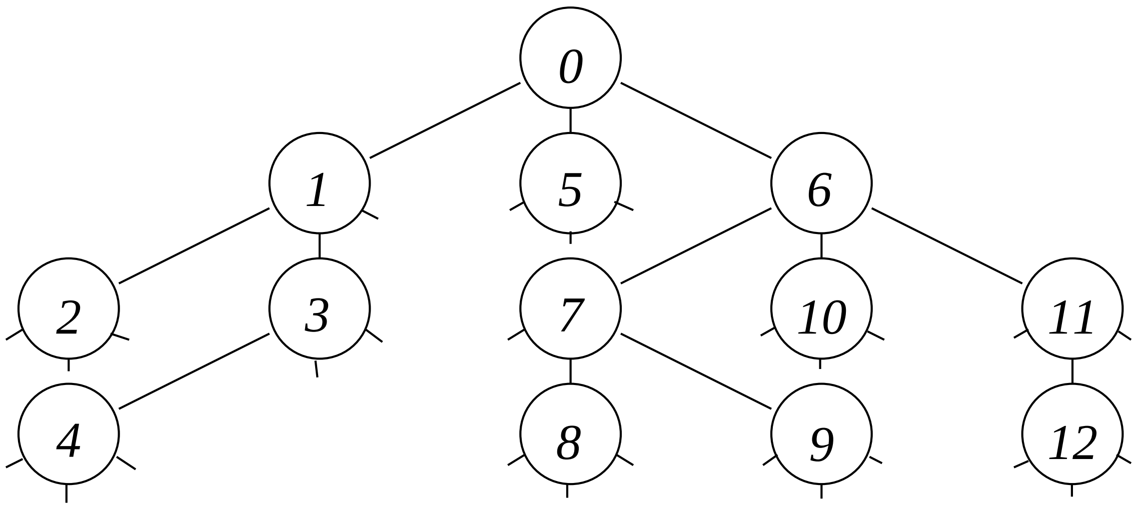
Opovídající datový typ zapsaný v Haskellu např. pro $n = 3$ je

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

Binární strom (s pevným pořadím následníků)



Ternární strom (s pevným pořadím následníků)



Binární stromy

Def: *Binární strom* je strom arity 2.

Základní operace nad binárním stromem

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

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

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

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

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

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

Poznámka: Základní operace mají konstantní složitost, slouží k popisu datové struktury a k její implementaci a k implementaci dalších operací, jako například vyhledání / přidání / odebrání uzlu, vyvážení stromu a podobně.

Pro základní operace, každou hodnotu x typu B a každé binární stromy l, r typu T musí platit

Axiomy binárního 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 binárních 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 dána bázová množina (typ) B. Nechť $b \in B$. Pak pro každé přirozené číslo $k \geq 0$ a každou k-prvkovou posloupnost neprázdných stromů nad B je dvojice $(b, [T_1, \ldots, T_k])$ neprázdným *stromem s neomezeným větvením* nad B.

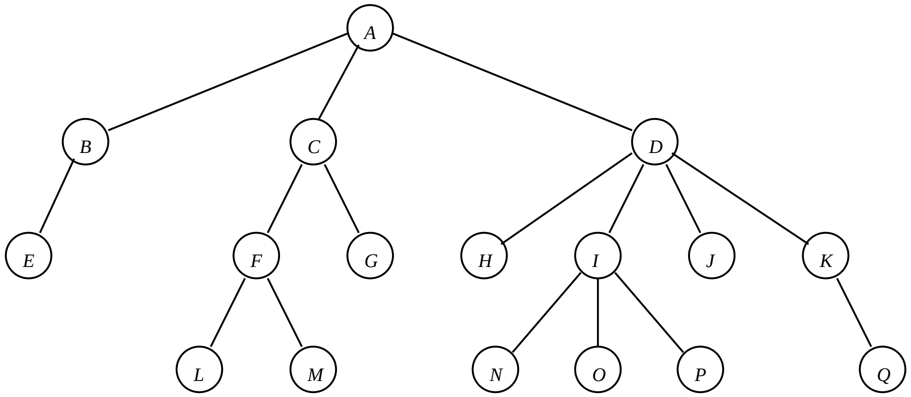
Poznámka: Druhou složkou uspořádané dvojice $(b, [T_1, \ldots, T_k])$ je k-prvková posloupnost stromů, tj. seznam délky k. Je-li $k = 0$, je seznam prázdný a dvojice reprezentuje jednouzlový strom (list) s ohodnocením b.

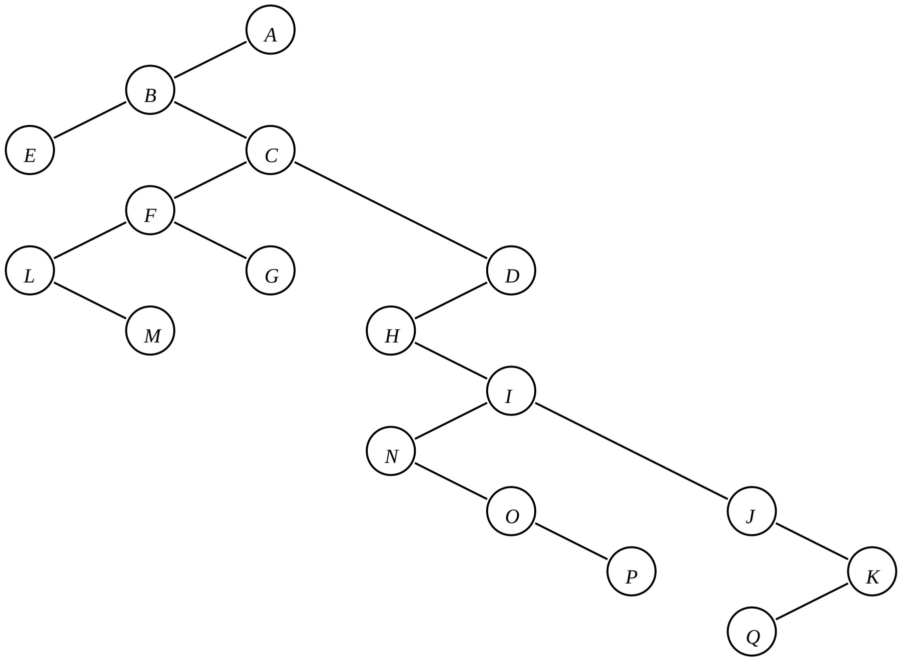
Stromy s neomezeným větvením se od stromů pevné arity liší tím, že číslo k není předem pevně dané, ale může být v jednom stromu pro různé uzly různé.

Poznámka: Stromy s neomezeným větvením lze reprezentovat binárními stromy.

Stromy s neomezeným větvením a binární 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

Vyhledávání

Def: Nechť $(K, \leq)$ je úplně uspořádaná množina tzv. *klíčů*, V je libovolná množina tzv. doplňujících údajů. Nechť $U = K \times V$ je množina dvojic (k, v), v níž se každý klíč k vyskytuje nejvýše jednou (tj. každý záznam z množiny U je určen jednoznačně svým klíčem).

Problému nalézt k danému klíči k záznam $(k, v) \in U$ se říká *vyhledávací problém*.

Poznámka: Například záznamy mohou být osobní data a klíči jsou rodná čísla, anebo záznamy jsou údaje o knihách v knihovně a klíče jsou knihovní signatury apod.

Poznámka: V praxi má většinou smysl pouze případ, kdy množina V je netriviální, aby bylo co hledat.

V ukázkových algoritmech se však doplňující údaje často neuvažují, tj. vyhledávají se jen klíče. Příslušné rozšíření těchto algoritmů je totiž přímočaré.

Algoritmus binárního vyhledávání

```
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 binárního vyhledávání má logaritmickou časovou složitost, tj. jeho složitost je v $\Theta(\log n)$, kde n je délka prohledávaného pole.

Poznámka: Extrasekvenční paměťová složitost algoritmu je konstantní, protože rekursivní volání v něm jsou *prostá*.

Cvičení: Implementujte algoritmus binárního vyhledávání bez použití rekurse.

Cvičení: Dokažte totální korektnost algoritmu binárního vyhledávání.

Hašování

O množině K všech možných klíčů obecně předpokládáme jen to, že na ní existuje úplné uspořádání. Někdy však tato množina může mít další speciální vlastnosti, jichž lze využít pro efektivní vyhledávání. Například je-li $K = \{1, \ldots, m\}$ a číslo m je malé, můžeme data z množiny V uložit do jednorozměrného pole a klíče využít jako indexy. Vyhledávání v takovéto datové struktuře je efektivní – stejně rychlé jako indexování 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 důležité, aby výpočet hodnot funkce h byl rychlý, v ideálním případě aby měl konstantní časovou složitost. Funkci h nazýváme *hašovací funkcí* a pole indexované jejími hodnotami nazýváme *hašovací tabulkou*.

Hašovací tabulky

Hašovací tabulka je datová struktura, pomocí níž lze prakticky efektivně realizovat „slovníkové“ operace vyhledání, přidání a zrušení položky.

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

Hašovací tabulka je jednorozměrné pole H indexované čísly $1, \ldots, n$. Převod klíčů na čísla realizuje *hašovací funkce* $h : K \to \{1, \ldots, n\}$. Výpočet hodnot hašovací funkce musí být efektivní, nejlépe 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 ukládat do n-prvkového 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 následující podmínka. Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, která budou skutečně uložena (tedy $|K'| \leq n$). Pak požadujeme, aby z každé množiny klíčů, které hašovací funkce zobrazí na stejný index, byl v množině K' nejvýše jeden klíč. Má-li pro pevně danou množinu K' klíčů skutečně uložených dat hašovací funkce tuto vlastnost, nazýváme ji *dokonalou hašovací funkcí pro klíče z* K'. Výhodou tabulek s dokonalými hašovacími funkcemi je optimální složitost vyhledání, vložení i zrušení prvku; je stejná jako pro pole.

Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, která budou skutečně uložena. Klíče z množiny K' nazveme *použité klíče*.

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

Věta: Má-li výpočet hašovací funkce konstantní složitost a hašování je dokonalé pro množinu použitých klíčů K', pak operace vyhledání, vložení, resp. zrušení prvku v hašovací tabulce mají stejnou časovou složitost, jako vyhledání, vložení, resp. zrušení prvku v poli.

Nevýhodou dokonalých hašovacích funkcí je, že závisí na množině K'. Tato množina musí být známa předem, abychom mohli dokonalou hašovací funkci sestrojit. V praxi však ukládaná data předem neznáme a tato data se mění. Proto v praxi používané hašovací funkce většinou nebývají dokonalé a musí se počítat s takzvanými *kolizemi*.

Kolize je případ, kdy má být více dat uloženo na jedné 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 pole, ale 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 rovnoměrné, ale dobře navržená hašovací funkce transformuje toto rozložení na rovnoměrné v množině indexů $\{1, \ldots, n\}$. To znamená, že pro náhodně zvolený klíč $k \in K'$ a index i, $1 \leq i \leq n$, je pravděpodobnost, že hodnota $h(k) = i$, stejná nezávisle na k a i a rovná $\dfrac{1}{n}$.

Nyní pro jednoduchost předpokládejme, 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 celá 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 nezávisle na K (což je v praxi splněno) mají hodnoty $h(k)$ rovnoměrné 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 $14, 5, 28, 19, 15, 20, 33, 12, 17, 10, 18, 24$.

Binární vyhledávací stromy

Def: *Binární vyhledávací strom* je binární strom nad úplně uspořádanou 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 binárními vyhledávacími stromy

Datový typ

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

Zjišťování 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
```

Vyhledávání

```
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
```

Vkládání 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 vyhledávací 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 binární strom vzniklý vložením resp. zrušením uzlu z vyhledávacího stromu pomocí funkcí `insert` resp. `delete` bude opět vyhledávací.

Cvičení: Modifikujte funkce `member`, `search`, `insert`, `delete` tak, aby pracovaly s realističtějším vyhledávacím 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 vyhledávání ve vyhledávacích stromech

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

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

kde h je hloubka vyhledávacího stromu, h' je hloubka jeho levého podstromu, h'' je hloubka jeho pravého podstromu a c, c' jsou konstanty.

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

Složitost vyhledávání ve vyhledávacích stromech

Složitosti operací vyhledání, přidání a zrušení položky ve vyhledávacím stromě jsou přímo úměrné 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

Nazvané podle G. M. Adeľsona-Veľského a E. M. Landise.

Def: Vyhledávací binární strom je *AVL*, když hloubka levého a pravého podstromu libovolného uzlu se liší nejvýše o jednu.

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

AVL stromy tedy mají logaritmickou hloubku — použijeme-li je jako vyhledávací stromy, pak má operace vyhledání položky logaritmickou složitost.

Def: *Fibonacciho strom* řádu 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 *minimální* (vzhledem k počtu uzlů) AVL strom hloubky $k-1$.

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

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

Minimalita je důsledkem předchozích dvou bodů: odebráním nějakého uzlu z jiné než nejpravější větve by někde vznikly sousední podstromy s hloubkami lišícími se aspoň o dvě; odebráním uzlu z nejpravější větve by se snížila hloubka celého stromu.

Věta: Hloubka AVL stromu v závislosti na počtu jeho uzlů je vždy v $\Theta(\text{log})$.

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

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

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

Protože Fibonacciho strom je minimální AVL strom, máme, že počet uzlů *každého* AVL stromu hloubky h roste aspoň tak rychle jako φ^h. Obráceně, každý AVL strom velikosti n má hloubku $O(\log_\varphi n)$.

Víme, že hloubka každého binárního stromu velikosti n je $\Omega(\log_2 n)$.

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

Operace na AVL stromech

Datová struktura:

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

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

Vyhledávání v AVL stromu se neliší od vyhledávání v běžném vyhledávacím stromu.

Po přidání nebo odebrání položky ovšem může nastat situace, že vyhledávací strom přestane splňovat podmínku AVL. Pak je nutné pozměnit strukturu stromu.

Přidávání položky (nového uzlu) do AVL stromu

Stejné jako u běžného vyhledávacího stromu, ale s kontrolou vyváženosti.

Přidávaný uzel označíme x.

Pokud se poruší vyváženost stromu, nalezne se nejmenší podstrom F, který je nevyvážený. Označíme-li h jeho hloubku před přidáním uzlu x, bude po přidání jeho hloubka $h + 1$. Kořen stromu F označíme f.

Bez újmy na obecnosti lze předpokládat, že uzel x je přidáván do *levého* podstromu stromu F. Nechť B je levý podstrom stromu F, G je pravý podstrom stromu F.

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

Rozlišíme dva případy:

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

   Vytvoříme strom

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

Dva dílčí podpřípady jsou:

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

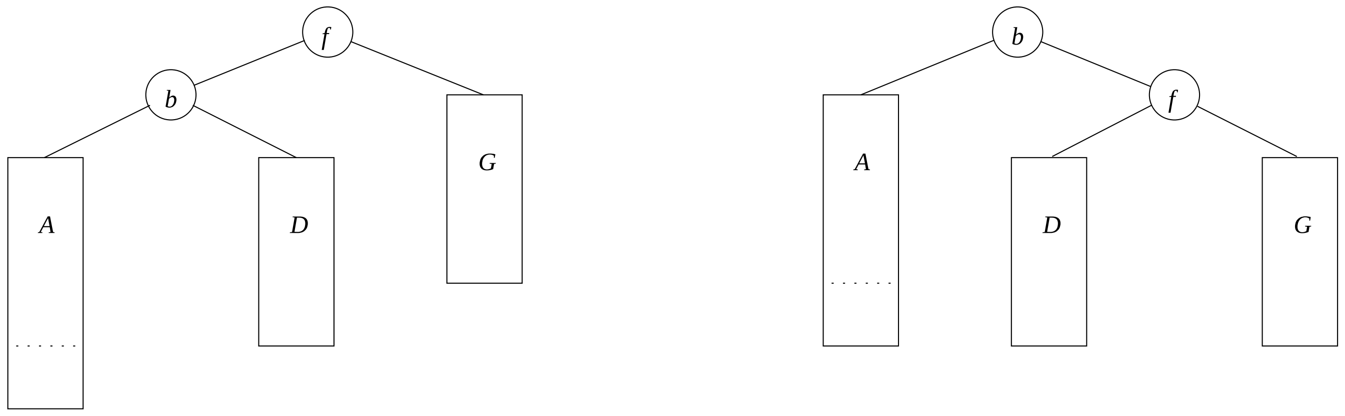
(b) Uzel x byl přidán do stromu E. Pak C má hloubku $h-3$, E má hloubku $h-2$.

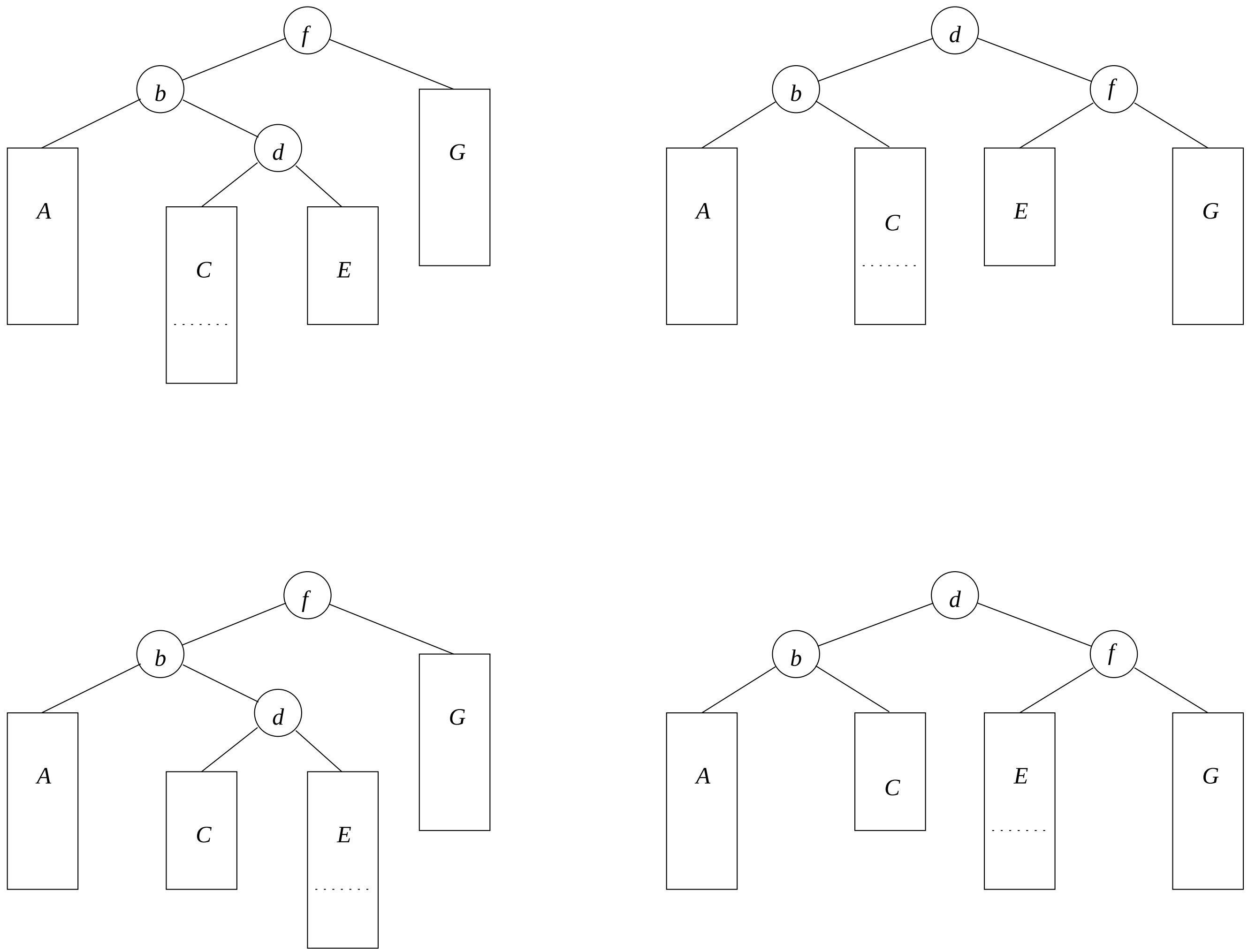
V obou případech však strom T má hloubku h, je AVL a má stejné uzly jako strom F.

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

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

Byl-li uzel x přidáván do *pravého* podstromu stromu F, je postup analogický (stranově převrácený).


f
b
G
A
D
b
f
A
D
G


f
b
d
A
C
E
G
d
b
f
A
C
E
G
f
b
d
A
C
E
G
d
b
f
A
C
E
G

Složitost operace přidání položky do AVL

Hloubka AVL stromu je logaritmická vzhledem k počtu jeho uzlů.

Přidání uzlu x: $\Theta(\log n)$.

Nalezení nejmenšího nevyváženého podstromu:

Protože hloubky podstromů jsou spočteny a uloženy v datové struktuře, stačí po cestě k uzlu x testovat, zda rozdíl spočtených hloubek levého a pravého podstromu každého uzlu je v množině $\{-1, 0, 1\}$. Nejnižší uzel, který tuto podmínku nesplňuje, je kořen podstromu F. Jeho nalezení trvá $\Theta(\log n)$.

Hloubky podstromů se přepočítávají jen po cestě od kořene k x, tedy složitost této úpravy je $\Theta(\log n)$.

Celková složitost přidání uzlu je tedy $\Theta(\log n)$.

Rušení položky (uzlu) z AVL stromu

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

Odebíraný list označíme x.

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

Bez újmy na obecnosti lze předpokládat, že uzel x byl odebrán z *levého* podstromu stromu B. Nechť A je levý podstrom stromu B, F je pravý podstrom stromu B.

Strom F je neprázdný (jinak by rušení uzlu x nemohlo porušit vyváženost 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

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

   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 stát, že hloubka podstromu T bude menší než hloubka původního podstromu, který byl na jeho místě.

Proto je nutné proces vyvažování opakovat: nalezne se nejmenší nadstrom T_2 stromu $T = T_1$, který není vyvážený, vyvážíme ho, nalezneme další nevyvážený nadstrom T_3... atd., až je vyvážený celý strom, T_k.

Hloubky však stačí přepočítávat vždy od kořene stromu T ke kořenu nejbližšího vyvažovaného nadstromu.

Složitost operace rušení položky z AVL

Zrušení listu: $\Theta(\log n)$.

Nechť hloubka nejmenšího nevyváženého podstromu T_1 je h_1. Nalezení tohoto podstromu trvá $\Theta(h_1)$, jeho vyvážení (rotace uzlů) trvá konstantní dobu, přepočítání hloubek trvá $\Theta(h_1)$.

Nechť pro $1 < i \leq k$ je h_i délka cesty z kořene podstromu T_{i-1} do kořene stromu T_i. Pak každé nalezení dalšího nevyváženého podstromu T_i trvá $\Theta(h_i)$, jeho vyvážení trvá $\Theta(1)$, přepočítání hloubek trvá $\Theta(h_i)$.

To znamená, že celková 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 načítání se postupně vytváří AVL strom s uzly ohodnocenými těmito čísly. Nakreslete tento AVL strom v každém kroku vytváření (tj. po přidání každého uzlu).

Cvičení: Do AVL stromu, který je zpočátku prázdný, se postupně přidávají položky $4, 1, 2, 7, 5, 6$, pak se odebere položka 7, přidá položka 3, odeberou se postupně $5, 6, 1$. Určete stav AVL stromu v každém kroku.

Cvičení: AVL strom má uzly ohodnocené čísly 1 až 20 a má strukturu Fibonacciho stromu šestého řádu. Popište a nakreslete proces rušení listu obsahujícího klíč 2.

Cvičení: Fibonacciho strom čtvrtého řádu na sedmi uzlech je ohodnocen jako AVL strom čísly $1, 3, 5, 7, 9, 11, 13$. Do tohoto stromu přidáme jako další položku náhodně zvolené sudé číslo k, $0 \leq k \leq 14$. Jaká je pravděpodobnost, že budeme muset rotovat uzly, abychom zachovali AVL vyváženost?

Černobílé stromy

Černobílé stromy jsou binární vyhledávací stromy, jejichž uzly nesou kromě klíče další atribut — *barvu* — černou nebo bílou.

Def: *Černobílý strom* je binární vyhledávací 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 následníci (pokud existují) jsou černí.
3. Všechny větve obsahují stejný počet černých uzlů.

Poznámka: Vedle názvu *černobílé stromy* se lze setkat i s doslovnými překlady anglického názvu *red-black tree* (drzewo czerwono-czarne, rot-schwarzer Baum, ruĝnigra arbo, vörös-fekete fa,. . .).

Def: *Černá hloubka* černobílého stromu t je počet černých uzlů na libovolné větvi. Značíme ji $\mathsf{bh}(t)$.

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

Důkaz: Indukcí podle hloubky stromu se ukáže, že každý černobílý strom t' má aspoň $2^{\mathsf{bh}(t')} - 1$ uzlů. Odtud vyplývá $\mathsf{bh}(t') \leq \log_2(n+1)$. Ale podle definice černobílého stromu jeho hloubka nepřevyšuje dvojnásobek jeho černé hloubky. Odtud plyne tvrzení.

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

Černobílé stromy v Haskellu

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

Vyhledávání v černobílém stromu je stejné jako v nevyváženém vyhledávacím stromu.

Zjišťování 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
```

Vyhledávání

```
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
```

Přidávání uzlu do černobílého stromu

Přidávaný uzel bude bílý. Tím se nezmění černá hloubka podstromů, ale mohou se dostat pod sebe dva bílé uzly.

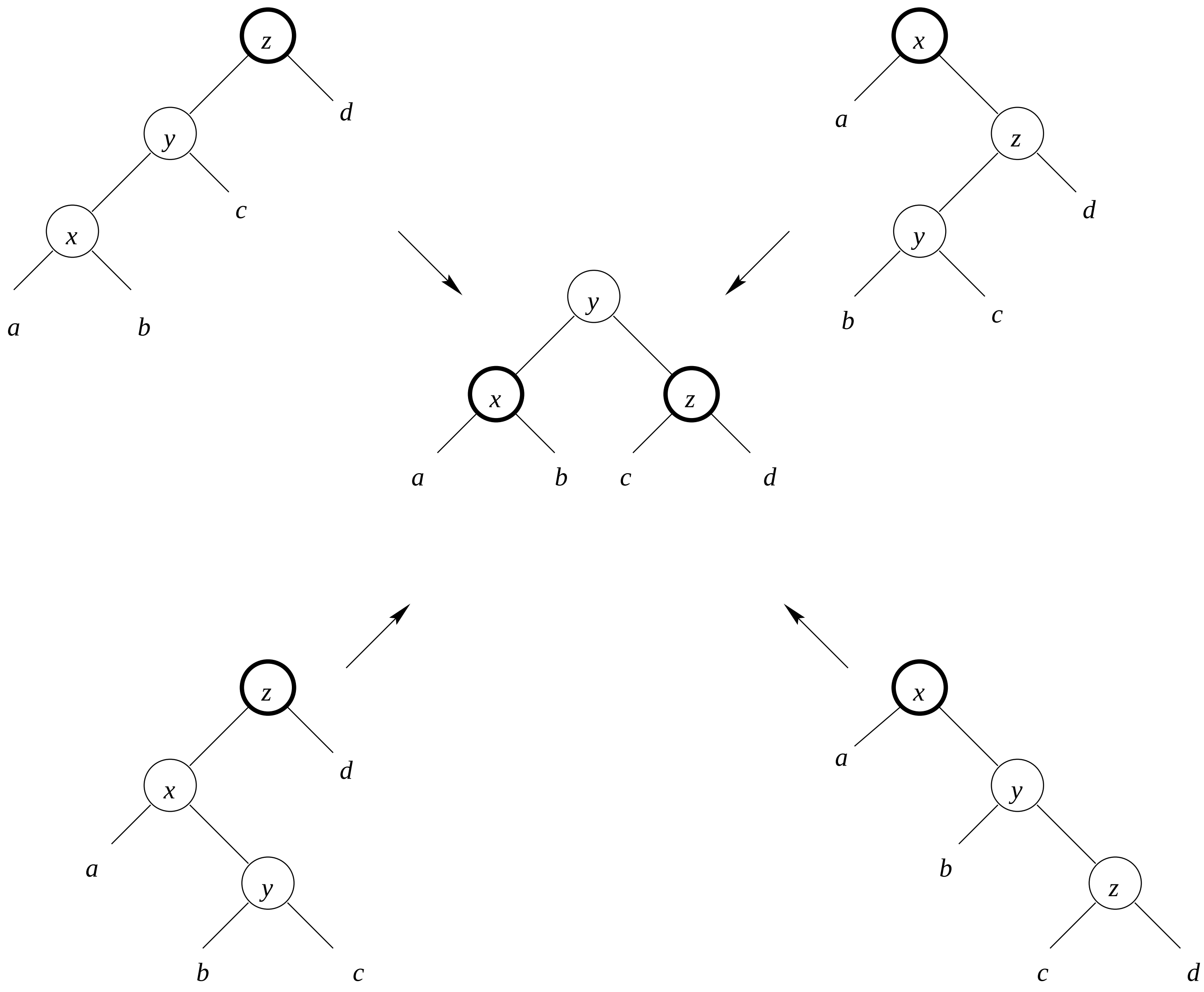
Se dvěma bílými uzly nad sebou a s černým uzlem nad nimi provedeme takovou rotaci, abychom snížili hloubku stromu, ale přebarvíme je tak, aby černá hloubka stromu zůstala zachována: kořen bude bílý a jeho dva následníci černí.

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

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

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

Složitost operace přidání uzlu

Věta: Operace `insert` přidání položky do černobílého stromu velikosti n má časovou složitost $\Theta(\log n)$.

Důkaz: Vyplývá z logaritmické hloubky černobílého stromu a z toho, že operace vyvážení `bal` má konstantní složitost.

Cvičení: Proč se při přidávání položky do černobílého stromu obarvuje kořen celého stromu na černo?

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

Cvičení: Nechť černožlutobílý strom je binární strom splňující podmínky:

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

Jaká je minimální a jaká maximální hloubka černožlutobílého stromu na n uzlech? Dokažte.

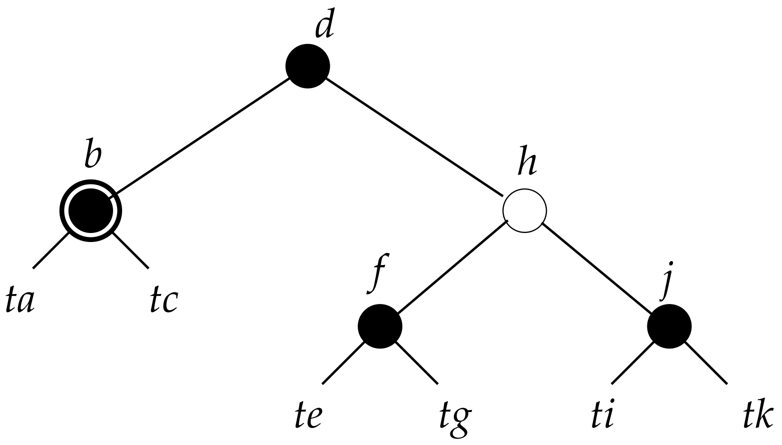
Rušení uzlu z černobílého stromu

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“.

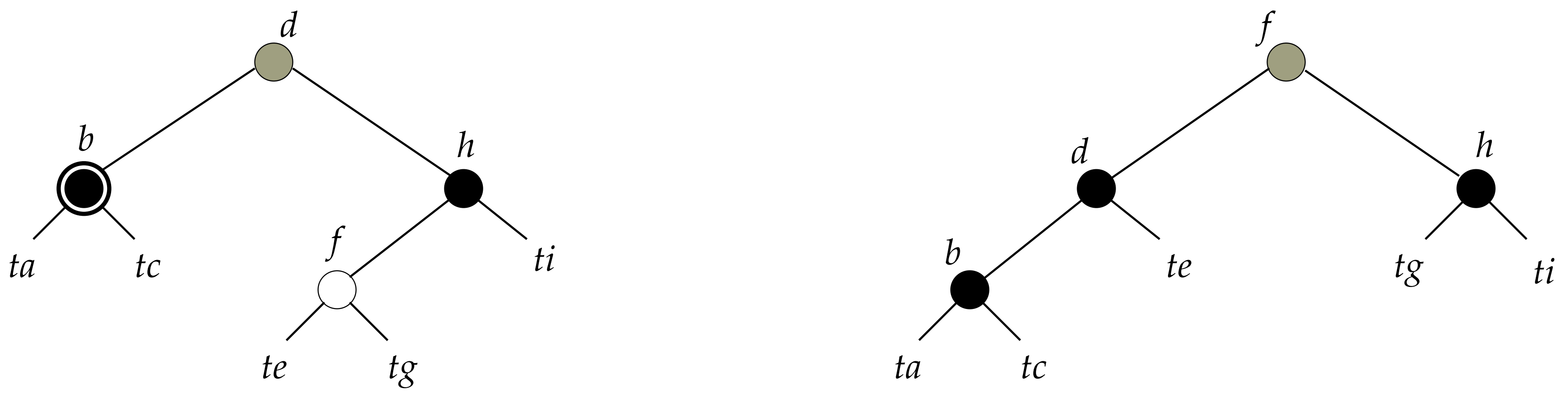
Stane-li se dvojnásobně černý kořen celého stromu, přebytečnou černou barvu z něho smažeme a necháme ho „jednou černý“. Tím se sníží černá hloubka celého stromu.

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



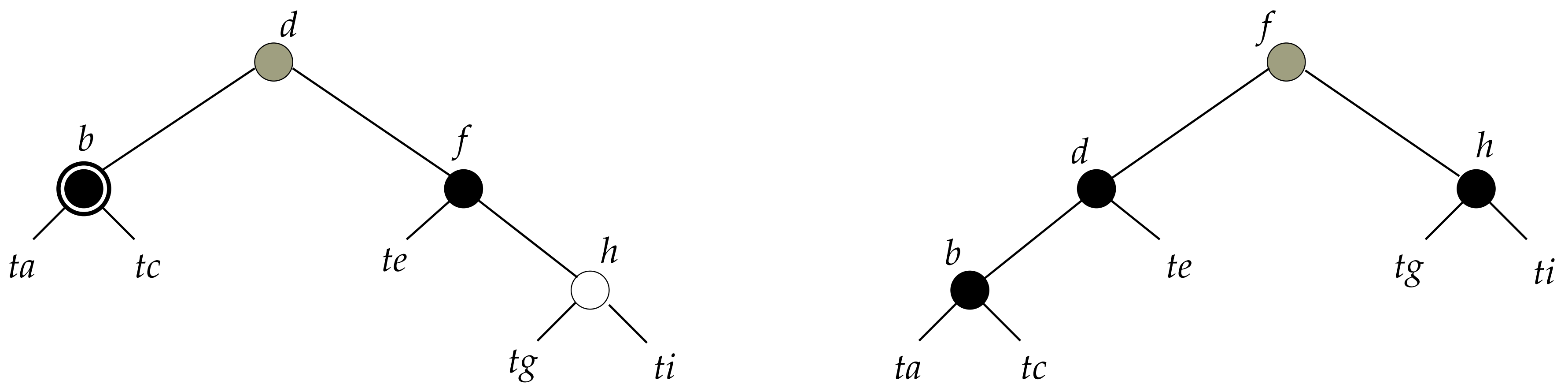
2. Bratr odbarvovaného uzlu je černý

2a bližší synovec je bílý



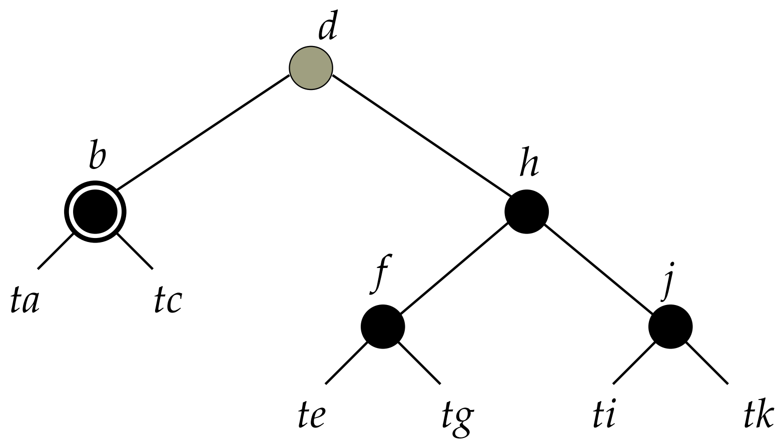
2. Bratr odbarvovaného uzlu je černý

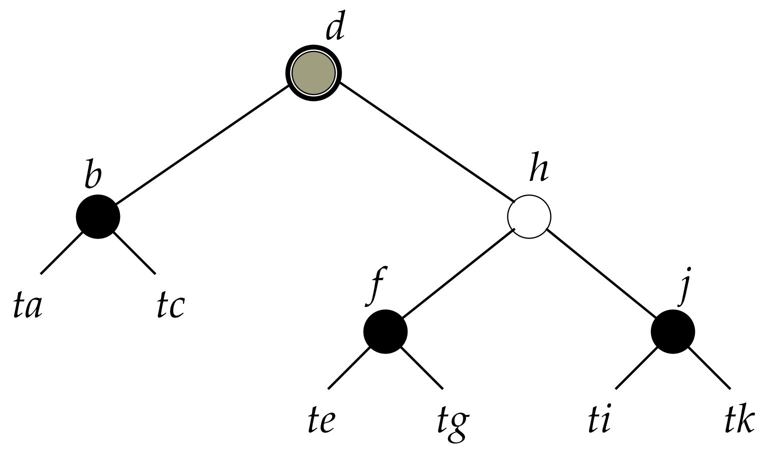
2b vzdálenější synovec je bílý



2. Bratr odbarvovaného uzlu je černý

2c žádný 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

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

Nalezení rušeného uzlu a náhrada 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 $\Theta(\log_k n)$.

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